XSQL开发说明

目录

[XSQL概述 5](#_Toc104565823)

[XSQL创建数据库对象 5](#_Toc104565824)

[XSQL创建多种数据库的对象 8](#_Toc104565825)

[XSQL创建多种数据库的对象并初始化数据 10](#_Toc104565826)

[XSQL应用级触发器 12](#_Toc104565827)

[XSQL数据库SQL自动生成 14](#_Toc104565828)

[XSQL常规List查询 16](#_Toc104565829)

[XSQL带参数的查询 18](#_Toc104565830)

[XSQL函数型占位符（主键ID的生成） 21](#_Toc104565831)

[XSQL自增长ID的返回 22](#_Toc104565832)

[XSQL占位符按IF条件取值（业务逻辑取值） 24](#_Toc104565833)

[XSQL动态参数<[ ... ]>的查询 28](#_Toc104565834)

[XSQL表名可变的SQL 30](#_Toc104565835)

[XSQL表字段名可变的SQL 31](#_Toc104565836)

[XSQL排序组合Order By可变的SQL 32](#_Toc104565837)

[XSQL表分区内查询的可变SQL 32](#_Toc104565838)

[XSQL分页页码可变的SQL 33](#_Toc104565839)

[XSQL统计行数Count(\*) 34](#_Toc104565840)

[XSQL分页查询 34](#_Toc104565841)

[XSQL懒人查询 36](#_Toc104565842)

[XSQL转义数据库关键字 38](#_Toc104565843)

[XSQL查询结果为Map结构的查询 39](#_Toc104565844)

[XSQL查询结果为表分区结构的查询 41](#_Toc104565845)

[XSQL表字段与Java属性名不同时的查询 42](#_Toc104565846)

[XSQL一对一复合型查询 43](#_Toc104565847)

[XSQL一对多复合型查询 47](#_Toc104565848)

[XSQL一行分拆成多行的查询（填充事件） 49](#_Toc104565849)

[XSQL一列分拆成多列的查询（填充事件） 49](#_Toc104565850)

[XSQL多行合并为一行的查询（填充事件） 49](#_Toc104565851)

[XSQL方便的写入CLob数据 50](#_Toc104565852)

[XSQL执行多个SQL语句 52](#_Toc104565853)

[XSQL批量执行 52](#_Toc104565854)

[XSQL预处理的批量执行 53](#_Toc104565855)

[XSQL树结构查询 54](#_Toc104565856)

[XSQL存储过程及函数（无入参的） 58](#_Toc104565857)

[XSQL存储过程及函数（带参数的） 60](#_Toc104565858)

[XSQL分域 63](#_Toc104565859)

[XSQL安全（防止SQL注入攻击） 64](#_Toc104565860)

[XSQL日期时间的处理 65](#_Toc104565861)

[XSQL执行统计分析 66](#_Toc104565862)

[Java执行XSQL 69](#_Toc104565863)

[Java注解@Xsql执行XSQL 69](#_Toc104565864)

修订版本记录

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **版本** | **说 明** | **操作** | **修订者** | **日 期** |
| V1.0 | 初始版本 | 创建 | [HY-ZhengWei](https://github.com/HY-ZhengWei) | 2018-04-18 |
| V1.0 | 审阅 | 审阅 | [Dirful](https://github.com/dirful)、[LHao](https://github.com/343136848)、[MaLong301](https://github.com/malong301) | 2018-05-02 |
| V1.1 | 简单补充一行分拆成多行、一列分拆成多列、多行合并成一行。 | 增加 | [HY-ZhengWei](https://github.com/HY-ZhengWei) | 2018-06-01 |
| V1.2 | 增加CLob写入的方法。 | 增加 | [HY-ZhengWei](https://github.com/HY-ZhengWei) | 2018-07-25 |
| V1.3 | 增加占位符按IF条件取值。 | 增加 | [HY-ZhengWei](https://github.com/HY-ZhengWei) | 2018-08-10 |
| V1.4 | 升级占位符按IF条件取值。支持if … else if … else if … else 复杂的判定规则 | 增加 | [HY-ZhengWei](https://github.com/HY-ZhengWei) | 2019-01-20 |
| V1.5 | 调用存储过程的具体说明 | 增加 | [HY-ZhengWei](https://github.com/HY-ZhengWei)、[LHao](https://github.com/343136848) | 2020-04-14 |
| V1.6 | 树结构查询 | 增加 | [HY-ZhengWei](https://github.com/HY-ZhengWei) | 2020-04-21 |
| V1.7 | 自增长ID的返回 | 增加 | [HY-ZhengWei](https://github.com/HY-ZhengWei) | 2020-05-27 |

对XSQL提出过优化、帮助及建议的朋友（按首次提出的时间排序）

[Dirful](https://github.com/dirful)、

[LHao](https://github.com/343136848)、

[YangPiena](https://github.com/yangpiena)、

[GH-XiangYiQian](https://github.com/GH-XiangYiQian)、

[Taohuaxinmu521](https://github.com/taohuaxinmu521)、

[MaLong301](https://github.com/malong301)、

[ZYsoftkitty](https://github.com/ZYsoftkitty)

# XSQL概述

XSQL是与MyBatis类似的数据库访问框架，比它强大，XSQL的组概念(XSQLGroup)是其它框架没有的，可能也是XSQL首次提出的，XSQL组拥有ETL及存储过程两种功能的优点。再加之XSQL云XSQLCloud功能，将支持并行计算、云计算等能力。

XSQL自身可独立运行及使用，也可运行在XJava平台上，作为XJava平台上一个应用使用。本文讲的均是基于XJava平台的开发方法。

# XSQL创建数据库对象

在项目（Web服务或桌面应用）启动时，自动判定数据库对象是否存在，当对象不存在时执行创建SQL创建对象。

方便好用的功能，只须给定一个空数据库，即能在项目启动时，构造完整个数据库。

数据库对象包括：表、视图、过程、函数、序列（Oracle）、索引、触发器、约束等数据概念的对象。

支持数据库类型如下：

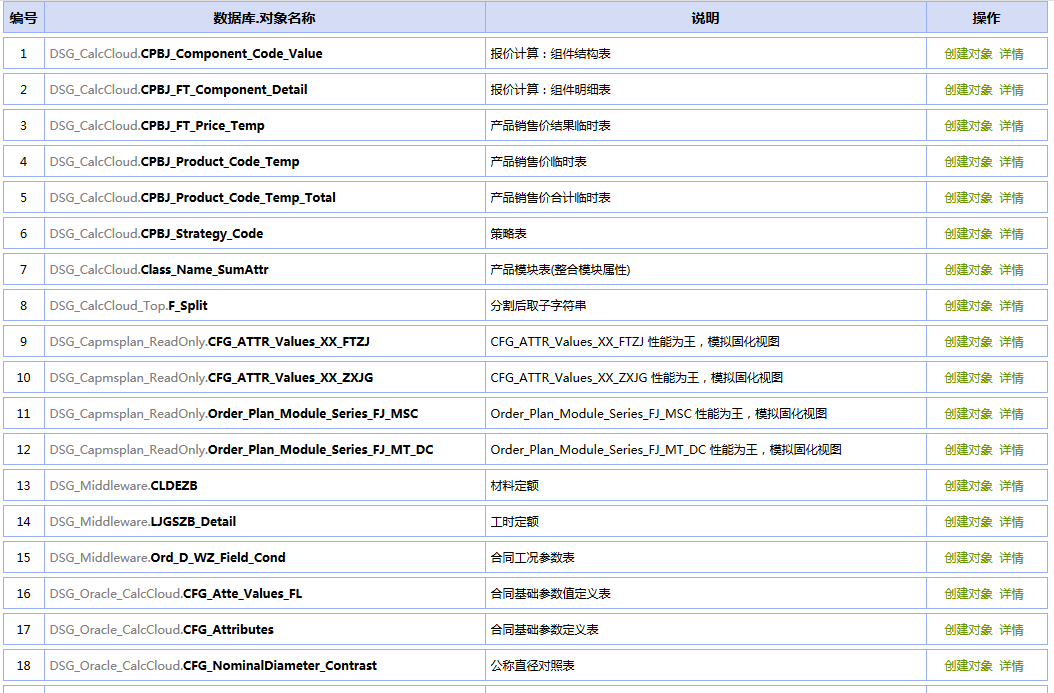
1. Oracle
2. MySQL
3. SQLServer
4. DB2
5. SQLite
6. PostgreSQL
7. 其它数据库可通过继承或扩展XSQLDBMetadata.xml配置文件来实现。

用此方法创建的数据库对象，均可通过 管理页面（<http://127.0.0.1:80/WebName/analyses>）反复重新创建（会先删除后重新创建），或查看创建数据库对象的列表，如下图。

管理页面需在web.xml中添加配置后，才能正常使用。配置 方法见org.hy.common.xml.plugins.analyse.AnalyseObjectServlet类中的说明。



创建对象列表，如下图：



**XML配置语法：**

1. **<sql>**
3. **<dataSourceGroup** ref="数据库连接池组的对象引用" **/>**
5. **<content>**
6. <![CDATA[
7. 数据库SQL。多个执行SQL间用;/分隔。
8. ]]>
9. **</content>**
11. **<comment>**注释说明。会在数据库对象创建时输出**</comment>**  <!-- 可选配置 -->
12. **<create>**数据库对象名称(不区分大小写)**</create>**   <!-- 此行请放在最后 -->
14. **</sql>**

**XML配置说明：**

1. <content>节点定义SQL语句，多个执行SQL语句间用分号加除号（**;/**）分隔。这样就能更好的支持数据库存储过程及函数的构造。
2. <create>节点一定要放在最后，当解析器遇到<create>节点时，就立刻执行<content>节点中定义的SQL语句，及显示<comment>节点的注释说明。
3. <create>节点中定义的数据库对象名称，会在<content>节点中定义的SQL中查找匹配，当未查找到时，会错异常，表示创建对象名称与创建SQL不匹配。
4. 请尽量在<content>节点中编写可重复数次执行的SQL语句。如，创建数据库对象A时，要向B表中插入一条配置数据，B表中也只能有一条对应数据库对象A的配置数据。此时，应当在INSERT语句前，再添加一条DELETE语句，预防性的先删除B表的配置数据后，再插入配置数据，防止重复数据。

**XML配置举例：**

1. **<sql** id="XSQL\_Create\_TActivityType"**>**
3. **<dataSourceGroup** ref="DSG\_XFlow" **/>**
5. **<content>**
6. <![CDATA[
7. CREATE TABLE TActivityType
8. (
9. activityTypeID     VARCHAR2(64)     NOT NULL
10. ,activityType       NVARCHAR2(128)   NOT NULL
11. ,infoComment        NVARCHAR2(2000)
12. ,orderNo            NUMBER(10)
13. );/
15. ALTER TABLE TActivityType ADD CONSTRAINT PK\_TActivityType PRIMARY KEY (activityTypeID);/
17. COMMENT ON TABLE  TActivityType                IS '工作流活动(节点)类型';/
18. COMMENT ON COLUMN TActivityType.activityTypeID IS '工作流活动类型ID';/
19. COMMENT ON COLUMN TActivityType.activityType   IS '工作流活动类型名称';/
20. COMMENT ON COLUMN TActivityType.infoComment    IS '备注说明';/
21. COMMENT ON COLUMN TActivityType.orderNo        IS '排列顺序';/
22. ]]>
23. **</content>**
25. **<comment>**工作流活动(节点)类型**</comment>**
26. **<create>**TActivityType**</create>**
28. **</sql>**

# XSQL创建多种数据库的对象

上面已经讲了如何创建数据库对象，但如何支持不同类型的数据库对象的创建呢？比如说，项目需要支持SQLServer、Oracle两种数据库的表创建怎么办呢？放心XSQL是支持的。

**XML配置语法：**

1. **<sql>**
3. **<dataSourceGroup** ref="DSG\_My" **/>** <!-- 数据库连接池组的对象引用 -->
5. **<content** if="ORACLE == this.getDataSourceGroup.getDbProductType"**>**
6. <![CDATA[
7. Oracle数据库的SQL语句。多个执行SQL间用;/分隔。
8. ]]>
9. **</content>**
11. **<content** if="SQLSERVER == this.getDataSourceGroup.getDbProductType"**>**
12. <![CDATA[
13. SQLServer数据库的SQL语句。多个执行SQL间用;/分隔。
14. ]]>
15. **</content>**
17. **<comment>**注释说明。会在数据库对象创建时输出**</comment>**  <!-- 可选配置 -->
18. **<create>**数据库对象名称(不区分大小写)**</create>**   <!-- 此行请放在最后 -->
20. **</sql>**

**XML配置说明：**

1. if关键字等式右侧的Fel表达式为真时，解析器才解析if所在的xml节点。
2. 通过<dataSourceGroup>节点引用或定义的数据库连接池组对象，来判定数据库类型。解析器再根据两个<content>节点定义的解析条件，判定条件真假，加载 不同运行时情况下对应的执行SQL语句。
3. getDbProductType()支持的数据库类型及类型名称，可在org.hy.common.db.DataSourceGroup类中找到定义。

**XML配置举例：**

1. **<sql** id="XSQL\_Create\_TActivityType"**>**
3. **<dataSourceGroup** ref="DSG\_XFlow" **/>**
5. **<content** if="ORACLE == this.getDataSourceGroup.getDbProductType"**>**
6. <![CDATA[
7. CREATE TABLE TActivityType
8. (
9. activityTypeID     VARCHAR2(64)     NOT NULL
10. ,activityType       NVARCHAR2(128)   NOT NULL
11. ,infoComment        NVARCHAR2(2000)
12. ,orderNo            NUMBER(10)
13. );/
15. ALTER TABLE TActivityType ADD CONSTRAINT PK\_TActivityType PRIMARY KEY (activityTypeID);/
17. COMMENT ON TABLE  TActivityType                 IS '工作流活动(节点)类型';/
18. COMMENT ON COLUMN TActivityType.activityTypeID  IS '工作流活动类型ID';/
19. COMMENT ON COLUMN TActivityType.activityType    IS '工作流活动类型名称';/
20. COMMENT ON COLUMN TActivityType.infoComment     IS '备注说明';/
21. COMMENT ON COLUMN TActivityType.orderNo         IS '排列顺序';/
22. ]]>
23. **</content>**
25. **<content** if="SQLSERVER == this.getDataSourceGroup.getDbProductType"**>**
26. <![CDATA[
27. CREATE TABLE TActivityType
28. (
29. activityTypeID     VARCHAR(64)     NOT NULL
30. ,activityType       NVARCHAR(128)   NOT NULL
31. ,infoComment        NVARCHAR(2000)
32. ,orderNo            INT
33. );/
35. ALTER TABLE TActivityType ADD CONSTRAINT PK\_TActivityType PRIMARY KEY (activityTypeID);/
37. EXEC P\_TableComment  TActivityType                 ,'工作流活动(节点)类型';/
38. EXEC P\_ColumnComment TActivityType ,activityTypeID ,'工作流活动类型ID';/
39. EXEC P\_ColumnComment TActivityType ,activityType   ,'工作流活动类型名称';/
40. EXEC P\_ColumnComment TActivityType ,infoComment    ,'备注说明';/
41. EXEC P\_ColumnComment TActivityType ,orderNo        ,'排列顺序';/
42. ]]>
43. **</content>**
45. **<comment>**工作流活动(节点)类型**</comment>**
46. **<create>**TActivityType**</create>**
48. **</sql>**

注：P\_TableComment和P\_ColumnComment是设置表或表字段注释的两个存储过程，因SQLServer数据库添加注释的命令太过于复杂而特意编写的。这两个存储过程也是用XSQL配置构建的。

# XSQL创建多种数据库的对象并初始化数据

上面已经讲了如何创建多种数据库的对象。又如何方便的初始化表中的数据呢？不同数据库的INSERT语句格式是一样，上面针对数据库类型分开编写对应SQL语句，难道INSERT语句也是重复的写两边吗？答案，当然是不用的。

这时就要用到XSQL的 “[应用级触发器](#_XSQL应用级触发器)”功能了。具体功能不在此赘述，请详见相关章节。

**XML配置语法：**

1. **<sql>**
3. **<dataSourceGroup** ref="DSG\_My" **/>** <!-- 数据库连接池组的对象引用 -->
5. **<content** if="ORACLE == this.getDataSourceGroup.getDbProductType"**>**
6. <![CDATA[
7. Oracle数据库的SQL语句。多个执行SQL间用;/分隔。
8. ]]>
9. **</content>**
11. **<content** if="SQLSERVER == this.getDataSourceGroup.getDbProductType"**>**
12. <![CDATA[
13. SQLServer数据库的SQL语句。多个执行SQL间用;/分隔。
14. ]]>
15. **</content>**
17. <trigger>
18. <create ref="另一个XSQL对象的引用"/> <!-- 可在此XSQL对象中执行相同的SQL语句 -->
19. <create ref="支持多个应用级触发器"/>
20. </trigger>
22. **<comment>**注释说明。会在数据库对象创建时输出**</comment>**  <!-- 可选配置 -->
23. **<create>**数据库对象名称(不区分大小写)**</create>**   <!-- 此行请放在最后 -->
25. **</sql>**

**XML配置说明：**

1. <trigger>节点将在本节点XSQL执行后执行，类似于数据库的After触发器。
2. <trigger>节点中可以添加一个或多个<create>节点。支持多个应用级触发器。
3. <trigger>节点中的<create>节点的数据库可以与本XSQL节点的数据库不一样，即支持跨库操作。

**XML配置举例：**

1. **<sql** id="XSQL\_Create\_TActivityType\_Init\_Insert"**>**
3. **<dataSourceGroup** ref="DSG\_XFlow" **/>**
5. **<content>**
6. <![CDATA[
7. INSERT  INTO TActivityType VALUES('AT001' ,'开始' ,1);/
8. INSERT  INTO TActivityType VALUES('AT999' ,'结束' ,999);/
9. ]]>
10. **</content>**
12. **<comment>**工作流活动(节点)类型表的初始化数据**</comment>**
14. **</sql>**


18. **<sql** id="XSQL\_Create\_TActivityType"**>**
20. **<dataSourceGroup** ref="DSG\_XFlow" **/>**
22. **<content** if="ORACLE == this.getDataSourceGroup.getDbProductType"**>**
23. <![CDATA[
24. CREATE TABLE TActivityType
25. (
26. activityTypeID     VARCHAR2(64)     NOT NULL
27. ,activityType       NVARCHAR2(128)   NOT NULL
28. ,infoComment        NVARCHAR2(2000)
29. ,orderNo            NUMBER(10)
30. );/
32. ALTER TABLE TActivityType ADD CONSTRAINT PK\_TActivityType PRIMARY KEY (activityTypeID);/
34. COMMENT ON TABLE  TActivityType                 IS '工作流活动(节点)类型';/
35. COMMENT ON COLUMN TActivityType.activityTypeID  IS '工作流活动类型ID';/
36. COMMENT ON COLUMN TActivityType.activityType    IS '工作流活动类型名称';/
37. COMMENT ON COLUMN TActivityType.infoComment     IS '备注说明';/
38. COMMENT ON COLUMN TActivityType.orderNo         IS '排列顺序';/
39. ]]>
40. **</content>**
42. **<content** if="SQLSERVER == this.getDataSourceGroup.getDbProductType"**>**
43. <![CDATA[
44. CREATE TABLE TActivityType
45. (
46. activityTypeID     VARCHAR(64)     NOT NULL
47. ,activityType       NVARCHAR(128)   NOT NULL
48. ,infoComment        NVARCHAR(2000)
49. ,orderNo            INT
50. );/
52. ALTER TABLE TActivityType ADD CONSTRAINT PK\_TActivityType PRIMARY KEY (activityTypeID);/
54. EXEC P\_TableComment  TActivityType                 ,'工作流活动(节点)类型';/
55. EXEC P\_ColumnComment TActivityType ,activityTypeID ,'工作流活动类型ID';/
56. EXEC P\_ColumnComment TActivityType ,activityType   ,'工作流活动类型名称';/
57. EXEC P\_ColumnComment TActivityType ,infoComment    ,'备注说明';/
58. EXEC P\_ColumnComment TActivityType ,orderNo        ,'排列顺序';/
59. ]]>
60. **</content>**
62. **<trigger>**
63. **<create** ref="XSQL\_Create\_TActivityType\_Init\_Insert"**/>**
64. **</trigger>**
66. **<comment>**工作流活动(节点)类型**</comment>**
67. **<create>**TActivityType**</create>**
69. **</sql>**

注：应用级触发器执行的XSQL对象XSQL\_Create\_TActivityType\_Init\_Insert应在触发源的XSQL对象XSQL\_Create\_TActivityType之前被解析构造出来。

# XSQL应用级触发器

数据表触发器大家都知道。应用级触发器是什么呢？它并非在数据库创建的触发器，而是在应用层面创建的触发器。数据库触发器创建后，将对所有应用程序均生效，但应用级触发器只对某个具体应用生效，专属于应用程序的。

XSQL对象A被调用时，自动触发执行另一个XSQL对象B执行的机制。它类似于数据库的After触发器，特性如下：

1. 对Insert、Update、Delete语句有效外，还对 SELECT语句、存储过程、函数及其它DDL、DML、DCL、TCL均生效，均可触发。
2. 一个XSQL可以触发多个触发器，并且可以递触发（即触发器的触发器）。
3. XSQL触发源的执行入参，会传递给所有XSQL触发器，并作为其执行入参。
4. 因为每个XSQL触发器均一个XSQL对象，每个XSQL对象可以有自己的数据库，所以触发源与触发器间、触发器与触发器间均可实现跨数据库的触发器功能。
5. XSQL触发器执行的时长，是不统计在触发源XSQL的执行时长中的。
6. XSQL触发器分为“同步模式”和“异步模式”。
   1. 在同步模式的情况下，所有XSQL触发器依次顺序执行，前一个执行完成，后下一个才执行。前一个执行异常后，其后的均不再执行。
   2. 在异步模式的情况下，每个XSQL触发器均是一个独立的线程，所有XSQL触发器几乎是同时执行的。
7. XSQL触发器执行异常后，是不会回滚先前主XSQL的操作的（即每个触发器每个操作都是一个独立的事务）。
8. XSQL触发源执行异常时，可以通过XSQLTrigger.errorCode属性控制XSQL触发器是否执行。默认XSQLTrigger.errorCode为True，即触发源异常时，XSQL触发器也被触发执行。

**XML配置语法：**

1. **<sql>**
3. **<dataSourceGroup** ref="数据库连接池组的对象引用" **/>**
5. **<content>**
6. <![CDATA[
7. SQL语句
8. ]]>
9. **</content>**
11. <trigger>
12. <create ref="另一个XSQL对象的引用"/> <!-- 可在此XSQL对象中执行相同的SQL语句 -->
13. <create ref="支持多个应用级触发器" />
14. <createBackup ref="另一个数据库连接池组的对象引用" />
15. </trigger>
17. **</sql>**

**XML配置说明：**

1. <trigger>节点将在本节点XSQL执行后执行，类似于数据库的After触发器。
2. <trigger>节点中可以添加一个或多个<create>节点。支持多个应用级触发器。
3. <trigger>节点中的<create>节点的数据库可以与本XSQL节点的数据库不一样，即支持跨库操作。
4. <createBackup>节点是另一个数据库连接池组的对象引用。一定是与触发源XSQL<dataSourceGroup>节点不一样的对象引用，即两个数据库。其主要用途是：备份或镜像数据库。当用户对A数据库操作时，将同样的操作在B数据库也执行一次。

**XML配置举例：**

1. **<sql>**
3. **<dataSourceGroup** ref="DSG\_XXDB" **/>**
5. **<trigger>**
6. **<createBackup** ref="DSG\_LogWeb"**/>**
7. **</trigger>**
9. **<content>**
10. <![CDATA[
11. INSERT  INTO ProductAttribute
12. (
13. modelID
14. ,productID
15. ,hasUseCond
16. ,nameCode
17. ,name
18. ,required
19. ,seqNo
20. ,sign
21. ,valueCode
22. ,value
23. )
24. VALUES (
25. :UUID
26. ,:SaveProductID
27. ,:hasUseCond
28. ,:nameCode
29. ,:name
30. ,:required
31. ,:seqNo
32. ,:sign
33. ,:valueCode
34. ,:value
35. )
36. ]]>
37. **</content>**
39. **</sql>**

注：

* 1. 上面的XSQL在执行后，将同时向两个数据库插入一样的数据。
  2. 参数:UUID是[自定义的函数型参数](#_XSQL函数型参数的插入更新)。
  3. 配置举例2见：[XSQL创建多种数据库的对象并初始化数据](#_XSQL创建多种数据库的对象并初始化数据)。

# XSQL数据库SQL自动生成

可以通过Java类自动生成对应的SQL语句。

**特性1：** 规范SQL格式。不尽是简单的生成了SQL语句，而且还规范了SQL语句的格式。

**特性2：**字段排序。排序规则与编写的Java类文件中的属性顺序相同。

**特性3：**按字段类型差异化处理占位符格式。

1. // 生成SELECT语句
2. Help.toSQLSelect(Java类.**class** ,首字母是否大写);
3. Help.toSQLSelect(Java类.**class** ,"表的别名" ,首字母是否大写);
4. Help.toSQLSelect(Java类.**class** ,"表的别名" ,"Java类名称" ,首字母是否大写);
6. // 生成INSERT语句
7. Help.toSQLInsert(Java类.**class**);
8. Help.toSQLInsert(Java类.**class** ,首字母是否大写);
10. // 生成UPDATE语句
11. Help.toSQLUpdate(Java类.**class**);
12. Help.toSQLUpdate(Java类.**class** ,首字母是否大写);

**Java代码说明：**

1. Help.toSQLSelect(Java类.**class** ,"表的别名" ,"Java类名称" ,首字母是否大写);主要用于[一对一复合型查询](#_XSQL一对一复合型查询) 或 [一对多复合型合查询](#_XSQL一对多复合型查询) 的SQL语句生成。

**自动生成SQL的举例：**

1. **SELECT**  A.participants
2. ,A.routes
3. ,A.activityID
4. ,A.templateID
5. ,A.activityName
6. ,A.activityTypeID
7. ,A.activityType
8. ,A.x
9. ,A.y
10. ,A.z
11. ,A.height
12. ,A.width
13. ,A.iconURL
14. ,A.infoComment
15. ,A.createrID
16. ,A.creater
17. ,A.createTime
18. ,A.lastUserID
19. ,A.lastUser
20. ,A.lastTime
21. **FROM**  ActivityInfo  A
22. **INSERT**  **INTO** ActivityInfo
23. (
24. participants
25. ,routes
26. ,activityID
27. ,templateID
28. ,activityName
29. ,activityTypeID
30. ,activityType
31. ,x
32. ,y
33. ,z
34. ,height
35. ,width
36. ,iconURL
37. ,infoComment
38. ,createrID
39. ,creater
40. ,createTime
41. ,lastUserID
42. ,lastUser
43. ,lastTime
44. )
45. **VALUES** (
46. :participants
47. , :routes
48. ,':activityID'
49. ,':templateID'
50. ,':activityName'
51. ,':activityTypeID'
52. , :activityType
53. , :x
54. , :y
55. , :z
56. , :height
57. , :width
58. ,':iconURL'
59. ,':infoComment'
60. ,':createrID'
61. ,':creater'
62. ,':createTime'
63. ,':lastUserID'
64. ,':lastUser'
65. ,':lastTime'
66. )

UPDATE语句请按需删除或保留<[ 和 ]> 两个限定符。

1. **UPDATE**  ActivityInfo
2. **SET**
3. <[    ,participants   =  :participants      ]>
4. <[    ,routes         =  :routes            ]>
5. <[    ,activityID     = ':activityID'       ]>
6. <[    ,templateID     = ':templateID'       ]>
7. <[    ,activityName   = ':activityName'     ]>
8. <[    ,activityTypeID = ':activityTypeID'   ]>
9. <[    ,activityType   =  :activityType      ]>
10. <[    ,x              =  :x                 ]>
11. <[    ,y              =  :y                 ]>
12. <[    ,z              =  :z                 ]>
13. <[    ,height         =  :height            ]>
14. <[    ,width          =  :width             ]>
15. <[    ,iconURL        = ':iconURL'          ]>
16. <[    ,infoComment    = ':infoComment'      ]>
17. <[    ,createrID      = ':createrID'        ]>
18. <[    ,creater        = ':creater'          ]>
19. <[    ,createTime     = ':createTime'       ]>
20. <[    ,lastUserID     = ':lastUserID'       ]>
21. <[    ,lastUser       = ':lastUser'         ]>
22. <[    ,lastTime       = ':lastTime'         ]>
23. **WHERE**  1 = 2

# XSQL常规List查询

最常规的列表查询。

**XML配置语法：**

1. **<sqls>**
3. **<sql>**
5. **<dataSourceGroup** ref="数据库连接池组的对象引用" **/>**
7. **<content>**
8. <![CDATA[
9. SELECT \* FROM 表名
10. ]]>
11. **</content>**
13. **<result>**
14. **<row>**行数据对应的Java对象**</row>**
15. **<cfill>**setter(colValue)**</cfill>**  <!-- 将字段填充到Java对象的方法 -->
16. **</result>**
18. **</sql>**
20. **</sqls>**

**XML配置说明：**

1. <dataSourceGroup>节点定义操作哪个数据库。是数据库连接池组的对象。
2. 映射关系：查询字段名称与<row>节点定义的Java对象的setter()方法匹配时，即能填充Java对象实例。查询字段名称是不区分大小写的，但为书写美观，建议使用驼峰命名法。当查询字段名称匹配不到setter()方法时，不报错，只是不填充Java对象实例，其它能匹配到还将正常填充。
3. <content>节点定义执行SQL语句。用<![CDATA[ ... ]]>它括住SQL语句是为了防止SQL语句中出现特殊字符时，XML解析器无法识别。
4. <result>节点定义查询结果集如何映射为Java对象。所以，此节点只出现在查询的XSQL配置中。
5. <result><row>节点定义数据库表的每行数据映射为Java的哪个对象。其值为Java类的全路径。
6. <result><cfill>节点定义数据库表字段以哪种方法映射为Java对象的属性值。其关键字有以下几种：
   1. row：表示行级对象
   2. row.xxx：中间有个点。xxx表示Java对象的成员属性名称，需定义成员属性的getter()方法才能使用。row.xxx即表示支持面向对象。xxx也应当是数据库查询SQL语句的查询字段之一。
   3. rowNo：表示行号
   4. colNo：表示列号
   5. colName：表示列名称
   6. colValue：表示列上的数值
   7. setter：表示用Java对象的成员属性对应的Setter()方法填充Java对象实例。**setter(colValue)**是固定写法之一。
   8. 自定义字段级填充方法。
      1. 如，<row>节点的行级对象被定义为java.util.HashMap时，<cfill>节点配置为：<cfill>**put(colName ,colValue)**</cfill>。即使用HashMap对象的put方法填充实例。
      2. 如，<row>节点的行级对象被定义为java.util.ArrayList时，<cfile>节点配置为：<cfill>**add(colValue)**</cfill>。即使用ArrayList对象的add方法填充实例。
      3. <cfill>定义的方法名是动态可变的、可自定义的（setter、put、add），可随外界变化而变化，千变万化XSQL都能映射。
7. <result><table>节点定义查询结果集（表级对象）被映射为Java的哪个对象。其值为Java类的全路径。默认值为java.util.ArrayList，所以上面语法中可以不配置。
8. <result><fill>节点定义行级对象填充到表级对象的填充方法名。与<cfill>节点类似，不再赘述了。默认值为**add(row)**，与<result><table>相互配对。

**XML配置举例：**

1. **<sqls>**
3. **<sql** id="XSQL\_XFlow\_ActivityType\_Query"**>**
5. **<dataSourceGroup** ref="DSG\_XFlow" **/>**
7. **<content>**
8. <![CDATA[
9. SELECT  A.activityTypeID
10. ,A.activityType
11. ,A.infoComment
12. ,A.orderNo
13. FROM  TActivityType  A
14. ORDER  BY A.orderNo
15. ]]>
16. **</content>**
18. **<result>**
19. **<row>**org.hy.xflow.engine.bean.ActivityType**</row>**
20. **<cfill>**setter(colValue)**</cfill>**
21. **</result>**
23. **</sql>**
25. **</sqls>**

建议：为了查询性能，查询语句应整齐的逐一罗列查询字段，并非简单的用 \* 表示。

# XSQL带参数的查询

XSQL常规List查询中的举例是未带查询参数的，而实际开发中常常用到带查询参数的SELECT、DELETE、UPDATE语句，XSQL使用“占位符”实现查询参数的关联绑定。

占位符：以英文冒号开头后跟自定义的变量名称。有两种以下形式：

1. :变量名称。
2. :变量名称.属性名称。

占位符也支持面向对象的编程理念，即，“类.属性”（中间有个点）的方式来层层映射。

占位符也支持面向函数的编程理念，见[XSQL函数型参数](#_XSQL函数型参数的插入更新)。

变量名称，可以自定义，只须与XSQL执行方法query、execute、executeUpdate、executeUpdates、getSQLCount等的入参相匹配映射上即可。

占位符与XSQL执行方法入参的映射关系：

1. 入参类型为java.util.Map类型时，Map.key的值即是占位符中的变量名称，变量名称以弱化的方式区分大小写，即先区分大小写配对，未能匹配成功时再按不区分大小写的方式配对。Map.value为替换占位符后的实现执行SQL语句的片段内容。当Map.value为值对象时，也可使用面向对象的编程理念。
2. 当入参类型为“setter、getter值对象”时，变量名称为值对象的getter方法短名称，即成员属性名。变量名称不区分大小写。
3. 入参值为NULL时，占位符将替换填充为NULL字样的文字。

占位符可应用于的SQL语句及位置：

1. 可应用于任何SQL语句，DDL、DQL、DML、DCL、TCL等。
2. 可应用于SQL语句的任一位置。如，表名称后，表示动态表；Where条件后，表示动态条件。只要确保最终生成的是正确语法的SQL语句即可。举例如下：
   1. [表名可变的SQL](#_XSQL表名可变的SQL)
   2. [表字段名可变的SQL](#_XSQL表字段名可变的SQL)
   3. [排序组合Order By可变的SQL](#_XSQL排序组合Order_By可变的SQL)
   4. [分页页码可变的SQL](#_XSQL分页页码可变的SQL)
   5. [表分区内查询的可变SQL](#_XSQL表分区内查询的可变SQL)（Oracle特性）

**XML配置语法：**

1. **<sqls>**
3. **<sql>**
5. **<dataSourceGroup** ref="数据库连接池组的对象引用" **/>**
7. **<content>**
8. <![CDATA[
9. SELECT  \*
10. FROM  表名
11. WHERE  字符类型的字段Name = ':占位符name'
12. AND  数值类型的字符Age  =  :占位符age
13. ]]>
14. **</content>**
16. **<result>**
17. **<row>**行数据对应的Java对象**</row>**
18. **<cfill>**setter(colValue)**</cfill>**  <!-- 将字段填充到Java对象的方法 -->
19. **</result>**
21. **</sql>**
23. **</sqls>**

**XML配置说明：**

1. 字符类型的表字段应按SQL语法在两边各加一个单引号的限定符。但如果是预解析执行方法的XSQL请不要加单引号限定符。举例见[XSQL预处理的批量执行](#_XSQL预处理的批量执行)。
2. 数值类型的表字段，直接编写占位符即可。部分数据库加上单引号限定符也能执行成功。
3. 其它节点说明详见“[XSQL常规List查询](#_XSQL常规List查询)”，此不再赘述。

**XML配置举例：**

1. **<sql** id="XSQL\_LogInfo\_Register\_QueryByID"**>**
3. **<dataSourceGroup** ref="DSG\_LogWeb" **/>**
5. **<content>**
6. <![CDATA[
7. SELECT  A.sysID
8. ,A.registerType
9. ,A.dbBuildTime
10. ,A.createTime
11. ,A.updateTime
12. FROM  TLogRegister  A
13. WHERE  A.sysID = ':sysID'
14. ]]>
15. **</content>**
17. **<result>**
18. **<row>**org.hy.log.model.LogRegister**</row>**
19. **<cfill>**setter(colValue)**</cfill>**
20. **</result>**
22. **</sql>**

# XSQL函数型占位符（主键ID的生成）

“[XSQL带参数的查询](#_XSQL带参数的查询)”一节中讲到占位符可实现“类.属性”的面向对象编程。占位符映射的值是由调用方提供的，在调用XSQL期间是固定不变的。如，占位符:name映射值“张三”，执行XSQL时是不会改变，同样的参数执行多次也不改变。

本小节讲的函数型参数，可实现将函数引用当参数传递，函数的返回值作最终与占位符映射的实现参数值。使占位符支持面向函数编程。

此功能在实现主键ID生成的场景上十分好用，特别是在[批量执行](#_XSQL批量执行)的XSQL上。做法如下：

1. 创建函数。实例化一个org.hy.common. MethodReflect类的对象，MethdoRelfect可动态反射任一类的方法。如，生成UUID的预设函数org.hy.common.MethodReflect.getMethodReflectUUID()。
2. 调用XSQL时传入上面创建的函数，并于占位符绑定，如下Java代码
3. XSQL                v\_XSQL   = XJava.getXSQL("XML文件中配置ID");
4. Map<String ,Object> v\_Params = **new** HashMap<String ,Object>();
6. v\_Params.put("UUID" ,MethodReflect.getMethodReflectUUID());
8. v\_XSQL.execute(v\_Params);

上面代码中的UUID是占位符的变量名称，可自行定义的。

**XML配置举例：**

1. **<sql>**
3. **<dataSourceGroup** ref="DSG\_XXDB" **/>**
5. **<content>**
6. <![CDATA[
7. INSERT  INTO ProductAttribute
8. (
9. modelID
10. ,productID
11. ,hasUseCond
12. ,nameCode
13. ,name
14. ,required
15. ,seqNo
16. ,sign
17. ,valueCode
18. ,value
19. )
20. VALUES (
21. :UUID
22. ,:SaveProductID
23. ,:hasUseCond
24. ,:nameCode
25. ,:name
26. ,:required
27. ,:seqNo
28. ,:sign
29. ,:valueCode
30. ,:value
31. )
32. ]]>
33. **</content>**
35. **</sql>**

# XSQL自增长ID的返回

当数据库（如MySQL、SQLServer）表绑定自增长ID字段时，此方法可获取Insert语句写入数据时DB自动生成的自增长ID的值。

有三种方式可以获取自增长ID

* 1. **方便简单的：**

XML配置中添加 <getID>true</ getID >，XSQL.executeUpdate(…)系列方法返回时的含义从原来的影响记录行数，变为影响首个记录的自增长ID的值。主要针对单行记录的写入（Insert）时快捷应用场景。

能做到兼容老版本的程序，代码零修改，仅通过配置xml实现自增长ID的获取。

* 1. **批量的：**

无须配置XML，使用XSQL.executeInsert(…)系列的方法，能同时返回所有自增长ID（集合）及影响记录行数。

* 1. **注解@Xsql的：**

上面两种情况均支持。并有三种写法，用于多种场景的使用。

**方便简单的XML配置举例：**

1. **<sql>**
3. **<dataSourceGroup** ref="DSG\_XXDB" **/>**
5. **<getID>**true**</getID>**
6. **<content>**
7. <![CDATA[
8. INSERT  INTO ProductAttribute
9. (
10. name
11. ,value
12. )
13. VALUES (
14. :name
15. ,:value
16. )
17. ]]>
18. **</content>**
20. **</sql>**

批量的Java代码举例：

1. public void test\_XSQLInsertGetID()
2. {
3. XSQLData v\_Ret = XJava.getXSQL("XSQL\_自增长ID的返回").executeInsert();
5. if ( v\_Ret.getIdentitys().size() >= 1 )
6. {
7. $Logger.info("MySQL自增ID = " + v\_Ret.getIdentitys().get(0));
8. }
9. else
10. {
11. $Logger.info("数据库未绑定or未获取到自增长ID");
12. }
13. }

注解@Xsql的代码举例：隐式定义1

通过接口方法返回类型 XSDLData来隐式定义要求获取自增长ID的集合。

1. @Xjava(id="XxxDAO" ,value=XType.XSQL)
2. public interface IXxxDAO
3. {
4. */\*\**
5. \* 批量添加多条数据
6. \*/
7. @Xsql("XSQL\_Insert\_Datas")
8. public XSQLData insertDatas(List<DataInfo> i\_Datas);
9. }

注解@Xsql的代码举例：隐式定义2

通过接口方法返回类型 List<Integer>来隐式定义要求获取自增长ID的集合。

1. @Xjava(id="XxxDAO" ,value=XType.XSQL)
2. public interface IXxxDAO
3. {
4. */\*\**
5. \* 批量添加多条数据
6. \*/
7. @Xsql("XSQL\_Insert\_Datas")
8. public List<Integer> insertDatas(List<DataInfo> i\_Datas);
9. }

注解@Xsql的代码举例：显式定义

[通过@Xsql.getID](mailto:通过@Xsql.getID)属性来显式定义要求获取自增长ID。

1. @Xjava(id="XxxDAO" ,value=XType.XSQL)
2. public interface IXxxDAO
3. {
4. */\*\**
5. \* 添加一条数据
6. \*/
7. @Xsql(id="XSQL\_Insert\_Data" ,getID=true)
8. public int insertData(DataInfo i\_Data);
9. }

优先级与冲突说明

* 1. 注解@Xsql优先级高于XML配置<getID />

即，当@Xsql要求返回自增长ID，XML配置<getID>false</getID>时，以@Xsql为准。

* 1. 注解@Xsql的隐式的优先级高于显式的

即，当@Xsql的隐式的生效时（返回类型为XSQLData或List<Integer>），@Xsql.getID=false时，以隐式的为准。

# XSQL占位符按IF条件取值（业务逻辑取值）

常规占位符映射的值是什么，生成的最终执行SQL的参数就是什么。当我们还要进一步实现某个或某些占位符带业务逻辑的取值时，可以给占位符添加取值条件。

占位符X按某个设定的条件取值，当条件满足(True)时取值A，否则取值B。

取值A、B可以如下四种取值类型：

* 1. 占位符X映射的值 （条件满足时的默认值）；
  2. 另一个占位符Y映射的值，须以英文冒号为前缀开头；
  3. 常量字符串；
  4. 空指针NULL值（条件不满足时的默认值）。

取值A、B中的占位符及占位符X均支持以下高级功能：

* 1. 支持xx.yy.zz面向对象；
  2. 支持函数型占位符。即占位符是一个方法的引用。占位符对应的数值，通过引用方法的返回值获取；
  3. 不区分大小写。

当用于动态标识 <[ ... ]>之内的占位符，可以控制动态标识内的SQL段是否拼接到最终执行SQL上执行。原则是：动态标识 <[ ... ]>之内的任意一个占位符为NULL时，动态标识内的SQL段将不参与最终运行SQL的执行。详见“[XSQL动态参数<[ ... ]>的查询](#_XSQL动态参数<[_..._]>的查询)”。

**XML配置语法：**

1. **<sql>**
3. **<dataSourceGroup** ref="数据库连接池组的对象引用" **/>**
5. **<content>**
6. <![CDATA[
7. SELECT  \*
8. FROM  表名称
9. WHERE  字段名称01 = ':点位符01'
10. AND  字段名称02 = ':点位符02'
11. AND  字段名称03 = ':点位符03'
12. ]]>
13. **</content>**
15. <!-- 条件01 -->
16. <!-- 实现Java编程语言中的 if ... else ... 的判定 -->
17. **<condition>**
18. **<name>**点位符01**</name>**
19. **<if>**Fel表达式**</if>**
20. **<true>**条件满足时的取值。默认为：THIS**</true>**
21. **<false>**条件不满足时的取值。默认为：NULL**</false>**
22. **</condition>**
24. <!-- 条件02：简写方式 -->
25. <!-- 实现Java编程语言中的 if ... else ... 的判定 -->
26. **<condition>**
27. **<name>**点位符02**</name>**   <!-- 必填项。注意不填写占位符前的冒号: -->
28. **<if>**Fel表达式**</if>**   <!— 可选项。当为空时(不配置此属性)，条件永远为真 -->
29. **</condition>**
30. <!-- 条件03 -->
31. <!-- 实现Java编程语言中的 if .. else if ... else ... 的多条件复杂判定 -->
32. **<condition>**
33. **<name>**点位符03**</name>**
34. **<if>**Fel表达式 01**</if>**
35. **<true>**Fel表达式 01满足时的取值。默认为：THIS**</true>**
36. **<if>**Fel表达式 02**</if>**
37. **<true>**Fel表达式 02满足时的取值。默认为：THIS**</true>**
38. **<if>**Fel表达式 03**</if>**
39. **<true>**Fel表达式 03满足时的取值。默认为：THIS**</true>**
40. **<false>**所有判定均不满足时的取值。默认为：NULL**</false>**
41. **</condition>**
42. <!-- 条件04：简写方式 -->
43. <!-- 实现Java编程语言中的 if .. else if ... else ... 的多条件复杂判定 -->
44. **<condition>**
45. **<name>**点位符04**</name>**
46. **<if>**Fel表达式 01**</if>**   <!-- 判定满足时取值THIS -->
47. **<if>**Fel表达式 02**</if>**
48. **<if>**Fel表达式 03**</if>**
49. **<false>**所有判定均不满足时的取值。默认为：NULL**</false>**
50. **</condition>**
52. **<result>**
53. **<row>**java.util.HashMap**</row>**
54. **<cfill>**put(colName ,colValue)**</cfill>**
55. **</result>**
57. **</sql>**

**XML配置说明：**

1. <condition>节点：条件节点，可以零到多个占位符条件。
2. <name>节点：占位符名称，对哪个占位符采取条件判定。
3. <if>节点：条件表达式，采用Fel表达式。可选的，当为空时，条件永远为真。可以有多个节点，可支持 if … else if … else if … else … 的Java编程语法。按顺序依次判定条件。任一<if>节点满足时，其后的<if>节点将不再判定。
4. <true>节点：条件满足时的取值。可选的，默认值是<name>节点指定占位符映射的值。可以有多个节点，可支持 if … else if … else if … else 的Java编程语法。
5. <false>节点：条件不满足时的取值。可选的，默认值是Java概念上的NULL。只出现在最后一个节点位置上，只需一个<false>节点就OK了。表示 if … else if … else … 中的else 情况，即所有<if>节点均不满足时的取值。

**XML配置举例：**

1. **<sql** id="XSQL\_User\_Condition"**>**
3. **<content>**
4. <![CDATA[
5. SELECT  A.userName
6. ,A.pwd
7. FROM  User  A
8. WHERE  A.userName = ':userName'
9. AND  A.pwd      = ':pwd'
10. <[ AND A.orgName = ':orgName' ]>
11. ]]>
12. **</content>**
14. <!-- 当用户名称为admin时，自动翻译为administrator用户名称登陆 -->
15. **<condition>**
16. **<name>**userName**</name>**
17. **<if>**:userName == 'admin'**</if>**
18. **<true>**administrator**</true>**
19. **<false>**:userName**</false>**   <!-- 也可以写成 this -->
20. **</condition>**
22. <!-- 密码验证 -->
23. **<condition>**
24. **<name>**password**</name>**
25. **<if>**<![CDATA[{:pwd}.length() >= 6 && {:pwd}.length() <= 16]]>**</if>**
26. **<false>**密码长度验证未通过**</false>**
27. **</condition>**
29. <!-- 部门名称是“根目录”时，不参与Where条件过滤 -->
30. <!-- 部门名称是“技术部”时，用’Hello World’字符串参与Where条件过滤 -->
31. <!-- 部门名称是“科技部”时，用’AI’字符串参与Where条件过滤 -->
32. <!-- 部门名称是其它情况时，用部门名称参与Where条件过滤 -->
33. **<condition>**
34. **<name>**orgName**</name>**
35. **<if>**<![CDATA[:orgName == '根目录']]>**</if>**
36. **<true>**NULL**</true>**
37. **<if>**<![CDATA[:orgName == '技术部']]>**</if>**
38. **<true>**Hello World**</true>**
39. **<if>**<![CDATA[:orgName == '科技部']]>**</if>**
40. **<true>**AI**</true>**
41. **<false>**THIS**</false>**
42. **</condition>**
44. **</sql>**

当入参参数是userName=’admin’、pwd=’123’、orgName=’根目录’时，生成的运行SQL如下：

1. **SELECT**  A.userName
2. ,A.pwd
3. **FROM**  User  A
4. **WHERE**  A.userName = 'administrator'
5. AND  A.pwd      = '密码长度验证未通过'

当入参数是userName=’root’、pwd=’12345678’ 、orgName=’根目录’时，生成的运行SQL如下：

1. **SELECT**  A.userName
2. ,A.pwd
3. **FROM**  User  A
4. **WHERE**  A.userName = 'root'
5. AND  A.pwd      = '12345678'

当入参数是userName=’root’、pwd=’12345678’ 、orgName=’我的部门’时，生成的运行SQL如下：

1. **SELECT**  A.userName
2. ,A.pwd
3. **FROM**  User  A
4. **WHERE**  A.userName = 'root'
5. AND  A.pwd      = '12345678'
6. AND  A.orgName  = '我的部门'

当入参数是userName=’root’、pwd=’12345678’ 、orgName=’技术部’时，生成的运行SQL如下：

1. **SELECT**  A.userName
2. ,A.pwd
3. **FROM**  User  A
4. **WHERE**  A.userName = 'root'
5. AND  A.pwd      = '12345678'
6. AND  A.orgName  = 'Hello World'

当入参数是userName=’root’、pwd=’12345678’ 、orgName=’科技部’时，生成的运行SQL如下：

1. **SELECT**  A.userName
2. ,A.pwd
3. **FROM**  User  A
4. **WHERE**  A.userName = 'root'
5. AND  A.pwd      = '12345678'
6. AND  A.orgName  = 'AI'

# XSQL动态参数<[ ... ]>的查询

XSQL在XML配置中编写的其实是：SQL模板。当带占位符参数时，并非最终可执行的SQL语句，需要填充替换占位符生成最终可执行的SQL语句。上面已经讲了通过带参数改变执行结果的方法，但无论参数值怎么变最终可执行SQL语句的主体结构不变，仅是参数值的变化。

如何动态SQL语句主体结构呢？

使用动态标识 <[ 和 ]> 。它们括起来的区域内所有占位符均有值（不为NULL）时，动态标记区域内的SQL才生效，才将参与到最终SQL的执行中。也因为此，参与动态标识区域内的占位符映射的Java类型就应当为对象，如，Integer、Double、Float，而不是int、double、float这样的基本类型，它们是没有NULL的情况的。

动态标识是与占位符配合一起使用的。当然，动态标记的区域内没有占位符，也不影响XSQL的执行，但。

动态标识可应用于的SQL语句及位置：

1. 可应用于任何SQL语句，DDL、DML、DCL、TCL等。
2. 可应用于SQL语句的任一位置。只要确保最终生成的是正确语法的SQL语句即可。

**XML配置语法：**

1. **<sqls>**
3. **<sql>**
5. **<dataSourceGroup** ref="数据库连接池组的对象引用" **/>**
7. **<content>**
8. <![CDATA[
9. SELECT  \*
10. FROM  表名
11. WHERE  字符类型的字段Name = ':占位符name'
12. <[   AND  数值类型的字符Age  =  :占位符age   ]>
13. ]]>
14. **</content>**
16. **<result>**
17. **<row>**行数据对应的Java对象**</row>**
18. **<cfill>**setter(colValue)**</cfill>**  <!-- 将字段填充到Java对象的方法 -->
19. **</result>**
21. **</sql>**
23. **</sqls>**

**XML配置说明：**

1. 占位符age：当值为18时（不为NULL），最终生成的执行SQL才有“AND 字段Age = 18 ”这样的查询条件。为NULL时没有此查询条件。
2. 其它节点说明详见“[XSQL常规List查询](#_XSQL常规List查询)”，此不再赘述。

**XML配置举例：**

1. **<sql** id="XSQL\_XFlow\_Template\_QueryByID\_NameVersion"**>**
3. **<dataSourceGroup** ref="DSG\_XFlow" **/>**
5. **<content>**
6. <![CDATA[
7. SELECT  A.templateID
8. ,A.version
9. ,A.templateName
10. ,A.infoComment
11. ,A.createrID
12. ,A.creater
13. ,A.createTime
14. ,A.lastUserID
15. ,A.lastUser
16. ,A.lastTime
17. ,A.isValid
18. ,A.isDelete
19. FROM  TTemplate  A
20. WHERE  1 = 1
21. <[   AND  A.templateID   = ':templateID'   ]>
22. <[   AND  A.templateName = ':templateName'
23. AND  A.version      = ':version'      ]>
24. ]]>
25. **</content>**
27. **<result>**
28. **<row>**org.hy.xflow.engine.bean.Template**</row>**
29. **<cfill>**setter(colValue)**</cfill>**
30. **</result>**
32. **</sql>**
33. **<sql** id="XSQL\_XFlow\_FlowInfo\_QueryByWorkID\_ServiceDataID"**>**
35. **<dataSourceGroup** ref="DSG\_XFlow" **/>**
37. **<content>**
38. <![CDATA[
39. SELECT  A.workID
40. ,A.workFatherID
41. ,A.flowTemplateID
42. ,A.serviceDataID
43. ,A.createrID
44. ,A.creater
45. ,A.createOrgID
46. ,A.createOrg
47. ,A.createTime
48. ,A.lastUserID
49. ,A.lastUser
50. ,A.lastTime
51. ,A.lastOrgID
52. ,A.lastOrg
53. ,A.lastProcessID
54. ,A.isValid
55. ,A.isDelete
56. FROM  TFlowInfo  A
57. <[   WHERE  A.serviceDataID = ':serviceDataID'   ]>
58. <[   WHERE  A.workID        = ':workID'          ]>
59. ]]>
60. **</content>**
62. **<result>**
63. **<row>**org.hy.xflow.engine.bean.FlowInfo**</row>**
64. **<cfill>**setter(colValue)**</cfill>**
65. **</result>**
67. **</sql>**

# XSQL表名可变的SQL

在不支持表分区的数据库中，我们如何实现按时期划分分类存储数据呢？创建前缀一样后面跟时间的一系列表，所以在操作这一系列表时，表的名称是随时间条件可变的SQL。

**XML配置举例：**

1. **<sql** id="XSQL\_LogInfo\_Register\_Add"**>**
3. **<dataSourceGroup** ref="DSG\_LogWeb" **/>**
5. **<content>**
6. <![CDATA[
7. INSERT  INTO TLog\_:sysID
8. (
9. id
10. ,logID
11. ,logType
12. ,logClass
13. ,logContent
14. ,logInfo
15. ,operatorNo
16. ,operationType
17. ,operationRemark
18. ,operationTime
19. ,waitTime
20. )
21. VALUES (
22. ':id'
23. ,':logID'
24. ,':logType'
25. ,':logClass'
26. ,':logContent'
27. ,':logInfo'
28. ,':operatorNo'
29. ,':operationType'
30. ,':operationRemark'
31. ,':operationTime'
32. ,':waitTime'
33. )
34. ]]>
35. **</content>**
37. **</sql>**

**举例说明：**

1. 详见语法说明见“[XSQL带参数的查询](#_XSQL固定参数的查询)”，此不再赘述。
2. 举例是按不同访问系统按系统编号写入不同分类的日志表中。

# XSQL表字段名可变的SQL

话不多说，直接上例子。详见语法说明见“[XSQL带参数的查询](#_XSQL固定参数的查询)”，此不再赘述。

**XML配置举例**

1. **<sql>**
3. **<value>**
4. <![CDATA[
5. UPDATE  TLogRegister
6. SET  :columnName = ':columnValue'
7. WHERE  ID          = ':id'
8. ]]>
9. **</value>**
11. **</sql>**

# XSQL排序组合Order By可变的SQL

话不多说，直接上例子。详见语法说明见“[XSQL带参数的查询](#_XSQL固定参数的查询)”，此不再赘述。

**XML配置举例**

1. **<sql>**
3. **<dataSourceGroup** ref="DSG\_LogWeb" **/>**
5. **<content>**
6. <![CDATA[
7. SELECT  A.sysid
8. ,A.msgKey
9. FROM  TMessageKey  A
10. WHERE  A.sysid IS NOT NULL
11. AND  A.sid   IS NULL
12. ORDER  BY :orderBys
13. ]]>
14. **</content>**
16. **<result>**
17. **<table>**java.util.HashMap**</table>**
18. **<fill>**put(row.sysid ,row.msgKey)**</fill>**
19. **<row>**org.hy.log.model.MessageKey**</row>**
20. **<cfill>**setter(colValue)**</cfill>**
21. **</result>**
23. **</sql>**

# XSQL表分区内查询的可变SQL

以Oracle数据库特性举例。话不多说，直接上例子。详见语法说明见“[XSQL带参数的查询](#_XSQL固定参数的查询)”，此不再赘述。

**XML配置举例**

1. **<sql>**
3. **<dataSourceGroup** ref="DSG\_LogWeb" **/>**
5. **<content>**
6. <![CDATA[
7. SELECT  A.logInfo
8. ,A.createTime
9. FROM  TLog PARTITION (:PartName)   A
10. WHERE  A.createTime <= TO\_DATE(':createTime' ,'yyyy-MM-dd HH24:mi:ss')
11. ]]>
12. **</content>**
14. **<result>**
15. **<row>**org.hy.log.model.LogInfo**</row>**
16. **<cfill>**setter(colValue)**</cfill>**
17. **</result>**
19. **</sql>**

# XSQL分页页码可变的SQL

XSQL已预设了常见数据库的分页功能，详见“[XSQL分页查询](#_XSQL分页查询)”。本节展示部分数据库用XSQL实现分页的XML配置。

**XML配置举例（MySQL数据库的）：**

1. **<sql** id="XSQLPaging\_MYSQL"**>**
3. **<value>**
4. <![CDATA[
5. :SQLPaging LIMIT :StartIndex ,:PagePerCount
6. ]]>
7. **</value>**
9. **</sql>**

**XML配置举例（Oracle数据库的）：**

1. **<sql** id="XSQLPaging\_ORACLE"**>**
3. **<value>**
4. <![CDATA[
5. SELECT  HYPaging\_W.\*
6. FROM (
7. SELECT  ROWNUM AS rowNumber ,HYPaging\_N.\*
8. FROM  ( :SQLPaging )  HYPaging\_N
9. WHERE  ROWNUM <= :StartIndex + :PagePerCount
10. )  HYPaging\_W
11. WHERE  HYPaging\_W.rowNumber > :StartIndex
12. ]]>
13. **</value>**
15. **</sql>**

**举例说明：**

1. 详见语法说明见“[XSQL带参数的查询](#_XSQL固定参数的查询)”，此不再赘述。

# XSQL统计行数Count(\*)

统计数据库表的数量，如SELECT COUNT(1) FROM 表名称。

**XML配置举例：**

1. **<sql** id="XSQL\_XXX\_XXX\_Query\_DataCount"**>**
3. **<dataSourceGroup** ref="DSG\_LogWeb" **/>**
5. **<content>**
6. <![CDATA[
7. SELECT  COUNT(1) AS DataCount
8. FROM  CPBJ\_Product\_Code   A
9. ,CPBJ\_FT\_Price\_Temp  B
10. WHERE  A.product\_code  = B.product\_code
11. AND  A.product\_class = B.product\_class
12. AND  A.product\_class = ':product\_class'
13. ]]>
14. **</content>**
16. **</sql>**

**Java代码举例：**

1. @Xjava(value=XType.XSQL)
2. **public** **interface** 接口名称
3. {
5. @Xsql("XSQL\_XXX\_XXX\_Query\_DataCount")
6. **public** **long** queryCount(Object i\_Params);
8. }

# XSQL分页查询

XSQL已预设了常见数据库的分页功能。

支持数据库类型如下：

1. Oracle
2. MySQL
3. SQLServer
4. DB2
5. SQLite
6. PostgreSQL
7. 其它数据库可通过继承或扩展XSQLPaging.xml配置文件来实现。

开发者只要编写正常查询语句的XSQL的XML配置即可。不用编写具体数据库的分页方法。

如，MySQL数据库的A表分页，无须编写Limit 关键字的部分，由XSQL合成：

1. **<sqls>**
3. **<sql>**
5. **<dataSourceGroup** ref="数据库连接池组的对象引用" **/>**
7. **<content>**
8. <![CDATA[
9. SELECT \* FROM A表
10. ORDER BY 排序字段
11. ]]>
12. **</content>**
14. **<result>**
15. **<row>**行数据对应的Java对象**</row>**
16. **<cfill>**setter(colValue)**</cfill>**  <!-- 将字段填充到Java对象的方法 -->
17. **</result>**
19. **</sql>**
21. **</sqls>**

Java调用代码如下：

1. XSQL         v\_XSQL  = XSQL.queryPaging("XML文件中配置ID" 或 XSQL对象);
2. List<Object> v\_Datas = v\_XSQL.query(查询参数的值对象或Map对象);
3. // v\_Datas即为分页的结果数据

代码说明：

* 1. XSQL.queryPaging()方法是按某个常规查询语句的XSQL对象生成一个分页查询语句的XSQL对象。
  2. 再用新生成的分页查询的XSQL对象执行查询动作。

对于SQLServer及DB2两个数据库，需在常规查询SQL的查询字段处添加rowNumber查询字段及排序方式，才能分页查询，这均两个数据库特性有关。

**SELECT** ROW\_NUMBER() OVER(**ORDER** **BY** id) **AS** rowNumber ,...... **FROM** 表名

# XSQL懒人查询

可能你只想临时编写一个XSQL用于测试，或者真的是懒的定义<row>节点的行级对象。当不想定义与数据库表对应映射的值对象时，是否也能通过XSQL查询呢？

答案当然是，可以的。通过<table>、<row>组合能生成不限于以下举例的Java数据结构。

1. List<List<Object>>：默认情况，即不在XML中配置<result>节点。
2. List<Map<String ,Object>>
3. List<Set<Object>>
4. Set<List<Object>>
5. Set<Map<String ,Object>>
6. Set<Set<Object>>
7. Map<String ,Map<String ,Object>>：需要rowNo作为外层Map的key值。
8. Map<String ,List<Object>>：需要rowNo作为Map的key值。
9. Map<String ,Set<Object>>：需要rowNo作为Map的key值。

<row>节点的默认值是：java.util.ArrayList，即使你不在XML中配置<row>节点也是可以返回查询结果的。这种查询结果的数据结构不太常用。常用的懒人查询是将<row>节点定义为：java.util.Hashtable或java.util.HashMap的数据结构，常用于轻量级ETL的XSQL组（XSQLGroup）中。

**XML配置语法：**

1. **<sqls>**
3. **<sql>**
5. **<dataSourceGroup** ref="数据库连接池组的对象引用" **/>**
7. **<content>**
8. <![CDATA[
9. SELECT \* FROM 表名
10. ]]>
11. **</content>**
13. **<result>**   <!--可选配置-->
14. **<row>**java.util.HashMap**</row>**   <!--行数据对应的Java对象-->
15. **<cfill>**put(colName ,colValue)**</cfill>**<!-- 将字段填充到Java对象的方法 -->
16. **<cstyle>**UPPER**</cstyle>** <!--字段名称的样式。可选配置 -->
17. **</result>**
19. **</sql>**
21. **</sqls>**

**XML配置说明：**

1. 语法中多数参数已在“[XSQL常规List查询](#_XSQL常规List查询)”阐述过，这里只做补充说明。
2. 每行数据用一个Map结构表示，Map.key为字段列名称，Map.value为字段数值。
3. <cstyle>节点用来控制Map.key中字段列名称的大小写样式默认为全部大写。可选参数如下（可参见org.hy.common.db.DBNameStyle）：
   1. UPPER：字段列名称全部大写。
   2. LOWER：字段列名称全部小写。
   3. NORMAL：字段名称按数据库默认样式显示。即SQL语句中是什么样式的就显示什么样式。如下SQL语句生成的Map.key字段列名称样式为：id ,Name ,nameCode ,AGE。
4. **SELECT** id
5. ,Name
6. ,nameCode
7. ,AGE
8. **FROM**  表名称
9. <cstyle>节点为可选配置。并且内容是不区分大小写的。写成Normal、NORMAL、normal等均可以。

**XML配置举例：**

1. **<sql** id="XSQL\_MailTime\_Step01\_Query"**>**
3. **<dataSourceGroup** ref="DSG\_LogWeb" **/>**
5. **<content** if="MYSQL == DSG\_LogWeb.getValue"**>**
6. <![CDATA[
7. SELECT  A.ID
8. ,A.Title
9. ,A.Content
10. ,A.Reciver
11. ,A.SendTime
12. FROM  TMailTime  A
13. WHERE  A.SendTime <= Now()
14. ]]>
15. **</content>**
17. **<content** if="SQLSERVER == DSG\_LogWeb.getValue"**>**
18. <![CDATA[
19. SELECT  A.ID
20. ,A.Title
21. ,A.Content
22. ,A.Reciver
23. ,A.SendTime
24. FROM  TMailTime  A
25. WHERE  A.SendTime <= GetDate()
26. ]]>
27. **</content>**
29. **<result>**
30. **<row>**java.util.HashMap**</row>**
31. **<cfill>**put(colName ,colValue)**</cfill>**
32. **<cstyle>**NORMAL**</cstyle>**
33. **</result>**
35. **</sql>**

# XSQL转义数据库关键字

带参数操作数据库的情况下，如果参数中出现的数据库关键字“单引号”时，可能会出现非法SQL，即语法不正确的SQL语句。

下面SQL，当占位符name的值为“张’三”（中间有一个单引号）时，在不配置XSQL.contentDB.keyReplace属性为false时，会执行异常。

keyReplace属性的默认值 为：true。

对于预处理机制执行SQL是不存在上述情况的。

**XML配置语法：**

1. **<sql>**
3. **<dataSourceGroup** ref="数据库连接池组的对象引用" **/>**
5. **<contentDB>**
6. **<keyReplace>**true**</keyReplace>**
7. **</contentDB>**
9. **<content>**
10. <![CDATA[
11. SQL语句
12. ]]>
13. **</content>**
15. **</sql>**

**XML配置举例**

1. **<sql** id="XSQL\_LogInfo\_Add"**>**
3. **<dataSourceGroup** ref="DSG\_LogWeb" **/>**
5. **<contentDB>**
6. **<keyReplace>**true**</keyReplace>**
7. **<safeCheck>**false**</safeCheck>**
8. **</contentDB>**
10. **<content>**
11. <![CDATA[
12. INSERT  INTO TLog
13. (
14. id
15. ,sysID
16. ,logID
17. ,logType
18. ,logClass
19. ,logContent
20. ,logInfo
21. ,operatorNo
22. ,operationType
23. ,operationRemark
24. ,operationTime
25. ,waitTime
26. )
27. VALUES (
28. ':id'
29. ,':sysID'
30. ,':logID'
31. ,':logType'
32. ,':logClass'
33. ,':logContent'
34. ,':logInfo'
35. ,':operatorNo'
36. ,':operationType'
37. ,':operationRemark'
38. ,':operationTime'
39. ,':waitTime'
40. )
41. ]]>
42. **</content>**
44. **</sql>**

# XSQL查询结果为Map结构的查询

查询结果为Map结果的查询。 相信你能理解是什么意思。常用于数据字典的查询。因为Map结构可以通过key快速定位value。

**XML配置语法：**

1. **<sqls>**
3. **<sql>**
5. **<dataSourceGroup** ref="数据库连接池组的对象引用" **/>**
7. **<content>**
8. <![CDATA[
9. SELECT \* FROM 表名
10. ]]>
11. **</content>**
13. **<result>**
14. **<table>**java.util.HashMap**</row>**       <!-- 表级的对象类型 -->
15. **<fill>**put(row.成员属性名 ,row)**</fill>** <!-- 行级对象填充到表级对象的填充方法 -->
16. **<row>**行数据对应的Java对象**</row>**
17. **<cfill>**setter(colValue)**</cfill>**  <!-- 用什么方法将每个字段填充到Java对象 -->
18. **</result>**
20. **</sql>**
22. **</sqls>**

**XML配置说明：**

1. 语法中多数参数已在“[XSQL常规List查询](#_XSQL常规List查询)”阐述过，这里只做补充说明。
2. <table>节点定义查询结果集（表级对象）被映射为Java的哪个对象。其值为Java类的全路径。定义它为java.util.HashMap的结构来存储数据库查询结果集。当然，换成java.util.Hashtable、java.util.LinkedHashMap都是可以的。
3. <fill><fill>节点定义行级对象填充到表级对象的填充方法名。定义它使用HashMap.put()方法将每一行数据存放在表级对象中，即存储在HashMap中。
4. row.成员属性名：成员属性名为Java对象的成员属性名称，需定义成员属性的getter()方法才能使用。Java对象是<row>节点定义的。成员属性名，也应当是数据库查询SQL语句的查询字段之一。

**XML配置举例：**

1. **<sql** id="XSQL\_MessageKey\_Query\_SysID"**>**
3. **<dataSourceGroup** ref="DSG\_LogWeb" **/>**
5. **<content>**
6. <![CDATA[
7. SELECT  A.sysid
8. ,A.msgKey
9. FROM  TMessageKey  A
10. WHERE  A.sysid IS NOT NULL
11. AND  A.sid   IS NULL
12. ORDER  BY A.sysid
13. ]]>
14. **</content>**
16. **<result>**
17. **<table>**java.util.HashMap**</table>**
18. **<fill>**put(row.sysid ,row.msgKey)**</fill>**
19. **<row>**org.hy.log.model.MessageKey**</row>**
20. **<cfill>**setter(colValue)**</cfill>**
21. **</result>**
23. **</sql>**

注：上面XSQL返回的Java结构为HashMap<String ,MessageKey>，其HashMap.key为Messagekey.getSysId()方法的值。

# XSQL查询结果为表分区结构的查询

表分区相信大家都知道，即按某一个字段信息将数据分类存储的结构。

Java的数据结构形式为Map<Object ,List<Object>>。

Map.key为分区字段。

Map.value为分区数据，即List<Object>。

表分区结构的最大好处是：可以通过分区字段快速定位一组数据。

现已封装好如下表分区结构的类，方便大家使用：

1. org.hy.common. TablePartition：常规表分区
2. org.hy.common. TablePartitionSet：表中每个分区是Set集合的结构。
3. org.hy.common. TablePartitionLink：有分区顺序的表分区。
4. org.hy.common. TablePartitionRID：分区主键索引的表分区。表中每个分区是Map集合的结构。

**XML配置语法：**

1. **<sqls>**
3. **<sql>**
5. **<dataSourceGroup** ref="数据库连接池组的对象引用" **/>**
7. **<content>**
8. <![CDATA[
9. SELECT \* FROM 表名
10. ]]>
11. **</content>**
13. **<result>**
14. **<table>**org.hy.common. TablePartition**</row>**  <!-- 表级的对象类型 -->
15. **<fill>**putRow(row.成员属性名 ,row)**</fill>** <!-- 行级对象填充到表级对象的填充方法 -->
16. **<row>**行数据对应的Java对象**</row>**
17. **<cfill>**setter(colValue)**</cfill>**  <!-- 用什么方法将每个字段填充到Java对象 -->
18. **</result>**
20. **</sql>**
22. **</sqls>**

**XML配置说明：**

1. 与“[XSQL查询结果为MAP结构的查询](#_XSQL查询结果为Map结构的查询)”类似，只是<table>和<fill>两节点不同。
2. row.成员属性名：将哪个成员属性名当作表分区的分区字段。
3. <fille>方法名可查询相关类的源码，这里先简单罗列一个。
   1. TablePartition 使用 putRow(分区字段 ,分区数据) 填充。
   2. TablePartitionSet 使用 putRow(分区字段 ,分区数据) 填充。
   3. TablePartitionLink使用 putRow(分区字段 ,分区数据) 填充。
   4. TablePartitionRID 使用 putRow(分区字段，主键索引 ,分区数据) 填充。主键索引也是Java对象的成员属性，需定义成员属性的getter()方法才能使用。

**XML配置举例：**

1. **<sql** id="XSQL\_XFlow\_ActivityRoute\_QueryByTemplateID"**>**
3. **<dataSourceGroup** ref="DSG\_XFlow" **/>**
5. **<content>**
6. <![CDATA[
7. SELECT  A.arID
8. ,A.activityID
9. ,A.templateID
10. ,A.nextActivityID
11. ,A.conditionXJavaID
12. ,A.conditionMethod
13. ,A.conditionValue
14. ,A.conditionVType
15. ,A.title
16. ,A.infoComment
17. ,A.createrID
18. ,A.creater
19. ,A.createTime
20. ,A.lastUserID
21. ,A.lastUser
22. ,A.lastTime
23. FROM  TActivityRoute  A
24. WHERE  A.templateID = ':templateID'
25. ]]>
26. **</content>**
28. **<result>**
29. **<table>**org.hy.common.TablePartition**</table>**
30. **<fill>**putRow(row.activityID ,row)**</fill>**
31. **<row>**org.hy.xflow.engine.bean.ActivityRoute**</row>**
32. **<cfill>**setter(colValue)**</cfill>**
33. **</result>**
35. **</sql>**

# XSQL表字段与Java属性名不同时的查询

数据库表A有个name\_code字段，对应的Java类A.java的成员属性叫nameCode，两名称不一样，如何映射呢？

答案是：通过SQL查询语句的别名来实现映射关系，给name\_code起个别名叫nameCode即可，如下。

1. **SELECT**  name\_code  **AS** nameCode
2. **FROM**  表A

# XSQL一对一复合型查询

Java类A.java的成员属性类型为Java类B.java，对象 A与B是一对一关系，对象A包含B的“类中类”。如何生成这样的数据结构的XSQL查询呢？

答案还是：通过SQL查询语句的别名来实现映射关系。赋予别名面向对象的概念，通过“类.属性”（中间有个点）来层层映射，将别名的用途发挥到极致。如下形式。

1. **SELECT**  A.字段A1
2. ,A.字段A2
3. ,A.字段An
4. ,B.字段B1   **AS** "B.字段B1"
5. ,B.字段B2   **AS** "B.字段B2"
6. ,B.字段Bn   **AS** "B.字段Bn"
7. ,C.字段B1   **AS** "C.字段C1"
8. ,C.字段B1   **AS** "C.字段C2"
9. ,C.字段B1   **AS** "C.字段Cn"
10. **FROM**  表A  A
11. ,表B  B
12. ,表C C
13. **WHERE**  A.BID = B.ID
14. **AND**  A.CID = C.ID

同时，也可以支持一个类中包含多个子类的情况的。

**XML配置举例：**

1. **<sql** id="XSQL\_XFlow\_ActivityInfo\_QueryByTemplateID"**>**
3. **<dataSourceGroup** ref="DSG\_XFlow" **/>**
5. **<content>**
6. <![CDATA[
7. SELECT  A.activityID
8. ,A.templateID
9. ,A.activityName
10. ,A.activityTypeID
11. ,B.activityTypeID  AS "activityType.activityTypeID"
12. ,B.activityType    AS "activityType.activityType"
13. ,B.infoComment     AS "activityType.infoComment"
14. FROM  TActivityInfo  A
15. ,TActivityType B
16. WHERE  A.activityTypeID = B.activityTypeID
17. AND  A.templateID     = ':templateID'
18. ]]>
19. **</content>**
21. **<result>**
22. **<row>**org.hy.xflow.engine.bean.ActivityInfo**</row>**
23. **<cfill>**setter(colValue)**</cfill>**
24. **</result>**
26. **</sql>**

XSQL中的行级Java对象定义如下：

1. **public** **class** ActivityInfo
2. {
3. /\*\* 工作流活动ID \*/
4. **private** String activityID;
6. /\*\* 工作流的模板ID \*/
7. **private** String templateID;
9. /\*\* 活动名称 \*/
10. **private** String activityName;
12. /\*\* 活动类型ID \*/
13. **private** String activityTypeID;
15. /\*\* 活动类型 \*/
16. **private** ActivityType activityType;

19. /\*\*
20. \* 获取：工作流活动ID
21. \*/
22. **public** String getActivityID()
23. {
24. **return** **this**.activityID;
25. }

28. /\*\*
29. \* 设置：工作流活动ID
30. \*
31. \* @param i\_ActivityID
32. \*/
33. **public** **void** setActivityID(String i\_ActivityID)
34. {
35. **this**.activityID = i\_ActivityID;
36. }

39. /\*\*
40. \* 获取：工作流的模板ID
41. \*/
42. **public** String getTemplateID()
43. {
44. **return** **this**.templateID;
45. }

48. /\*\*
49. \* 设置：工作流的模板ID
50. \*
51. \* @param i\_TemplateID
52. \*/
53. **public** **void** setTemplateID(String i\_TemplateID)
54. {
55. **this**.templateID = i\_TemplateID;
56. }

59. /\*\*
60. \* 获取：活动名称
61. \*/
62. **public** String getActivityName()
63. {
64. **return** **this**.activityName;
65. }

68. /\*\*
69. \* 设置：活动名称
70. \*
71. \* @param i\_ActivityName
72. \*/
73. **public** **void** setActivityName(String i\_ActivityName)
74. {
75. **this**.activityName = i\_ActivityName;
76. }

79. /\*\*
80. \* 获取：活动类型ID
81. \*/
82. **public** String getActivityTypeID()
83. {
84. **return** **this**.activityTypeID;
85. }

88. /\*\*
89. \* 设置：活动类型ID
90. \*
91. \* @param i\_ActivityTypeID
92. \*/
93. **public** **void** setActivityTypeID(String i\_ActivityTypeID)
94. {
95. **this**.activityTypeID = i\_ActivityTypeID;
96. }

99. /\*\*
100. \* 获取：活动类型
101. \*/
102. public ActivityType getActivityType()
103. {
104. return activityType;
105. }

108. /\*\*
109. \* 设置：活动类型
110. \*
111. \* @param activityType
112. \*/
113. **public** **void** setActivityType(ActivityType activityType)
114. {
115. this.activityType = activityType;
116. }

119. }

Java类中类ActivityType定义如下：

1. **public** **class** ActivityType
2. {
4. /\*\* 工作流活动类型ID \*/
5. **private** String  activityTypeID;
7. /\*\* 工作流活动类型名称 \*/
8. **private** String  activityType;
10. /\*\* 备注说明 \*/
11. **private** String  infoComment;
13. /\*\* 排列顺序 \*/
14. **private** Integer orderNo;


18. /\*\*
19. \* 获取：工作流活动类型ID
20. \*/
21. **public** String getActivityTypeID()
22. {
23. **return** **this**.activityTypeID;
24. }

27. /\*\*
28. \* 设置：工作流活动类型ID
29. \*
30. \* @param i\_ActivityTypeID
31. \*/
32. **public** **void** setActivityTypeID(String i\_ActivityTypeID)
33. {
34. **this**.activityTypeID = i\_ActivityTypeID;
35. }

38. /\*\*
39. \* 获取：工作流活动类型名称
40. \*/
41. **public** String getActivityType()
42. {
43. **return** **this**.activityType;
44. }

47. /\*\*
48. \* 设置：工作流活动类型名称
49. \*
50. \* @param i\_ActivityType
51. \*/
52. **public** **void** setActivityType(String i\_ActivityType)
53. {
54. **this**.activityType = i\_ActivityType;
55. }

58. /\*\*
59. \* 获取：备注说明
60. \*/
61. **public** String getInfoComment()
62. {
63. **return** **this**.infoComment;
64. }

67. /\*\*
68. \* 设置：备注说明
69. \*
70. \* @param i\_InfoComment
71. \*/
72. **public** **void** setInfoComment(String i\_InfoComment)
73. {
74. **this**.infoComment = i\_InfoComment;
75. }

78. /\*\*
79. \* 获取：排列顺序
80. \*/
81. **public** Integer getOrderNo()
82. {
83. **return** orderNo;
84. }

87. /\*\*
88. \* 设置：排列顺序
89. \*
90. \* @param orderNo
91. \*/
92. **public** **void** setOrderNo(Integer orderNo)
93. {
94. **this**.orderNo = orderNo;
95. }
97. }

# XSQL一对多复合型查询

Java类A.java的成员属性类型为List<Java类B.java>（或Set<…> 或Map<…>），对象 A与B是一对多关系，类似于树结构中父节点（主表）与子节点（子表）的关系。

使用XSQL.result. relationKeys属性来实现。relationKeys标记出能表示一对多关系中归属同一对象的关系字段，组合关系的多个字段间用逗号分隔。

relationKeys关系字段一般为主表中的主键字段，或是主表存在于子表中的外键字段。

注意：SQL语句需按关系字段排序。

**XML配置语法：**

1. **<sqls>**
3. **<sql>**
5. **<dataSourceGroup** ref="数据库连接池组的对象引用" **/>**
7. **<content>**
8. <![CDATA[
9. SELECT  A.字段A1
10. , A.字段A2
11. , A.字段An
12. , B.字段B1 AS "B.字段B1"
13. , B.字段B2 AS "B.字段B2"
14. , B.字段Bn AS "B.字段Bn"
15. FROM  主表 A
16. ,子表 B
17. WHERE A.主键字段 = B.外键字段
18. ORDER BY A.关系字段
19. ]]>
20. **</content>**
22. **<result>**
23. **<row>**行数据对应的Java对象**</row>**
24. **<cfill>**setter(colValue)**</cfill>**  <!-- 将字段填充到Java对象的方法 -->
25. **<relationKeys>**关系字段**</relationKeys>** <!--组合关系的多个字段间用逗号分隔-->
26. **</result>**
28. **</sql>**
30. **</sqls>**

**XML配置举例：**

比如，查询数据库所有表及表字段的信息。

表字段对应的Java对象：ColumnInfo.java

表信息对应的Java对象：TableInfo.jara

1. **public** **class** TableInfo
2. {
4. /\*\* 表名称 \*/
5. **private** String           nameDB;
7. /\*\* 字段信息 \*/
8. **private** List<ColumnInfo> tableColumns;
10. }

表名称nameDB属性是TableInfo与ColumnInfo两对象的关系纽带。在编写SQL时，应一次性查询出两表组合而成的所有关系数据，假如有3个表，每个表都有5个字段，查询结果的数据量应为：3 \* 5 = 15 条记录。用Oracle数据库举例的SQL语句如下：

1. **<sql** id="XSQL\_TablesColumns"**>**
3. **<dataSourceGroup** ref="DSG\_Template" **/>**
5. **<content>**
6. <![CDATA[
7. SELECT  A.table\_Name     AS nameDB
8. ,B.column\_Name    AS "tableColumns.name"
9. ,B.data\_Type      AS "tableColumns.typeDB"
10. ,B.data\_Length    AS "tableColumns.length"
11. FROM  User\_Tables              A
12. ,User\_Tab\_Columns         B
13. WHERE  A.table\_Name = B.table\_Name
14. ORDER  BY A.table\_Name ,B.column\_id
15. ]]>
16. **</content>**
18. **<result>**
19. **<row>**org.hy.common.xml.junit.template.TableInfo**</row>**
20. **<cfill>**setter(colValue)**</cfill>**
21. **<relationKeys>**nameDB**</relationKeys>**
22. **</result>**
24. **</sql>**

# XSQL一行分拆成多行的查询（填充事件）

暂时详见 org.hy.common.xml.XSQLResultFillEvent。后面补充说明

# XSQL一列分拆成多列的查询（填充事件）

暂时详见 org.hy.common.xml.XSQLResultFillEvent。后面补充说明

# XSQL多行合并为一行的查询（填充事件）

暂时详见 org.hy.common.xml.XSQLResultFillEvent。后面补充说明

# XSQL方便的写入CLob数据

常规写入CLob字段类型数据要分成两步：

1：Insert写入常规字段，同时用EMPTY\_CLOB()关键字占位；

2：再通过SELECT查询获取CLob数据流，写入CLob长文本信息，同时加FOR UPDATE锁。如下两组SQL。

1. **INSERT** **INTO** tablename
2. (
3. id
4. ,clobColumn
5. )
6. **VALUES** (
7. 1
8. ,EMPTY\_CLOB()
9. );
10. **SELECT**  clobColumn
11. **FROM**  tablename
12. **WHERE**  id = 1
13. **FOR**  **UPDATE**

用XSQL实现就方便很多了，只须一步，只须Insert写入常规字段。之后的长文本信息的写入，XSQL能自动完成，无须开发者实现。对于开发者来说，CLob字段相当于普通类型的写入。

**XML配置语法：**

1. **<sql>**
3. **<dataSourceGroup** ref="数据库连接池组的对象引用" **/>**
5. <!-- 多个用逗号，分隔 -->
6. **<lobName>**CLob字段01 ,CLob字段02**</lobName>**
7. **<lobWheres>**主键id名称 = ':主键的占位符'**</lobWheres>**
9. **<content>**
10. <![CDATA[
11. INSERT  INTO 表名称
12. (
13. 主键id名称
14. ,普通字段
15. ,CLob字段01
16. ,CLob字段02
17. )
18. VALUES (
19. ':主键的占位符'
20. ,':普通字段的占位符'
21. ,EMPTY\_CLOB()
22. ,EMPTY\_CLOB()
23. )
24. ]]>
25. **</content>**
27. **</sql>**

**XML配置说明：**

1. <lobName>节点：大数据字段类型(如,CLob)的字段名称，多个字段名称间用逗号，分隔。只用于Insert、Update语句。
2. <lobWheres>节点：写入大数据字段类型(如,CLob)的所在行的查询条件SQL片段(不包含WHERE关键字)。只用于Insert、Update语句。此节点中可以写占位符，并且占位符也支持面向对象，如下举例。
3. <lobName>、<lobWheres>两节点的信息将组合生成如下SQL
4. **SELECT**  <lobName>节点
5. **FROM**  tablename          -- 表名称自动识别
6. **WHERE**  <lobWheres>节点
7. **FOR**  **UPDATE**

**XML配置举例：**

1. **<sql>**
3. **<dataSourceGroup** ref="DSG\_Oracle\_CalcCloud" **/>**
5. **<lobName>**notes,cause**</lobName>**
6. **<lobWheres>**order\_no = ':order\_no'**</lobWheres>**
8. **<content>**
9. <![CDATA[
10. INSERT  INTO ORD\_D\_Order\_Head
11. (
12. order\_no
13. ,orderNo8
14. ,dept
15. ,forder\_no
16. ,CLob字段01
17. ,status
18. ,CLob字段02
19. ,dataCreateTime
20. )
21. VALUES (
22. ':order\_no'
23. ,':orderNo8'
24. ,':dept'
25. ,':forder\_no'
26. ,EMPTY\_CLOB()
27. ,':status'
28. ,EMPTY\_CLOB()
29. ,SYSDATE
30. )
31. ]]>
32. **</content>**
34. **</sql>**

# XSQL执行多个SQL语句

XSQL被Java调用一次，有能力顺次向数据库发起多个SQL语句的执行命令。通常来说，这些SQL语句均为非查询语句的DDL、DML、DCL、TCL语句等。

每个SQL语句以分号加除号 **;/** 结尾，表示一个完整SQL语句的结束。

为什么使用 **;/** 两个符号组成的标记表示结尾呢？原因是，只有这样才能更好的支持数据库存储过程及函数等复杂对象的构造。

Java在调用此类XSQL时，应使用XSQL.execute()系列的方法执行。此方法只返回是否执行成功。

执行某一个SQL语句异常后，其后的SQL语句将不再执行，本SQL语句前的已执行成功 的SQL语句不会回滚，由外界控制提交commit的方式除外，但因DDL等语句数据库也不支持回滚操作，所以创建后也无法同一事务中回滚。

**XML配置举例：**

详见“[XSQL创建数据库对象](#_XSQL创建数据库对象)”

# XSQL批量执行

批量执行与“[XSQL执行多个SQL语句](#_XSQL执行多个SQL语句)”是两种执行方式。批量执行指：相同SQL模板（相同主体语句的SQL）的XSQL循环执行多次，执行次数可由调用方控制，并统一提交或分批提交的操作。可以简单的理解为将XSQL放在for循环中执行。

但“XSQL执行多个SQL语句”是指调用一次XSQL后，XSQL向数据库发起多个不同SQL语句的执行命令，执行SQL的数量由<content>节点预先设定好，调用方不控制。

批量执行一般用于：Insert、Update、Delete语句。与单次执行的XSQL的XML配置一样，两者无差异，只是在Java代码调用时，调用方法不一样而已。

批量执行的Java代码：

XJava.getXSQL("XML文件中配置ID").executeUpdates(List<值对象 或 Map对象>);

单次执行的Java代码：

XJava.getXSQL("XML文件中配置ID").executeUpdate(值对象 或 Map对象);

XSQL. batchCommit属性可控制批量执行时，是否分批提交及每次达到多少提交一次。默认为零值，表示批量操作完成后整体提交。batchCommit属性也可配置在XML文件中，如下：

1. **<sql>**
3. **<dataSourceGroup** ref="数据库连接池组的对象引用"**/>**
5. **<batchCommit>**100**</batchCommit>**
7. **<content>**
8. <![CDATA[
9. Insert、Update、Delete语句
10. ]]>
11. **</content>**
12. **</sql>**

# XSQL预处理的批量执行

如果只是简单的数据操作（Insert、Update、Delete），没用各种各样复杂可变的SQL等（如，[表名可变的SQL](#_XSQL表名可变的SQL)），为了提高SQL的执行性能，建议使用预处理的批量执行。

预处理的批量执行与上一节的批量执行大体一样。只有两点不同的地方需要注意。

**注意1：**

对于数据库任一的数据类型（如，字符串类型的字段），不再用两单引号作限定符。

**注意2：**

当两个或以上的占位符变量名称的前缀相同时，应按名称长度降序的顺序编写其占位符出现在SQL语句的位置。防止短名称的占位符错误替换了长名称前缀相同的占位符。

预处理批量执行的Java代码：

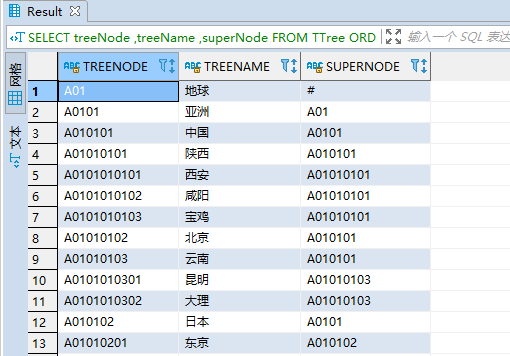
XJava.getXSQL("XML文件中配置ID").executeUpdatesPrepared(List<值对象 或 Map对象>);

XSQL. batchCommit属性与[批量执行](#_XSQL批量执行)一样。

# XSQL树结构查询

对于多级省地市、多层组织机构、系统菜单等树结构的数据查询、Java结构如何组织呈现呢？

用省地市大家都熟悉的树结构举例吧，如下图



XSQL配置文件\*.xml就按常规SQL写，如下图。

1. **<sqls>**
3. **<sql>**
5. **<dataSourceGroup** ref="数据库连接池组的对象引用" **/>**
7. **<content>**
8. <![CDATA[
9. SELECT treeNode AS cityID
10. ,treeName AS cityName
11. ,superNode AS superID
12. FROM TTree
13. ]]>
14. **</content>**
16. **<result>**
17. **<table>**org.hy.common.TreeObject**</table>**  <!-- 集合类名固定写法 -->
18. **<fill>**put(row)**</fill>** <!-- 填充方法名称 -->
19. **<row>**xx.xx.model.CityInfo**</row>** <!-- 实现接口TreeObjectNode的对象 -->
20. **<cfill>**setter(colValue)**</cfill>**   <!-- 固定写法 -->
21. **</result>**
23. **</sql>**
25. **</sqls>**

**XML配置说明：**

1. <table>节点：TreeObject是个Map集合。
2. <fill>节点：使用TreeObject.put(节点ID ,父节点ID ,节点对象) 方法填充Map集合元素。
3. <row>节点：实现org.hy.common.TreeObjectNode接口值对象Model。见如下文件

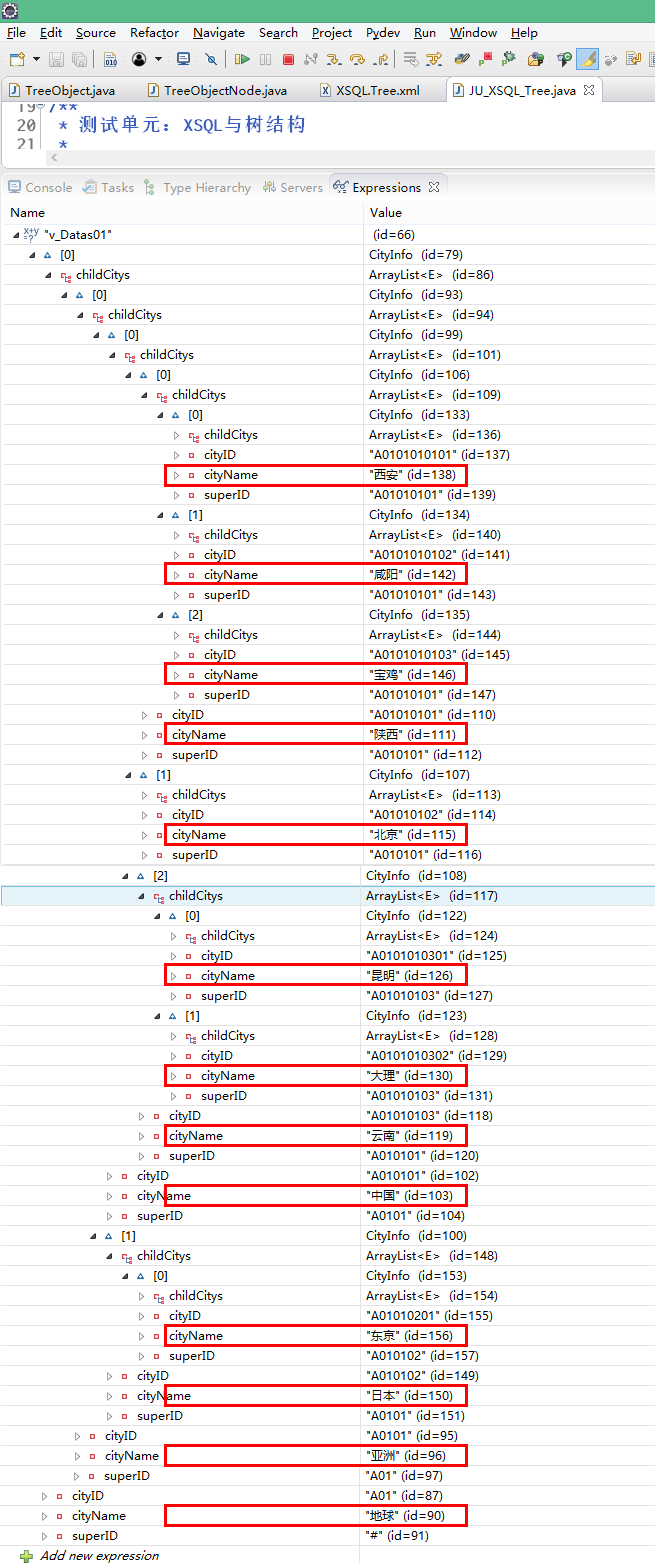


1. 对数据零要求：节点ID编码无要求、数据排序无要求(先入父后入子性能最高)、树结构层次无限制。

**Java代码举例：**

1. // 获取XSQL对象
2. XSQL v\_XSQL = XJava.getXSQL("XSQL\_Tree");
3. // 查询并获取返回结果（树结构）
4. **TreeObject<CityInfo>** v\_Tree = (TreeObject<CityInfo>)v\_XSQL.**query()**;
5. // 将查询结果输出为一个List集合（也是树结构）
6. List<CityInfo> v\_Datas = v\_Tree.**getTreeList()**;
8. // 将查询结果输出为一个Map集合（也是树结构，Map是有顺序的）
9. Map<String ,CityInfo> v\_Datas = v\_Tree.**getTreeMap()**;

Java结构展示，如下图



生成的树结构展示：

1. {
2. "childCitys": [
3. {
4. "childCitys": [
5. {
6. "childCitys": [
7. {
8. "childCitys": [
9. {
10. "childCitys": [],
11. "cityID": "A0101010101",
12. "cityName": "西安",
13. "superID": "A01010101"
14. },
15. {
16. "childCitys": [],
17. "cityID": "A0101010102",
18. "cityName": "咸阳",
19. "superID": "A01010101"
20. },
21. {
22. "childCitys": [],
23. "cityID": "A0101010103",
24. "cityName": "宝鸡",
25. "superID": "A01010101"
26. }
27. ],
28. "cityID": "A01010101",
29. "cityName": "陕西",
30. "superID": "A010101"
31. },
32. {
33. "childCitys": [],
34. "cityID": "A01010102",
35. "cityName": "北京",
36. "superID": "A010101"
37. },
38. {
39. "childCitys": [
40. {
41. "childCitys": [],
42. "cityID": "A0101010301",
43. "cityName": "昆明",
44. "superID": "A01010103"
45. },
46. {
47. "childCitys": [],
48. "cityID": "A0101010302",
49. "cityName": "大理",
50. "superID": "A01010103"
51. }
52. ],
53. "cityID": "A01010103",
54. "cityName": "云南",
55. "superID": "A010101"
56. }
57. ],
58. "cityID": "A010101",
59. "cityName": "中国",
60. "superID": "A0101"
61. },
62. {
63. "childCitys": [
64. {
65. "childCitys": [],
66. "cityID": "A01010201",
67. "cityName": "东京",
68. "superID": "A010102"
69. }
70. ],
71. "cityID": "A010102",
72. "cityName": "日本",
73. "superID": "A0101"
74. }
75. ],
76. "cityID": "A0101",
77. "cityName": "亚洲",
78. "superID": "A01"
79. }
80. ],
81. "cityID": "A01",
82. "cityName": "地球",
83. "superID": "#"
84. }

Java及数据说明：

1. **节点ID不为空**：树节点ID。不可为空或空指针。

2. **多个顶级根节点**：允许存在多个顶级根节点。即，没有只能有一个顶级根节点的限制。

3. **顶级根节点**： i\_SuperID=空值并不是判定为顶级根节点的标准。标准是：当任何节点均不引用它时，i\_SuperID才能被判定为顶级根节点

4. **不可递归**：不可在树结构中出现递归引用。相信大家能理解，不再赘述原因

5. **重复覆盖**：重复节点ID在构建树时，相同节点ID将会覆盖，其子节点将被级联删除

6. **节点顺序**： 顶级根节点的顺序按 “先来后到” 排序。子节点的顺序由最终用户决定。即实现接口 TreeObjectNode 方来维护。

注：版本最低支持的要求

1. <dependency>
2. <groupId>**org**</groupId>
3. <artifactId>**hy.common.base**</artifactId>
4. <version>**1.0.152**</version>
5. </dependency>

# XSQL存储过程及函数（无入参的）

XSQL调用数据库存储过程及函数。因无入参的相对简单，先讲无入参的，其后再讲有参数。

如返回为游标（查询结果集），也能映射为Java对象。详细常量说明见：org.hy.common.xml.XSQLCallParam类。

**XML配置语法：**

1. **<sql>**
3. **<dataSourceGroup** ref="数据库连接池组的对象引用" **/>**
5. **<type>**存储过程P/函数F的类型**</type>**
7. **<content>**存储过程/函数的名称**</content>**
9. **</sql>**

**XML配置说明：**

1. <type>节点有四个取值可选，但此处只用前两个
   1. P：表示存储过程
   2. F：表示函数
   3. N：增、删、改、查的SQL语句。它也是默认值。
   4. C：DDL、C创建表，创建对象等。
2. <content>节点为存储过程或函数的名称。

**XML配置举例：**

1. **<sql>**
3. **<dataSourceGroup** ref="DSG" **/>**
5. **<type>**P**</type>**
7. **<content>**P\_C\_HY**</content>**
9. **</sql>**

注：存储过程P\_C\_HY的调用。无入参参数，无返回值。

1. **<sql** id="XSQL\_Function"**>**
3. **<dataSourceGroup** ref="DSG" **/>**
5. **<type>**F**</type>**
7. **<content>**F\_C\_HY**</content>**
9. **<addParam>**
10. **<name>**Return**</name>**
11. **<jdbcType>**java.sql.Types.VARCHAR**</jdbcType>**
12. **<type** ref="this.$Type\_OUT" **/>**
13. **</addParam>**
15. **</sql>**

注：函数F\_C\_HY的调用。无入参参数，函数返回值类型为VARCHAR。

Java调用代码如下：

1. XSQL   v\_XSQL  = XJava.getXSQL("XML文件中配置ID");
2. Object v\_Datas = v\_XSQL.**call**(查询参数的值对象或Map对象);
3. // 调用存储过程、函数用call方法，与查询、插入、更新和删除的XSQL执行方法不同。

# XSQL存储过程及函数（带参数的）

带参数的相对复杂一点儿。

此功能，在掌握轻量级ETL的XSQL组（XSQLGroup）功能后，几乎不怎么用了，所以配置写法上优化的少了些。

**XML配置语法：**

1. **<sql>**
3. **<dataSourceGroup** ref="数据库连接池组的对象引用" **/>**
5. **<type>**存储过程P/函数F的类型**</type>**
7. **<content>**存储过程/函数的名称**</content>**
9. <!-- 添加输入类型的参数 -->
10. **<addParam>**
11. **<name>**参数名称**</name>**
12. **<jdbcType>**参数类型**</jdbcType>**
13. **<type>**IN**</type>**           <!-- 默认为 IN 参数，所以可以没有<type>节点 -->
14. **</addParam>**

17. <!-- 添加输出类型的参数 -->
18. **<addParam>**
19. **<name>**参数名称**</name>**
20. **<jdbcType>**参数类型**</jdbcType>**
21. **<type>**OUT**</type>**
22. **</addParam>**
24. **</sql>**

**XML配置说明：**

1. <type>节点：通过它标记出参数是输入类的参数，还是输出类的参数。默认值 为：IN，输入类的参数。此节点有三个可选值
   1. IN ：输入类型。
   2. OUT：输出类型。
   3. IN OUT：输入输出类型。
2. <name>节点为参数名称，与占位符同义，但不用前加冒号。
3. <jdbcType>节点为参数类型。
   1. 当为输入类型参数时，jdbcType = string ，转为 setParamMethod = CallableStatement.setString(int ,String) 方法。
   2. 当为输出类型参数时，jdbcType = java.sql.Types.VARCHAR ，转为 jdbcTypeID = 12 。

**XML配置举例：**

1. <!-- 调用存储过程 -->
2. **<sql** id="XSQL\_Call\_Procedure"**>**
4. **<dataSourceGroup** ref="DSG" **/>**
6. **<type>**P**</type>**
8. **<content>**P\_C\_HY**</content>**
10. <!-- 添加输入类型的参数 -->
11. **<addParam>**
12. **<name>**beginTime**</name><jdbcType>**DATE**</jdbcType>**
13. <!-- 默认为 IN 参数，所以可以没有<type>标记 -->
14. **</addParam>**
16. **<addParam>**
17. **<name>**endTime**</name>**
18. **<jdbcType>**DATE**</jdbcType>**
19. **<type>**IN**</type>**               <!-- <type>标记写法1 -->
20. **</addParam>**
22. **<addParam>**
23. **<name>**info**</name>**
24. **<jdbcType>**STRING**</jdbcType>**
25. **<type** ref="this.$Type\_IN" **/>**   <!-- <type>标记写法2 -->
26. **</addParam>**
28. **<addParam>**
29. **<name>**age**</name>**
30. **<jdbcType>**INT**</jdbcType>**
31. **<type** ref="param.$Type\_IN" **/>**     <!-- <type>标记写法3 -->
32. **</addParam>**
34. <!-- 添加输出类型的参数 -->
35. **<addParam>**
36. **<name>**cursor01**</name>**
37. **<jdbcType>**oracle.jdbc.OracleTypes.CURSOR**</jdbcType>**
38. **<type>**OUT**</type>**
39. **</addParam>**
41. **<addParam>**
42. **<name>**beginTime**</name>**
43. **<jdbcType>**java.sql.Types.DATE**</jdbcType>**
44. **<type** ref="this.$Type\_OUT" **/>**
45. **</addParam>**
47. **<addParam>**
48. **<name>**endTime**</name>**
49. **<jdbcType>**java.sql.Types.DATE**</jdbcType>**
50. **<type** ref="this.$Type\_OUT" **/>**
51. **</addParam>**
53. **<addParam>**
54. **<name>**info**</name>**
55. **<jdbcType>**java.sql.Types.VARCHAR**</jdbcType>**
56. **<type** ref="this.$Type\_OUT" **/>**
57. **</addParam>**
59. **<addParam>**
60. **<name>**age**</name>**
61. **<jdbcType>**java.sql.Types.INTEGER**</jdbcType>**
62. **<type** ref="this.$Type\_OUT" **/>**
63. **</addParam>**
65. **</sql>**
66. <!-- 调用函数 -->
67. <!-- 注意Oracle的函数返回类型为Boolean是无须注册的 -->
68. **<sql** id="XSQL\_Call\_Function"**>**
70. **<dataSourceGroup** ref="DSG" **/>**
72. **<type>**F**</type>**
74. **<content>**F\_C\_HY**</content>**
76. <!-- 先添加函数返回值 -->
77. **<addParam>**
78. **<name>**Return**</name>**
79. **<jdbcType>**java.sql.Types.VARCHAR**</jdbcType>**
80. **<type** ref="this.$Type\_OUT" **/>**
81. **</addParam>**
83. <!-- 添加输入类型的参数 -->
84. **<addParam>**
85. **<name>**beginTime**</name>**
86. **<jdbcType>**DATE**</jdbcType>**
87. <!-- 默认为 IN 参数，所以可以没有<type>标记 -->
88. **</addParam>**
90. **<addParam>**
91. **<name>**endTime**</name>**
92. **<jdbcType>**DATE**</jdbcType>**
93. **<type>**IN**</type>**              <!-- <type>标记写法1 -->
94. **</addParam>**
96. **<addParam>**
97. **<name>**info**</name>**
98. **<jdbcType>**STRING**</jdbcType>**
99. **<type** ref="this.$Type\_IN" **/>**  <!-- <type>标记写法2 -->
100. **</addParam>**
102. **<addParam>**
103. **<name>**age**</name>**
104. **<jdbcType>**INT**</jdbcType>**
105. **<type** ref="param.$Type\_IN" **/>**     <!-- <type>标记写法3 -->
106. **</addParam>**
108. <!-- 添加输出类型的参数 -->
109. **<addParam>**
110. **<name>**cursor01**</name>**
111. **<jdbcType>**oracle.jdbc.OracleTypes.CURSOR**</jdbcType>**
112. **<type>**OUT**</type>**
113. **</addParam>**
115. **<addParam>**
116. **<name>**beginTime**</name>**
117. **<jdbcType>**java.sql.Types.DATE**</jdbcType>**
118. **<type** ref="this.$Type\_OUT" **/>**
119. **</addParam>**
121. **<addParam>**
122. **<name>**endTime**</name>**
123. **<jdbcType>**java.sql.Types.DATE**</jdbcType>**
124. **<type** ref="this.$Type\_OUT" **/>**
125. **</addParam>**
127. **<addParam>**
128. **<name>**info**</name>**
129. **<jdbcType>**java.sql.Types.VARCHAR**</jdbcType>**
130. **<type** ref="this.$Type\_OUT" **/>**
131. **</addParam>**
133. **<addParam>**
134. **<name>**age**</name>**
135. **<jdbcType>**java.sql.Types.INTEGER**</jdbcType>**
136. **<type** ref="this.$Type\_OUT" **/>**
137. **</addParam>**
139. **</sql>**

Java调用代码如下：

1. XSQL   v\_XSQL  = XJava.getXSQL("XML文件中配置ID");
2. Object v\_Datas = v\_XSQL.**call**(查询参数的值对象或Map对象);
3. // 调用存储过程、函数用call方法，与查询、插入、更新和删除的XSQL执行方法不同。

# XSQL分域

暂时详见 org.hy.common.xml.XSQLDomain。后面补充说明

实现相同数据库结构下的，多个数据库间的分域功能。

多个数据库间的相同SQL语句，不用重复写多次，只须通过"分域"动态改变数据库连接池组即可。

# XSQL安全（防止SQL注入攻击）

通过XSQL.contentDB.safeCheck属性控制是否检测XSQL安全。默认为开启状态。

在某些特殊情况下，如开发者想将完整的执行SQL语句保存在数据库中，当SQL执行日志记录时，此时的XSQL对象需要关闭安全检查，将safeCheck设置 为false即可。

**XML配置语法：**

1. **<sql>**
3. **<dataSourceGroup** ref="数据库连接池组的对象引用" **/>**
5. **<contentDB>**
6. **<safeCheck>tr**ue**</safeCheck>**
7. **</contentDB>**
9. **<content>**
10. <![CDATA[
11. SQL语句
12. ]]>
13. **</content>**
15. **</sql>**

**XML配置举例：**

1. **<sql** id="XSQL\_LogInfo\_Add"**>**
3. **<dataSourceGroup** ref="DSG\_LogWeb" **/>**
5. **<contentDB>**
6. **<keyReplace>**true**</keyReplace>**
7. **<safeCheck>**false**</safeCheck>**
8. **</contentDB>**
10. **<content>**
11. <![CDATA[
12. INSERT  INTO TLog
13. (
14. id
15. ,sysID
16. ,logID
17. ,logType
18. ,logClass
19. ,logContent
20. ,logInfo
21. ,operatorNo
22. ,operationType
23. ,operationRemark
24. ,operationTime
25. ,waitTime
26. )
27. VALUES (
28. ':id'
29. ,':sysID'
30. ,':logID'
31. ,':logType'
32. ,':logClass'
33. ,':logContent'
34. ,':logInfo'
35. ,':operatorNo'
36. ,':operationType'
37. ,':operationRemark'
38. ,':operationTime'
39. ,':waitTime'
40. )
41. ]]>
42. **</content>**
44. **</sql>**

# XSQL日期时间的处理

XSQL采用数据库自身的日期时间处理方法，只要符合数据库要求即可。Java对应的类型建议使用org.hy.common.Date类。下面罗列出常用数据库的日期转换方法，仅供参考。

1. Oracle数据库
   1. 当前时间转字符：TO\_CHAR(SYSDATE , ' yyyy-MM-dd HH24:mi:ss')
   2. 字符转时间：TO\_DATE ('2012-01-01 01:01:01' ,'yyyy-MM-dd HH24:mi:ss')
2. MySQL数据库
   1. 当前时间转字符：DATE\_FORMAT(NOW() ,'%Y-%m-%d %H:%i:%s')
   2. 字符转时间：STR\_TO\_DATE('2018-01-01 01:01:01' ,'%Y-%m-%d %H:%i:%s')
3. SQLServer数据库
   1. 当前时间转字符：CONVERT(VARCHAR(32) ,GETDATE() ,121) 。转换后的格式YYYY-MM-DD HH24:MI:SS
   2. 字符转时间：CONVERT(DATETIME ,'2018-01-01 01:01:01' ,121)

# XSQL执行统计分析

每个XSQL对象均有记录执行统计的数据，可方便提供给开发者统计分析。此外，预设有统计分析页面（<http://127.0.0.1:80/WebName/analyses>），如下图。

统计分析页面需在web.xml中添加配置后，才能正常使用。配置 方法见org.hy.common.xml.plugins.analyse.AnalyseObjectServlet类中的说明。



数据库访问量（支持集群）

数据库组合SQL访问量（支持集群）：



数据库连接池组使用情况（支持集群）：



查看SQL执行日志：



查看SQL执行异常日志（有异常时才有内容）：



# Java执行XSQL

XSQL是与MyBatis类似的数据库访问框架。XSQL自身可独立运行及使用，也可运行在XJava平台上，作为XJava平台上一个应用使用。本文讲的均是基于XJava平台的开发方法。

使用下面Java代码获取一个配置在XML文件中的XSQL对象。

1. XSQL v\_XSQL = XJava.getXSQL("XML文件中的XSQL节点的ID");
3. v\_XSQL.query(…);   // 查询数据库的一系列方法。
4. v\_XSQL.executeUpdate(…);  // 插入、更新、删除数据库的一系列方法。
5. v\_XSQL.execute(…);   // 数据库所有操作的一系列方法。
6. v\_XSQL.getSQLCount(…);   // 查询统计数据库记录行数的一系列方法。
7. v\_XSQL.queryBigData(…);   // 大数据查询及处理的一系列方法。
8. XSQL.queryPaging(…);   // 分页查询的一系列方法。

# Java注解@Xsql执行XSQL

暂时详见 org.hy.common.xml.annotation.Xsql。后面补充说明