SQL 高级知识第 2 版

李岳 著

公众号:【SQL 数据库开发】出品

前言

《SQL 高级知识》系列自发布以来，给有基础的小伙伴进一步提升 SQL 技能，带来了一定的帮助，虽然已经发布了多篇高级知识系列，但是其中缺少一些示例讲解等内容。于是经过重新编写，形成了目前的《SQL 高级知识第二版》，SQL 的高级知识点太多，这里列举的一些高级知识点，都是平常使用频率较高的。小伙伴们可以在公众号的"阅读原文"的地方获取示例数据库 SQL_Road 脚本.sql，第二版中的所有截图和代码均出自此示例数据库。

第一章 临时表的用法

1.1 临时表定义

临时表与实体表类似，只是在使用过程中，临时表是存储在系统数据库 tempdb 中。当我们不再使用临时表的时候，临时表会自动删除。

1.2 临时表分类

临时表分为本地临时表和全局临时表，它们在名称、可见性以及可用性上有区别。

1.3 临时表的特性

对于临时表有如下几个特点：

- **本地临时表**

  就是用户在创建表的时候添加了"#"前缀的表，其特点是根据数据库连接独立。只有创建本地临时表的数据库连接有表的访问权限，其它连接不能访问该表；

不同的数据库连接中，创建的本地临时表虽然"名字"相同，但是这些表之间相互并不存在任何关系；在 SQLSERVER 中，通过特别的命名机制保证本地临时表在数据库连接上的独立性，意思是你可以在不同的连接里使用相同的本地临时表名称。

- **全局临时表**

  是用户在创建表的时候添加"**##**"前缀的表，其特点是所以数据库连接均可使用该全局临时表，当所有引用该临时表的数据库连接断开后自动删除。

  全局临时表相比本地临时表，命名上就需要注意了，与本地临时表不同的是，全局临时表名不能重复。

临时表利用了数据库临时表空间，由数据库系统自动进行维护，因此节省了物理表空间。并且**由于临时表空间一般利用虚拟内存，大大减少了硬盘的 I/O 次数，因此也提高了系统效率**。

临时表在事务完毕或会话完毕数据库会自动清空，不必记得用完后删除数据

1.4 本地临时表

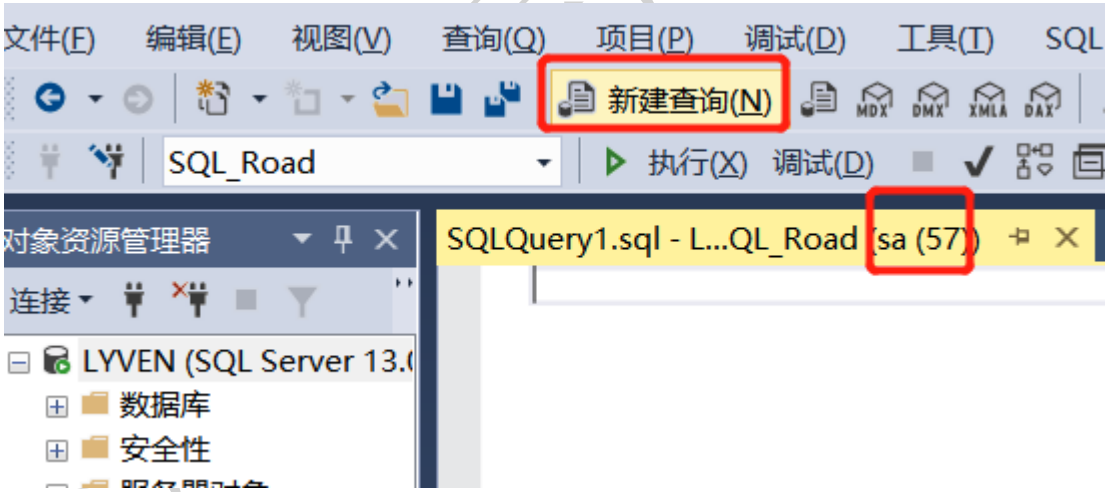
本地临时表的名称以单个数字符号"#" 打头；它们仅对当前的用户连接（也就是创建本地临时表的 connection）是可见的；当用户从 SQL Server 实例断开连接时被删除。

1.5 本地临时表实例

我们以 Customers 表为实例，表数据如下：

客户ID	姓名	地址	城市	邮编	省份
1	张三	北京路27号	上海	200000	上海市
2	李四	南京路12号	杭州	310000	浙江省
3	王五	花城大道17号	广州	510000	广东省
4	马六	江夏路19号	武汉	430000	湖北省
5	赵七	西二旗12号	北京	100000	北京市
6	宋一	花城大道21号	广州	510000	广东省
7	刘二	长安街121号	北京	100000	北京市

我们新建一个连接，每当"新建查询"就代表打开了一个连接，连接的 ID 就是 sa 后面的数字，我们的这个连接 ID 是 57.



下面我们在这个查询页面建立一个临时表。

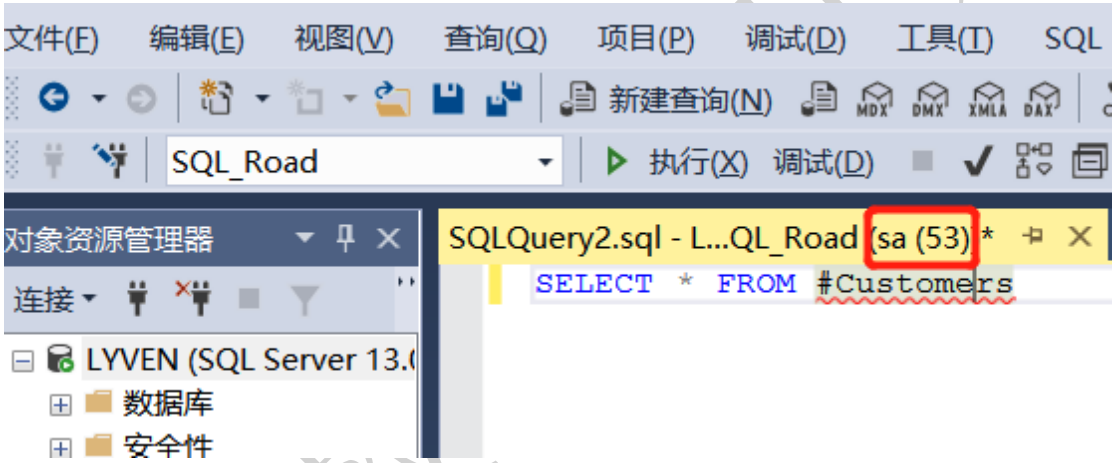
```
SELECT * INTO #Customers FROM Customers
```

这样我们就建好了一个临时表，可以查询一下临时表#Customers 的数据。与 Customers 内容一致。

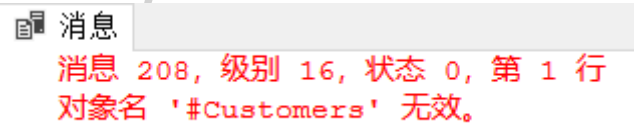
```
SELECT * FROM #Customers
```

客户ID	姓名	地址	城市	邮编	省份
1	张三	北京路27号	上海	200000	上海市
2	李四	南京路12号	杭州	310000	浙江省
3	王五	花城大道17号	广州	510000	广东省
4	马六	江夏路19号	武汉	430000	湖北省
5	赵七	西二旗12号	北京	100000	北京市
6	宋一	花城大道21号	广州	510000	广东省
7	刘二	长安街121号	北京	100000	北京市

如果我们再打开一个页面，同样查询#Customers 表会怎么样呢?



我们在新开的查询页面执行上述查询语句，得到的结果如下:



说明本地临时表不支持跨连接查询。只能在当前连接(或者当前查询页面)访问。

那本地临时表具体在什么地方呢？它又是怎么存放的呢?

这就是我们刚才建立的临时表，在系统中并不是用#Cusomters 表示的。

1.6 全局临时表

全局临时表的名称以两个数字符号 "##"打头，创建后对任何数据库连接都是可见的，当所有引用该表的数据库连接从 SQL Server 断开时被删除。

1.7 全局临时表实例

我们还是按照上面的步骤走一遍

先打开一个查询页面，输入如下查询语句：

```
SELECT * INTO ##Customers FROM Customers
```

执行完上面的查询语句后，我们关掉查询页面，再重新开一个页面查询

##Customers 中的内容

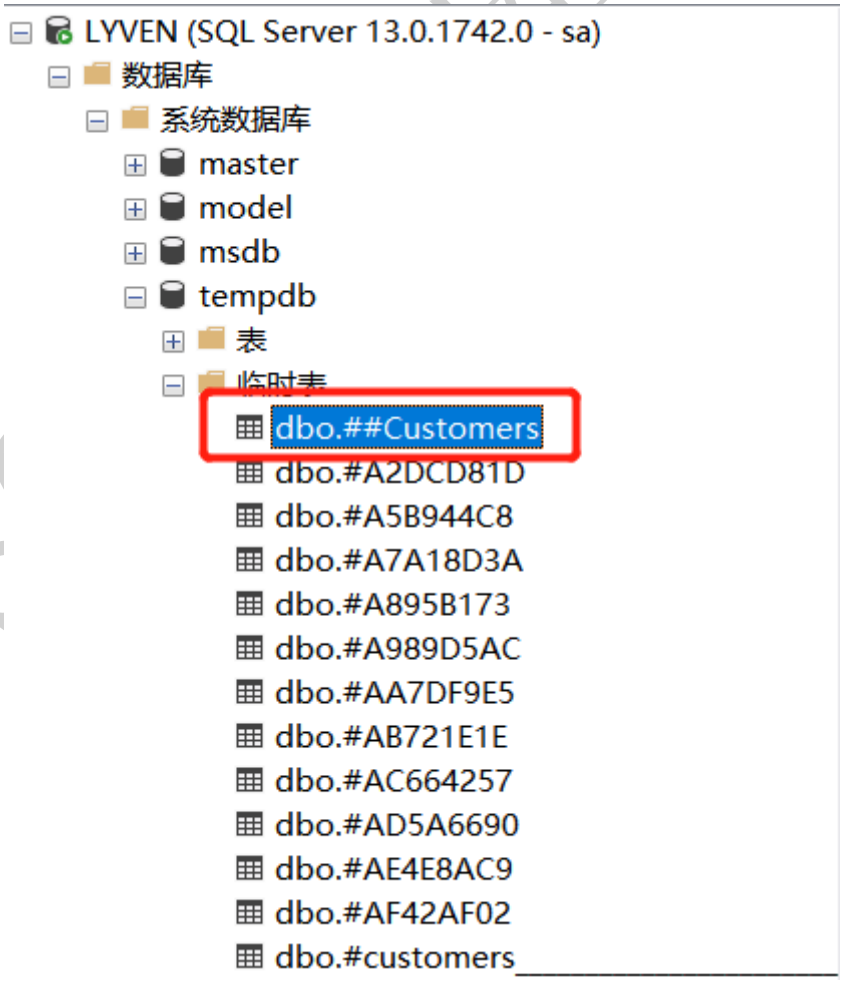
```
SELECT * FROM ##Customers
```

结果如下：

客户ID	姓名	地址	城市	邮编	省份
1	张三	北京路27号	上海	200000	上海市
2	李四	南京路12号	杭州	310000	浙江省
3	王五	花城大道17号	广州	510000	广东省
4	马六	江夏路19号	武汉	430000	湖北省
5	赵七	西二旗12号	北京	100000	北京市
6	宋一	花城大道21号	广州	510000	广东省
7	刘二	长安街121号	北京	100000	北京市

此时并不会像本地临时表那样报错了。

全局临时表的位置如下：



它的名称与我们自定义的名称一致，系统不会额外添加其他信息。

1.8 临时表的用途

介绍完临时表，我们来说说如何用它来进行优化

临时表的优化一般使用在子查询较多的情况下，也称为嵌套查询。我们写如下子查询：

```
SELECT * FROM sales.Temp_Salesorder
WHERE SalesOrderDetailID IN
(SELECT SalesOrderDetailID FROM sales.SalesOrderDetail
WHERE UnitPrice IN
(SELECT UnitPrice FROM sales.SalesOrderDetail WHERE UnitPrice>0)
)
```

这是一个比较简单的两层嵌套子查询，我们看一下执行情况：

```
SQL Server 分析和编译时间:
   CPU 时间 = 31 毫秒，占用时间 = 37 毫秒。

(122317 行受影响)
表 'Workfile'。扫描计数 0，逻辑读取 0 次，物理读取 0 次，预读 0 次，l
表 'Worktable'。扫描计数 0，逻辑读取 0 次，物理读取 0 次，预读 0 次，
表 'Temp_SalesOrder'。扫描计数 1，逻辑读取 23915 次，物理读取 0 次，
表 'SalesOrderDetail'。扫描计数 2，逻辑读取 2492 次，物理读取 0 次，
```

可以看到这里的逻辑读取是比较高的。

我们用临时表重新来看下执行情况如何，我们将第一二层的查询结果插入到#temp中，然后从临时表中查询结果。

```
SELECT  SalesOrderDetailID  INTO  #temp  FROM  sales.SalesOrderDetail
WHERE  UnitPrice  IN  (SELECT  UnitPrice  FROM  sales.SalesOrderDetail
  WHERE  UnitPrice>0)

SELECT  *  FROM  sales.Temp_Salesorder
WHERE  SalesOrderDetailID  IN
(SELECT  SalesOrderDetailID  FROM  #temp)
```

执行情况如下：

```
(122317 行受影响)
表 'Workfile'。扫描计数 0，逻辑读取 0 次，物理读取 0 次
表 'Worktable'。扫描计数 0，逻辑读取 0 次，物理读取 0
表 'Temp_SalesOrder'。扫描计数 1，逻辑读取 23915 次，
表 '#temp_____________________________________________

读 0 次。
预读 0 次。
0 次，lob 预读 0 次。
_____________________00000000009F'。扫描计数 1，逻辑读取 196
```

相比上一次的逻辑读，成倍的减少了逻辑读取次数。在对查询的性能进行调节时，如果逻辑读值下降，就表明查询使用的服务器资源减少，查询的性能有所提高。如果逻辑读值增加，则表示调节措施降低了查询的性能。在其他条件不变的情况下，一个查询使用的逻辑读越少，其效率就越高，查询的速度就越快。

因此我们可以看出临时表在比较复杂的嵌套查询中是可以提高查询效率的。

1.9 批注

临时表不管是在 SQL Server 还是其他平台都有使用，其在查询优化方面可以极大的提高查询效率，而 SQL Server 平台的临时表相比其他平台更容易创建和使用，

其优越性不言而喻。所以如果平时工作或学习过程中，临时表可以作为一个必备技能经常使用。

第二章 CASE WHEN 的用法

2.1 CASE 函数的类型

CASE 具有两种格式，简单 CASE 函数和 CASE 搜索函数。这两种方式，大部分情况下可以实现相同的功能。

2.2 简单 CASE 函数

语法

```
CASE column
WHEN <condition> THEN value
WHEN <condition> THEN value
.....
ELSE value END
```

示例

```
CASE  sex
     WHEN  '1'  THEN  '男'
          WHEN  '2'  THEN  '女'
ELSE  '其他'  END
```

2.3 CASE 搜索函数

语法

CASE

WHEN <condition> [,<condition>] THEN value

WHEN <condition> [,<condition>] THEN value

.....

ELSE value END

示例

```
CASE  WHEN sex = '1'  THEN '男'
        WHEN sex = '2'  THEN '女'
ELSE '其他'  END
```

简单 CASE 函数重在简洁，但是它只适用于这种单字段的单值比较，而 CASE 搜索函数的优点在于适用于所有比较(包括多值比较)的情况。

例如

```
CASE  WHEN  sex = '1'  AND  age>18  THEN '成年男性'
        WHEN  sex = '2'  AND  age>18  THEN '成年女性'
ELSE '其他'  END
```

注意：CASE 函数只返回第一个符合条件的值，剩下的 CASE 部分将会被自动忽略。比如说，下面这段 SQL，你永远无法得到“第二类”这个结果

```
CASE  WHEN Type IN ('a','b') THEN '第一类'
        WHEN Type IN ('a')       THEN '第二类'
ELSE '其他类'  END
```

2.4 CASE 行转列

CASE 用的比较广泛的功能就是行转列，就是将记录行里的数据按条件转换成具体的列。看如下的一个示例：

```
IF OBJECT_ID('Score') IS NOT NULL  DROP  TABLE  Score
GO
CREATE  TABLE  Score(姓名  NVARCHAR(10),课程  NVARCHAR(10),分数  INT)
INSERT  INTO  Score  VALUES  (N'张三',N'语文',74)
INSERT  INTO  Score  VALUES  (N'张三',N'数学',83)
```

```
INSERT INTO Score VALUES (N'张三',N'物理',93)
INSERT INTO Score VALUES (N'李四',N'语文',74)
INSERT INTO Score VALUES (N'李四',N'数学',84)
INSERT INTO Score VALUES (N'李四',N'物理',94)
GO
SELECT * FROM Score
GO
```

```
IF OBJECT_ID('Score') IS NOT NULL DROP TABLE Score
GO
CREATE TABLE Score(姓名 NVARCHAR(10),课程 NVARCHAR(10),分数 INT)
INSERT INTO Score VALUES (N'张三',N'语文',74)
INSERT INTO Score VALUES (N'张三',N'数学',83)
INSERT INTO Score VALUES (N'张三',N'物理',93)
INSERT INTO Score VALUES (N'李四',N'语文',74)
INSERT INTO Score VALUES (N'李四',N'数学',84)
INSERT INTO Score VALUES (N'李四',N'物理',94)
GO
SELECT * FROM Score
GO
```

执行完成后的结果如图：

姓名	课程	分数
张三	语文	74
张三	数学	83
张三	物理	93
李四	语文	74
李四	数学	84
李四	物理	94

现在我们想实现这样的功能，就是将各学科作为单独的列来显示各个学生各科的成绩。我们可以对课程里的记录做如下的行列转换：

```
SELECT 姓名,
MAX(CASE 课程 WHEN N'语文' THEN 分数 ELSE 0 END) 语文,
MAX(CASE 课程 WHEN N'数学' THEN 分数 ELSE 0 END) 数学,
```

```
MAX(CASE 课程 WHEN N'物理' THEN 分数 ELSE 0 END) 物理
FROM Score
GROUP BY 姓名
```

执行结果如下：

姓名	语文	数学	物理
李四	74	84	94
张三	74	83	93

2.5 行转列新方法

这样就很好的完成了行列的转换了，当然这只是一个比较简单的例子，SQL Server 2005 版之后有单独的行列转换功能 **PIOVT**，以下查询同样可以得到上面的结果：

```
SELECT * FROM Score
PIVOT( MAX(分数) FOR 课程 IN (语文,数学,物理)) A
```

其中 FOR 后面的是我们即将进行行转列的列部分

IN 里面的是我们行转列之后的列

MAX 是聚合 IN 里面的内容，也可以是其他聚合函数：SUM，MIN，COUNT 等

PIVOT 写法比较固定，是 CASE WHEN 的一种简略写法。

2.6 批注

CASE 是我们在日常工作中使用非常频繁的一个功能，可以很好的将我们需要的数据单独的显示在一列里面，有助于对数据有个比较清晰的掌握。与 Excel 的转置有点类似，但是其功能的多样性又比 Excel 更强一点。

第三章 派生表

3.1 派生表的定义

派生表是在外部查询的 FROM 子句中定义的，只要外部查询一结束，派生表也就不存在了。

3.2 派生表的作用

派生表可以简化查询，避免使用临时表。相比手动生成临时表性能更优越。派生表与其他表一样出现在查询的 FROM 子句中。

例如：

```
SELECT * FROM (
SELECT * FROM Customers WHERE 城市='广州'
) Cus
```

其中 Cus 就是派生表

3.3 派生表的特征

- 所有列必须要有名称，出现无列名的要重命名
- 列名称必须是要唯一，相同名称肯定是不允许的
- 不允许使用 ORDER BY(除非指定了 TOP)

注意：派生表是一张虚表，在数据库中并不存在，是我们自己创建的，目的主要是为了缩小数据的查找范围，提高查询效率。

3.4 派生表嵌套

如果需要用一个本身就引用了某个派生表的查询，去定义另一个派生表，最终得到的就是嵌套派生表。

例子：查询每年处理客户数超过 70 的订单年度和每年所处理的客户数量。

方法一：不使用派生表

```
SELECT
YEAR(orderdate) AS Orderyear,
COUNT(DISTINCT custid) AS Numcusts
FROM Sales.Orders
GROUP BY YEAR(Orderdate)
HAVING COUNT(DISTINCT Custid) > 70;
```

方法二：使用派生表

```
SELECT Orderyear, Numcusts
FROM (
SELECT Orderyear, COUNT(DISTINCT Custid) AS Numcusts
FROM (
SELECT YEAR(Orderyear) AS Orderyear, Custid
FROM Sales.Orders) AS D1
GROUP BY Orderyear
) AS D2
WHERE Numcusts > 70;
```

嵌套查询看起来非常复杂，嵌套查询也是很容易产生问题的一个方面。在这个例子中，使用嵌套派生表的目的是为了重用列别名。但是，由于嵌套增加了代码的复杂性，所以对于本例考虑使用方案一。

3.5 批注

派生表在我们日常查询中经常使用到，用法也比较灵活，主要是执行效率会相对较高，因为它可以提前缩小查询范围。但是也不可乱用，特别是在不需要使用派生表而强制使用，效果反而会适得其反。

第四章 DBLINK

4.1 DBLINK 的定义

当我们要跨本地数据库，访问另外一个数据库表中的数据时，本地数据库中就必须要创建远程数据库的 DBLINK,通过 DBLINK 本地数据库可以像访问本地数据库一样访问远程数据库表中的数据。

4.2 创建 DBLINK 的语法

定义 DBLINK 类型

```
EXEC master.dbo.sp_addlinkedserver
@server = '远程 IP 地址',
@srvproduct='DBLINK 类型(默认 SQL Server)'
```

定义 DBLINK 连接属性

```
EXEC master.dbo.sp_addlinkedsrvlogin
@rmtsrvname='远程 IP 地址',
@useself='False',
@locallogin=NULL,
@rmtuser='远程数据库用户名',
@rmtpassword='远程数据库密码'
```

以上两步要一起执行才能生成 DBLINK 连接。

4.3 创建 DBLINK 连接示例

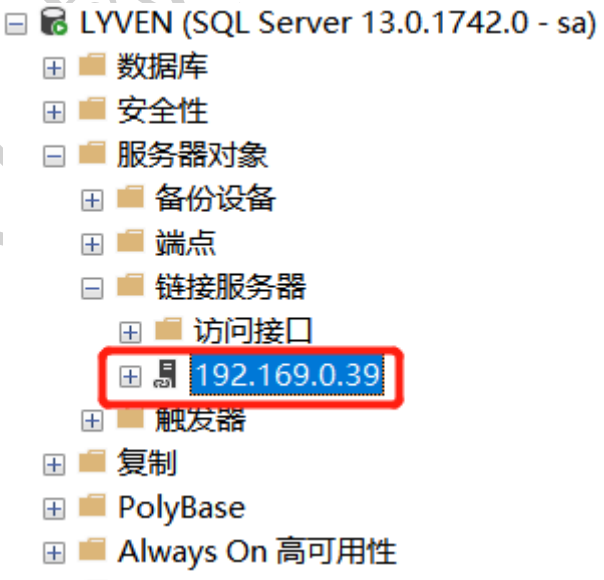
本地数据库 IP 地址是 192.168.0.35，已知局域网有一台 IP 地址为 192.169.0.39 的数据库服务器，其账户和密码分别是 sa 和!QAZ1234，那么我们应该这样创建 DBLINK 连接：

```
USE  master
GO

EXEC master.dbo.sp_addlinkedserver
@server  =  '192.168.0.39',
@srvproduct='SQL Server'

EXEC master.dbo.sp_addlinkedsrvlogin
@rmtsrvname='192.168.0.39',
@useself='False',
@locallogin=NULL,
@rmtuser='sa',
@rmtpassword='!QAZ1234'
Go
```

执行完后我们会看到在 SSMS 的服务器对象下面有一个创建好的 DBLINK 连接，如下图：



4.4 DBLINK 的作用

前面的定义已经说明，通过 DBLINK 本地数据库可以像访问本地数据库一样访问远程数据库表中的数据。

DBLINK 示例

以本地 Customers 表和远程数据库 192.168.0.39 里 SQL_Road 数据库下的 Orders 表为例

客户ID	姓名	地址	城市	邮编	省份
1	张三	北京路27号	上海	200000	上海市
2	李四	南京路12号	杭州	310000	浙江省
3	王五	花城大道17号	广州	510000	广东省
4	马六	江夏路19号	武汉	430000	湖北省
5	赵七	西二旗12号	北京	100000	北京市
6	宋一	花城大道21号	广州	510000	广东省
7	刘二	长安街121号	北京	100000	北京市

Customers 表

订单ID	客户ID	员工ID	订单日期	发货ID
1	3	9	2018-09-21 09:18:34.000	3
2	4	9	2018-06-28 23:12:15.000	5
3	6	3	2018-09-21 16:56:34.000	3
4	3	7	2018-09-28 11:24:54.000	4
5	1	4	2018-09-30 14:46:21.000	4

远程数据库中的 Orders 表

我们想用本地的 Customers 表关联远程数据库 192.168.0.39 里 SQL_Road 数据库下的 Orders 表里的数据，可以这样写 SQL：

```
SELECT  c.姓名,o.订单日期  FROM  Customers c
JOIN  [192.168.0.38].SQL_Road.dbo.Orders o  ON  c.客户 ID=o.客户 ID
```

结果如下：

姓名	订单日期
王五	2018-09-21 09:18:34.000
马六	2018-06-28 23:12:15.000
宋一	2018-09-21 16:56:34.000
王五	2018-09-28 11:24:54.000
张三	2018-09-30 14:46:21.000

这样我们就将本来隔绝的两个表通过 DBLINK 关联上了。

4.5 删除 DBLINK

当我们不需要 DBLINK 的时候，可以通过以下方式进行删除

```
EXEC  master.dbo.sp_dropserver
@server='192.169.0.39',
@droplogins='droplogins'
```

这样就将刚创建的 DBLINK 删除了。

4.6 批注

DBLINK 是我们日常查询管理经常要使用到的一个利器，可以很方便的将原本隔开的两个数据库建立起连接。为我们跨库查询提供一个非常便捷的方法。

第五章 集合

5.1 集合的定义

集合是由一个和多个元素构成的整体，在 SQL Server 中的表就代表着事实集合，而其中的查询就是在集合的基础上生成的结果集。SQL Server 的集合包括交集（INTERSECT），并集（UNION），差集（EXCEPT）。

5.2 交集 INTERSECT

可以对两个或多个结果集进行连接，形成“交集”。返回左边结果集和右边结果集中都有的记录，且结果不重复(这也是集合的主要特性)

交集限制条件

- 子结果集要具有相同的结构。
- 子结果集的列数必须相同
- 子结果集对应的数据类型必须可以兼容。
- 每个子结果集不能包含 order by 和 compute 子句。

交集示例

我们用以下两个表中的数据作为示例

Cno	Name
1	广州
2	深圳
3	珠海
4	北京
5	上海

City1

Cno	Name
1	南京
2	无锡
3	苏州
4	北京
5	上海

City2

取以上两个表的交集，我们可以这样写 SQL

```
SELECT * FROM City1
INTERSECT
SELECT * FROM City2
```

结果如下：

Cno	Name
4	北京
5	上海

其中北京和上海是上面两个表共有的结果集。

这和我们的内连接(INNER JOIN)有点类似，以上 SQL 也可以这样写

```
SELECT c1.* FROM City1 c1
INNER JOIN City2 c2
ON c1.Cno=c2.Cno AND c1.Name=c2.Name
```

结果与上面结果相同。

5.3 并集 UNION

可以对两个或多个结果集进行连接，形成“并集”。子结果集所有的记录组合在一起形成新的结果集。其中使用 UNION 可以得到不重复（去重）的结果集，使用 UNION ALL 可能会得到重复（不去重）的结果集。

并集限制条件

- 子结果集要具有相同的结构。
- 子结果集的列数必须相同
- 子结果集对应的数据类型必须可以兼容。
- 每个子结果集不能包含 order by 和 compute 子句。

UNION 示例

还是以上面的 City1 和 City2 为例，取两个表的并集，我们可以这样写 SQL：

```
SELECT * FROM City1
UNION
SELECT * FROM City2
```

结果如下：

Cno	Name
1	广州
1	南京
2	深圳
2	无锡
3	苏州
3	珠海
4	北京
5	上海

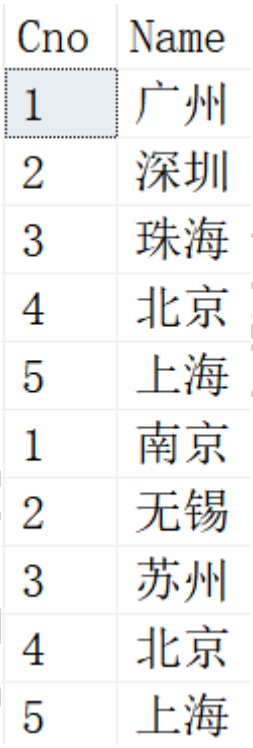
我们看到，北京和上海去掉了重复的记录，只保留了一次

UNION ALL 示例

我们再看看使用 UNION ALL 会怎么样?

```
SELECT * FROM City1
UNION ALL
SELECT * FROM City2
```

结果如下:



Cno	Name
1	广州
2	深圳
3	珠海
4	北京
5	上海
1	南京
2	无锡
3	苏州
4	北京
5	上海

与上面的 UNION 相比，UNION ALL 仅仅是对两个表作了拼接而已，北京和上海依然在下面重复出现了，而且细心的读着应该发现了，UNION 还会对结果进行排序，而 UNION ALL 不会。

5.4 差集 EXCEPT

可以对两个或多个结果集进行连接，形成“差集”。返回左边结果集合中已经有的记录，而右边结果集中没有的记录。

差集限制条件

- 子结果集要具有相同的结构。
- 子结果集的列数必须相同
- 子结果集对应的数据类型必须可以兼容。
- 每个子结果集不能包含 order by 和 compute 子句。

差集示例

以 City1 和 City2 为例，我们想取 City1（左表）和 City2（右表）的差集，可以这样写 SQL：

```
SELECT * FROM City1
EXCEPT
SELECT * FROM City2
```

结果如下：

Cno	Name
1	广州
2	深圳
3	珠海

我们看到，因为北京和上海在两个表都存在，差集为了只显示左表中有的，而右表中没有的，就把这两个给过滤掉了。

此外我们常说的关联条件其实也是集合的一种，是通过子表的笛卡尔积按不同的关联条件过滤之后得到的结果集。有兴趣的同学可以阅读一下《Microsoft SQL SERVER 2008 技术内幕 T-SQL 查询》，这本书中有关于集合论的具体阐述。

5.5 批注

集合是我们数据处理过程中的理论基础，可以通过集合的观点去很好的理解不同的查询语句。每一个物理表就是一个集合，当我们要对表进行操作的时候，将它们看成对集合的操作就很好理解了。

第六章　分组集

6.1 分组集的定义

是多个分组的并集，用于在一个查询中，按照不同的分组列对集合进行聚合运算，等价于对单个分组使用"UNION ALL"，计算多个结果集的并集。

6.2 分组集种类

SQL Server 的分组集共有三种 GROUPING SETS, CUBE, 以及 ROLLUP， 其中 CUBE 和 ROLLUP 可以当做是 GROUPING SETS 的简写版

6.3 GROUPING SETS

GROUPING SETS 子句允许你指定多个 GROUP BY 选项。增强了 GROUP BY 的功能。

可以通过一条 SELECT 语句实现复杂繁琐的多条 SELECT 语句的查询。并且更加的高效，解析存储一条 SQL 于语句

GROUP SETS 示例

我们以 Customers 表为例，其内容如下：

客户ID	姓名	地址	城市	邮编	省份
1	张三	北京路27号	上海	200000	上海市
2	李四	南京路12号	杭州	310000	浙江省
3	王五	花城大道17号	广州	510000	广东省
4	马六	江夏路19号	武汉	430000	湖北省
5	赵七	西二旗12号	北京	100000	北京市
6	宋一	花城大道21号	广州	518000	广东省
7	刘二	长安街121号	北京	100000	北京市

我们先分别对城市和省份进行分组，统计出他们的数量

```
SELECT  城市,NULL  省份,COUNT(城市)  FROM  Customers
GROUP  BY  城市
UNION  ALL
SELECT  NULL,省份,COUNT(省份)  FROM  Customers
GROUP  BY  省份
```

结果为：

城市	省份	数量
北京	NULL	2
广州	NULL	2
杭州	NULL	1
上海	NULL	1
武汉	NULL	1
NULL	北京市	2
NULL	广东省	2
NULL	湖北省	1
NULL	上海市	1
NULL	浙江省	1

再使用 GROUPING SETS 来统计

```
SELECT
城市,
省份,
COUNT(客户 ID) 数量
FROM  Customers
GROUP  BY  GROUPING  SETS  (城市,省份)
```

结果如下

城市	省份	数量
NULL	北京市	2
NULL	广东省	2
NULL	湖北省	1
NULL	上海市	1
NULL	浙江省	1
北京	NULL	2
广州	NULL	2
杭州	NULL	1
上海	NULL	1
武汉	NULL	1

其实上下两个结果是一样的，只是 UNION ALL 不排序，而 GROUPING SETS 增加了排序。这样不仅减少了代码，而且这样的效率会比 UNION ALL 的效率高。通常 GROUPING SETS 使用在组合分析中。

6.4 ROLLUP

ROLLUP 也是 GROUPING SETS 的一种简略写法，我们举例说明。

我们先使用 GROUPING SETS 的多层组合

```
SELECT
省份,
城市,
COUNT(1) 数量
FROM Customers
GROUP BY GROUPING SETS (
省份,(省份,城市)
)
```

其结果为：

省份	城市	数量
北京市	北京	2
北京市	NULL	2
广东省	广州	2
广东省	NULL	2
湖北省	武汉	1
湖北省	NULL	1
上海市	上海	1
上海市	NULL	1
浙江省	杭州	1
浙江省	NULL	1

我们使用 ROLLUP 可以这样写

```
SELECT
省份,
城市,
COUNT(客户 ID) 数量
FROM  Customers
GROUP  BY  省份,城市  WITH  ROLLUP
```

其结果为:

省份	城市	数量
北京市	北京	2
北京市	NULL	2
广东省	广州	2
广东省	NULL	2
湖北省	武汉	1
湖北省	NULL	1
上海市	上海	1
上海市	NULL	1
浙江省	杭州	1
浙江省	NULL	1
NULL	NULL	7

我们来解读一下 ROLLUP 的作用，其作用是对每个列先进行一次分组，并且对第一列的数据在每个组内还进行一次汇总，最后对所有的数据再进行一次汇总，所以相比 GROUPING SETS 会多了个所以数据的汇总。这个在对组内进行聚合时是经常使用到的。

6.5 CUBE

而 CUBE 相比 ROLLUP 就更多一个维度了，我们还是距离说明。

```
SELECT
省份,
城市,
COUNT(客户 ID) 数量
FROM  Customers
GROUP  BY  省份,城市  WITH  CUBE
```

结果如下：

省份	城市	数量
北京市	北京	2
NULL	北京	2
广东省	广州	2
NULL	广州	2
浙江省	杭州	1
NULL	杭州	1
上海市	上海	1
NULL	上海	1
湖北省	武汉	1
NULL	武汉	1
NULL	NULL	7
北京市	NULL	2
广东省	NULL	2
湖北省	NULL	1
上海市	NULL	1
浙江省	NULL	1

在 ROLLUP 的基础上，还会将第一列每组的汇总数据额外显示在最后。

6.6 批注

分组集类似于 Excel 的透视图，可以对各类数据进行组内计算，这里不止可以进行数量统计，也可以进行求和，最大最小值等操作。是我们在进行数据分析时候经常使用到的一组功能。

第七章 MERGE INTO

7.1 MERGE 的定义

MERGE 关键字是一个神奇的 DML 关键字，它能将 INSERT，UPDATE，DELETE 等操作并为一句，根据与源表联接的结果，对目标表执行插入、更新或删除操作。

7.2 MERGE 的语法

MERGE INTO target_table
USING source_table
ON condition
WHEN MATCHED THEN
operation
WHEN NOT MATCHED THEN
operation;

注意：其中最后语句分号不可以省略，且源表既可以是一个表也可以是一个子查询语句。

7.3 MERGE 的用法

merge 无法多次更新同一行，也无法更新和删除同一行

当源表和目标表不匹配时：

- 若数据是源表有目标表没有，则进行插入操作;
- 若数据是源表没有而目标表有，则进行更新或者删除数据操作

当源表和目标表匹配时：

- 进行更新操作或者删除操作

7.4 MERGE 的使用场景

- 数据同步
- 数据转换

- 基于源表对目标表做 INSERT,UPDATE,DELETE 操作

  我们常用的是第三种场景

7.5 MERGE 使用限制

- 在 MERGE MATCHED 操作中，只能允许执行 UPDATE 或者 DELETE 语句。
- 在 MERGE NOT MATCHED 操作中，只允许执行 INSERT 语句。
- 一个 MERGE 语句中出现的 MATCHED 操作，只能出现一次 UPDATE 或者 DELETE 语句，否则就会出现下面的错误：*An action of type 'WHEN MATCHED' cannot appear more than once in a 'UPDATE' clause of a MERGE statement.*

7.6 MERGE 示例

下面我们通过一个示例来介绍一下该如何使用 MERGE，我们以 Customers 表和 Orders 表为例。数据如下：

客户ID	姓名	地址	城市	邮编	省份
1	张三	北京路27号	上海	200000	上海市
2	李四	南京路12号	杭州	310000	浙江省
3	王五	花城大道17号	广州	510000	广东省
4	马六	江夏路19号	武汉	430000	湖北省
5	赵七	西二旗12号	北京	100000	北京市
6	宋一	花城大道21号	广州	518000	广东省
7	刘二	长安街121号	北京	100000	北京市

Customers

订单ID	客户ID	员工ID	订单日期	发货ID
1	3	9	2018-09-21 09:18:34.000	3
2	4	9	2018-06-28 23:12:15.000	5
3	6	3	2018-09-21 16:56:34.000	3
4	3	7	2018-09-28 11:24:54.000	4
5	1	4	2018-09-30 14:46:21.000	4

Orders

Q：当 Customers 表里的客户有购买商品，我们就更新一下他们的下单时间，将他们的下单时间往后推迟一小时，如果客户没有购买商品，那么我们就将这些客户的信息插入到订单表里。

根据上面的要求我们可以这样写 SQL：

```
MERGE  INTO  Orders O
--确定目标表 Orders
USING  Customers C  ON  C.客户 ID=O.客户 ID
--从源表 Customers 确定关联条件 C.客户 ID=O.客户 ID
WHEN  MATCHED
--当匹配时对目标表的订单日期执行更新操作
THEN  UPDATE  SET  O.订单日期=DATEADD(HOUR,1,O.订单日期)
WHEN  NOT  MATCHED  BY  TARGET
--当不匹配时对目标表进行插入操作
THEN  INSERT  (客户 ID,员工 ID,订单日期,发货 ID)
VALUES  (C.客户 ID,NULL,NULL,NULL)
;
```

我们看一下 Orders 表里的结果：

订单ID	客户ID	员工ID	订单日期			发货ID
1	3	9	2018-09-21	10	18:34.000	3
2	4	9	2018-06-29	00	12:15.000	5
3	6	3	2018-09-21	17	56:34.000	3
4	3	7	2018-09-28	12	24:54.000	4
5	1	4	2018-09-30	15	46:21.000	4
6	2	NULL	NULL			NULL
7	5	NULL	NULL			NULL
8	7	NULL	NULL			NULL

我们发现与 Customers 表里匹配上的订单日期被修改了，订单日期往后推迟了一小时，而没有匹配上的在订单表尾部增加了几行记录。这就是 MERGE 的实际应用了。

7.7 OUTPUT 子句

MERGE 还能与 OUTPUT 一起使用，可以将刚刚做过变动的数据进行输出，我们以上面的示例为基础，进行示范。

```
MERGE  INTO  Orders O
--确定目标表 Orders
USING  Customers C  ON  C.客户 ID=O.客户 ID
--从源表 Customers 确定关联条件 C.客户 ID=O.客户 ID
WHEN  MATCHED
--当匹配时对目标表的订单日期执行更新操作
THEN  UPDATE  SET  O.订单日期=DATEADD(HOUR,1,O.订单日期)
WHEN  NOT  MATCHED  BY  TARGET
--当不匹配时对目标表进行插入操作
THEN  INSERT  (客户 ID,员工 ID,订单日期,发货 ID)
VALUES  (C.客户 ID,NULL,NULL,NULL)
OUTPUT  $action  AS  [ACTION],Inserted.订单日期 ,
Inserted.客户 ID,Inserted.发货 ID,Inserted.员工 ID
--用 OUTPUT 输出刚刚变动过的数据
;
```

执行上述语句结果如下：

ACTION	订单日期	客户ID	发货ID	员工ID
UPDATE	2018-09-30 16:46:21.000	1	4	4
UPDATE	NULL	2	NULL	NULL
UPDATE	2018-09-21 11:18:34.000	3	3	9
UPDATE	2018-09-28 13:24:54.000	3	4	7
UPDATE	2018-06-29 01:12:15.000	4	5	9
UPDATE	NULL	5	NULL	NULL
UPDATE	2018-09-21 18:56:34.000	6	3	3
UPDATE	NULL	7	NULL	NULL

从上图我们看到，执行的动作都是更新，这里的动作只有 UPDATE 和 DELETE，插入也属于更新，此外我们看到订单日期又往后推迟了一小时，是因为我们又一次执行了往后增加一小时的更新操作，其他的字段没变。

7.8 批注

MERGE 功能比较丰富，以上我们只是简单介绍了一些常用功能，还有其他一些用法，有兴趣的可以搜索一下并动手尝试。在我们要对表做多种操作时，这种写法不仅可以节省代码，而且有时候还可以提高执行效率。

第八章 存储过程

8.1 存储过程的定义

存储过程其实就是已预编译为可执行过程的一个或多个 SQL 语句。通过调用和传递参数即可完成该存储过程的功能。

8.2 创建存储过程语法

CREATE PROC | PROCEDURE procedure_name

[{@参数数据类型} [=默认值] [OUTPUT],

{@参数数据类型} [=默认值] [OUTPUT],

....

]

AS

sql_statements

GO

简单示例

```
CREATE  PROC sp_test
@param1  INT,
@param2  VARCHAR(16)
AS
SELECT  *  FROM  test
WHERE  id=@param1
AND  t_no=@param2;
GO
```

上面就是一个简单的示例。

注意：存储过程在创建阶段可以带参数或不带参数，不带参数的一般是执行一些不需要传递参数的语句就可以完成的功能，带参数那就是需要传递参数的 SQL 语句，就像上面的示例，传递了两个参数给 SQL 语句。带参数的一定要定义参数类型，是字符型的还要定义长度，给参数加默认值是可选的。

8.3 存储过程的优点

提高性能

SQL 语句在创建过程时进行分析和编译。存储过程是预编译的，在首次运行一个存储过程时，查询优化器对其进行分析、优化，并给出最终被存在系统表中的存储计划，这样，在执行过程时便可节省此开销。

降低网络开销

存储过程调用时只需用提供存储过程名和必要的参数信息，从而可降低网络的流量。

便于进行代码移植

数据库专业人员可以随时对存储过程进行修改，但对应用程序源代码却毫无影响，从而极大的提高了程序的可移植性。

更强的安全性

- 系统管理员可以对执行的某一个存储过程进行权限限制，避免非授权用户对数据的访问
- 在通过网络调用过程时，只有对执行过程的调用是可见的。因此，恶意用户无法看到表和数据库对象名称、嵌入自己的 Transact-SQL 语句或搜索关键数据。
- 使用过程参数有助于避免 SQL 注入攻击。因为参数输入被视作文字值而非可执行代码，所以，攻击者将命令插入过程内的 Transact-SQL 语句并损害安全性将更为困难。
- 可以对过程进行加密，这有助于对源代码进行模糊处理。

8.4 存储过程的缺点

逻辑处理吃力

SQL 本身是一种结构化查询语言，但不是面向对象的的，本质上还是过程化的语言，面对复杂的业务逻辑，过程化的处理会很吃力。同时 SQL 擅长的是数据查询而非业务逻辑的处理，如果如果把业务逻辑全放在存储过程里面，违背了这一原则。

修改参数复杂

如果需要对输入存储过程的参数进行更改，或者要更改由其返回的数据，则您仍需要更新程序集中的代码以添加参数、更新调用，等等，这时候估计会比较繁琐了。

开发调试复杂

由于 IDE 的问题，存储过程的开发调试要比一般程序困难。

无法应用缓存

虽然有全局临时表之类的方法可以做缓存，但同样加重了数据库的负担。如果缓存并发严重，经常要加锁，那效率实在堪忧。

不支持群集

数据库服务器无法水平扩展，或者数据库的切割（水平或垂直切割）。数据库切割之后，存储过程并不清楚数据存储在哪个数据库中。

8.5 创建不带参数的存储过程

示例：查询订单表中订单总数

```
--查询存储过程
IF OBJECT_ID (N'PROC_ORDER_COUNT', N'P') IS NOT NULL
    DROP  PROCEDURE    PROC_ORDER_COUNT;
GO
CREATE  PROCEDURE    PROC_ORDER_COUNT
AS
        SELECT  COUNT(OrderID)  FROM  Orders;
GO
```

执行上述存储过程：

```
EXEC  PROC_ORDER_COUNT;
```

8.6 创建带参数的存储过程

示例：根据城市查询订单数量

```
--查询存储过程，根据城市查询总数
IF OBJECT_ID (N'PROC_ORDER_COUNT', N'P') IS NOT NULL
    DROP  PROCEDURE    PROC_ORDER_COUNT;
GO
CREATE  PROCEDURE    PROC_ORDER_COUNT(@city  NVARCHAR(50))
AS
        SELECT  COUNT(OrderID)  FROM  Orders  WHERE  City=@city
GO
```

执行上述存储过程：

```
EXEC  PROC_ORDER_COUNT N'广州';
```

8.7 参数带通配符

```
--查询订单编号头两位是LJ的订单信息，含通配符
IF OBJECT_ID (N'PROC_ORDER_INFO', N'P') IS NOT NULL
    DROP  PROCEDURE    PROC_ORDER_INFO;
GO
CREATE  PROCEDURE    PROC_ORDER_INFO
      @OrderID  NVARCHAR(50)='LJ%'   --默认值
AS
        SELECT  OrderID,City,OrderDate,Price  FROM  Orders
        WHERE  OrderID  LIKE  @OrderID;
GO
```

执行上述存储过程：

```
EXEC  PROC_ORDER_INFO;
EXEC  PROC_ORDER_INFO N'LJ%';
EXEC  PROC_ORDER_INFO N'%LJ%';
```

8.8 带输出参数

```
--根据订单查询的信息，返回订单的城市及单价
IF OBJECT_ID (N'PROC_ORDER_INFO ', N'P') IS NOT NULL
        DROP  PROCEDURE    PROC_ORDER_INFO ;
```

```
GO
CREATE  PROCEDURE    PROC_ORDER_INFO
        @orderid  NVARCHAR(50),   --输入参数
        @city  NVARCHAR(20)  OUT,   --输出参数
        @price  FLOAT  OUTPUT                --输入输出参数
AS
        SELECT  @city=City,@price=Price  FROM  Orders
        WHERE  OrderID=@orderid  AND  Price=@price;
GO
```

执行上述存储过程：

```
DECLARE  @orderid  NVARCHAR(50),
        @city  NVARCHAR(20),
        @price  INT;
SET  @orderid= N'LJ0001';
SET  @price =  35.21;
EXEC PROC_ORDER_INFO @orderid,@city OUT, @price OUTPUT;
SELECT  @city, @price;
```

8.9 存储过程中的插入

```
--新增订单信息
IF OBJECT_ID (N'PROC_INSERT_ORDER', N'P') IS NOT NULL
    DROP  PROCEDURE    PROC_INSERT_ORDER;
GO
CREATE  PROCEDURE    PROC_INSERT_ORDER
    @orderid  NVARCHAR(50),
    @city  NVARCHAR(20),
    @price  FLOAT
AS
        INSERT  INTO  Orders(OrderID,City,Price)
        VALUES(@orderid,@city,@price)
GO
```

执行上述存储过程：

```
EXEC  PROC_INSERT_ORDER N'LJ0001',N'广州',35.21;
```

8.10 存储过程中的更新

```
--修改订单信息
IF OBJECT_ID (N'PROC_UPDATE_ORDER', N'P') IS NOT NULL
    DROP  PROCEDURE    PROC_UPDATE_ORDER;
```

```
GO
CREATE  PROCEDURE    PROC_UPDATE_ORDER
    @orderid  NVARCHAR(50),
    @city  NVARCHAR(20),
    @price  FLOAT
AS
        UPDATE  Orders  SET  OrderID=@orderid,City=@city,Price=@price;
GO
```

执行上述存储过程：

```
EXEC PROC_UPDATE_ORDER N'LJ0001',N'上海',37.21;
```

8.11 存储过程中的删除

```
--删除订单信息
IF OBJECT_ID (N'PROC_DELETE_ORDER', N'P') IS NOT NULL
    DROP  PROCEDURE    PROC_DELETE_ORDER;
GO
CREATE  PROCEDURE    PROC_DELETE_ORDER
    @orderid  NVARCHAR(50),
 AS
        DELETE    FROM  Orders  WHERE  OrderID=@orderid;
GO
```

执行上述存储过程：

```
EXEC PROC_DELETE_ORDER N'LJ0001';
```

8.12 重复编译存储过程

重复编译存储过程的目的是为了提高存储过程的执行效率。

```
--重复编译
IF OBJECT_ID (N'PROC_ORDER_WITH_RECOMPILE', N'P') IS NOT NULL
    DROP  PROCEDURE    PROC_ORDER_WITH_RECOMPILE;
GO
CREATE  PROCEDURE    PROC_ORDER_WITH_RECOMPILE
WITH  RECOMPILE   --重复编译
AS
        SELECT  *  FROM  Orders;
GO
```

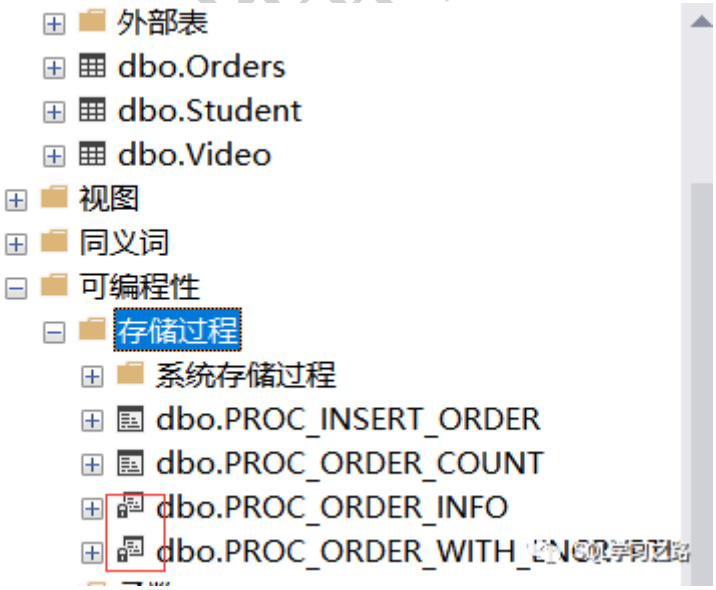
8.13 加密存储过程

```
--查询存储过程，进行加密，加密后不能查看和修改源脚本
IF OBJECT_ID (N'PROC_ORDER_WITH_ENCRYPTION', N'P') IS NOT NULL
    DROP  PROCEDURE    PROC_ORDER_WITH_ENCRYPTION;
GO
CREATE  PROCEDURE    PROC_ORDER_WITH_ENCRYPTION
WITH  ENCRYPTION  --加密
AS
       SELECT  *  FROM  Orders;
GO
```

执行上述存储过程：

```
EXEC  PROC_ORDER_WITH_ENCRYPTION
```

执行完的效果如图：



8.14 批注

存储过程在业务量小的系统中应用比较广泛，将逻辑全部交给数据库进行处理，可以快速完成一些逻辑功能。但是针对业务量比较大的系统，一般是禁止使用存储过程的，因为对数据库的压力太大。

第九章　开窗函数 OVER

9.1 OVER 的定义

OVER 用于为行定义一个窗口，它对一组值进行操作，不需要使用 GROUP BY 子句对数据进行分组，能够在同一行中同时返回基础行的列和聚合列。

9.2 OVER 的语法

OVER ([PARTITION BY column] [ORDER BY culumn])

PARTITION BY 子句进行分组;

ORDER BY 子句进行排序。

窗口函数 OVER()指定一组行，开窗函数计算从窗口函数输出的结果集中各行的值。开窗函数不需要使用 GROUP BY 就可以对数据进行分组，还可以同时返回基础行的列和聚合列。

9.3 OVER 的用法

OVER 开窗函数必须与聚合函数或排序函数一起使用，聚合函数一般指 SUM(),MAX(),MIN,COUNT(),AVG() 等常见函数。排序函数一般指 RANK(),ROW_NUMBER(),DENSE_RANK(),NTILE()等。

9.4 OVER 在聚合函数中使用的示例

我们以 SUM 和 COUNT 函数作为示例来给大家演示。

```
--建立测试表和测试数据
CREATE  TABLE  Employee
(
ID  INT    PRIMARY  KEY,
Name  VARCHAR(20),
GroupName  VARCHAR(20),
Salary  INT
)
INSERT  INTO    Employee
VALUES(1,'小明','开发部',8000),
           (4,'小张','开发部',7600),
           (5,'小白','开发部',7000),
           (8,'小王','财务部',5000),
           (9,  null,'财务部',NULL),
           (15,'小刘','财务部',6000),
           (16,'小高','行政部',4500),
           (18,'小王','行政部',4000),
           (23,'小李','行政部',4500),
           (29,'小吴','行政部',4700);
```

9.5 SUM 后的开窗函数

```
SELECT  *,
         SUM(Salary)  OVER(PARTITION  BY  Groupname) 每个组的总工资,
         SUM(Salary)  OVER(PARTITION  BY  groupname  ORDER  BY  ID) 每个组的累计总
工资,
         SUM(Salary)  OVER(ORDER  BY  ID) 累计工资,
         SUM(Salary)  OVER() 总工资
from  Employee
```

结果如下：

ID	Name	GroupName	Salary	每个组总工资	每个组的累计总工资	累计工资	总工资
1	小明	开发部	8000	22600	8000	8000	51300
4	小张	开发部	7600	22600	15600	15600	51300
5	小白	开发部	7000	22600	22600	22600	51300
8	小王	财务部	5000	11000	5000	27600	51300
9	NULL	财务部	NULL	11000	5000	27600	51300
15	小刘	财务部	6000	11000	11000	33600	51300
16	小高	行政部	4500	17700	4500	38100	51300
18	小王	行政部	4000	17700	8500	42100	51300
23	小李	行政部	4500	17700	13000	46600	51300
29	小吴	行政部	4700	17700	17700	51300	51300

其中开窗函数的每个含义不同，我们来具体解读一下：

SUM(Salary) OVER (PARTITION BY Groupname)

只对 PARTITION BY 后面的列 Groupname 进行分组，分组后求解 Salary 的和。

SUM(Salary) OVER (PARTITION BY Groupname ORDER BY ID)

对 PARTITION BY 后面的列 Groupname 进行分组，然后按 ORDER BY 后的 ID 进行排序，然后在组内对 Salary 进行累加处理。

SUM(Salary) OVER (ORDER BY ID)

只对 ORDER BY 后的 ID 内容进行排序，对排完序后的 Salary 进行累加处理。

SUM(Salary) OVER ()

对 Salary 进行汇总处理

9.6 COUNT 后的开窗函数

```
SELECT  *,
                COUNT(*)  OVER(PARTITION  BY  Groupname ) 每个组的个数,
                COUNT(*)  OVER(PARTITION  BY  Groupname  ORDER  BY  ID) 每个组的累积个数,
                COUNT(*)  OVER(ORDER  BY  ID) 累积个数 ,
                COUNT(*)  OVER() 总个数
from  Employee
```

返回的结果如下图：

ID	Name	GroupName	Salary	每个组的个数	每个组的累积个数	累积个数	总个数
1	小明	开发部	8000	3	1	1	10
4	小张	开发部	7600	3	2	2	10
5	小白	开发部	7000	3	3	3	10
8	小王	财务部	5000	3	1	4	10
9	NULL	财务部	NULL	3	2	5	10
15	小刘	财务部	6000	3	3	6	10
16	小高	行政部	4500	4	1	7	10
18	小王	行政部	4000	4	2	8	10
23	小李	行政部	4500	4	3	9	10
29	小吴	行政部	4700	4	4	10	10

后面的每个开窗函数就不再一一解读了，可以对照上面 SUM 后的开窗函数进行一一对照。

9.7 OVER 在排序函数中使用的示例

我们对 4 个排序函数一一演示

```
--先建立测试表和测试数据
WITH t AS
(SELECT  1  StuID,'一班'  ClassName,70  Score
UNION  ALL
SELECT  2,'一班',85
UNION  ALL
SELECT  3,'一班',85
UNION  ALL
SELECT  4,'二班',80
UNION  ALL
SELECT  5,'二班',74
UNION  ALL
SELECT  6,'二班',80
)
SELECT  *  INTO  Scores  FROM  t;
SELECT  *  FROM  Scores
```

9.8 ROW_NUMBER()

定义：ROW_NUMBER()函数作用就是将 SELECT 查询到的数据进行排序，每一条数据加一个序号，他不能用做于学生成绩的排名，一般多用于分页查询，比如查询

前 10 个 查询 10-100 个学生。ROW_NUMBER()必须与 ORDER BY 一起使用，否则会报错。

对学生成绩排序

```
SELECT *,
ROW_NUMBER() OVER (PARTITION BY ClassName ORDER BY SCORE DESC) 班内排序,
ROW_NUMBER() OVER (ORDER BY SCORE DESC) AS 总排序
FROM Scores;
```

结果如下：

StuID	ClassName	Score	班内排序	总排序
2	一班	85	1	1
3	一班	85	2	2
4	二班	80	1	3
6	二班	80	2	4
5	二班	74	3	5
1	一班	70	3	6

这里的 PARTITION BY 和 ORDER BY 的作用与我们在上面看到的聚合函数的作用一样，都是用来进行分组和排序使用的。

此外 ROW_NUMBER()函数还可以取指定顺序的数据。

```
SELECT * FROM (
SELECT *, ROW_NUMBER() OVER (ORDER BY SCORE DESC) AS 总排序
FROM Scores
) t WHERE t.总排序=2;
```

结果如下：

StuID	ClassName	Score	总排序
3	一班	85	2

9.9 RANK()

定义：RANK()函数，顾名思义排名函数，可以对某一个字段进行排名，这里和ROW_NUMBER()有什么不一样呢？ROW_NUMBER()是排序，当存在相同成绩的学生时，ROW_NUMBER()会依次进行排序，他们序号不相同，而 Rank()则不一样。如果出现相同的，他们的排名是一样的。下面看例子:

示例

```
SELECT ROW_NUMBER() OVER (ORDER BY SCORE DESC) AS [RANK],*
FROM Scores;

SELECT RANK() OVER (ORDER BY SCORE DESC) AS [RANK],*
FROM Scores;
```

结果:

StuID	ClassName	Score	总排序
2	一班	85	1
3	一班	85	2
4	二班	80	3
6	二班	80	4
5	二班	74	5
1	一班	70	6

StuID	ClassName	Score	总排序
2	一班	85	1
3	一班	85	1
4	二班	80	3
6	二班	80	3
5	二班	74	5
1	一班	70	6

其中上图是 ROW_NUMBER()的结果，下图是 RANK()的结果。当出现两个学生成绩相同是里面出现变化。RANK()是 1-1-3-3-5-6，而 ROW_NUMBER()则还是 1-2-3-4-5-6，这就是 RANK()和 ROW_NUMBER()的区别了。

9.10 DENSE_RANK()

定义：DENSE_RANK()函数也是排名函数，和 RANK()功能相似，也是对字段进行排名，那它和 RANK()到底有什么不同那？特别是对于有成绩相同的情况，DENSE_RANK()排名是连续的，RANK()是跳跃的排名，一般情况下用的排名函数就是 RANK() 我们看例子：

示例

```
SELECT
RANK() OVER (ORDER BY SCORE DESC) AS [RANK],*
FROM Scores;

SELECT
DENSE_RANK() OVER (ORDER BY SCORE DESC) AS [RANK],*
FROM Scores;
```

结果如下：

StuID	ClassName	Score	总排序
2	一班	85	1
3	一班	85	1
4	二班	80	3
6	二班	80	3
5	二班	74	5
1	一班	70	6

StuID	ClassName	Score	总排序
2	一班	85	1
3	一班	85	1
4	二班	80	2
6	二班	80	2
5	二班	74	3
1	一班	70	4

上面是 RANK()的结果，下面是 DENSE_RANK()的结果

9.11 NTILE()

定义：NTILE()函数是将有序分区中的行分发到指定数目的组中，各个组有编号，编号从 1 开始，就像我们说的'分区'一样 ，分为几个区，一个区会有多少个。

```
SELECT *,NTILE(1) OVER (ORDER BY SCORE DESC) AS 分区后排序 FROM Scores;

SELECT *,NTILE(2) OVER (ORDER BY SCORE DESC) AS 分区后排序 FROM Scores;

SELECT *,NTILE(3) OVER (ORDER BY SCORE DESC) AS 分区后排序 FROM Scores;
```

结果如下：

StuID	ClassName	Score	分区后排序
2	一班	85	1
3	一班	85	1
4	二班	80	1
6	二班	80	1
5	二班	74	1
1	一班	70	1

StuID	ClassName	Score	分区后排序
2	一班	85	1
3	一班	85	1
4	二班	80	1
6	二班	80	2
5	二班	74	2
1	一班	70	2

StuID	ClassName	Score	分区后排序
2	一班	85	1
3	一班	85	1
4	二班	80	2
6	二班	80	2
5	二班	74	3
1	一班	70	3

就是将查询出来的记录根据 NTILE 函数里的参数进行平分分区。

9.12 批注

OVER 开窗函数是我们工作中经常要使用到的，特别是在做数据分析计算的时候，经常要对数据进行分组排序。上面我们额外介绍了聚合函数和排序函数的与 OVER 结合的使用方法，此外还有很多与 OVER 一起使用的函数，比如 LEAD 函数，

LAG 函数，STRING_AGG 函数等等都会使用到开窗函数 OVER，其使用方法也要务必掌握。

第十章 变量

10.1 变量的定义

SQL Server 中的变量就是一个参数，可以对这个参数进行赋值。

10.2 变量的分类

变量分为局部变量和全局变量，局部变量用@来标识，全局变量用@@来标识（常用的全局变量一般都是已经定义好的）

10.3 声明变量

变量在使用前必须先声明才能够使用。

申明局部变量语法

DECLARE @变量名 数据类型;

例如：

DECLARE @A INT;

这样就声明了一个整数型的变量@A

10.4 局部变量赋值

声明完了变量就可以给变量赋值了，变量赋值有两种方式 SET 或 SELECT

语法

SET 变量名=值

SELECT 变量名 1=值 1,变量名 2=值 2

从上面的语法大家可能已经看出两种赋值方式的区别了，SET 只能给一个变量赋值，SELECT 可以给多个变量赋值。

例如

```
SET @A=3
SELECT @A=字段名 1，@B=字段名 2 FROM TABLE
```

10.5 变量常用场景

变量一般用作参数去给字段赋值，即将变量的值反过来赋值给字段。

我们以表 Customers 作为示例表

客户ID	姓名	地址	城市	邮编	省份
1	张三	北京路27号	上海	200000	上海市
2	李四	南京路12号	杭州	310000	浙江省
3	王五	花城大道17号	广州	510000	广东省
4	马六	江夏路19号	武汉	430000	湖北省
5	赵七	西二旗12号	北京	100000	北京市
6	宋一	花城大道21号	广州	510000	广东省
7	刘二	长安街121号	北京	100000	北京市

```
DECLARE @ID INT
DECLARE @NAME VARCHAR2(50)
DECLARE @ADDRESS VARCHAR2(50)
--用 SET 方法给变量赋值 ， 此方法一次只能给一个变量赋值
SET @ID=1
--将部门 ID 为 1 的客户姓名和地址，赋值给@NAME 和@ADDRESS 变量 ，此方法能一次多个变量赋值
SELECT @NAME=姓名,@ADDRESS=地址 FROM Customers WHERE 客户 ID=@ID
```

```
--查询变量里的结果
SELECT  @NAME,@ADDRESS
```

结果如下：

(无列名)	(无列名)
张三	北京路27号

如果我们想查询其他 ID 的姓名和地址，只需要更改一下@ID 的值即可。

Q：可能有人会问，我直接把值写在客户 ID 后面不就可以了吗？为什么写这么长一段内容来要使用变量呢?

这里有两个原因

1、使用简便

当一个查询里同一个字段需要修改的地方较多的时候，我们只需要修改这个字段对应的变量内容，那么所有的字段对应的值都会一起跟着修改。

例如

要查询学生们对应的不同老师的信息：

```
DECLARE  @ID  INT
SET  @ID=1
SELECT  *  FROM  TEST  WHERE  Teacher=@ID  AND  Student='张三'
UNION  ALL
SELECT  *  FROM  TEST  WHERE  Teacher=@ID  AND  Student='李四'
UNION  ALL
SELECT  *  FROM  TEST  WHERE  Teacher=@ID  AND  Student='王五'
UNION  ALL
SELECT  *  FROM  TEST  WHERE  Teacher=@ID  AND  Student='马六'
UNION  ALL
SELECT  *  FROM  TEST  WHERE  Teacher=@ID  AND  Student='赵七'
```

我们只需要修改@ID 的值，下面的所有查询的 ID 都会变更。

2、可以提高查询效率。

当我们使用查询的使用，数据库在执行这个查询语句的时候，如果不使用变量来修改值，实际上是两个查询。

例如：

```
SELECT * FROM TEST WHERE Student='张三'
SELECT * FROM TEST WHERE Student='李四'
```

执行这两个查询，数据库会制定两个执行计划，而制定执行计划是需要消耗系统资源的。

而如果我们改成：

```
DECLARE @NAME VARCHAR(20)
SET @NAME='张三'
SELECT * FROM TEST WHERE Student=@NAME
```

当我们修改@NAME 的值为'李四'的时候，数据库还是会使用之前的执行计划。这样就节省了时间。

10.6 全局变量

全局变量使用@@来表示，一般都是系统预定义的一些全局变量。常用的全局变量有

@@ERROR ——最后一个 SQL 错误的错误号

@@IDENTITY ——最后一次插入的标识值

@@LANGUAGE ——当前使用的语言的名称

@@MAX_CONNECTIONS – 可以创建的同时连接的最大数目

@@ROWCOUNT —-受上一个 SQL 语句影响的行数

@@SERVERNAME ——本地服务器的名称

@@TRANSCOUNT ——当前连接打开的事物数

@@VERSION ——SQL Server 的版本信息

例如查询数据库的版本号

```
PRINT  @@VERSION
```

结果：

```
PRINT @@VERSION
```

100 %

消息

```
Microsoft SQL Server 2016 (RTM-GDR) (KB4019088) - 13.0.1742.0 (X64)
    Jul  5 2017 23:41:17
    Copyright (c) Microsoft Corporation
    Enterprise Edition (64-bit) on Windows 10 Home China 6.3 <X64> (Build 17763: )
```

查询本地服务器的名称

```
PRINT  @@SERVERNAME
```

结果：

```
PRINT @@SERVERNAME
```

100 %

消息

```
LYVEN
```

这些信息都存储在全局变量中，当发生改变时，全局变量的值也会跟着改变。

以上就是变量的一些相关内容，如有什么疑问，可以在底下留言，我会一一回复的。

10.7 批注

变量的应用范围比较广，特别是在存储过程，游标还有动态 SQL 中都有应用。作用也比较明显，在查询优化方面也是一个不错的选择。此外还有很多全局变量可以供我们在平时的开发中去使用，有兴趣的可以去探究一下其他全局变量的用法。

第十一章 递归

11.1 递归查询原理

SQL Server 中的递归查询是通过 CTE(表表达式)来实现。至少包含两个查询，第一个查询为定点成员，定点成员只是一个返回有效表的查询，用于递归的基础或定位点；第二个查询被称为递归成员，使该查询称为递归成员的是对 CTE 名称的递归引用是触发。在逻辑上可以将 CTE 名称的内部应用理解为前一个查询的结果集。

11.2 递归查询的终止条件

递归查询没有显式的递归终止条件，只有当第二个递归查询返回空结果集或是超出了递归次数的最大限制时才停止递归。是指递归次数上限的方法是使用 MAXRECURION。

11.3 递归查询的优点

效率高，大量数据集下，速度比程序的查询快。

11.4 递归的常见形式

WITH CTE AS (
SELECT column1,column2... FROM tablename WHERE conditions
UNION ALL
SELECT column1,column2... FROM tablename
INNER JOIN CTE ON conditions
)

11.5 递归查询示例

创建测试数据，有一个员工表 Employee，ManagerID 是 UserID 的父节点，这是一个非常简单的层次结构模型。

```
USE  SQL_Road
GO
CREATE    TABLE  Employee
(
       UserID  INT,
       ManagerID  INT,
       Name  NVARCHAR(10)
)
 INSERT    INTO  dbo.Employee
 SELECT  1,-1,N'Boss'
 UNION    ALL
 SELECT  11,1,N'A1'
 UNION    ALL
 SELECT  12,1,N'A2'
 UNION    ALL
 SELECT  13,1,N'A3'
 UNION    ALL
 SELECT  111,11,N'B1'
 UNION    ALL
 SELECT  112,11,N'B2'
 UNION    ALL
 SELECT  121,12,N'C1'
```

查询一下 Employee 表里的数据

UserID	ManagerID	Name
1	-1	Boss
11	1	A1
12	1	A2
13	1	A3
111	11	B1
112	11	B2
121	12	C1

查询每个 User 的的直接上级 Manager

```
WITH CTE AS(
  SELECT  UserID,ManagerID,Name,Name  AS  ManagerName
  FROM  dbo.Employee
  WHERE  ManagerID=-1
  UNION  ALL
  SELECT  c.UserID,c.ManagerID,c.Name,p.Name  AS  ManagerName
  FROM  CTE P
  INNER  JOIN  dbo.Employee c  ON  p.UserID=c.ManagerID
)

SELECT  UserID,ManagerID,Name,ManagerName
FROM  CTE
```

结果如下：

UserID	ManagerID	Name	ManagerName
1	-1	Boss	Boss
11	1	A1	Boss
12	1	A2	Boss
13	1	A3	Boss
121	12	C1	A2
111	11	B1	A1
112	11	B2	A1

我们来解读一下上面的代码

1、查询 ManagerID=-1，作为根节点，这是递归查询的起始点。

2、迭代公式是 UNION ALL 下面的查询语句。在查询语句中调用中 CTE，而查询语句就是 CTE 的组成部分，即“自己调用自己”，这就是递归的真谛所在。

所谓迭代，是指每一次递归都要调用上一次查询的结果集，UNION ALL 是指每次都把结果集并在一起。

3、迭代公式利用上一次查询返回的结果集执行特定的查询，直到 CTE 返回 NULL 或达到最大的迭代次数，默认值是 32。最终的结果集是迭代公式返回的各个结果集的并集，求并集是由 UNION ALL 子句定义的，并且只能使用 UNION ALL

查询路径

下面我们通过层次结构查询子节点到父节点的 PATH，我们对上面的代码稍作修改：

```
WITH CTE AS(
  SELECT  UserID,ManagerID,Name,CAST(Name  AS  NVARCHAR(MAX))  AS  LPath
  FROM  dbo.Employee
  WHERE  ManagerID=-1
  UNION  ALL
  SELECT  c.UserID,c.ManagerID,c.Name,p.LPath+'->'+c.Name  AS  LPath
```

```
  FROM  CTE P
  INNER  JOIN  dbo.Employee c  ON  p.UserID=c.ManagerID
  )

SELECT  UserID,ManagerID,Name,LPath
FROM  CTE
```

其中 CAST(Name AS NVARCHAR(MAX))是将 Name 的长度设置为最大，防止字段过长超出字段长度。具体结果如下：

UserID	ManagerID	Name	LPath
1	-1	Boss	Boss
11	1	A1	Boss->A1
12	1	A2	Boss->A2
13	1	A3	Boss->A3
121	12	C1	Boss->A2->C1
111	11	B1	Boss->A1->B1
112	11	B2	Boss->A1->B2

以上就是递归查询的一些知识介绍了，自己可以动手实验一下，这个一般在面试中也经常会考察面试者，希望能帮助到大家~

第十二章 流程控制

12.1 流程控制的定义

一般是指用来控制程序执行和流程分至点额命令，一般指的是逻辑计算部分的控制。

12.2 流程控制种类

常见的流程控制有以下 8 种

BEGIN ... END	WAITFOR	GOTO
WHILE	IF ... ELSE	BREAK
RETURN	CONTINUE	

下面给大家具体介绍每种流程控制的用法。

12.3 BEGIN...END

BEGIN ... END 语句用于将多个 T-SQL 语句合为一个逻辑块。当流程控制语句必须执行一个包含两条或两条以上的 T-SQL 语句的语句块时，使用 BEGIN ... END 语句。

语法

```
BEGIN
sql_statement...
END
```

示例

我们在数据库中打印出我们公众号的名称"SQL 数据库开发"

```
DECLARE  @A  VARCHAR(20)
SET  @A='SQL 数据库开发'
BEGIN
```

```
SELECT  @A
END
```

结果如下：

(无列名)
SQL数据库开发

这里的 SELECT @A 就是一条被执行的命令语句。

12.4 IF [...ELSE]

IF [...ELSE]表示可以只使用 IF，也可以 IF 和 ELSE 一起使用，表示条件判断。当满足某个条件使，就执行 IF 下面的语句，否则执行 ELSE 下面的语句

IF 语法

IF <条件表达式>

{命令行 | 程序块}

IF 示例

如果某字符串的长度大于 5，就打印该字符串

```
DECLARE  @A  VARCHAR(20)
SET  @A='SQL 数据库开发'
IF  LEN(@A)>5
SELECT  @A
```

结果：

(无列名)
SQL数据库开发

这里结果与上面的 BEGIN...END 一样，但是如果我们将条件改成大于 8，结果可能就不是这样的了，小伙伴们可以试一下。

IF...ELSE 语法

IF <条件表达式>

{命令行 | 程序块}

ELSE {命令行 | 程序块}

IF...ELSE 示例

如果字符串的长度大于 10，就打印该字符串，否则打印"字符串长度太短"

```
DECLARE  @A  VARCHAR(20)
SET  @A='SQL 数据库开发'
IF  LEN(@A)>10
SELECT  @A
ELSE
SELECT  '字符串长度太短'
```

结果：

(无列名)
字符串长度太短

很明显字符串"SQL 数据库开发"长度不大于 10，所以返回 ELSE 里的结果了。

12.5 WHILE

WHILE 是循环控制，当满足 WHILE 后面的条件后，就可以循环执行 WHILE 下面的语句。通常与 CONTINUE 和 BREAK 一起使用，Break 命令让程序完全跳出循环语句，结束 WHILE 命令，CONTINUE 是让命令继续返回执行

语法

WHILE <条件表达式>

{命令行 | 程序块}

CONTINUE

{命令行 | 程序块}

BREAK

{命令行 | 程序块}

示例

有 1 到 10 这样一组数字，从 1 按顺序开始，遇到偶数就跳过，遇到奇数就打印出来，当遇到 9 就结束打印。

```
DECLARE @i int;
SET @i = 0;
WHILE(@i < 10)
BEGIN
        SET @i = @i + 1;
    IF(@i % 2 = 0)
    BEGIN
                PRINT ('跳过偶数数' + CAST(@i AS varchar));
        CONTINUE;
    END
        ELSE IF (@i = 9)
    BEGIN
                PRINT ('到' + CAST(@i AS varchar) + '就结束打印');
        BREAK;
```

```
        END
        PRINT @i;
END
```

结果如下：

我们只正常打印出来了 1-3-5-7 其他的不是跳过就是到 9 就结束了。

12.6 RETURN

RETURN 语句用于使程序从一个查询、存储过程或批量处理中无条件返回，其后面的语句不再执行。如果在存储过程中使用 return 语句，那么此语句可以指定返回给调用应用程序、批处理或过程的整数；如果没有为 RETURN 指定整数值，那么该存储过程将返回 0。

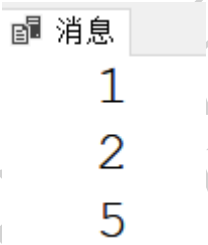
语法

```
RETURN [整数表达式]
```

示例

```
BEGIN
        PRINT(1);
    PRINT(2);
    RETURN ;
    PRINT(3);   --在 RETURN 之后的代码不会被执行，因为会跳过当前批处理
        PRINT(4);
END
GO
BEGIN
        PRINT(5);
END
```

结果如下：



RETURN 后面的 3-4 都没打印，说明在当前批处理的 RETURN 后都没执行，而新起的 BEGIN...END 不受上面的 RETURN 影响，所以打印了 5

12.7 GOTO

GOTO 命令用来改变程序执行的流程，使程序跳转到标识符指定的程序行再继续往下执行。

GOTO 命令虽然增加了程序设计的灵活性，但破坏了程序的结构化，使程序结构变得复杂而且难以测试。

注意：

- 语句标识符可以是数字或者字母的组合，但必须以":"结束。而在 GOTO 语句后的标识符不必带":"。
- GOTO 语句和跳转标签可以在存储过程、批处理或语句块中的任何地方使用，但不能超出批处理的范围。

语法

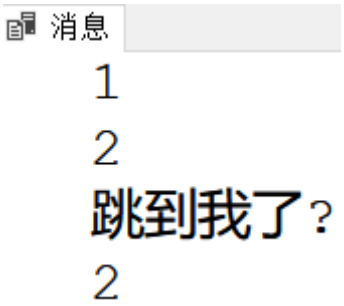
```
GOTO 标识符
```

示例

```
DECLARE @i INT;
SET @i = 1;
PRINT @i;
SET @i = 2;
PRINT @i;
GOTO ME;
SET @i = 3; --这行被跳过了
PRINT @i;

ME:PRINT('跳到我了?');
PRINT @i
```

结果如下：



从上面可以看出，当跳到 ME 的时候，GOTO 之前的数有打印，之后的数就跳过了

12.8 WAITFOR

用于挂起语句的执行，直到指定的时间点或者指定的时间间隔。

注意：

WAITFOR 常用语某个特定的时间点或时间间隔自动执行某些任务。在 WAITFOR 语句中不能包含打开游标，定义视图这样的操作。在包含事务的语句中不要使用 WAITFOR 语句，因为 WAITFOR 语句在时间点或时间间隔执行期间将一直拥有对象的锁，当事务中包含 WAITFOR 语句，事务的其他语句又需要访问被锁住的数据对象事就容易发生死锁现象。

指定时间点的语法

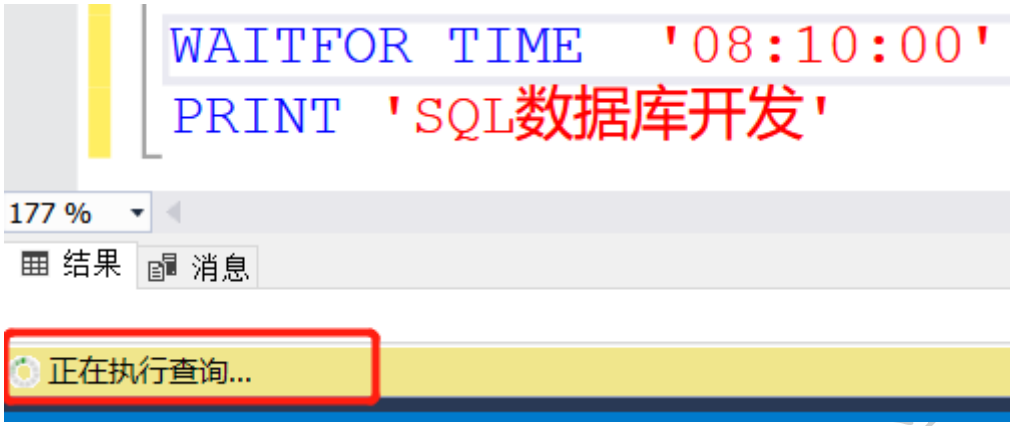
WAITFOR　TIME <具体时间>

示例

在'08:10:00'执行打印字符串"SQL 数据库开发"

```
WAITFOR TIME  '08:10:00'
PRINT  'SQL 数据库开发'
```

如果你执行这句话，那如果在今天这个点之前，那么等到这个时候它就会打印字符串，如果在今天这个点之后，那你需要等到第二天的这个时间点才会打印。在未执行之前查询窗口是一直"正在执行查询..."状态



指定等待时间间隔的语法

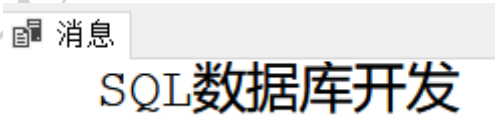
WAITFOR DELAY 'INTERVAR'

INTERVAR 为时间间隔，指定执行 WAITFOR 语句之前需要等待的时间，最多为 24 小时。

示例

```
WAITFOR DELAY '00:00:03'
PRINT 'SQL 数据库开发'
```

在等到 3 秒钟后，会打印出字符串



12.9 批注

流程控制是 SQL 开发中经常需要使用到的，特别是条件判断 IF...ELSE，循环执行 WHILE 是经常使用的，对于想在 SQL 开发中有所提高的同学，务必要掌握这几个流程控制的用法。

第十三章 动态 SQL

在介绍动态 SQL 前我们先看看什么是静态 SQL

13.1 静态 SQL

静态 SQL 语句一般用于嵌入式 SQL 应用中，在程序运行前，SQL 语句必须是确定的，例如 SQL 语句中涉及的列名和表名必须是存在的。静态 SQL 语句的编译是在应用程序运行前进行的，编译的结果会存储在数据库内部。而后程序运行时，数据库将直接执行编译好的 SQL 语句，降低运行时的开销。

13.2 动态 SQL

动态 SQL 语句是在应用程序运行时被编译和执行的，例如，使用 DB2 的交互式工具 CLP 访问数据库时，用户输入的 SQL 语句是不确定的，因此 SQL 语句只能被动态地编译。动态 SQL 的应用较多，常见的 CLI 和 JDBC 应用程序都使用动态 SQL。

13.3 动态 SQL 作用

- 自动化管理任务。例如：对于数据库实例中的每个数据库，查询其元数据，为其执行 BACKUP DATABASE 语句。
- 改善特定任务的性能。例如，构造参数化的特定查询，以重用以前缓存过的执行计划。

- 对实际数据进行查询的基础上，构造代码元素。例如，当事先不知道再 PIVOT 运算符的 IN 子句中应该出现哪些元素时，动态构造 PIVOT 查询。

13.4 动态 SQL 执行方法

使用 EXEC(EXECUTE 的缩写)命令和使用 SP_EXECUTERSQL。

EXEC 命令执行

语法

EXECUTE (SQL 语句)

注：EXECUTE 命令有两个用途，一个是用来执行存储过程，另一个是执行动态 SQL

不带参数示例

在变量@SQL 中保存了一个字符串，该字符串中包含一条查询语句，再用 EXEC 调用保存在变量中的批处理代码，我们可以这样写 SQL：

```
EXEC ('SELECT * FROM Customers')
```

结果如下：

客户ID	姓名	地址	城市	邮编	省份
1	张三	北京路27号	上海	200000	上海市
2	李四	南京路12号	杭州	310000	浙江省
3	王五	花城大道17号	广州	510000	广东省
4	马六	江夏路19号	武汉	430000	湖北省
5	赵七	西二旗12号	北京	100000	北京市
6	宋一	花城大道21号	广州	518000	广东省
7	刘二	长安街121号	北京	100000	北京市

与我们直接执行 SELECT * FROM Customers 一样。

带参数示例

还是上面的示例，我们换一种写法

```
DECLARE  @SQL  AS  VARCHAR(100);
DECLARE  @Column  AS  VARCHAR(20);
SET  @Column  =  '姓名'
SET  @SQL  =  'SELECT '  + @Column  +  ' FROM Customers';
EXEC (@SQL)
```

结果如下：

姓名
李四
刘二
马六
宋一
王五
张三
赵七

SP_EXECUTERSQL 执行

语法

EXEC SP_EXECUTERSQL 参数 1,参数 2,参数 3

注意：SP_EXECUTERSQL 是继 EXEC 后另一种执行动态 SQL 的方法。使用这个存储过程更加安全和灵活，因为它支持输入和输出参数。注意的是，与 EXEC 不同的是,SP_EXECUTERSQL 只支持使用 Unicode 字符串作为其输入的批处理代码。

示例

构造了一个对 Customers 表进行查询的批处理代码，在其查询过滤条件中使用一个输入参数@CusID

```
DECLARE  @SQL  AS  NVARCHAR(100);
SET  @SQL=N'SELECT * FROM Customers
WHERE 客户 ID=@CusID;';

EXEC SP_EXECUTESQL
        @STMT=@SQL,
        @PARMS=N'@CusID AS INT',
        @CusID=1;
```

结果如下：

客户ID	姓名	地址	城市	邮编	省份
1	张三	北京路27号	上海	200000	上海市

代码中将输入参数取值指定为 1，但即使采用不同的值在运行这段代码，代码字符串仍然保存相同。这样就可以增加重用以前缓存过的执行计划的机会。

13.5 批注

动态 SQL 在日常工作中可能接触的比较少，但是其功能是非常强大的，可以直接嵌套在代码里进行操作数据，但是也很容易出错，特别是在进行命令拼接时候要非常仔细。这里只是给大家简单介绍一下其使用方法，需要深入使用还需要多多研究。

第十四章 触发器

14.1 触发器的定义

触发器（trigger）是 SQL Server 提供给程序员和数据分析员来保证数据完整性的一种方法，它是与表事件相关的特殊的存储过程，它的执行不是由程序调用，也不是手工启动，而是由事件来触发，比如当对一个表进行操作（ INSERT，DELETE， UPDATE）时就会激活它执行。

14.2 触发器的作用

触发器的主要作用就是其能够实现由主键和外键所不能保证的复杂参照完整性和数据的一致性，它能够对数据库中的相关表进行级联修改，提高比 CHECK 约束更复杂的的数据完整性，并自定义错误消息。

触发器的主要作用主要有以下接个方面

- 强制数据库间的引用完整性
- 级联修改数据库中所有相关的表，自动触发其它与之相关的操作
- 跟踪变化，撤销或回滚违法操作，防止非法修改数据
- 返回自定义的错误消息，约束无法返回信息，而触发器可以
- 触发器可以调用更多的存储过程

14.3 触发器的优点

- 触发器是自动的。当对表中的数据做了任何修改之后立即被激活。
- 触发器可以通过数据库中的相关表进行层叠修改。

- 触发器可以强制限制。这些限制比用 CHECK 约束所定义的更复杂。与 CHECK 约束不同的是，触发器可以引用其他表中的列。

14.4 触发器的分类

SQL Server 包括三种常规类型的触发器：DML 触发器、DDL 触发器和登录触发器。

DML(数据操作语言,Data Manipulation Language)触发器

DML 触发器是一些附加在特定表或视图上的操作代码，当数据库服务器中发生数据操作语言事件时执行这些操作。

SQL Server 中的 DML 触发器有三种：

- INSERT 触发器:向表中插入数据时被触发；
- DELETE 触发器：从表中删除数据时被触发；
- UPDATE 触发器：修改表中数据时被触发。

当遇到下列情形时，应考虑使用 DML 触发器：

- 通过数据库中的相关表实现级联更改
- 防止恶意或者错误的 INSERT、DELETE 和 UPDATE 操作，并强制执行 CHECK 约束定义的限制更为复杂的其他限制。
- 评估数据修改前后表的状态，并根据该差异才去措施。

DDL(数据定义语言,Data Definition Language)触发器

DDL 触发器是当服务器或者数据库中发生数据定义语言(主要是 CREATE，DROP，ALTER 开头的语句)事件时被激活使用，使用 DDL 触发器可以防止对数据架构进行的某些更改或记录数据中的更改或事件操作。

14.5 登录触发器

登录触发器将为响应 LOGIN 事件而激发存储过程。与 SQL Server 实例建立用户会话时将引发此事件。登录触发器将在登录的身份验证阶段完成之后且用户会话实际建立之前激发。因此，来自触发器内部且通常将到达用户的所有消息（例如错误消息和来自 PRINT 语句的消息）会传送到 SQL Server 错误日志。如果身份验证失败，将不激发登录触发器。

14.6 触发器的工作原理

触发器触发时：

- 系统自动在内存中创建 **INSERTED** 表或 **DELETED** 表；
- 只读，不允许修改，触发器执行完成后，自动删除。

INSERTED 表：

- 临时保存了插入或更新后的记录行；
- 可以从 INSERTED 表中检查插入的数据是否满足业务需求；
- 如果不满足，则向用户发送报告错误消息，并回滚插入操作。

DELETED 表：

- 临时保存了删除或更新前的记录行；
- 可以从 DELETED 表中检查被删除的数据是否满足业务需求；
- 如果不满足，则向用户报告错误消息，并回滚插入操作。

INSERTED 表和 DELETED 表对照：

修改操作记录	**INSERTED 表**	**DELETED 表**
增加(INSERT)记录	存放新增的记录	/
删除(DELETE)记录	/	存放被删除的记录
修改(UPDATE)记录	存放更新后的记录	存放更新前的记录

14.7 创建触发器

创建触发器的语法：

```
CREATE TRIGGER trigger_name ON table_name
[WITH ENCRYPTION]
FOR | AFTER | INSTEAD
OF [DELETE, INSERT, UPDATE]
AS
T-SQL 语句
GO
```

注：

WITH ENCRYPTION 表示加密触发器定义的 SQL 文本

DELETE, INSERT, UPDATE 指定触发器的类型

14.8 触发器示例

创建学生表

```
create table student(
stu_id int identity(1,1) primary key,
stu_name varchar(10),
stu_gender char(2),
stu_age int
)
```

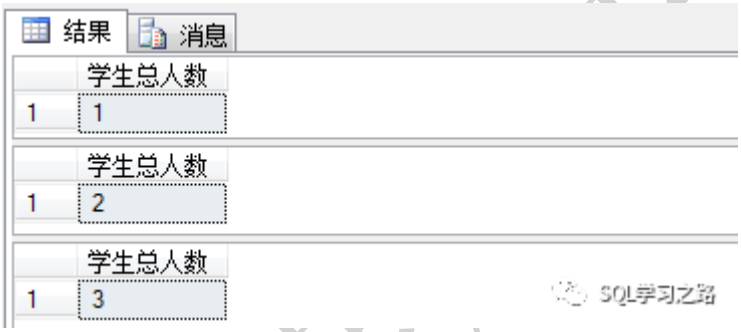
创建 INSERT 触发器

```
--创建 INSERT 触发器
create trigger trig_insert
on student after insert
as
begin
    --判断 student_sum 表是否存在
    if object_id(N'student_sum',N'U') is null
    --创建存储学生人数的 student_sum 表
    create table student_sum(
    stuCount int default(0)
    );
    declare @stuNumber int;
    select @stuNumber = count(*)from student;
    --判断表中是否有记录
    if not exists (select * from student_sum)
    insert into student_sum values(0);
    update student_sum set stuCount =@stuNumber;
    --把更新后总的学生数插入到 student_sum 表中
end

--测试触发器 trig_insert
--功能是向 student 插入数据的同时级联插入到 student_sum 表中，更新 stuCount
--因为是后触发器，所以先插入数据后，才触发触发器 trig_insert;
insert into student(stu_name,stu_gender,stu_age)
```

```
values('吕布','男',30);
select  stuCount 学生总人数  from  student_sum;
insert  into  student(stu_name,stu_gender,stu_age)
values('貂蝉','女',30);
select  stuCount 学生总人数  from  student_sum;
insert  into  student(stu_name,stu_gender,stu_age)
values('曹阿瞒','男',40);
select  stuCount 学生总人数  from  student_sum;
```

执行上面的语句后，结果如下图所示：

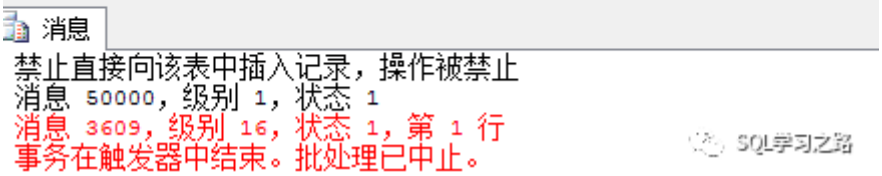


既然定义了学生总数表 student_sum 表是向 student 表中插入数据后才计算学生总数的，所以学生总数表应该禁止用户向其中插入数据

```
--创建 insert_forbidden,禁止用户向 student_sum 表中插入数据
create  trigger  insert_forbidden
on  student_sum  after  insert
as
begin
    RAISERROR('禁止直接向该表中插入记录，操作被禁止',1,1)
    --raiserror 是用于抛出一个错误
    rollback  transaction
end
--触发触发器 insert_forbidden
insert  student_sum (stuCount)
values(5);
```

结果如下：

```
--触发触发器insert_forbidden
insert student_sum (stuCount) values(5);
```



创建 DELETE 触发器

用户执行 DELETE 操作，就会激活 DELETE 触发器，从而控制用户能够从数据库中删除数据记录，触发 DELETE 触发器后，用户删除的记录会被添加到 DELETED 表中，原来表的相应记录被删除，所以在 DELETED 表中查看删除的记录。

```
--创建 delete 触发器
create  trigger  trig_delete
on  student  after  delete
as
begin
    select  stu_id  as  已删除的学生编号,
    stu_name stu_gender,
    stu_age
    from  deleted
end;

--执行一条 delete 语句触发 trig_delete 触发器
delete  from  student  where  stu_id=1;
```

结果如下：

```
--执行一一条delete语句触发trig_delete触发器
delete from student where stu_id=1;
```

结果 消息

	已删除的学生编号	stu_gender	stu_age
1	1	吕布	30

创建 UPDATE 触发器

UPDATE 触发器是当用户在指定表上执行 UPDATE 语句时被调用被调用，这种类型的触发器用来约束用户对数据的修改。UPDATE 触发器可以执行两种操作：更新前的记录存储在 DELETED 表中，更新后的记录存储在 INSERTED 表中。

```
--创建 update 触发器
create  trigger  trig_update
on  student  after  update
as
begin
    declare  @stuCount  int;
    select  @stuCount=count(*)  from  student;
    update  student_sum  set  stuCount =@stuCount;

    select  stu_id  as  更新前学生编号,
    stu_name  as  更新前学生姓名  from  deleted;

    select  stu_id  as  更新后学生编号,
    stu_name  as  更新后学生姓名  from  inserted;
end
--创建完成，执行一条 update 语句触发 trig_update 触发器
update  student  set  stu_name='张飞'
where  stu_id=2;
```

结果如下：

```
--创建完成，执行一条update语句触发trig_update触发器
update student set stu_name='张飞' where stu_id=2;
```

结果 消息

	更新前学生编号	更新前学生姓名
1	2	貂蝉

	更新后学生编号	更新后学生姓名
1	2	张飞

创建替代触发器

与前面介绍的三种 AFTER 触发器不同，SQL Server 服务器在执行 AFTER 触发器的 SQL 代码后，先建立临时的 INSERTED 表和 DELETED 表，然后执行代码中对数据库操作，最后才激活触发器中的代码。而对于替代(INSTEAD OF)触发器,SQL Server 服务器在执行触发 INSTEAD OF 触发器的代码时，先建立临时的 INSERTED 表和 DELETED 表，然后直接触发 INSTEAD OF 触发器，而拒绝执行用户输入的 DML 操作语句。

```
--创建 instead of 触发器
create  trigger  trig_insteadOf
on  student instead  of  insert
as
begin
    declare  @stuAge  int;
    select  @stuAge=(select  stu_age  from  inserted)
    if(@stuAge >120)
    select  '插入年龄错误'  as  '失败原因'
end
```

创建完成，执行一条 INSERT 语句触发触发器 trig_insteadOf

```
insert into student (stu_name,stu_gender,stu_age)values('子龙','男',130);
```

结果 消息

	失败原因
1	插入年龄错误

14.9 批注

触发器在早期的数据处理过程中经常使用到，特别是在处理一些因某些动作而需要对其他表进行调整的逻辑时。但是随着数据量的增长，触发器对数据库的性能影响越来越大，容易造成数据库性能降低。所以触发器在数据量大的场景是禁止使用的，但是其逻辑处理功能还是被一直保留，说明其还是有较深的应用场景，需要我们掌握它的相关用法。