

SQL

- 关系数据库回顾

- T-SQL基本知识

发 展 命名规则

数据类型 基本运算

常用函数 流程控制

- 数据定义语言

数据库 表 索引 视图

- 数据修改语言

Insert Update Delete

- 数据查询语言

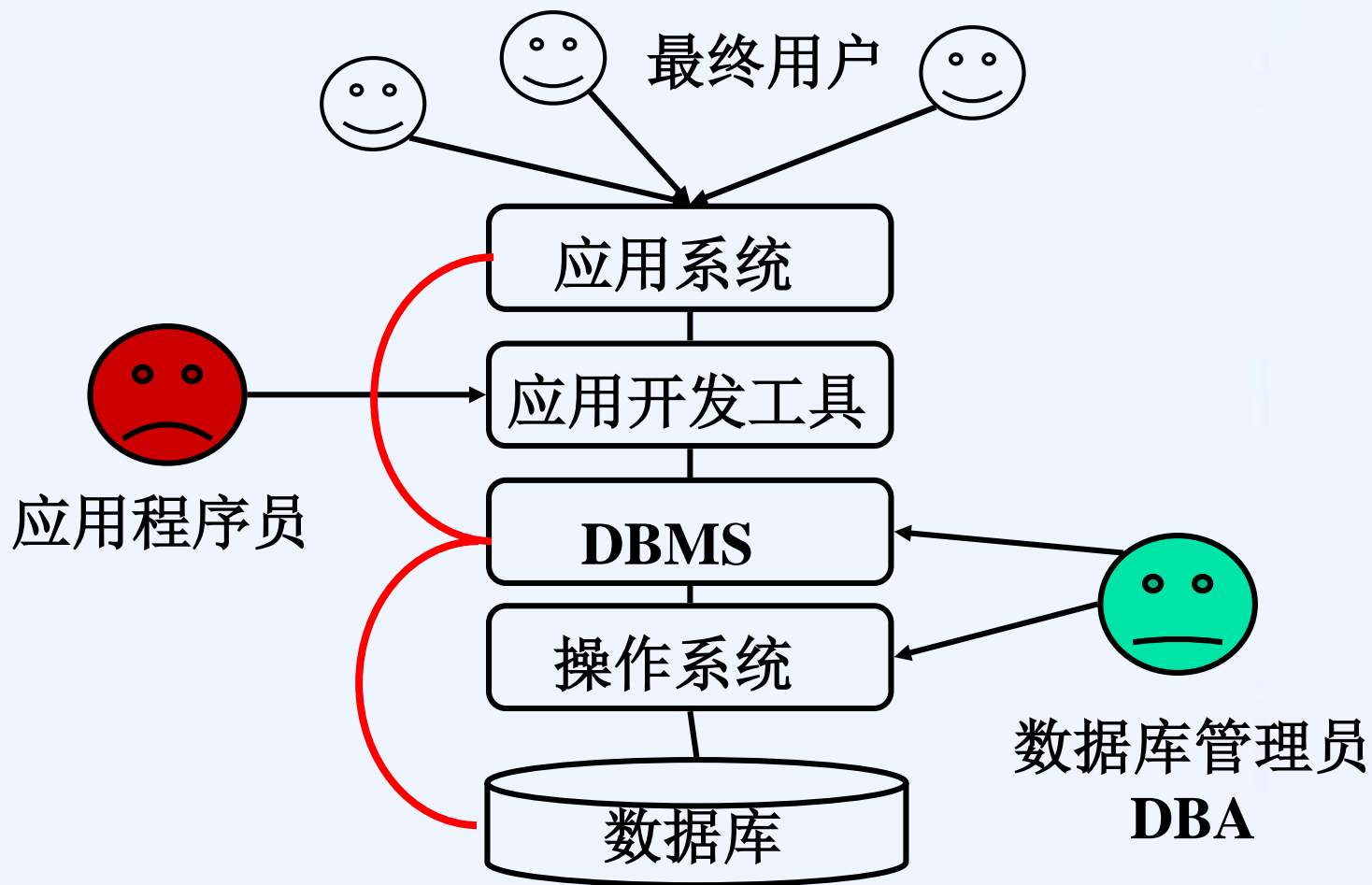
Select

- 存储过程

- 触发器

- 游标

数据库系统的构成



关系数据库简介

常见的RDBMS有：DB2, Oracle, MS SQL Server等。

关系数据库中常见的对象：

◆表(关系)：

【例】下表为Students表

code	myname	birthday	score
200101184022	王涛	1983/3/9	84
200201184358	于栋	1982/10/15	95

字段
属性

记录
元组

❖关系模式：对关系的描述，一般表示为：

关系名(属性1, 属性2,属性n)

其中加下划线表示这些属性的集合构成键码。



关系数据库

关系数据库中常见的对象：

◆表(关系)

◆索引：为表建立的一种实现快速查询的工具。可分为**簇索引**(聚集索引)和**非簇索引**(非聚集索引)。

■簇索引是行的物理顺序和行的索引顺序相同的索引。

使用簇索引可以使UPDATE和SELECT操作大大加速。

每个表只能有一个簇索引。

定义簇索引键时使用的列越少越好。

■非簇索引是指定表的逻辑顺序的索引。行的物理顺序和索引顺序不尽相同。

每个表可以建立多个非簇索引。



索引

◆簇索引：一般对符合下列情况的列建立簇索引：

- 含有有限(但不是很少)数目唯一值的列，

例如某表中含有美国的50个州缩写名的列。

- WHERE子句中含有“**BETWEEN**、>、>=、<=、<”运算符的查询列，

【例】 `SELECT * FROM sale_tab
WHERE ord_date BETWEEN '5/1/2003' AND '6/1/2003'`

- 可能返回大结果集的查询列，

【例】 `SELECT * FROM phonebook
WHERE last_name='Smith'`

◆非簇索引：对下列情况考虑使用非簇类索引：

- 含有大量唯一值的列。

- 返回很小的或单行结果集的查询。



关系数据库

关系数据库中常见的对象：

◆ **表(关系)**

◆ **索引**

◆ **存储过程**：多个SQL语句的有机的集合。

◆ **触发器**：一种特殊的存储过程，当满足一定的条件时，会触发它执行。多用来保证表中数据的一致性。

◆ **视图**：可以看成是虚拟表或存储查询。视图不作为独特的对象存储，数据库内存储的是SELECT语句。

【例】下表为Students表， ‘SELECT code FROM Students’

code	myname	birthday	score
200101184022	王涛	1983/3/9	84
200201184358	于栋	1982/10/15	95

SQL的发展

◆SQL：结构化查询语言。

- 1974年由CHAMBERLIN和BOYEE提出，叫SEQUEL(A Structured English Query Language)，IBM公司对其进行了修改，并用于其SYSTEM R关系数据库系统中；
- 1976年推出了第二版本SEQUEL/S；
- 1980年更名为SQL。
- 1982年，ANSI (美国国家标准化组织)开始制定SQL标准，1986年，公布了SQL语言的第一个标准SQL86。1987年，ISO(国际标准化组织)通过了SQL86标准。
- 1989年，ISO对SQL86进行了补充，推出了SQL89标准。
- 1992年，ISO又推出了SQL92标准，也称为SQL2。



SQL的发展

- 目前SQL99(也称为SQL3)已完成, 增加了面向对象的功能。
- SQL发展成了RDBMS的标准接口, 被各RDBMS支持。
- 各RDBMS对SQL又有扩展。各厂商的扩展并无太大冲突。Transact SQL(简称T-SQL)就是Sybase公司的扩充版本。

本章将以MS SQL Server 2000为背景来介绍T-SQL。



SQL命令分类

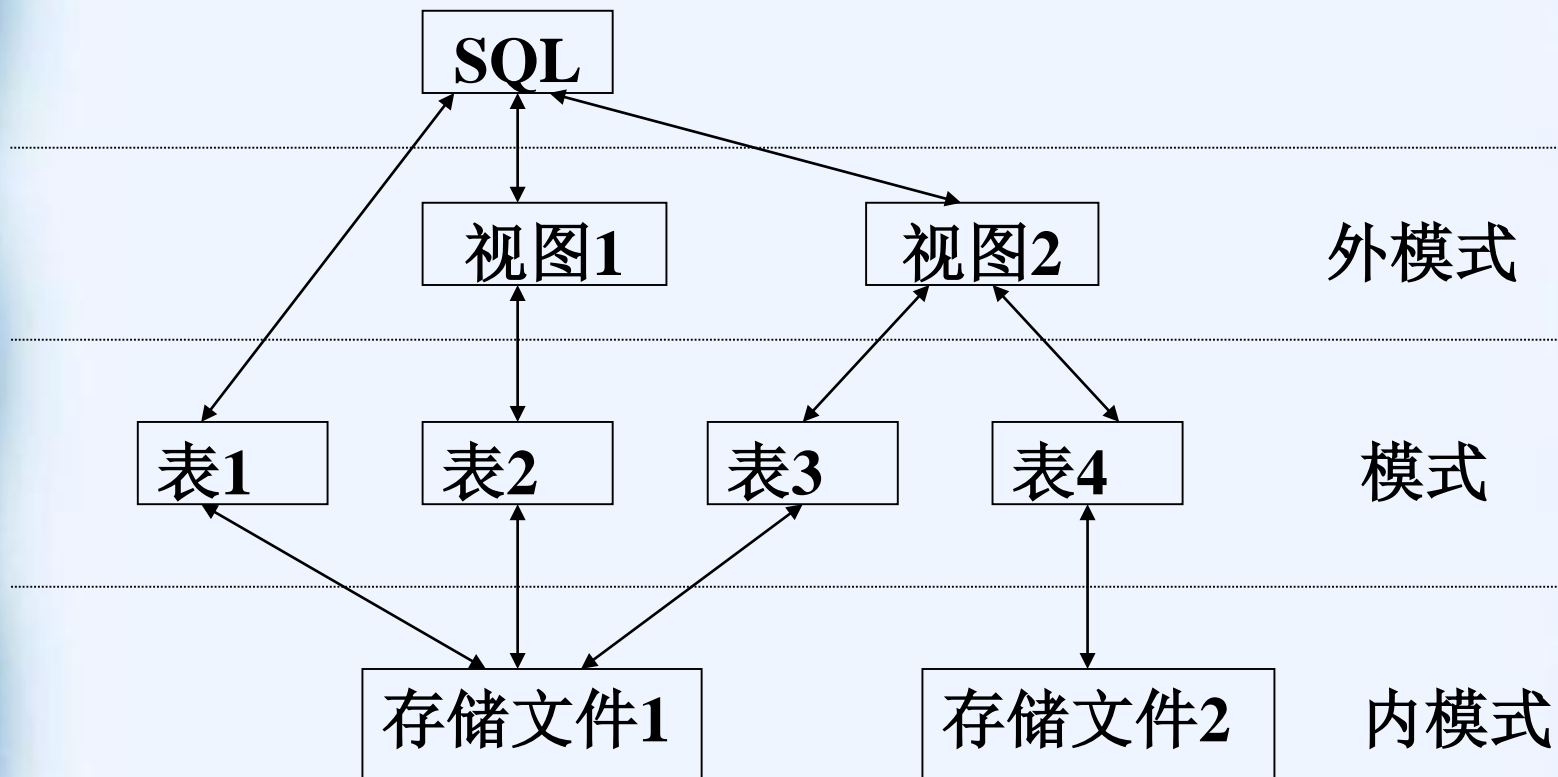
◆标准的SQL命令包括下面几类：

- 数据定义命令DDL：包括创建、修改、删除数据库、表、索引、视图等，如CREATE、ALTER、DROP.....。
- 数据修改命令DML：包括对数据的插入、删除和修改，如INSERT、DELETE、UPDATE.....。
- 数据查询命令DQL：即SELECT，可以实现选择、投影、连接等关系操作。
- 数据控制命令DCL：用于保证数据存取的安全性，如口令、授权等。
- 事务控制命令：用于保证数据的完整性、一致性、可恢复性等。



SQL运作模式

◆三级模式结构:



1、命名规则

- ◆ 必须以字母a-z 和 A-Z、下划线 (_)、at 符号 (@) 或者数字符号 (#)开头。
 - 后续字符可以是：字母、十进制数字、@、#、美元符号 (\$) 或下划线。
- ◆ 在 SQL Server 中
 - @开始的标识符表示局部变量或参数。
 - 以#开始的标识符表示临时表或过程。
 - 以双数字符号 (##) 开始的标识符表示全局临时对象。
 - 某些 T-SQL 函数的名称以@@开始。为避免混淆这些函数，建议不要使用以@@开始的名称。
- ! 标识符不能是保留字的大写和小写形式。
- ! 标识符不允许嵌入空格或其它特殊字符。



变量的声明和赋值

- 变量的声明: **DECLARE** *@local_variable data_type*
变量在声明后均初始化为 NULL。
- 给变量赋值: **SET** *@local_variable =*
或 **SELECT** *@local_variable =*

【例】

```
DECLARE @find varchar(30)
SET @find = 'Ring%'
SELECT au_lname, au_fname, phone FROM authors
WHERE au_lname LIKE @find
```

【例】

```
DECLARE @pub_id char(4), @hire_date datetime
SET @pub_id = '0877'
SET @hire_date = '1/1/2003'
```



2、基本数据类型

- 整数 **int** 从 -2^{31} 到 $2^{31}-1$ 的整数。

- 比特 **bit** 1 或 0 的整数。

- 近似数字

float 从 $-1.79E + 308$ 到 $1.79E + 308$ 的浮点精度数字。

real 从 $-3.40E + 38$ 到 $3.40E + 38$ 的浮点精度数字。

- 字符串

char(n) 固定长度的非 Unicode 字符数据，最长 8,000 个字符。

varchar(n) 可变长度的非 Unicode 字符数据，最长 8,000 个字符。

不指定 n ，则默认长度为 1。

 Unicode 字符

采用双字节对字符进行编码，有前缀 N。

【例】'Michél' 是字符串常量，N'Michél' 则是 Unicode 常量。



基本数据类型

- 日期

datetime 从 1753 年 1 月 1 日到 9999 年 12 月 31 日的日期和时间数据，精确到百分之三秒(或 3.33 毫秒)。

◆ SQL Server 可以识别以下格式的日期和时间：

- 字母日期格式：

【例】 'April 15, 1998'

- ~~数字~~ 数字日期格式：数字 分隔符 数字 分隔符 数字

- 数字的顺序一般为mdy。

- 分隔符有斜杠 (/)、连字符 (-) 或句号 (.) 。

【例】 '4/15/1998'、'1990-10-02'、'04-15-1996'

- 未分隔的字符串格式：

【例】 '19981207'

3、基本运算₁₊, ₂逻辑运算符, ₃注释

1、+(字符串串联) 将两个字符串串联到一个表达式中。

语法 *expression + expression*

参数 image、ntext 或 text 除外的字符和二进制数据类型。

两个表达式都必须具有相同的数据类型，或者其中的一个表达式必须能够隐式地转换为另一种数据类型。有时必须使用字符数据的显式转换 (通过函数CONVERT或 CAST) 。

【例】 SELECT (au_lname + ', ' + au_fname) AS Name
FROM authors

2、逻辑运算符 =、>、<、>=、<=、<>、!<、!=、!>。

【例】 WHERE price > (SELECT MIN(price) FROM titles)

3、注释：-- 起头的

或者 /* *text_of_comment* */

【例】 -- Choose the student's database.



基本运算

4BETWEEN, 5IN, 6LIKE

4、**BETWEEN** 指定测试范围。

语法 *test_expression* [NOT] **BETWEEN** *begin_expression*
AND *end_expression*

【例】 WHERE *ypayment* NOT BETWEEN 4095 AND 12000

5、**IN** 确定给定的值是否与子查询或列表中的值相匹配。

语法 *test_expression* [NOT] **IN** (*subquery*| *expression*[,...*n*])

【例】 WHERE *state* IN ('CA', 'IN', 'MD')

WHERE *state* = 'CA' OR *state* = 'IN' OR *state* = 'MD'

【例】 WHERE *au_id* NOT IN

(SELECT *au_id* FROM *author* WHERE *age* < 50)

6、**LIKE** 确定给定的字符串是否与指定的模式匹配。

语法 *match_expression* [NOT] **LIKE** *pattern*

【例】 WHERE *au_name* LIKE 'Jo%'



基本运算7通配符

7、通配符

- **%** 匹配包含零个或多个字符的任意字符串。
- **[]** 匹配指定范围内或者属于方括号所指定的集合中的任意单个字符。

【例】 WHERE au_lname LIKE '[A-C]arsen'
WHERE au_lname LIKE '[ABC]arsen'

【例】 WHERE au_lname LIKE '50%'
WHERE au_lname LIKE '50[%]'

- **[^]** 匹配不处于指定范围内或者不属于方括号内指定集合中的任意单个字符。

【例】 WHERE au_lname LIKE 'de[^a]%'

【例】 **[^ab]** 等价于 **[^a^b]** 表示一位字符不能是a或b。
[^a][^b] 表示两个相连字符相继不能是a和b。

- **_** 匹配任意单个字符。

【例】 WHERE au_fname LIKE '_ean'
WHERE au_fname LIKE '%ean'



8、EXISTS(*subquery*)

如果子查询包含行，则返回 **TRUE**。

子查询的 **SELECT** 语句不允许有 **COMPUTE** 子句和 **INTO** 关键字。

【例】

```
SELECT DISTINCT pub_name
```

```
FROM publishers
```

```
WHERE EXISTS
```

```
(SELECT * FROM titles
```

```
WHERE pub_id = publishers.pub_id AND type = 'business')
```



基本运算

9模, 10~12位运算

9、%(模)提供两数相除后的余数。

语法 *dividend % divisor*

参数 *dividend* 必须是整型。*divisor* 可以是任何数字数据类型。

结果类型 *int*

10、&(按位 AND)在两个整型值之间执行按位逻辑与运算。

语法 *expression & expression*

11、|(按位 OR)将两个给定的整型值转换为二进制表达式，对其执行按位逻辑或运算。

语法 *expression | expression*

12、~(按位 NOT)将某个给定的整型值转换为二进制表达式，对其执行按位逻辑非运算。

语法 *~ expression*



13、ALL 用标量值与单列集中的值进行比较。

scalar_exp **逻辑运算符 ALL**(*subquery*)

- *scalar_exp* 是任何有效的表达式。
- *subquery* 是返回单列结果集的子查询。返回列的数据类型必须与 *scalar_exp* 的数据类型相同。而且该子查询 **SELECT** 语句不允许使用 **ORDER BY** 子句、**COMPUTE** 子句和 **INTO** 关键字。
- 如果所有给定的比较对(*scalar_exp*, *x*)均为 **TRUE**，其中 *x* 是单列集中的值，则返回 **TRUE**；否则返回 **FALSE**。



14、SOME | ANY用标量值与单列集中的值进行比较。

scalar_expression { = | < > | != | > | > = | ! > | < | < = | ! < }
{ SOME | ANY } (*subquery*)

对于 (*scalar_expression*, *x*)(其中 *x* 是单列集中的值), 当指定的比较有为TRUE的时, **SOME** 或 **ANY** 返回 **TRUE**。否则返回 **FALSE**。

4、常用函数1COUNT

1、COUNT 返回组中项目的数量。

语法 **COUNT**({ [**ALL** | **DISTINCT**] *expression* | * })

参数

- **ALL** 返回非空值的数量。缺省为**ALL**。
- **DISTINCT** 指定 **COUNT** 返回唯一非空值的数量。
- *expression* 一个表达式(一般为属性名), 其数据类型不能是 **text**、**image** 或 **ntext**。不允许使用聚合函数和子查询。

!COUNT(*) 不能与 **DISTINCT** 一起使用。返回包括 **NULL** 值和重复值的数量。

【例】 **SELECT COUNT(DISTINCT city) FROM students**

【例】 **SELECT COUNT(*) FROM courses**



常用函数2SUM, 3AVG

2、SUM 返回表达式中值的和。SUM 只能用于数字列。空值将被忽略。

语法 SUM([ALL | DISTINCT] *expression*)

参数

- ALL 对所有的值进行聚合函数运算。缺省为ALL。
- DISTINCT 指定 SUM 返回唯一值的和。
- *expression*是常量、列或函数(bit数据类型的除外)。

【例】 SELECT SUM(score) FROM sScores

3、AVG 返回组中值的平均值。空值将被忽略。

语法 AVG([ALL | DISTINCT] *expression*)

【例】 SELECT AVG(score) FROM sScores



常用函数4MAX, 5MIN

4、MAX 返回表达式的最大值。

语法 **MAX**([**ALL** | **DISTINCT**] *expression*)

■**MAX** 可用于数字列、字符列和 **datetime** 列，但不能用于 **bit** 列。

【例】 **SELECT MAX(age) FROM students**

5、MIN 返回表达式的最小值。

语法 **MIN**([**ALL** | **DISTINCT**] *expression*)

■**MIN** 可用于数字列、**char** 列、**varchar** 列或 **datetime** 列，但不能用于 **bit** 列。

【例】 **SELECT MIN(age) FROM students**



常用函数6字符处理

6、常用的字符处理函数

- **LOWER**(*char_expression*) 将大写转换为小写字符。
- **UPPER**(*char_expression*) 将小写转换为大写字符。
- **RIGHT**(*char_expre*, *integer_expre*) 返回从右边开始指定个数的字符。 **LEFT** 类似。

【例】 `SELECT '入学年份'=LEFT(st_id,4) FROM students`

- **SUBSTRING**(*expression*, *start*, *length*) 返回字符串的一部分。

【例】 `SELECT x = SUBSTRING('abcdef', 2, 3)`

- **LEN**(*string_expression*) 返回给定字符串表达式的字符个数，不包含尾随空格。
- **REPLACE**(*'string_exp1'*, *'string_exp2'*, *'string_exp3'*) 用3替换1中出现的所有第2个给定字符串表达式。
- **SPACE**(*integer_expression*) 返回指定数量的空格组成的字符串。



常用函数

7 日期处理, 8 ISNUMERIC, 9 ISDATE

7、常用的日期处理函数

- **MONTH**(*date*) 返回代表指定日期月份的整数。

【例】 `SELECT "Month Number" = MONTH('03/12/2003')`

! SQL Server 将 0 解释为 '01/01/1900'。

【例】 `SELECT MONTH(0), DAY(0), YEAR(0)`

- **GETDATE**() 返回当前系统日期和时间。

【例】 `SELECT GETDATE()`

【例】 `CREATE TABLE Score(
stu_id char(11) NOT NULL,
cos_name varchar(40) NOT NULL,
exam_date datetime DEFAULT GETDATE())`

8、 **ISNUMERIC**(*expression*) 判断是否为有效数字类型。

9、 **ISDATE**(*expression*) 判断是否为有效的日期。



5、流程控制 1GO, 2BEGIN, 3WHILE

1、**GO** 表示提交一批T-SQL语句。**单独占行。**

2、**BEGIN...END** 将一系列T-SQL语句作为一个组同时执行。可以嵌套。

语法 **BEGIN**
 {*sql*语句}
 END

3、**WHILE** 设置循环语句的条件。

- 只要指定的条件为真，就重复执行语句。
- 可以使用**BREAK**和**CONTINUE**关键字在循环内部控制**WHILE**循环中语句的执行。

■ 语法 **WHILE** *Boolean_expression*
 { *sql*语句 }
 [**BREAK**]
 { *sql*语句 }
 [**CONTINUE**]



流程控制 例

```
【例】 WHILE (SELECT AVG(score) FROM st_score) <40
BEGIN
    UPDATE st_score SET score = score *1.2
    IF (SELECT MAX(score) FROM st_score) >90
        BREAK
    ELSE
        CONTINUE
END
```

如果平均成绩小于40分，则将st_score表中的成绩都乘以1.2。
如果最高成绩大于90分，则中止。否则继续，直到平均成绩不再小于40分。

!超过一句的SQL语句块要用**BEGIN**..... **END**包含。



流程控制 4IF

4、IF...ELSE

语法: IF *Boolean_expression*
 {*sql语句*}
 [ELSE
 {*sql语句*}]

【例】 IF (SELECT AVG(score) FROM st_score) < 80
 BEGIN
 PRINT '本课程成绩的最低分是'
 SELECT MIN(score) AS Title FROM st_score
 END
 ELSE
 PRINT '本课程平均成绩超过80分'

- **PRINT** '字符串' 直接打印字符串给用户。



流程控制 5CASE

5、CASE 计算条件列表并返回多个可能结果表达式之一。

语法 (1)简单CASE函数:

```
CASE input_expression  
WHEN when_exp THEN result_exp[ ...n ]  
[ELSE sql语句]  
END
```

(2) CASE搜索函数:

```
CASE  
WHEN Boolean_exp THEN result_exp [ ...n ]  
[ELSE sql语句]  
END
```

返回第一个取值为 TRUE 的 *result_exp*。如果没有取值为 TRUE 的, 又没有指定 ELSE 子句, 则返回 NULL 值。



流程控制 例

【例】 **SELECT** '系别' =
CASE SUBSTRING(stu_id,7,3)
WHEN '184' **THEN** '控制系'
WHEN '181' **THEN** '电信系'
ELSE '未知系别'
END, name **AS** Name **FROM** students

返回学生的系别名称和姓名。

【例】 **SELECT** '成绩' =
CASE
WHEN score **IS NULL** **THEN** '未考'
WHEN score >= 85 **THEN** '优秀'
WHEN score >= 75 and score < 85 **THEN** '良好'
ELSE '及格'
END, name **FROM** st_score

返回学生的成绩等级和姓名。



流程控制 6WAITFOR, 7RETURN

6、WAITFOR 指定触发语句块、存储过程或事务执行的时间、时间间隔或事件。

语法 **WAITFOR** { **DELAY** '*time*' | **TIME** '*time*' }

参数

DELAY指示SQL Server一直等到指定的时间过去，最长可达 24 小时。

'*time*'要等待的时间。不能指定日期。

TIME指示 SQL Server 等待到指定时间。

7、RETURN 从查询或过程中无条件退出。不执行位于 **RETURN** 之后的语句。

语法 **RETURN** [*integer_expression*]



流程控制 8GOTO, 9ROLLBACK, 10PRINT

8、GOTO 将执行流变更到标签处。跳过 GOTO 之后的 Transact-SQL 语句，在标签处继续处理。GOTO 语句和标签可在过程、批处理或语句块中的任何位置使用。GOTO 语句可嵌套使用。

语法 定义标签 *label*：改变执行 **GOTO** *label*

9、ROLLBACK TRANSACTION 滚回刚才执行的事务

10、PRINT '字符串' 直接打印字符串给用户。

数据库的建立、删除

1、建立

```
CREATE DATABASE UserTest      --数据库名
ON (NAME=UserTest_Data,      --数据设备名
FILENAME='D:\TEST_Data.mdf',  --数据文件
SIZE=1,                        --初始大小为1M
MAXSIZE=500,                   --最大只能到500M
FILEGROWTH=10%)               --以10% 的速度增大空间
LOG ON
(NAME=UserTest_Log,           --日志设备名
FILENAME='D:\TEST_Log.ldf',   --日志文件
SIZE=1,
MAXSIZE=500,
FILEGROWTH=10%)
```

默认为
MB

也可用
数值

2、删除 **DROP DATABASE** UserTest



数据库的使用

3、使用

USE UserTest

则以下的对表进行操作的SQL语句均在UserTest 中。

? 如下语句能够正确执行吗？

CREATE DATABASE UserTest

USE UserTest

出错，因为一批SQL语句中间没有GO的话，会认为一起提交。改正：

CREATE DATABASE UserTest

GO

USE UserTest

? 正在使用的数据库能够被删除吗？

表的建立

1、建立表：**CREATE TABLE** 表名(
列名1 数据类型1,列名2 数据类型2,主键说明,外部键说明)

◆基本构件：关键字、表名、列名、数据类型。

!表名在整个数据库中要唯一。

◆主键的定义：

CONSTRAINT 主键约束的名字 **PRIMARY KEY**
[CLUSTERED | NONCLUSTERED] (作为主键的列名)

!主键不能定义在可以为空的列上。

!每个表只能创建一个 **PRIMARY KEY** 约束。

◆外部键的定义：

CONSTRAINT 外部键约束的名字 **FOREIGN KEY** (本表的列名) **REFERENCES** 另一个表2 (表2中对应的列名)

!表2中对应的列名必须为表2的主键。

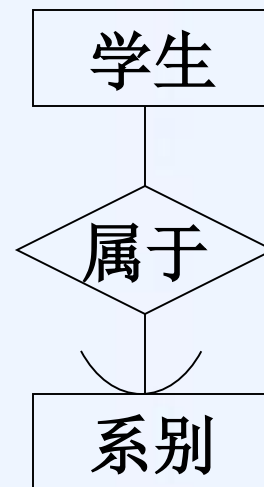
!约束名在整个数据库中要唯一。



表的建立

1、建立表

```
CREATE TABLE Students(  
  st_id integer NOT Null UNIQUE,  
  st_name varchar(50) NOT Null,  
  pwd varchar(20) Null DEFAULT '0000',  
  birthday datetime Null  
    CONSTRAINT cc_birth  
      CHECK(Year(birthday)>1985),  
  st_dept integer NOT Null,  
    CONSTRAINT pk_Students PRIMARY KEY  
      NONCLUSTERED(st_id),  
    CONSTRAINT fk_Students FOREIGN KEY(st_dept)  
      REFERENCES st_Dept(deptno)  
)
```



表的建立

```
CREATE TABLE st_Dept(  
deptno integer NOT Null  
        CONSTRAINT pk_st_Dept PRIMARY KEY,  
deptname varchar(50) NOT Null  
        CONSTRAINT upper_name  
                CHECK(deptname=UPPER(deptname)),  
fdtime datetime DEFAULT GETDATE()  
)
```



表的修改

2、修改表

ALTER TABLE *table*

[**ALTER COLUMN** 列名 数据类型等属性]

| **ADD** [列的定义|约束的定义]

| **DROP** {[CONSTRAINT] 约束名|COLUMN 列名}

|{**CHECK** | **NOCHECK**} CONSTRAINT {ALL|约束名}

|{**ENABLE** | **DISABLE**} TRIGGER {ALL|触发器名}



表的修改 例

【例】 不检查出生年份是否>1985， 则

ALTER TABLE Students **NOCHECK CONSTRAINT** cc_birth

【例】

ALTER TABLE Students **ALTER COLUMN** st_name integer
Null

!不能改已经对它建立了约束的列的属性，包括主键、外部键、UNIQUE等。

【例】 **ALTER TABLE** Students **ADD CONSTRAINT**
pk2_Students **PRIMARY KEY NONCLUSTERED** (st_name)正
确吗？

【例】 **ALTER TABLE** Students **DROP** pk_Students



表的删除

3、删除表

DROP TABLE 表名

【例】 **DROP TABLE** Students

!不能删除正由一个 **FOREIGN KEY** 约束引用的表。必须先删除引用的 **FOREIGN KEY** 约束或引用的表。

【例】 **DROP TABLE** Students

GO

DROP TABLE st_Dept

索引的建立

1、建立

CREATE [**UNIQUE**] [**CLUSTERED** | **NONCLUSTERED**]
INDEX 索引名

ON {表名|视图名} (列名 [**ASC** | **DESC**] [,...*n*])
--升序或降序。默认为 ASC。

【例】

CREATE UNIQUE NONCLUSTERED INDEX Idx_st_Dept
ON st_Dept(deptname)

在表st_Dept的deptname列上建立唯一非簇类索引
Idx_st_Dept。

◆索引名在表或视图中要唯一。



索引的建立和删除

!注意:

CREATE INDEX Idx_Test **ON** Test (colm1,colm2)与
CREATE INDEX Idx_Test **ON** Test (colm2,colm1)二者不同。

前者以colm1为主顺序，后者以colm2为主顺序。

2、删除索引

DROP INDEX '表名.索引名|视图名.索引名' [,...n]

【例】

DROP INDEX st_Dept. Idx_st_Dept

视图的建立和删除

1、建立

CREATE VIEW [<数据库名>.] [<用户名>.] 视图名 [(列名
[,...n])] –若不指定列名，则与select保持一致

[WITH < view_attribute > [,...n]]

AS *select_statement* [WITH CHECK OPTION]

--强制视图上执行的所有数据修改语句都必须符合由
select_statement 设置的准则。

CREATE VIEW 创建一个虚拟表，该表以另一种方式表示一个或多个表中的数据。

!CREATE VIEW 必须是查询批处理中的第一条语句。



视图的建立和删除

**【例】 CREATE VIEW TestView1(col1,col2,col3,col4,col5)
AS**

**SELECT * FROM Students WHERE Year(birthday)>1988
GO**

CREATE VIEW TestView2(col1,col2,col3) AS

**SELECT a.st_name,b.deptno,b.deptname FROM Students
a,st_Dept b WHERE a.st_dept=b.deptno**

!给视图指定列名个数必须与SELECT语句列数相同。



视图的建立和删除

视图只有满足下列条件才可更新：

- *select_statement* 在选择列表中没有聚合函数(聚合函数对一组值执行计算并返回单一的值，如AVG, COUNT,SUM,MIN,MAX)，也不包含 TOP、GROUP BY、UNION或 DISTINCT 子句。在 FROM 子句的子查询中允许使用聚合函数。
- *select_statement* 的选择列表中没有派生列。派生列是由任何非简单列表表达式(使用函数或运算符等)所构成的结果集列。
- *select_statement* 中的 FROM 子句至少引用一个表。



视图的建立和删除

【例】 以下视图是不可更新的：

CREATE VIEW TestView **AS**

SELECT GETDATE() **AS** CurrentDate,

@ @LANGUAGE **AS** CurrentLanguage,

CURRENT_USER **AS** CurrentUser

2、删除

DROP VIEW { 视图名 } [,...n]

【例】 **DROP VIEW** TestView

数据修改语言INSERT

- ◆1、INSERT将新行添加到表或视图。

INSERT [**INTO**]{ 表名|视图名} {[(列名列表)]
{ **VALUES**({DEFAULT|NULL|表达式})
| *derived_table*}

- DEFAULT**强制该列值为为该列定义的默认值。

如果对于某列并不存在默认值，并且该列允许 **NULL**，则插入 **NULL**。

- derived_table* 任何有效的 **SELECT** 语句



INSERT

Students(st_id,st_name,pwd,birthday,st_dept)
st_Dept(deptno,deptname,fdtime)

【例】

INSERT st_Dept **VALUES**(184, 'AUTOMATIC CONTROL',
DEFAULT)

INSERT Students **VALUES**(2000184013, '王菲', **DEFAULT**,
'1976-01-01', 184)



上面两句之间用不用加上GO语句？

!如果要在有外部键的表中插入数据，则对应外部键的列值必须在被参照表中存在。

【例】

INSERT Students

SELECT st_id+10,Left(st_name,1)+'燕',pwd,'4-3-1985',st_dept
FROM Students **WHERE** Left(st_name,1)='王'



INSERT

INSERT语句注意事项:

- 主键列不能为空和重复。
- 不允许为空的列必须给予赋值。
- 赋的值要满足约束条件。

数据修改语言UPDATE

◆2、 UPDATE更改表中的现有数据。

UPDATE { 表名 | 视图名 }

SET

列名 = { *expression* | DEFAULT | NULL }

[FROM { < table_source > }] --指定条件时要用到的表

[WHERE < search_condition >]

! FROM 子句中指定的表的别名不能作为 SET *column_name* 子句中要修改的列名的限定符使用。



UPDATE

st_score(st_id,Eng_score,Math_score,AC_score)

students(st_id,st_name,pwd,birthday,st_dept)

courses(course_id,course_name,type,percents,teacher_id)

【例】 将系编号为 '184' 的学生的英语成绩乘以课程难度系数。假设 '0001' 为英语课代码，percents 为课程难度系数。

UPDATE st_score

SET t.Eng_score = t.Eng_score * s.percents

FROM st_score t, courses s

WHERE s.course_id = '0001' **AND** t.st_id **IN**

(**SELECT** students.st_id **FROM** students **WHERE**
students.st_dept = '184')

上面的内容是否有错？

改正： **SET** Eng_score = t.Eng_score * s.percents



数据修改语言DELETE

◆3、DELETE从表中删除行。

DELETE [**FROM**]

{表名 | 视图名}

[**FROM** { < table_source > } [,...*n*]] --指定条件的表

[**WHERE** { < search_condition > }]

【例】**DELETE** authors

? 删除表authors?

- 不带参数使用 **DELETE**将删除表中所有行。



DELETE

titles(title_id,title,pub_id,type)

titleauthor(title_id,au_id)

authors(au_id,name,address)

【例】 **DELETE FROM** titleauthor
WHERE title_id **IN** (**SELECT** title_id **FROM** titles
WHERE title **LIKE** '%computers%')

【例】 **DELETE** titleauthor
FROM titleauthor **AS** a **INNER JOIN** titles **AS** b
ON a.title_id=b.title_id
WHERE titles.title **LIKE** '%computers%'

二者等价。

【例】 **DELETE** authors
FROM (**SELECT** TOP 10 * **FROM** authors) **AS** t1
WHERE authors.au_id = t1.au_id



数据查询语言SELECT

◆基本语法:

<u>SELECT</u> <i>select list</i>	--要查询的内容
[INTO <i>new_table</i>]	--将结果存放到一个新表
<u>FROM</u> <i>table_source</i>	--指定所使用的表、视图
[WHERE <i>search_condition</i>]	--查询的条件
[GROUP BY <i>group_by_expression</i>]	--分组
[ORDER BY <i>order_expression</i> [ASC DESC]]	--排序



SELECT子句

◆ **SELECT**[ALL|DISTINCT][TOP n [PERCENT]]

<select_list>

- 缺省为ALL。空值认为相等。
- TOP n [PERCENT]只从查询结果中输出前 n 行或前 $n\%$ 行。
- 带 PERCENT 时， n 必须是介于 0 和 100 之间的整数。
- 如果查询包含 ORDER BY 子句，将输出由 ORDER BY 子句排序的前 n 行(或前百分之 n 行)。

【例】 **SELECT DISTINCT TOP 10** Eng_score
 FROM st_score
 ORDER BY Eng_score **ASC**

输出最低的10种英语成绩的数值。



AS子句

◆ AS 子句可用于为结果集列指定不同的名称或别名。

【例】 **SELECT** EmpSSN **AS** "Employee Social Security Number" **FROM** EmpTable

【例】 st_score(st_id,Eng_score,Math_score,AC_score)
查询各个年级的英语平均成绩。

```
SELECT Left(st_id,4) AS grade# ,  
Sum(Eng_score)/Count(Eng_score) AS Avg_Eng  
FROM st_score  
GROUP BY Left(st_id,4) --GROUP BY子句不能利用grade#  
ORDER BY grade# ASC
```



UNION子句

◆可以在多个查询之间使用 **UNION** 运算符，以将查询的结果组合成单个结果集，

- 该结果集显示的列名是前一个SELECT的列名。
- 该结果集包含联合查询中的所有查询的全部行。
- 如果UNION连接的两个查询结果列不一致，则会将后一个查询结果转换成前一个查询结果的列的数据类型。

!如果两表列数不同，则不能用UNION。

【例】 **SELECT * FROM TestStu2 UNION SELECT * FROM TestStu1**

和

SELECT * FROM TestStu1 UNION SELECT * FROM TestStu2

不同。



SELECT 示例

Customers(Cust_Name,Country,City,Phone)

SouthAmericanCustomers (Cust_Name,Country,City,Phone)

【例】

SELECT Cust_Name,Country,City,Phone

INTO #CustomerResults

FROM Customers

WHERE Country **IN** ('USA','Canada') --北美记录

UNION

SELECT Cust_Name,Country,City,Phone

FROM SouthAmericanCustomers

INTO 子句指定名为 CustomerResults 的表包含由 Customers 和 SouthAmericanCustomers 表中指定列的并集组成的最终结果集。



SELECT 示例

st_score(st_id,Eng_score,Math_score,AC_score)

students(st_id,st_name,pwd,birthday,st_dept)

【例】 查询最高英语成绩的学生学号和英语成绩。

```
SELECT st_id,Eng_score  
FROM st_score  
WHERE Eng_score = (SELECT  
                Max(Eng_score) FROM st_score )  
ORDER BY st_id
```



SELECT 示例

titles(title_id,ytd_sales)

titleauthor(title_id,au_id)

authors(au_id,au_fname,au_lname)

【例】 **SELECT** ytd_sales **AS** Sales,
authors.au_fname + ' ' + authors.au_lname **AS** Author
INTO #Tmp_count

FROM titles **INNER JOIN** titleauthor **ON** titles.title_id =
titleauthor.title_id **INNER JOIN** authors **ON**
titleauthor.au_id = authors.au_id

ORDER BY Sales **DESC**, Author **ASC**



SELECT 示例

st_score(st_id,Eng_score,Math_score,AC_score)

students(st_id,st_name,pwd,birthday,st_dept)

【例】 查询英语成绩为100的学生的名字并存放在临时表 #highscore 中。

```
SELECT DISTINCT st_name INTO #highscore
FROM students
WHERE 100 IN (SELECT Eng_score FROM st_score
WHERE st_score.st_id = students.st_id)
```

```
【例】 SELECT DISTINCT st_name INTO #highscore
FROM students
WHERE st_id IN (SELECT st_id FROM st_score WHERE
st_score.Eng_score = 100)
```

```
【例】 SELECT DISTINCT st_name INTO #highscore
FROM students a,st_score b WHERE
a.st_id = b.st_id AND b.Eng_score = 100
```



SELECT 示例

customers(cust_name,address,postcode,acc_id)

accouts(acc_id,type,balance,fdtime)

【例】 查询存款类型存在check类型的银行客户名称。

SELECT DISTINCT cust_name

FROM customers

WHERE EXISTS (**SELECT** * **FROM** accouts **WHERE**

acc_id = customers.acc_id **AND** type = 'check')

【例】 **SELECT DISTINCT** cust_name

FROM customers

WHERE acc_id **IN** (**SELECT** acc_id **FROM** accouts

WHERE type = 'check')

【例】 **SELECT DISTINCT** cust_name

FROM customers **AS** a

INNER JOIN accouts **AS** b **ON** a. acc_id = b. acc_id

WHERE b. type = 'check'



SELECT步骤总结

- I.** 分析所要查询的内容和条件分布在哪些表中
- II.** 在SELECT子句中写要查询的内容
- III.** 在FROM子句中写所涉及的表
- IV.** 在WHERE子句中写所有的条件
 - 注意：利用“表名.”引用该表的列名。
 - 若要分组查询则加入GROUP BY子句。
 - 若要排序则利用ORDER BY子句。
 - 若要将查询结果保存在一个临时表中则利用INTO子句。



SELECT 示例

st_score(st_id,Eng_score,Math_score,AC_score)

st_highscore(st_id,score,course_id)

courses(course_id,course_name,type,percents,teacher_id)

【例】 查询各科成绩最高的学生的学号并将相关信息保存在 st_highscore 表中。

INSERT st_highscore (st_id,score,course_id)

SELECT a.st_id,a.Eng_score,b.course_id

FROM st_score a, courses b

WHERE Eng_score = (**SELECT** Max(Eng_score)

FROM st_score) **AND** b.course_name = '英语'

类似将其他课程的进行处理。

? 若为 st_score(st_id,course_id,score) 该如何处理?



SELECT 示例

三表联合查询

st_highscore(st_id,score,course_id)

students(st_id,st_name,pwd,birthday,st_dept)

courses(course_id,course_name,type,percents,teacher_id)

【例】 查询在 ‘import’类型的课程中考试过最高分的学生名字。

SELECT st_name

FROM students a, st_highscore b, courses c

WHERE a.st_id = b.st_id **AND** b.course_id = c.course_id **AND**
c.type = 'import'

SELECT st_name

FROM students

WHERE st_id **IN** (**SELECT** st_id **FROM** st_highscore **WHERE**
course_id **IN** (**SELECT** course_id **FROM** courses **WHERE** type
= 'import'))



为什么用存储过程

在利用数据库软件开发工具开发应用程序时，当客户端与数据库服务器通信时，会牵涉到更多因素。

首先客户端数据通过多层网络软件；其次，在服务器解释消息前也必须通过多层网络软件；服务器接受请求后，还必须对SQL语句进行解释、优化、执行；而执行结果在返回客户端的过程中同样需要两端的多层网络软件。

而且，客户端应用程序向服务器请求数据时，若返回结果集过大，也同样要花费过多的时间。

客户/服务器方式下的应用程序中，客户机应用通常包括多个SQL语句进程，用以完成相应的数据检索、数据更新等任务。如果每个数据库请求均以单个SQL语句的形式发送给服务器，则客户机和服务器之间的数据传送量显然大大增加。



为什么用存储过程

可以将多个SQL语句集合在一起形成存储过程，使用存储过程有如下好处：

- 性能：因为存储过程是在服务器上运行的，而服务器通常是一种功能更加强大的机器，它的执行时间要比在工作站中的执行时间短。

另外，由于数据库信息已经物理地在同一系统中准备好，因此就不必等待记录通过网络传递进行处理。相反，存储过程具有对数据库的立即地、准备好地访问，这使得信息处理极为迅速。

- 客户/服务器开发益处：存储过程可以在服务器上进行开发，这样，将客户方和服务器方的开发任务分离，可减少完成项目需要的时间。用户可以独立开发服务器方组件而不涉及客户方，但可在客户应用程序间重复使用服务器方组件。



为什么用存储过程

- **安全性：**存储过程不仅可以改善效能，而且可以作为安全机制，因为用户可以被授予执行一个存储过程的许可权，即使他在存储过程中引用的表或视图上没有许可权。
- **存储过程在第一次运行时被编译，并存储在当前数据库的系统表中。**当存储过程被编译时，它们被优化以选择访问表信息的最佳路径，这一优化要考虑到表中数据的实际形式、索引是否可用、表负载等一些因素。这些编译后的存储过程可大大加强系统的性能。



存储过程的建立

```
CREATE PROC[EDURE] 存储过程名 [;number]  
[{@ 参数名 数据类型} [=default] [OUTPUT] [...n]  
[WITH{RECOMPILE, ENCRYPTION}]
```

AS *sql_statement*

参数含义：

- *number* 是可选的整数，用来对同名的过程分组，以便使用一条 **DROP PROCEDURE** 语句即可将同组的过程一起除去。

【例】 存储过程可以命名为 **orderproc;1**、**orderproc;2** 等。
DROP PROCEDURE orderproc 语句将除去整个组。

- @参数名：每个存储过程的参数仅用于该过程本身。相同的参数名称可以用在其它过程中。默认情况下，参数只能代替常量，而不能用于代替表名、列名或其它数据库对象的名称。用户必须在执行过程时提供每个所声明参数的值（除非定义了该参数的默认值）。存储过程最多可以有 2,100 个参数。



存储过程的建立

- *default* 参数的默认值。如果定义了默认值，不必指定该参数的值即可执行过程。默认值必须是常量或 **NULL**。
- **OUTPUT** 表明参数是返回参数。该选项的值可以返回给 **EXEC[UTE]**。
- **RECOMPILE** 表明 SQL Server 不会将该过程的计划放入缓存，该过程将在运行时重新编译。
- **ENCRYPTION** 表示 SQL Server 加密 **syscomments** 表中包含 **CREATE PROCEDURE** 语句文本的条目。
- **AS** 指定过程要执行的操作。

注：在存储过程的定义中，可以使用 **RAISERROR** 函数返回用户定义的错误信息并设系统标志，记录发生错误。



存储过程的执行

EXEC[UTE]{[@return_status =]{存储过程名
[;number]|@procedure_name_var}{[@存储过程的参数
=]{value|@variable[**OUTPUT**]][**DEFAULT**]}

- **@return_status** 可选的整型变量，保存存储过程的返回状态。该变量必须提前被声明。
- **@procedure_name_var** 局部定义变量名，代表已存在的存储过程名称。
- **OUTPUT** 指定存储过程必须返回一个参数。该存储过程的匹配参数也必须由关键字 **OUTPUT** 创建。

如果使用 **OUTPUT** 参数，则参数值必须作为变量传递（即 **@存储过程的参数=@variable**）。



存储过程的删除

DROP PROCEDURE

从当前数据库中删除一个或多个存储过程或过程组。

DROP PROCEDURE { *procedure* } [,...*n*]

参数

procedure 是要删除的存储过程或存储过程组的名称。



存储过程示例

st_score(st_id,course_id,score)

【例】 CREATE PROCEDURE prc_tot_score @grade int,
@course_id string, @avg_score int OUTPUT

AS

SELECT @avg_score=AVG(score)

FROM st_score

WHERE Left(st_id,4)=@grade and course_id=@course_id

执行：假设0101为英语课程代码，要查找2001级英语课程的平均成绩。

DECLARE @tot_amt int

EXECUTE prc_tot_score 2001,'0101',@tot_amt output

SELECT @tot_amt



存储过程示例

用户信息表userinfo(id int(4) not null,
fullname varchar(50) not null,
password varchar(20) not null,
nickname varchar(50) not null)
编写存储过程检验用户是否存在。

【例】 CREATE PROCEDURE usercheck @infullname
varchar(50), @inpassword varchar(50), @outcheck char(3)
OUTPUT

AS

if exists(select * from userinfo where fullname=@infullname
and password=@inpassword)
select @outcheck='yes'
else
select @outcheck='no'



存储过程示例

【例】

```
CREATE PROCEDURE titles_sum @@TITLE  
varchar(40)='% ', @@SUM money OUTPUT
```

```
AS
```

```
SELECT 'Title Name' = title
```

```
FROM titles WHERE title LIKE @@TITLE
```

```
SELECT @@SUM = SUM(price)
```

```
FROM titles WHERE title LIKE @@TITLE
```

```
GO
```



存储过程示例

```
CREATE PROCEDURE au_info2 @lastname varchar(30) =  
'D%', @firstname varchar(18) = '%' AS
```

```
SELECT au_lname, au_fname, title, pub_name
```

```
FROM authors a INNER JOIN titleauthor ta ON a.au_id =  
ta.au_id INNER JOIN titles t ON t.title_id = ta.title_id  
INNER JOIN publishers p ON t.pub_id = p.pub_id
```

```
WHERE au_fname LIKE @firstname AND au_lname LIKE  
@lastname
```

```
GO
```

```
EXECUTE au_info2
```

```
EXECUTE au_info2 'Wh%'
```

```
EXECUTE au_info2 @firstname = 'A%'
```

```
EXECUTE au_info2 'Hunter', 'Sheryl'
```

```
EXECUTE au_info2 'H%', 'S%'
```



为什么用触发器

◆定义：触发器就是对某一个表的一定的操作触发某种条件，从而执行的一段程序。触发器是一个特殊的存储过程。

◆常见的触发器有三种：分别应用于Insert、Update、Delete事件。

?为什么要使用触发器？

【例】Student(StudentID,...)

BorrowRecord(BorrowRecord, StudentID, BorrowDate, ReturnDate) --学生借书记录表(流水号,学号,借出时间,归还时间)

用到的功能有：

- 1.如果更改了学生的学号，希望他的借书记录仍然与这个学生相关（也就是同时更改借书记录表的学号）；
- 2.如果该学生已经毕业，希望删除他的学号的同时，也删除它的借书记录。



触发器的建立

```
CREATE TRIGGER trigger_name ON { table | view }  
[ WITH ENCRYPTION ]  
{ { FOR | AFTER | INSTEAD OF } {[DELETE] [,]  
[INSERT] [, ] [UPDATE] }  
[ NOT FOR REPLICATION ]  
AS  
{ IF UPDATE ( column )  
  [ { AND | OR } UPDATE ( column ) ]  
  [ ...n ] }  
sql_statement [ ...n ] } }
```



触发器的建立-注意事项

- **CREATE TRIGGER** 必须是批处理中的第一条语句，并且只能应用到一个表中。
- 触发器只能在当前的数据库中创建，不过触发器可以引用当前数据库的外部对象。
- 如果指定触发器所有者名称以限定触发器，请以相同的方式限定表名。
- 在同一条 **CREATE TRIGGER** 语句中，可以为多种用户操作（如 **INSERT** 和 **UPDATE**）定义相同的触发器操作。
- 当触发器激发时，将向调用应用程序返回结果。若要避免返回结果，请不要包含返回结果的 **SELECT** 语句，也不要再在触发器中进行变量赋值的语句。
- 触发器中不允许进行 **DATABASE** 的建立、修改和删除操作。



触发器的建立-例

Student(StudentID,....)

BorrowRecord(BorrowRecord, StudentID, BorrowDate,
ReturnDate)

CREATE TRIGGER truStudent **ON** Student **FOR** UPDATE
AS

IF Update(StudentID)

BEGIN


UPDATE BorrowRecord

SET StudentID=i.StudentID

FROM BorrowRecord br, **Deleted** d, **Inserted** i

WHERE br.StudentID=d.StudentID

END

 **Deleted**与**Inserted**分别表示触发事件的表“旧的一条记录”和“新的一条记录”。



触发器的建立-例

Student(StudentID,....)

BorrowRecord(BorrowRecord, StudentID, BorrowDate,
ReturnDate)

CREATE TRIGGER trdStudent **ON** Student **FOR DELETE**
AS

DELETE BorrowRecord
FROM BorrowRecord br, Deleted d
WHERE br.StudentID=d.StudentID



触发器的修改

```
ALTER TRIGGER trigger_name ON ( table | view )  
[ WITH ENCRYPTION ]  
{ ( FOR | AFTER | INSTEAD OF ) { [ INSERT ] [ , ]  
[ UPDATE ] }  
[ NOT FOR REPLICATION ]  
AS  
    { IF UPDATE ( column ) [ { AND | OR }  
UPDATE( column ) ] [ ...n ] }  
    sql_statement [ ...n ] }
```



触发器的修改-例

```
CREATE TRIGGER royalty_reminder ON roysched  
WITH ENCRYPTION  
FOR INSERT, UPDATE  
AS RAISERROR (50009, 16, 10)
```

-- Now, alter the trigger.

```
ALTER TRIGGER royalty_reminder ON roysched FOR  
INSERT AS  
RAISERROR (50009, 16, 10)
```

--消息 50009 是 sysmessages 中的用户定义消息。



触发器的删除

DROP TRIGGER { *trigger* } [,...*n*]

【例】

**IF EXISTS (SELECT name FROM sysobjects WHERE
name = 'employee_insupd' AND type = 'TR')**

DROP TRIGGER employee_insupd

GO

触发器的建立

- **AFTER**: 指定触发器只有在触发 SQL 语句中指定的所有操作都已成功执行后才激发。默认为 **AFTER**。

- ! 不能在视图上定义 **AFTER** 触发器。

- **INSTEAD OF**: 指定执行触发器而不是执行触发 SQL 语句，从而替代触发语句的操作。

在表或视图上，每个 **INSERT**、**UPDATE** 或 **DELETE** 语句最多可以定义一个 **INSTEAD OF** 触发器。

- ! 不能在 **WITH CHECK OPTION** 的可更新视图上定义 **INSTEAD OF** 触发器。

触发器的建立

- { [DELETE] [,] [INSERT] [,] [UPDATE] }

指定在表或视图上执行哪些数据修改语句时将激活触发器。

必须至少指定一个选项。在触发器定义中允许使用以任意顺序组合的这些关键字。如果指定的选项多于一个，需用逗号分隔这些选项。

触发器的建立

- **NOT FOR REPLICATION**

当复制进程更改触发器所涉及的表时，不应执行该触发器。

触发器的建立

- **IF UPDATE** (*column*)

测试在指定的列上进行的 **INSERT** 或 **UPDATE** 操作，不能用于 **DELETE** 操作。

可以指定多列。

因为在 **ON** 子句中指定了表名，所以在 **IF UPDATE** 子句中的列名前不要包含表名。


触发器的建立

- **sql_statement**: 触发器的条件和操作。触发器条件指定其它准则，以确定 **DELETE**、**INSERT** 或 **UPDATE** 语句是否导致执行触发器操作。

触发器可以包含任意数量和种类的 Transact-SQL 语句。

触发器旨在根据数据修改语句检查或更改数据；它不应将数据返回给用户。触发器中的 Transact-SQL 语句常常包含控制流语言。

CREATE TRIGGER 语句中可使用 **deleted** 和 **inserted** 表，它们是逻辑（概念）表。这些表在结构上类似于定义触发器的表（也就是在其中尝试用户操作的表）；这些表用于保存用户操作可能更改的行的旧值或新值。

 一个 **Update** 的过程可以看作为：生成新的记录到 **Inserted** 表，复制旧的记录到 **Deleted** 表，然后删除 **Student** 记录并写入新纪录。



游标的使用

◆游标：就是像高级语言一样，是存放数据集，并逐条访问的一种机制。

◆游标使用的五步：

第一步：声明游标。两种形式：

1. **DECLARE** cursor_name [INSENSITIVE] [SCROLL]
CURSOR
FOR select_statement
[FOR {READ ONLY | UPDATE }[[OF column_list]]]
2. **DECLARE** cursor_name **CURSOR**[LOCAL | GLOBAL]
[FORWARD_ONLY | SCROLL] [STATIC | KEYSET |
DYNAMIC]
[READ_ONLY | SCROLL_LOCKS | OPTIMISTIC]
FOR select_statement
[FOR {READ ONLY | UPDATE }[[OF column_list]]]



游标的使用

1. **DECLARE** cursor_name [INSENSITIVE] [SCROLL]
CURSOR FOR select_statement
[FOR {READ ONLY | UPDATE }[OF column_list]]

INSENSITIVE关键字指明要为检索到的结果集建立一个临时拷贝，以后的数据从这个临时拷贝中获取。如果在后来游标处理的过程中，原表中数据发生了改变，对于该游标而言是不可见的。这种游标不允许数据更改。

SCROLL指明游标可以在任意方向上滚动。所有的fetch选项（**first**、**last**、**next**、**relative**、**absolute**）都可以使用。缺省则游标只能向前滚动（**next**）。

Select_statement指明SQL语句建立的结果集。不允许使用**COMPUTE**、**COMPUTE BY**、**FOR BROWSE**和**INTO**。

READ ONLY指明不允许数据修改。**UPDATE**则可以修改。

OF column_list指明结果集中可以进行修改的列。使用**UPDATE**关键字则所有的列都可进行修改。



游标的使用

2. **DECLARE** cursor_name **CURSOR**[LOCAL | GLOBAL]
[FORWARD_ONLY | SCROLL] [STATIC | KEYSET |
DYNAMIC] [READ_ONLY | SCROLL_LOCKS |
OPTIMISTIC] **FOR** select_statement
[FOR {READ ONLY | UPDATE }[OF column_list]]

LOCAL指明游标是局部的，只能在它所声明的过程中使用。
GLOBAL游标对连接全局可见。当连接结束时游标才不可用。
FORWARD_ONLY指明游标只能向前滚动。
STATIC与**INSENSITIVE**作用相同。**KEYSET**创建键集驱动游标。**DYNAMIC**指明游标将反映所有对结果集的修改。缺省为**DYNAMIC**。
SCROLL_LOCK表明，为了保证游标操作的成功，而对修改或删除行进行加锁。**OPTIMISTIC**则不加锁。



游标的使用

◆游标使用的五步:

第一步：声明游标

第二步：打开游标

OPEN{{[GLOBAL]cursor_name}|cursor_variable_name}

Cursor_variable_name是所引用游标的变量名。

在游标被打开后，系统变量@@cursor_rows可以用来检测结果集的行数。@@cursor_rows为负数时，表示游标正在被异步迁移，其绝对值为当前结果集的行数。



游标的使用

◆游标使用的五步:

第一步: 声明游标

第二步: 打开游标

第三步: 从游标中取值

FETCH[NEXT|PRIOR|FIRST|LAST|ABSOLUTE {n|@nvar}|
RELATIVE {n|@nvar}] **FROM**[GLOBAL]{cursor_name
|cursor_variable_name}
[INTO @variable_name][,.....n]]

ABSOLUTE n:取结果集中的第n行, n为0不能得到任何行。

RELATIVE n:取出当前行的前n行或后n行。值为正则取出的行在当前行前n行的位置上, 如果该值为负数, 则返回当前行的后n行。

FETCH执行状态存储在系统变量@@fetch_status中。如果**FETCH**成功, 则@@fetch_status为0。



游标的使用

◆游标使用的五步：

第一步：声明游标

第二步：打开游标

第三步：从游标中取值

第四步：关闭游标

CLOSE [GLOBAL]cursor_name|cursor_variable_name

游标关闭之后，不能再执行FETCH操作。如果还需要使用
FETCH语句，则要重新打开游标。

第五步：释放游标

DEALLOCATE

[GLOBAL]cursor_name|cursor_variable_name



游标的使用

【例】 对于表BorrowRecord(BorrowRecord, StudentID, StudentFeeID, BorrowDate, ReturnDate, Fee) --学生借书记录表(流水号, 学号, 费用结算号（外键）, 借出时间, 归还时间, 借书费用)

StudentFee(StudentFeeID, StudentID, BorrowBookAllFee) --学生费用结算表(费用结算号（主键）, 学号, 所有借书总费用)

两者关系为多对一的关系，关联字段为StudentFeeID。

由于某种原因StudentFee表的数据遭到了破坏，想将“所有借书总费用”重算。



游标的使用

--声明一个游标

```
DECLARE curStudentFee CURSOR  
FOR
```

```
SELECT StudentFeeID FROM StudentFee
```

--声明两个费用变量

```
DECLARE @mAllFee Money           --总费用  
DECLARE @iFeeID      Int         --借书结算号
```

--初始化

```
SET @mAllFee=0  
SET @iFeeID=0
```

--打开游标

```
OPEN curStudentFee
```



游标的使用

--循环并提取记录

FETCH FIRST FROM curStudentFee **INTO** @iFeeID

WHILE (@@Fetch_Status=0)

BEGIN

--从借书记录中计算某一学生的借书总记录的总费用

SELECT @mAllFee=Sum(Fee)

FROM BorrowRecord

WHERE StudentFeeID=@iFeeID

--更新到汇总表。

UPDATE StudentFee **SET** BorrowBookAllFee=@mAllFee

WHERE StudentFeeID=@iFeeID

FETCH NEXT FROM curStudentFee **INTO** @ iFeeID

END



游标的使用

--关闭游标

CLOSE curStudentFee

--释放游标

DEALLOCATE curStudentFee

! 游标的使用包括声明、打开、关闭、释放。

! @@Fetch_Status 游标提取状态标志，0表示正确。

【例】假设有学生成绩表和各系英语成绩汇总表，可用游标将学生英语成绩逐个汇总到各系英语成绩汇总表。

◆ 游标占用内存，并且会锁定表，效率不高。

执行一次Fetch相当于执行一次SELECT。