XSQL开发说明

目录

[XSQL创建数据库对象 3](#_Toc512249968)

[XSQL创建多种数据库的对象 5](#_Toc512249969)

[XSQL创建多种数据库的对象并初始化数据 7](#_Toc512249970)

[XSQL应用级触发器 9](#_Toc512249971)

[XSQL常规List查询 11](#_Toc512249972)

[XSQL带参数的查询 14](#_Toc512249973)

[XSQL函数型参数（主键ID的生成） 16](#_Toc512249974)

[XSQL动态参数的查询 17](#_Toc512249975)

[XSQL表名可变的SQL 19](#_Toc512249976)

[XSQL表字段名可变的SQL 20](#_Toc512249977)

[XSQL排序组合Order By可变的SQL 20](#_Toc512249978)

[XSQL表分区内查询的可变SQL 21](#_Toc512249979)

[XSQL分页页码可变的SQL 21](#_Toc512249980)

[XSQL分页查询 22](#_Toc512249981)

[XSQL懒人查询 22](#_Toc512249982)

[XSQL查询结果为Map结构的查询 24](#_Toc512249983)

[XSQL查询结果为表分区结构的查询 26](#_Toc512249984)

[XSQL表字段与Java属性名不同时的查询 28](#_Toc512249985)

[XSQL一对一复合型查询 28](#_Toc512249986)

[XSQL一对多复合型查询 32](#_Toc512249987)

[XSQL执行多个SQL语句 32](#_Toc512249988)

[XSQL批量执行 33](#_Toc512249989)

[XSQL预处理的批量执行 34](#_Toc512249990)

[XSQL分域 34](#_Toc512249991)

[XSQL日期时间的处理 34](#_Toc512249992)

# 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），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。



**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 == DSG\_My.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 == DSG\_My.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 == DSG\_XFlow.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 == DSG\_XFlow.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 == DSG\_My.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 == DSG\_My.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 == DSG\_XFlow.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 == DSG\_XFlow.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、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函数型参数（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>**

# 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分页查询

# 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

# 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分域

# 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。所以，只要符合数据库要求即可。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)