SQL脚本语言

批处理

- 批处理是包含一个或多个SQL语句的组，从应用程序一次性发送到服务器执行，服务器将批处理语句编译成一个可执行单元

批处理中的错误处理

- 编译错误使执行计划无法编译，从而导致批处理中的任何语句均无法执行
- 大多数运行时错误将停止执行当前语句和其后的语句
- 少数运行时错误(如违反约束)仅停止执行当前语句，而继续执行批处理中其它所有语句
- 如果批处理第二条语句在执行时失败，则第一条语句的结果不受影响，因为它已经执行

批处理中的错误处理

Batch 1

```
select * from S
select * fom SC
```

编译错误

Batch 2

```
select * from S
select * from CS
```

延迟解析表名、运行时错误

Batch 3

```
select * from CS
select * from S
```

延迟解析表名、运行时错误

Batch 4

```
select * from S
select G from SC
```

编译错误

局部变量

局部变量是可以保存特定类型的单个数据值的对象

变量通常用于：

- 作为计数器计算循环执行的次数
- 保存数据值以供控制流语句测试
- 保存由存储过程返回代码返回的数据值

变量的声明方式：**declare** @变量名称 数据类型

变量的赋值方式：**set** @变量名称 = SQL表达式

局部变量

```
declare  @find  varchar(10)
set  @find = '张%'
select     sno, sname, age
from       student
where      sname  like  @find
```

```
declare  @rows  int

set  @rows = ( select  count(*)  from  student )
```

控制流：SQL Server

控制流关键字：用于控制T-SQL语句、语句块和存储过程的执行流，使语句互相连接、关联和依存

begin...end	*waitfor*
goto	*while*
if...else	*break*
Return	*continue*

控制流：MySQL

控制流关键字：用于控制T-SQL语句、语句块和存储过程的执行流，使语句互相连接、关联和依存

begin...end	*if else*
case when	*repeat…until*
loop	*while*
leave	*iterate*

错误处理：SQL Server

```
begin try
    { SQL 语句 }
end try
begin catch
    { SQL语句 }
end catch
```

如果try部分中的SQL语句出现错误，则转入catch部分进行相应的错误处理

- error_number( )：返回错误号
- error_severity( )：返回严重性级别
- error_state( )：返回错误状态号
- error_procedure( )：返回发生错误的存储过程或触发器的名称
- error_line( )：返回例程中导致错误的行号
- error_message( )：返回描述错误的完整文本信息

SQL Server错误处理示例

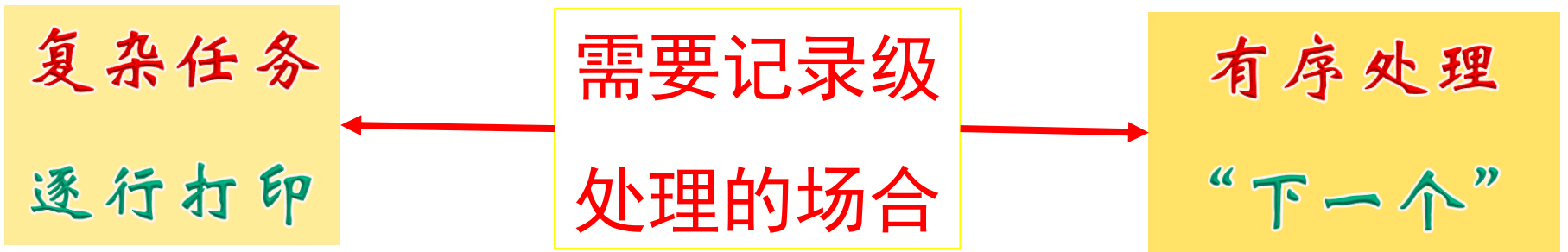
```
begin try
    select      1/0
end try
begin catch
select          error_number( )      as  ErrorNumber,
                error_severity( )    as  ErrorSeverity,
                error_state( )       as  ErrorState,
                error_procedure( )   as  ErrorProcedure,
                error_line( )        as  ErrorLine,
                error_message( )     as  ErrorMessage
end catch
```

ErrorNumber	ErrorSeverity	ErrorState	ErrorProcedure	ErrorLine	ErrorMessage
8134	16	1	NULL	2	Divide by zero error encountered.

SQL与过程化执行方式的差别

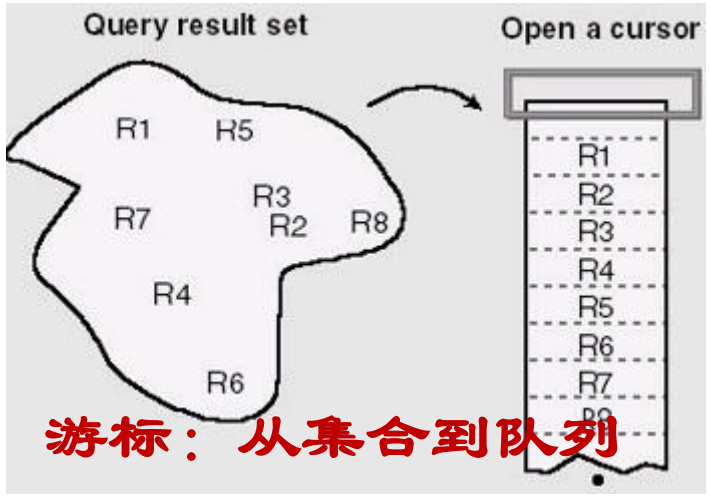
阻抗失配 impedance mismatch

- SQL：一次一集合
- 过程化执行：一次一记录



游标

- 游标：在查询结果的记录集合中移动的指针
- 需要游标的数据操作：当select语句的结果中包含多行时，使用游标可以逐个存取这些行



游标：从集合到队列

- 活动集：select语句返回的行的集合
- 当前行：活动集中当前处理的那一行

  游标即是指向当前行的指针

游标分类

- **滚动游标**：游标位置可以来回移动，可在活动集中取任意行
- **非滚动游标**：只能在活动集中顺序地取下一行
- **更新游标**：系统对游标指向的当前行加锁，当程序读下一行数据时，本行数据解锁，下一行数据加锁

完整的使用游标的过程

游标定义语句

Declare：定义一个游标，使之对应一个**select**语句

declare 游标名 [**insensitive**] [**scroll**] **cursor for**

select语句 [**for update** [**of**列表名]]

- **insensitive**：创建游标使用的数据临时复本，对游标进行提取操作时返回的数据不反映对基表所做的修改
- 若省略**insensitive**，对基表提交的更新都会反映在后面的提取中
- **for update**：表示该游标可用于对当前行的修改与删除

游标定义语句

open 游标名

- 打开一个游标，执行游标对应的查询
- 结果集合为该游标的活动集

fetch [**next** | **prior** | **first** | **last** | **current** | **relative n** | **absolute m**]

游标名 **into** [局部变量]

在活动集中将游标移到特定的行，取出该行数据放到相应的宿主变量中

游标定义语句

close 游标名

- 关闭游标，释放活动集及其所占资源
- 需要再使用该游标时，执行open语句

free 游标名

- 删除游标，以后不能再对该游标执行open语句

使用游标的例子：逐行更新

```
declare  my_curs  cursor for
        select  *  from  SC   where  cno = 'c1'
for update
open  my_curs
while  @@fetch_status = 0
        begin
                update  SC
                set  grade = grade *1.05
                where current of  my_curs
                fetch next from  my_curs
        end
close  my_curs
deallocate  my_curs
```

动态SQL：产生SQL的SQL

根据用户的输入构造SQL执行代码

用户输入

student_id = ‘s01’

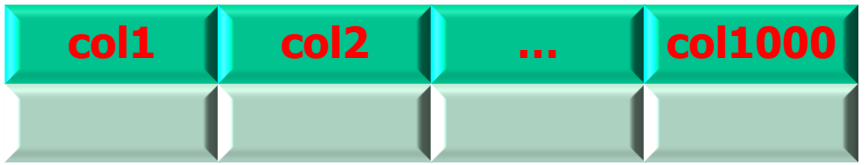
动态SQL

```
prepare  sqlState

from  “delete from  student  where  sno = ? "

execute  sqlState  using : student_id
```

delete from student **where** sno = ‘s01’

动态SQL：创建一个1000列的宽表

```
declare  i  int = 1,  sqlState text = ' '
set  sqlState = ' create table  wideTb('
    while  i < 1000
    begin
        set  sqlState = sqlState + 'col' + cast( i as varchar(10) )
                      +  char(13) + 'int' + ','
        set    i = i+1
    end
set  sqlState = sqlState + 'col1000 int )'
execute  sqlState
```



动态SQL：购物页面的动态筛选器

图书馆的检索界面

author	title	date
zhangsan	MySQL	null

call my_choice ('zhangsan', 'MySQL', null)

```
create procedure my_choice (myauthor, mytitle, mydate)
as
select author, title, date
from book
where ( author = myauthor or myauthor is null )
  and ( title = mytitle or mytitle is null )
  and ( date = mydate or mydate is null )
```

```
select author, title, date
from book
where ( author = 'zhangsan' or myauthor is null )
  and ( title = 'MySQL' or mytitle is null )
  and ( mydate is null )
```

弊端：or无法利用索引

动态SQL：动态筛选器

```
set   sql =    'select  author, title, date'
               + 'from  book'
               + 'where  1=1'
               + case when myauthor is not null  then  'and author = myauthor ' end
               + case when mytitle is not null  then  'and  title =mytitle ' end
               + case when mydate is not null  then  'and date = mydate' end ;
execute        sql
```

```
call  my_choice( myauthor, null, null )
```

```
select  author, title, date  from  book
where  1=1 and author = myauthor
```

存储过程与函数

存储过程

存储过程将程序在服务器中预先编译好并存储起来，然后应用程序只需简单地向服务器发出调用该存储过程的请求即可

存储过程的优点

- 执行效率高
- 重复使用
- 统一的操作流程
- 维护业务逻辑
- 安全性

存储过程的执行效率问题

如果存储过程的执行计划是一成不变的，则随着时间推移，该计划有可能变得是次优的、过时的

比如数据分布发生变化，后续创建了索引

随着存储过程输入参数的不同，有可能对应着不同的最优的执行计划

比如一个按性别检索信息的存储过程 **get_info**（@sex），由于数据的倾斜分布，对于输入'男'，最好的执行计划是表扫描，而对于输入'女'，最好的执行计划是利用性别列上的过滤索引

存储过程的命令格式：MySQL

```
create procedure 存储过程名 [参数 数据类型 ]

begin 存储过程SQL体 end
```

```
delimiter $$
create procedure get_Sname s_id varchar(10)
begin
    select sname from student
    where sno = s_id;
end $$
delimiter ; -- 将语句结束符修改回 ";"
```

执行存储过程：call get_Sname 's01'

嵌套存储过程：利用$n! = n * (n-1)!$计算阶乘

```
create procedure  factorial param1 int
as
declare  one_less  int, answer  int
if ( param1 < 0  or  param1 > 12 ) return -1
if ( param1 = 0  or  param1 = 1 ) select  answer=1
else   begin
                        select  one_less = @param1 - 1
                        call  answer = factorial one_less
                        if ( answer= -1 ) return -1
                        select  answer = answer * param1
        end
return ( answer )
```

存储过程练习题

创建一个存储过程 delete_one 操作，每次它删除表中重复行中的一个

例如假定表 T = {1, 2, 2, 3, 3, 3, 4, 4, 4, 4}

对其执行一次 delete_one，T = {2, 3, 3, 4, 4, 4}

再执行一次 delete_one，T = {3, 4, 4}

创建一个存储过程 delete_dup 操作，每次它只保留表中重复行中的一个

例如假定表 T= {1, 2, 2, 3, 3, 3, 4, 4, 4, 4}

对其执行 delete_dup，T = {1, 2, 3, 4}

用户定义函数

用户定义函数：用于封装经常执行的逻辑的子例程

用户定义函数与存储过程的区别

- 存储过程只能返回一个整数值，用户定义函数可以返回各种数据类型值
- 存储过程可以做任何数据库修改，用户定义函数不可以修改数据库
- 存储过程只能由exec来执行，不能用在表达式中，用户定义函数可以由exec来执行，也可以用于表达式中或from子句中
- 存储过程一般用于对数据库修改或设置，用户定义函数则适于提取数据

标量函数

```
create function  function_name
    ( [ parameter_name  data_type ] )
returns  return_data_type
return
    ( function_body )
```

标量函数

```
create function  AverageGrade ( c_number varchar(8) )
returns  int
return
        ( select      avg(grade)
        from          SC
        where         cno = c_number )
```

标量函数的调用

```
select      AverageGrade ( 'c01' )
```

```
select      sno, grade
from        SC
where       grade > AverageGrade( 'c01' )
   and      cno = 'c01'
```

表值函数：SQL Server

```
create function function_name （
    [@parameter_name data_type ] )
    returns  TABLE
return （ select-stmt )
```

表值函数：SQL Server

```
create function  StudentsByClass ( @c_number varchar(8) )
returns table
as
return
    ( select    sno, grade
    from        SC
    where       cno = @c_number )
```

```
select  *  from  StudentsByClass( 'c01' )
```

Apply操作符与表值函数：SQL Server

apply 可以看成是左右两个表的连接操作，右边的表是一个表值函数的返回结果，该表值函数的输入参数是左边表的某个或者几个列

返回每门课程的成绩排在前三的学生

```
create function fn_CourseTop3Grade( @course_no char(8) )
returns table
as
return
(
    select      top(3) sno, grade
    from        SC
    order by    grade desc
)
```

```
select  C.cno, R.sno, R.grade
from    course C cross apply
        fn_CourseTop3Grade( C.cno ) as R
order by C.cno
```

触发器定义

触发器的定义

触发器是一条语句，当对数据库做修改时，它自动被系统执行

- **ECA**：Event-Condition-Action（事件－条件－动作）
- **E**：指明监视哪些事件：Insert、delete、update
- **C**：指明什么条件下触发器被执行
- **A**：指明触发器执行的动作是什么
- 触发器里有两个动作：所监视的动作、所执行的动作

主动：一切习惯中最好的那个

主动数据库：pull vs push

触发器的作用

维护约束

- 防止在选定一门课后删除该课程

辅助缓存数据维护

- 当基础表发生改变时更新物化视图

商业规则

- 向客户发送短信通知其物流信息

监控

- 传感器感知到一氧化碳浓度级别提高，则开启通风系统

简化应用设计

- 将核心编程逻辑从异常处理中分离出来

行级触发器的定义

create trigger trigger-name { **before** | **after** }

{ **insert** | **delete** | **update** [of column-name] } **on** table-name

referencing { **old row as** identifier | **new row as** identifier }

for each row

{ **when**(search-condition)
begin atomic
triggered-SQL-statement
end }

行级触发器示例

emp(eno, ename, salary, job)，要求职工工资增幅不得超过10%

```
create trigger   raise_limit
after update of  salary  on  emp
   referencing new row as  nrow  old row as  orow
   for each row
   when (nrow.salary > 1.1 * orow.salary)
        begin
          print ”Salary increase 10%”
        end
```

行级触发器完成复杂业务处理（教材示例）

当帐户透支时将帐户余额设为0，并建一笔贷款，其金额为透支额

```
create trigger overdraft-trigger after update on account
   referencing new row as nrow for each row
   when nrow.balance < 0
   begin atomic
      insert into loan values( nrow.account-number,
                               nrow.branch-name, – nrow.balance )
      update account set balance = 0
      where account.account-number = nrow.account-number
   end
```

语句级触发器的定义

create trigger trigger-name { **before** | **after** }

{ **insert** | **delete** | **update** [of column-name] } **on** table-name

referencing { **old table as** identifier | **new table as** identifier }

for each statement

{ **when**(search-condition)
begin atomic
triggered-SQL-statement
end }

语句级触发器：聚集值作为监视对象时

emp(eno, ename, salary, job)，职工平均工资不得低于800

```
create trigger AvgSal_limit
after update of salary on emp
referencing new table as n_tb old table as o_tb
for each statement
when ( 800 > ( select avg(SAL) from emp )
   begin
        delete from emp
        where eno in ( select eno from n_tb )
        insert into emp ( select * from o_tb )
   end
```

before触发器：处理违反约束的更新

如果插入的成绩不及格，则将其改为60分

```
create trigger  pass-grade-trigger
before insert on  SC
referencing new row as  nrow
for each row
when ( nrow.grade < 60 )
    begin
            set  nrow.grade  = 60
    end
```

触发器： SQL Server

deleted 和 inserted 是逻辑(概念)表。这些表在结构上类似于定义触发器的表，用于保存用户操作可能更改的行的旧值或新值

```
create trigger  S_SC_Delete
on  student
after delete
as
        if  @@rowcount = 0  return
        delete from   SC
        where  SC.sno = deleted.sno
```

触发器： SQL Server

触发动作影响到多行时使用游标

逐行输出被删除的员工

```
create trigger reminder_trigger on emp
after delete as
        if @@rowcount = 0 return
        declare  @msg varchar(100), @eno char(10), @ename char(10)
        declare  cursorDeleted cursor for select eno, ename from deleted
        open     cursorDeleted
        fetch next from cursorDeleted into @eno, @ename
        while @@fetch_status = 0
        begin
                set      @msg = '被删除的员工主码是:' + @eno +
                                '被删除的员工姓名是:' + @.ename
                print @msg
                fetch next from cursorDeleted into @eno, @ename
        end
        close    cursorDeleted
        deallocate cursorDeleted
```

递归触发器

设计触发器，保证部门预算始终等于该部门预算与其所有子部门预算之和

dept_name	parent_name	budget
d1	d2	10
d2	d3	100
d3	null	500

d1的budget增加10

dept_name	parent_name	budget
d1	d2	20
d2	d3	110
d3	null	510

递归触发器

```
create trigger budget on dept after update
as
if ( select parent_name from inserted ) is null return
update      dept
set         budget = budget + ( select budget from inserted ) –
                              ( select budget from deleted )
where       dept_name = ( select parent_name from inserted )
```

替代触发器

```
create  view  computer_teacher   as
    ( select  tno, tname, salary
      from     teacher T, department D
      where    T.dno = D.dno
       and     D.dname = ' 计算机系' )
```

```
insert into computer_teacher ( 't01', 'tom', 800 )
```

系统的自动转换

```
insert into  teacher ( 't01', 'tom', 800, null, null )
```

我们期望的转换

```
insert into  teacher ( 't01', 'tom', 800, null, '计算机系系号' )
```

替代触发器：按指定意图进行更新

```
create trigger  insert_view  on  computer_teacher

instead of  insert

as

    declare  @d_no  char(10)

    set  @d_no =  ( select  dno  from  department  where  dname = '计算机系' )

    insert into  teacher ( inserted.tno, inserted.tname, inserted.salary, null , @d_no )
```

替代触发器：更新不可更新的视图

```
create view  join_view  as
    select  Table1.a as a1, Table2.a as a2
    from   Table1  join  Table2  on  Table1.a = Table2.a
```

Table1
a
1
2
4

Table2
a
1
2
2
3

join_view	
a1	a2
1	1
2	2
2	2

```
create trigger  delete_join  on  join_view
instead of  delete
as
    delete  Table1    where  a  in  ( select  a1  from  deleted )
    delete  Table2    where  a  in  ( select  a2  from  deleted )
```

```
delete from  join_view
where  a1=2
```

Table1
a
1
4

Table2
a
1
3

课堂小问答

触发器只能监视单个表上的更新事件

如果监视目标涉及到多个表该怎么办？

比如连接或者交集

触发器的冲突

当一个事件同时激活多个触发器时，触发顺序如何确定？

有序冲突解决方案

- 轮流计算触发器的前提条件。当一个条件求值为真时，执行相应的触发器；当执行完成时，考虑下一个触发器

分组冲突解决方案

- 同时计算所有触发器的前提条件，然后调度执行所有前提条件为真的触发器
- 发布-订阅触发器

触发器的冲突

Trigger 1

on	在课程注册表中插入一行
if	超过课程班容量
then	将未满足请求通知注册者

Trigger 2

on	在课程注册表中插入一行
if	超过课程班容量
then	将请求放入等待列表

方案一：设定触发器触发优先级；方案二：合并成一个触发器