**MCP**

**主讲老师：张胜利**

1. **pivot(行转列)和unpivot(列转行)**
2. **使用POVIT首先你需要在FROM子句内定义2个表：**

**A.一个称为源表(SourceTable)。**

**B.另一个称为数据透视表(PivotTable)。**

**语法：SELECT <未透视的列>, [第一个透视列] AS <列别名>,**

**[第二个透视列] AS <列别名>, ... [最后一个透视列] AS <列别名>**

**FROM ( <SELECT查询> ) AS <源表> PIVOT (**

**<聚合函数>(<列>) FOR [<需要转换为行的列>] IN ( [第一个透视列], [第二个透视列], … [最后一个透视列] ) ) AS <数据透视表> <可选的ORDER BY子句>;**

**以上的PIVOT子句内的第1…n个透视列的值均为需要转换为行的列的常量值,需要用[]括起（仅指PIVOT子句）**

**Eg:**

**select '总人数' as '班级' ,[初一 1班],[初二 1班],[初三 1班],[初三 2班]from (select class ,student\_no from Student)as sourcestu**

**pivot(count(student\_no) for class in ([初一 1班],[初二 1班],[初三 1班],[初三 2班]))as pivotstu ;**

1. **使用UNPIVOT 实现的功能其实与PIVOT恰恰相反**

**.语法同PIVOT, 但是UNPIVOT的子句没有聚合函数**

**SELECT <未逆透视的列>, [合并后的列] AS <列别名>,**

**[行值的列名] AS <列别名>**

**FROM ( <SELECT查询> ) AS <源表> UNPIVOT ( <行值的列名> FOR <将原来多个列合并到单个列的列名> IN ( [第一个合并列], [第二个合并列], ... [最后一个合并列] ) ) AS <数据逆透视表> <可选的ORDER BY子句>;**

**Eg:**

**---行转列：**

**with unpivotsource as (select '班级总人数' as '总人数',[初一 1班],[初二 1班],[初三 1班],[初三 2班] from (select class,student\_no from Student) as sourcestu**

**pivot(count(student\_no) for class in ([初一 1班],[初二 1班],[初三 1班],[初三 2班])) as pivotstu)**

**----列转行**

**select [班级],[总人数] from (select [初一 1班],[初二 1班],[初三 1班],[初三 2班] from unpivotsource ) as unsource**

**unpivot ([总人数] for [班级] in ([初一 1班],[初二 1班],[初三 1班],[初三 2班])) as un\_p**

**（二） APPLY运算符**

**使用 APPLY 运算符可以为实现查询操作的外部表表达式返回的每个行调用表值函数。表值函数作为右输入，外部表表达式作为左输入。通过对右输入求值来获得左输入每一行的计算结果，生成的行被组合起来作为最终输出。APPLY 运算符生成的列的列表是左输入中的列集，后跟右输入返回的列的列表。**

**APPLY 有两种形式： CROSS APPLY 和 OUTER APPLY。**

**CROSS APPLY 仅返回外部表中通过表值函数生成结果集的行。OUTER APPLY 既返回生成结果集的行，也返回不生成结果集的行，其中表值函数生成的列中的值为 NULL。**

**（三）存储过程**

**定义：**

**存储过程Procedure是一组为了完成特定功能的SQL语句集合，经编译后存储在数据库中，用户通过指定存储过程的名称并给出参数来执行。**

**存储过程中可以包含逻辑控制语句和数据操纵语句，它可以接受参数、输出参数、返回单个或多个结果集以及返回值。**

**由于存储过程在创建时即在数据库服务器上进行了编译并存储在数据库中，所以存储过程运行要比单个的SQL语句块要快。同时由于在调用时只需用提供存储过程名和必要的参数信息，所以在一定程度上也可以减少网络流量、简单网络负担。**

* **优点:**
* **存储过程加快系统运行速度，存储过程只在创建时编译，以后每次执行时不需要重新编译。**
* **存储过程可以封装复杂的数据库操作，简化操作流程，例如对多个表的更新，删除等。**
* **可实现模块化的程序设计，存储过程可以多次调用，提供统一的数据库访问接口，改进应用程序的可维护性。**
* **存储过程可以增加代码的安全性，对于用户不能直接操作存储过程中引用的对象，SQL  Server可以设定用户对指定存储过程的执行权限。**
* **存储过程可以降低网络流量，存储过程代码直接存储于数据库中，在客户端与服务器的通信过程中，不会产生大量的T\_SQL代码流量。**
* **缺点：**
* **数据库移植不方便，存储过程依赖与数据库管理系统， SQL Server 存储过程中封装的操作代码不能直接移植到其他的数据库管理系统中。**
* **不支持面向对象的设计，无法采用面向对象的方式将逻辑业务进行封装，甚至形成通用的可支持服务的业务逻辑框架.**
* **代码可读性差，不易维护。不支持集群。**

**语法:**

**create proc[edure] 过程名称**

**@参数名称 参数类型, --输入参数……@参数名称**

**参数类型 output --输出参数 As 执行语句**

**Eg:**

**Create procedure searchBooks(@bookId int,@bookAuth varchar(50),@bookName varchar(20) output)**

**as**

**set nocount on --不计返回数，有利于提高性能**

**select @bookName=book\_name from books where book\_id=@bookId and book\_auth=@bookAuth return @bookId**

**--执行过程**

**declare @id int,@name varchar(20)**

**exec @id=searchBooks 1,'孔子',@name output**

**print @id print @name**

**存储过程的加密：**

**with encryption --加密定义Sql语句文本**

**需要 在参数之后as之前加入 with encryption，因为加密后无法再看到定义文本，因而开发者需要保留好定义语句。**

**Eg: create proc proc\_temp\_encryption with encryption as  
select \* from student;**

**（四）事务（transaction）**

**作用：保持逻辑数据一致性与可恢复性。**

**常用语句：**

1. **Begin Transaction：标记事务开始。**
2. **Commit Transaction：事务已经成功执行，数据已经处理妥当。**
3. **Rollback Transaction：数据处理过程中出错，回滚到没有处理之前的数据状态，或回滚到事务内部的保存点。**
4. **Save Transaction：事务内部设置的保存点，就是事务可以不全部回滚，只回滚到这里，保证事务内部不出错的前提下。**

**TRY...CATCH**

**TRY…CATCH 构造包括两部分：一个 TRY 块和一个 CATCH 块。如果在 TRY 块内的 Transact-SQL 语句中检测到错误条件，则控制将被传递到 CATCH 块（可在此块中处理此错误）。**

**CATCH 块处理该异常错误后，控制将被传递到 END CATCH 语句后面的第一个 Transact-SQL 语句。如果 END CATCH 语句是存储过程或触发器中的最后一条语句，则控制将返回到调用该存储过程或触发器的代码。将不执行 TRY 块中生成错误的语句后面的 Transact-SQL 语句。如果 TRY 块中没有错误，控制将传递到关联的 END CATCH 语句后紧跟的语句。如果 END CATCH 语句是存储过程或触发器中的最后一条语句，控制将传递到调用该存储过程或触发器的语句。**

**TRY 块以 BEGIN TRY 语句开头，以 END TRY 语句结尾。在 BEGIN TRY 和 END TRY 语句之间可以指定一个或多个 Transact-SQL 语句。**

**CATCH 块必须紧跟 TRY 块。CATCH 块以 BEGIN CATCH 语句开头，以 END CATCH 语句结尾。在 Transact-SQL 中，每个 TRY 块仅与一个 CATCH 块相关联。**

**错误函数**

* **TRY...CATCH 使用下列错误函数来捕获错误信息：**
* **ERROR\_NUMBER() 返回错误号。**
* **ERROR\_MESSAGE() 返回错误消息的完整文本。此文本包括为任何可替换参数（如长度、对象名或时间）提供的值。**
* **ERROR\_SEVERITY() 返回错误严重性。**
* **ERROR\_STATE() 返回错误状态号。**
* **ERROR\_LINE() 返回导致错误的例程中的行号。**
* **ERROR\_PROCEDURE() 返回出现错误的存储过程或触发器的名称。**

**Eg:**

**BEGIN TRY**

**SELECT \* FROM sys.messages WHERE message\_id = 21; END TRY**

**BEGIN CATCH**

**SELECT ERROR\_NUMBER() AS ErrorNumber;**

**END CATCH;**

**---开启事务**

**begin tran**

**--错误扑捉机制，看好啦，这里也有的。并且可以嵌套。**

**begin try**

**--语句正确**

**insert into lives (Eat,Play,Numb) values ('猪肉','足球',1)**

**--Numb为int类型，出错**

**insert into lives (Eat,Play,Numb) values ('猪肉','足球','abc') --语句正确**

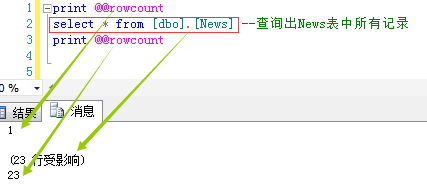
**insert into lives (Eat,Play,Numb) values ('狗肉','篮球',2)**

**end try**

**在SqlServer里，嵌套事务的层次是由@@TranCount全局变量反映出来的。  
每一次Begin Transaction都会引起@@TranCount加1。而每一次Commit Transaction都会使@@TranCount减1，而RollBack Transaction会回滚所有的嵌套事务包括已经提交的事务和未提交的事务，而使@@TranCount置0。  
Eg:  
 BEGIN TRAN --@@TRANCOUNT值为0   
 SELECT @@TRANCOUNT --值为1   
 BEGIN TRAN**

**SELECT @@TRANCOUNT --值为2   
  COMMIT TRAN   
  SELECT @@TRANCOUNT --值为1   
  ROLLBACK TRAN   
  SELECT @@TRANCOUNT --值为0   
 --又如  
  BEGIN TRAN --@@TRANCOUNT值为0   
  SELECT @@TRANCOUNT --值为1   
  SAVE TRAN t1   
 SELECT @@TRANCOUNT --值为1   
 BEGIN TRAN   
 SELECT @@TRANCOUNT --值为2   
 ROLLBACK TRAN t1   
 SELECT @@TRANCOUNT --注意这里的值为2   
  IF @@TRANCOUNT>0   
  ROLLBACK TRAN SELECT '处理结束', @@TRANCOUNT --为0   
    @@rowcount受影响的行数**

**在对数据表进行增删改查操作时，有一个系统变量@@rowcount 用来表达当前会话内，最后的一次操作对数据库造成了几行数据的影响。**

****

**XACT\_STATE 测试事务的状态**

**CATCH 块中的代码可以通过使用 XACT\_STATE 函数来测试事务的状态。如果会话中包含无法提交的事务，**

**XACT\_STATE 将返回 -1。如果 XACT\_STATE 返回 -1，则 CATCH 块将不能执行写日志的任何操作。**

**Xact\_state()有三种值：-1.事务不可提交，1.事务可提交；0.表示没有事务，此时commit或者rollback会报错**

**如果设置Set xact\_abort off在此不可以使用xact\_state来判断是否有不可提交的事务,只可以使用@@Trancount来判断是否有还未提交的事务，未提交的事务未必就是不可提交的事务，所以使用@@TranCount>0后就RollBack是不准确的.**

**这个容易和存储过程一起用，一个存储过程中可能有好几个事务，如果遇到错误，事务前的还是执行，错误事务本身回滚，但是存储过程不回滚。如果PB之类的调用存储过程，并且AUTOCOMMIT=TRUE的话，就还后上面分析的一样，如果AUTOCOMMIT=FALSE的话：**

**1、存储过程没有报错，COMMIT后还是全部执行**

**2、存储过程报错，ROLLBACK后存储过程所有的操作都回滚**

**--若发生约束违规时. SET XACT\_ABORT ON将使事务不可提交**

**SET XACT\_ABORT ON;**

**BEGIN TRY**

**BEGIN TRANSACTION;**

**--此表上存在一个FOREIGN KEY约束。**

**--这个语句会生成约束违规错误.**

**DELETE FROM Production.Product**

**WHERE ProductID = 980;**

**--如果删除操作成功，则提交事务。**

**- -CATCH块不会执行。**

**COMMIT TRANSACTION;**

**END TRY**

**BEGIN CATCH**

**- 为0,1或-1测试XACT\_STATE。**

**- 如果1，事务是可以提交的。**

**- 如果-1，事务有错误发生，应该 回滚。**

**- XACT\_STATE = 0意味着没有事务和**

**- 提交或回滚操作会产生错误。**

**- 测试事务不可提交**

**IF (XACT\_STATE()) = -1**

**BEGIN**

**PRINT 'The transaction is in an uncommittable state.' +**

**' Rolling back transaction.'**

**ROLLBACK TRANSACTION;**

**END;**

**IF (XACT\_STATE()) = 1**

**BEGIN**

**PRINT 'The transaction is committable.' +**

**' Committing transaction.'**

**COMMIT TRANSACTION;**

**END;**

**END CATCH;**