XSQL开发说明

# 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”功能了。具体功能不在此赘述，请详见org.hy.common.xml.XSQLTrigger类中的说明。

**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。

**XML配置举例：**

详见“[XSQL创建数据库对象](#_XSQL创建数据库对象)”

# 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

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”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. }