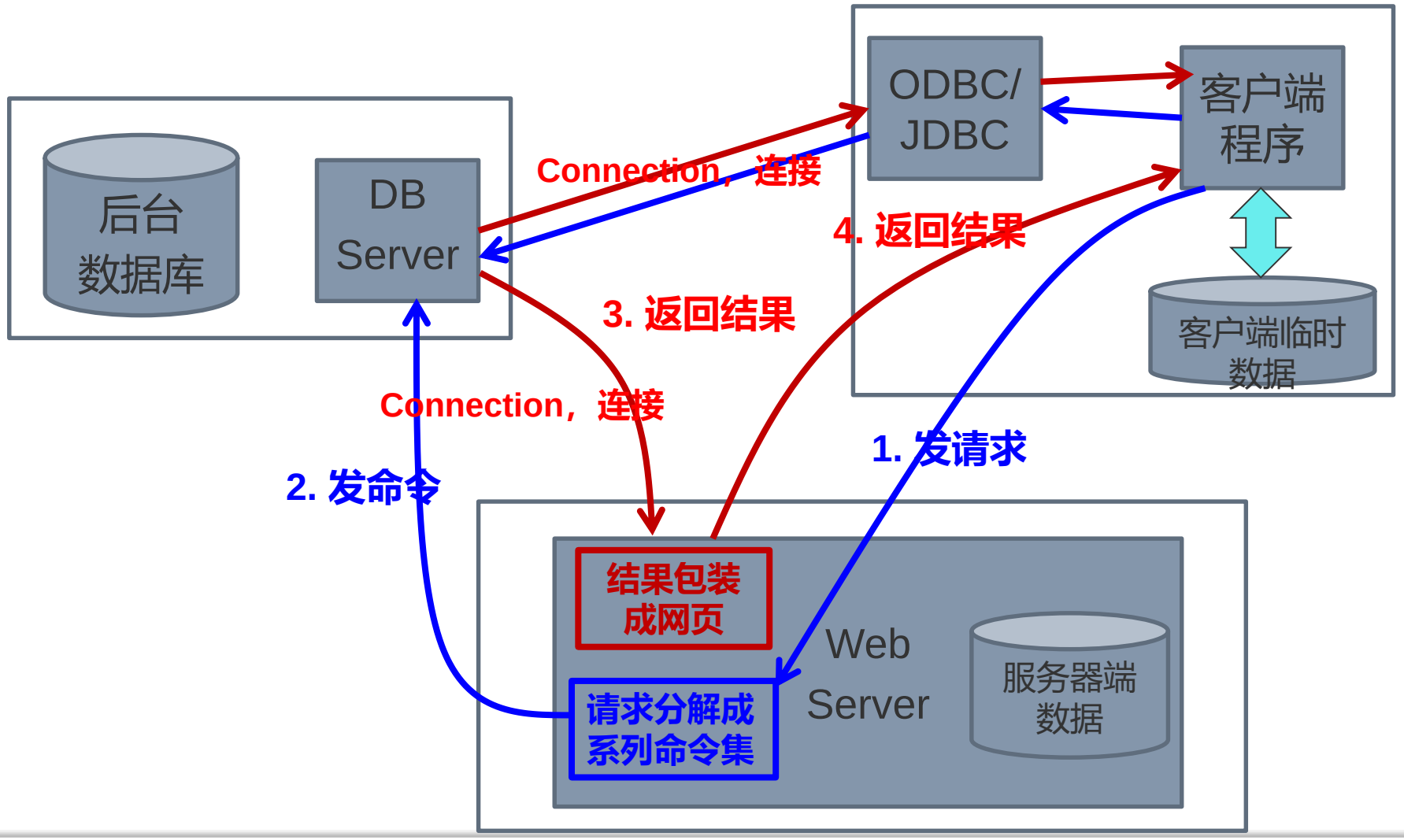
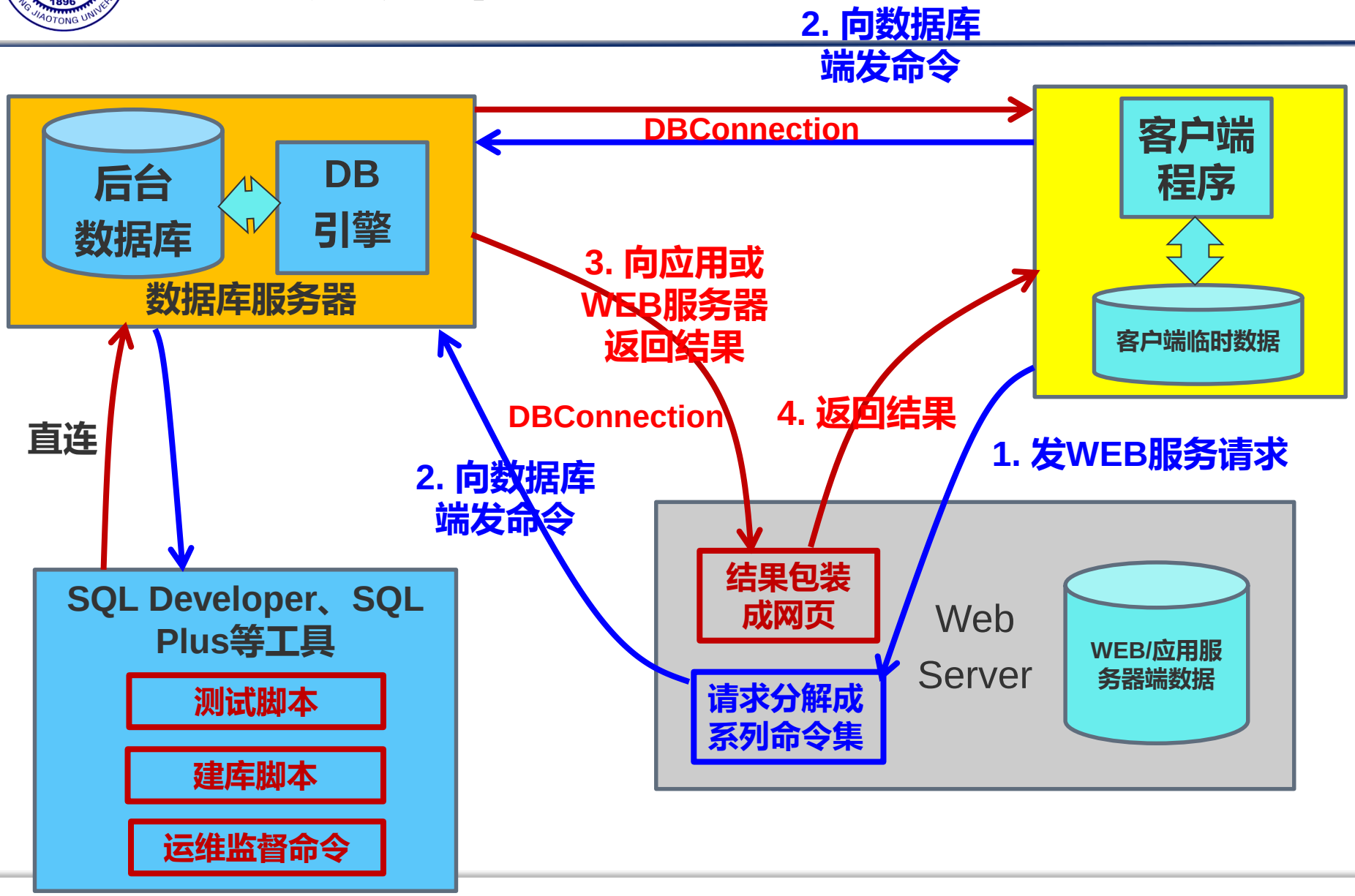
# 第八章 数据库编程

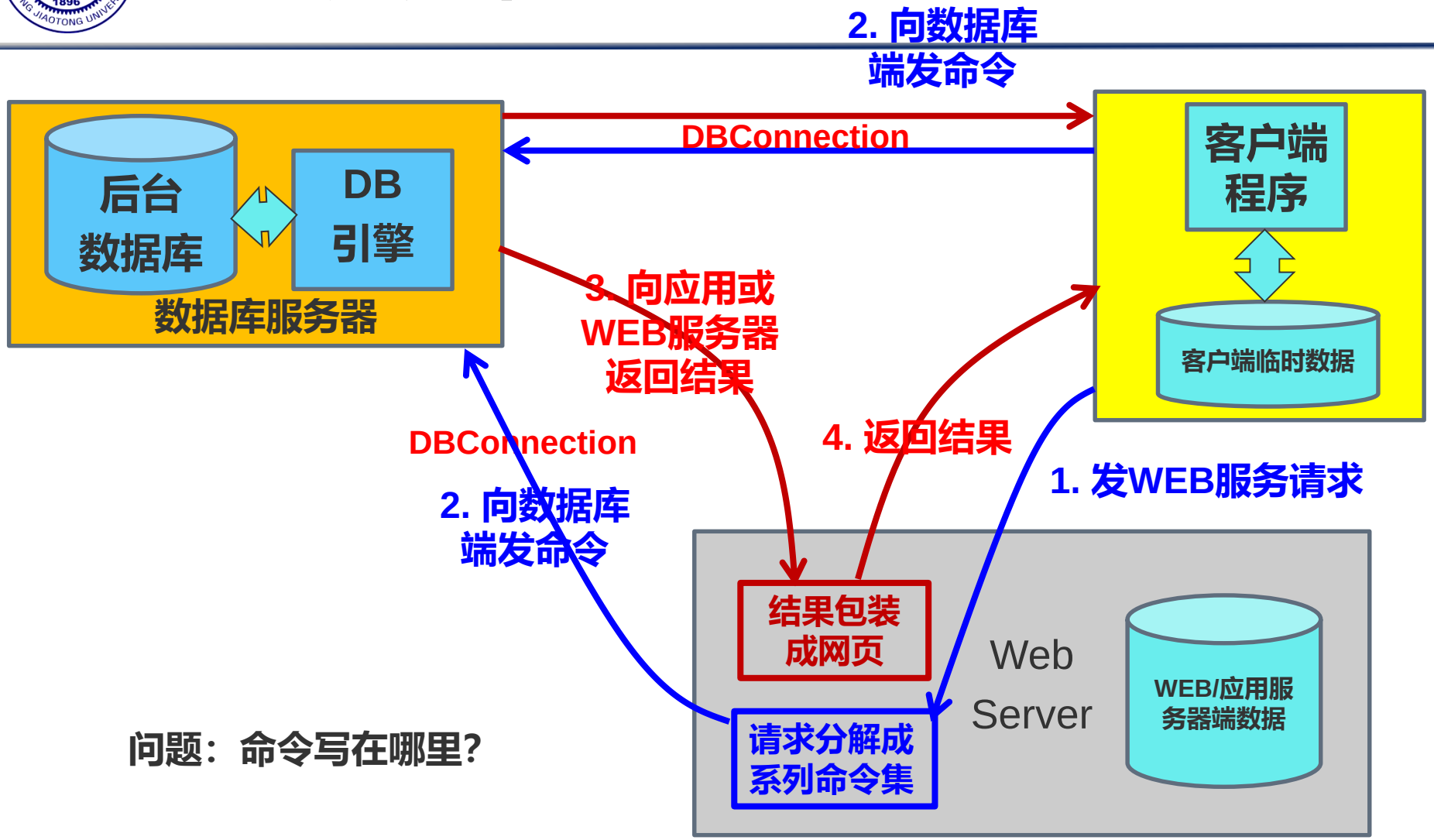
# 数据库系统架构模式与编程



# Web应用架构


2. 向数据库端发命令
DBConnection
后台数据库
DB引擎
数据库服务器
客户端程序
客户端临时数据
3. 向应用或WEB服务器返回结果
DBConnection
4. 返回结果
1. 发WEB服务请求
直连
2. 向数据库端发命令
SQL Developer、SQL Plus等工具
测试脚本
建库脚本
运维监督命令
结果包装成网页
请求分解成系列命令集
Web Server
WEB/应用服务器端数据

# Web应用架构




问题：命令写在哪里?

# 第八章 数据库编程

## 第一节 嵌入式SQL

## 第二节 ODBC和SQL/CLI

## 第三节 JDBC

## 第四节 存储过程

## 第五节 触发器

# 用户使用数据库的方式

- 交互式界面
- 应用程序

# 数据库程序设计方法

**将数据库命令嵌入到通用程序设计语言中**

嵌入到宿主程序设计语言中的数据库语句，要用特殊前缀（如EXEC SQL）加以标识。预编译器先扫描源程序代码，找出数据库语句，将这些语句抽取出来交给RDBMS处理。

# 数据库程序设计方法

## 使用数据库函数库

宿主程序设计语言通过使用函数库来实现数据库调用。实际的数据库查询和更新命令，以及所有必要的信息都以参数形式包含在函数调用中。

- 设计一种全新的语言

# 第一节 嵌入式SQL

1 嵌入式SQL及其处理过程

2 嵌入式SQL语句与主语言之间的通信

3 嵌入式SQL的使用情况

4 动态SQL

# 一. 嵌入式SQL及其处理流程

**嵌入式SQL是**将SQL语句嵌入到其它宿主语言编写的程序中，作为宿主语言的子语言，使宿主语言具备访问数据库的能力。

# 宿主语言与SQL语言的差别

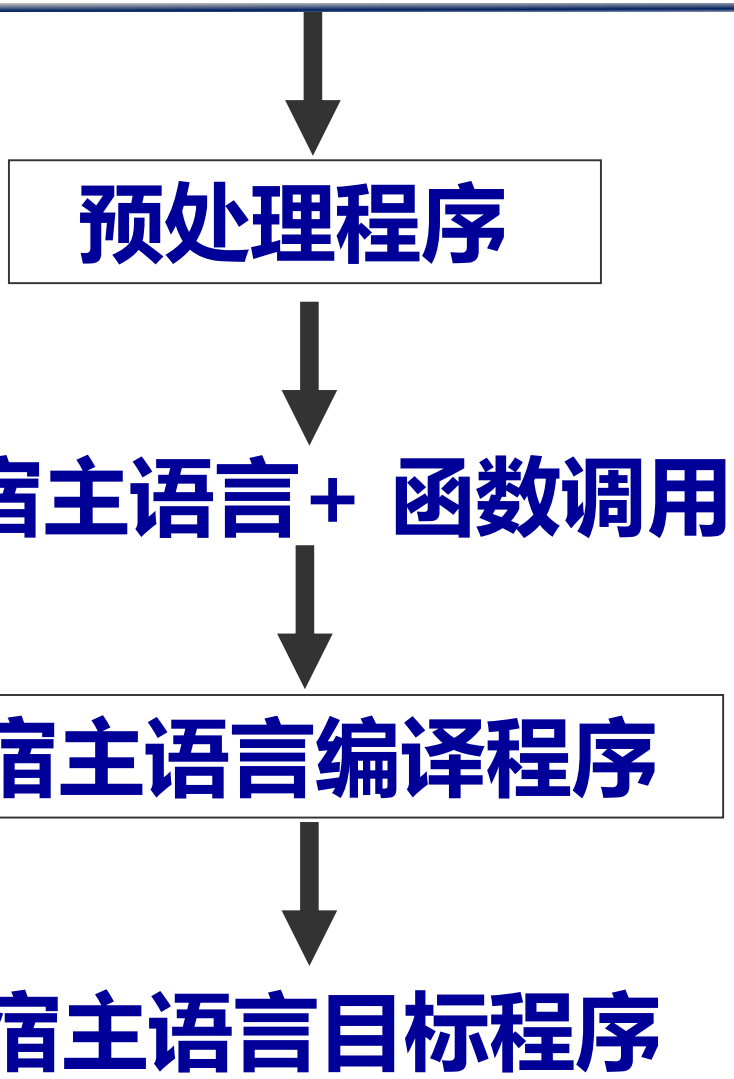
SQL语句按记录集合进行处理，一般用SQL对数据库进行存取操作。

宿主(高级)语言按每条记录进行处理，一般用高级语言进行程序流程的控制。

# 嵌入式SQL的实现：

**采用预编译方式**，先用预处理程序对源程序进行扫描，识别出SQL语句，把SQL语句转换成主语言调用语句，然后再用宿主语言的编译程序把源程序编译成目标程序。

# 源程序的预处理编译的具体过程图



# 二、嵌入式SQL语句与主语言之间的通信

SQL语句负责操纵数据库，而高级语言负责控制程序流程。它们之间如何通信?

（1）**SQL通信区（SQLCA）**：

- 向主语言传递SQL语句的执行状态信息
- 使主语言能够据此信息控制程序流程。

（2）**主变量**:

- 主语言向SQL语句提供参数
- 将SQL语句查询数据库的结果交主语言进一步处理

（3）**游标**:

- 解决集合性操作语言与过程性操作语言的不匹配。

# 1.SQL通信区SQLCA

SQL语句执行后，系统将当前的工作状态和运行环境的各种数据送到SQLCA中，应用程序从SQLCA中取出状态信息，决定接下来执行的语句。

（1）SQLCA是用

EXEC SQL INCLUDE SQLCA

语句加以定义的一个数据结构

SQLCA中有一个变量SQLCODE，存放每次执行SQL语句后返回的代码。

# 1.SQL通信区SQLCA

（2）应用程序执行完一条SQL语句后，测试SQLCODE的值了解SQL语句是否成功执行，以决定下一步如何处理，控制程序流程。

**如果SQL语句成功，则SQLCODE等于0**

**如果SQL语句失败，则SQLCODE不等于0,存放错误代码**

# 2. 主变量

嵌入式SQL语句中可以使用主语言的程序变量来输入或输出数据，将SQL语句中使用的主语言程序变量称为**主变量**（**共享变量**）。

```
string kk1, kk2

select sname into :kk1 from s

where sno = :kk2;
```

# 2. 主变量

所有主变量必须在:

EXEC SQL BEGIN DECLARE SECTION与

EXEC SQL END DECLARE SECTION

之间进行说明。随后SQL语句就可引用这些变量。

在SQL中引用主变量时，变量前需加“：”号，以便和本身字段变量相区别。

# 3. 游标

由于SQL语句处理的是记录集合，而主语句一次只能处理一个记录，因此需要用游标（cursor）来协调这两种不同的处理方式，把集合操作转换成单记录处理方式。

游标是系统为用户开设的一个数据缓冲区，存放SQL语句的执行结果，每个游标区有一个名字。通过游标逐一获取记录，并赋给主变量，交给主语言处理。

# 3. 游标

游标的使用:

(a)游标定义语句(DECLARE)

(b)游标打开语句(OPEN)

(c)游标取值语句(FETCH)

(d)游标关闭语句(CLOSE)

# (a)游标定义语句（DECLARE）

语法如下：

EXEC SQL DECLARE <游标名> CURSOR FOR
<select 语句> ;

EXEC SQL DECLARE C1 CURSOR FOR

SELECT sno, cno, grade FROM sc;

# (b)游标打开语句（OPEN）

该语句执行游标定义中的select语句，游标是一个指针，此时指向查询结果的第一个元组的前一位置。

OPEN语法如下：

EXEC SQL OPEN <游标名>；

EXEC SQL OPEN C1;

# (c)游标取值语句（FETCH）

把游标指向的行中的值取出，送到主变量，并将指针指向下一行。语法如下：

EXEC SQL FETCH <游标名>
INTO <变量表>；

变量表是由用逗号分开的主变量组成。 FETCH语句常置于宿主语言程序的循环结构中，并借助宿主语言的处理语句逐一处理查询结果中的一个个元组。

EXEC SQL FETCH C1 INTO :sno, :cno, :grade;

# (d)游标关闭语句(CLOSE)

关闭游标，释放结果集占用的缓冲区和其他资源，使它不再和查询结果相联系。

其语法如下：

EXEC SQL CLOSE <游标名>；

# 例子：

```
EXEC SQL INCLUDE SQLCA;          (1)定义SQL通信区
EXEC SQL BEGIN DECLARE SECTION;  (2)主变量说明开始
    CHAR sno[5];
    CHAR cno[3];
    INT grade;
EXEC SQL END DECLARE SECTION; 主变量说明结束
main()
{
        EXEC SQL DECLARE C1 CURSOR FOR    (3)定义游标
        SELECT sno, cno, grade FROM sc;
```

# 例子-续

```
EXEC SQL OPEN C1;                    (4)游标操作(打开游标)
for( ; ; )
{
    EXEC SQL FETCH C1 INTO :sno,:cno,:grade;
    (5)游标操作（推进游标指针并将当前数据放入主变量)
    if (sqlca.sqlcode <> 0) (6)利用SQLCA中状态信息决定何时退出循环
        break;
    printf("sno:%s,cno:%s,grade:%d",sno,cno, grade);
}
EXEC SQL CLOSE C1;                   (7)游标操作(关闭游标)
}
```

# 三、嵌入式SQL的使用情况

## 1. 不用游标的SQL语句：

★说明性语句

★数据定义语句

★数据控制语句

不需要游标、不需要主变量、不需要返回结果

★查询为单记录的SELECT语句

★非CURRENT形式的UPDATE语句

★非CURRENT形式的DELETE语句

★INSERT语句

# 三、嵌入式SQL的使用情况

## 2. 用游标的SQL语句

★查询结果为多记录的SELECT语句

★ CURRENT形式的UPDATE语句

★ CURRENT形式的DELETE语句

# 1. 不用游标的SQL 语句

(1) 如果是INSERT、DELETE和UPDATE语句，那么加上前缀标志“EXEC SQL” 和结束标志；，就能嵌入在宿主语言程序中使用。

(2) 对于SELECT语句，如果已知查询结果肯定是单元组，也可以加上前缀标志直接嵌入在主程序中使用，此时应在SELECT语句中增加一个INTO子句，指出找到的值应送到相应的主变量中去。

# (a) 说明性语句

```
EXEC SQL BEGIN DECLARE SECTION;
    char givensno[5], sn[9],  ss[1];
    int sa, raise;
EXEC SQL END DECLARE SECTION;
```

主变量

必须配对

# (b) 数据定义语句

```
EXEC SQL CREATE TABLE STUDENT
    （sno char(5) NOT NULL UNIQUE,
      sname char(20),
      sex char(1),
      age int,
      sdept char(15));
EXEC SQL DROP TABLE STUDENT;
```

## (c) 数据控制语句

```
EXEC SQL GRANT SELECT
      ON TABLE STUDENT
      TO USER1;
```

# (d) 查询结果为单记录的SELECT语句

```
EXEC SQL SELECT  sname, age, sex
                INTO :sn, :sa,  :ss     （输出主变量）
                FROM   S
                WHERE sno=:givensno;
```

## (e) INSERT语句

```
EXEC SQL  INSERT  INTO S(sno,sname, age)
   VALUES(:givensno, :sn, :sa);  （输入主变量）
```

# (f) 非CURRENT形式的DELETE语句

```
EXEC SQL DELETE   FROM   SC
        WHERE  sno = :givensno;    （输入主变量）
```

# (g) 非CURRENT形式的UPDATE语句

```
EXEC SQL UPDATE    SC
                SET  g = g + :raise     （输入主变量）
                WHERE  cno IN
                    (SELECT  cno   FROM   C
                     WHERE  cn = 'DB');
```

# 2.涉及游标的SQL 语句

## (1)查询结果为多记录的select语句的使用方式

(a)先用游标定义语句定义一个游标与某个select语句对应

(b)游标用OPEN语句打开后，执行相应的SELECT语句，把结果取到缓冲区,此时游标指向查询结果的第一行之前。

(c)执行fetch语句，游标向前推进一行，把游标指向的当前值送到主变量，供程序处理,循环重复，直到所有查询结果处理完毕。

(d)最后用CLOSE语句关闭游标。

# 例：用C编写程序输出不及格的学生名单

```
EXEC SQL DECLARE nopass CURSOR FOR
        SELECT sno，sname，g
        FROM  S，SC
        WHERE S.sno = SC.sno AND g < 60;
EXEC SQL OPEN  NOPASS;
EXEX SQL FETCH NOPASS INTO :m_sno，:m_sname，:m_grade
while (sqlca.sqlcode == 0)
 {
     printf(m_sno，m_sname，m_grade);
     EXEX SQL FETCH NOPASS INTO :m_sno,:m_sname,:m_grade;
 }
EXEC SQL CLOSE  NOPASS;
```

# 2.涉及游标的SQL 语句

## (2)对游标所指向元组的修改或删除操作 (即CURRENT形式的UPDATE语句和DELETE语句)

- 面向集合的操作，一次修改或删除所有满足条件的记录
- 如果只想修改或删除其中某个记录
  - 用带游标的SELECT语句查出所有满足条件的记录
  - UPDATE语句和DELETE语句中的子句：

    **WHERE CURRENT OF <游标名>**

    表示修改或删除的是最近一次取出的记录，即游标指针指向的记录

# 操作步骤

（1）用DECLARE语句说明游标，其中SELECT语句用FOR UPDATE OF <列名>，表明检索出的数据在指定列上可修改。

（2）用OPEN 语句打开游标，把所有满足条件的记录取到缓冲区。

# 操作步骤

（3）用FETCH语句推进游标指针，并将当前记录从缓冲区送到主变量。

（4）检查该记录是否是要修改或删除的记录，如果是，用UPDATE或DELETE语句修改或删除它。但UPDATE和DELETE语句必须用WHERE CURRENT OF <游标名>，表明要修改或删除的记录是游标所指向的记录。

第（3）、（4）步循环处理（依据SQLCA工作区中返回的SQLCODE代码控制循环）

（5）处理完毕用CLOSE语句关闭游标，释放缓冲区和其他资源。

# 例：

**例如对找到的某学生的成绩作如下操作：删除不及格的成绩，60-69分的成绩修改为70分，再显示该学生的成绩信息。**

```
EXEC SQL BEGIN DECLARE SECTION;
        char sno1[5],cno1[9], givensno[5];
        int g1;
EXEC SQL END DECLARE SECTION; //定义主变量
EXEC SQL  INCLUDE  SQLCA;//定义通信区
Void sel()
{
```

# 例（续）：

```
gets(givensno);
EXEC SQL DECLARE   scx  CURSOR FOR
    select  sno,  cno,  g
    from sc
    where sno=:givensno
    for update of  g;   //定义游标,成绩可修改
  EXEC   SQL   OPEN   scx;
 while(1)
    {EXEC SQL FETCH  scx
                    INTO :sno1,  :cno1,  :g1;
       if  SQLCA.SQLCODE<>0   break;
       if  (g1<60)
            EXEC SQL DELETE FROM sc
                WHERE CURRENT OF scx;
```

# 例（续）：

```
    Else
     {  if  (g1>=60 and g1<70)
          {  EXEC SQL UPDATE  sc
                      SET  g=70
                      WHERE  CURRENT OF scx;
             g1=70;
           }
        printf("%s, %s, %d",sno1, cno1 , g1);
      }
   }
   EXEC SQL CLOSE scx;
}
```

# 注意

当游标定义中的SELECT语句带有UNION或 ORDER BY子句时，或者该SELECT语句相当于定义了一个不可更新的视图时，不能使用CURRENT形式的 UPDATE和 DELETE语句

为什么?

# 四、 动态SQL

## （1）静态SQL语句

语句中使用的**主变量**、**查询目标列**、**条件**等在预编译时都是**确定**的，只有主变量的**值**在程序运行过程中**动态输入**。

```
SELECT  sname, age, sex
 INTO :sn, :sa,  :ss
FROM   S
WHERE sno=:givensno;
```

## （2）动态SQL语句

语句中的**查询条件**、要**查询的属性列**、要**查询的表**等在预编译时**不确定**，只有在运行过程中才能确定要提交的SQL语句。

# 动态SQL语句的三种类型：

## 1、直接执行的动态SQL

只用于非查询SQL语句的执行。应用程序定义一个字符串宿主变量，用于存放所要执行的SQL语句。然后，用EXEC SQL EXECUTE IMMEDIATE立即执行字符串宿主变量中的SQL语句。

SQL主变量：

程序主变量包含的内容是SQL语句的内容，而不是原来保存数据的输入或输出变量。

# 实例：删除满足某个输入条件的学生

```
EXEC SQL  BEGIN DECLARE SECTION;
char  sqlstring[200];
EXEC SQL  END DECLARE SECTION;
char cond[150];
…
strcpy(sqlstring, ”delete from student where”);
printf(“enter search condition:”);
scanf(“%s”, cond);
strcat(sqlstring, cond);
EXEC SQL EXECUTE IMMEDIATE :sqlstring;
…
```

# 2. 带动态参数的动态SQL

在该类语句中，使用参数符号？表示该位置的数据在运行时设定。动态参数的输入不是编译时完成绑定，而是通过**PREPARE**语句准备SQL主变量，用**EXECUTE**语句绑定数据或主变量完成

# 实例： 删除小于某个输入年龄的学生

```
EXEC SQL  BEGIN DECLARE SECTION;
char  sqlstring[200];
int s_age;
EXEC SQL  END DECLARE SECTION;
…
strcpy(sqlstring,”delete from student where age<?;”);
printf(“enter age for deleting:”);
scanf(“%d”,&s_age);
// 用prepare 语句定义sqlstring中的SQL语句为命令 smt
EXEC SQL PREPARE smt FROM:sqlstring;
// 用参数s_age取代？执行命令smt
EXEC SQL EXECUTE smt USING :s_age;
…
```

# 3. 查询类动态SQL

查询类动态SQL须返回查询结果，因为查询结果是单元组还是多元组，往往不能在编程序时确定，所以在该类语句中，一律以游标取数。

例：输入课程号，查询该课程的学生成绩，将查询结果按指定的方式排序

# 3. 查询类动态SQL--例题

```
EXEC SQL BEGIN DECLARE SECTION;
        char sqlstring[200];
        char sno [7];
        float grade;
        char givencno[6];
EXEC SQL END DECLARE SECTION; //定义主变量
…
char orderby[150];
strcpy(sqlstring, ”select sno,grade from sc where
cno=?”);
```

# 3. 查询类动态SQL--例题

```
//提示用户输入ORDER BY子句
printf("enter the ORDER BY clause:");
scanf("%s",orderby);
strcat(sqlstring,orderby);
 //提示用户输入要查询成绩的课程号
printf("enter the course number:");
scanf("%s",givencno);
// 准备查询语句
EXEC SQL PREPARE query FROM :sqlstring;
// 定义游标
EXEC SQL DECLARE   grade-cursor  CURSOR
FOR query;
```

# 3. 查询类动态SQL--例题

```
// 打开游标
EXEC  SQL  OPEN grade-cursor  USING : givencno;
// 取数
    while(1)
    {
          EXEC SQL FETCH   grade-cursor
                    INTO :sno, :grade;
          if  SQLCA.SQLCODE<>0   break;
          // 处理从游标取出的数据
          …
     }
  EXEC SQL CLOSE grade-cursor;
}
```

# 小结

1.在嵌入式SQL中，SQL语句与主语言语句分工非常明确

- SQL语句
  - 直接与数据库打交道，取出数据库中的数据。
- 主语言语句
  - 控制程序流程
  - 对取出的数据做进一步加工处理

# 小结（续）

2.SQL语言是面向集合的，一条SQL语句原则上可以产生或处理多条记录

3.主语言是面向记录的，一组主变量一次只能存放一条记录，仅使用主变量并不能完全满足SQL语句向应用程序输出数据的要求，嵌入式SQL引入了游标的概念，用来协调这两种不同的处理方式

# 第二节 ODBC和SQL/CLI

Open Database Connectivity

# 第二节 ODBC和SQL/CLI

1 数据库互连概述

2 ODBC应用系统的体系结构

3 ODBC数据源管理

4 开发ODBC应用程序的典型步骤

# 一、数据库互连概述

## 为什么要提出ODBC?

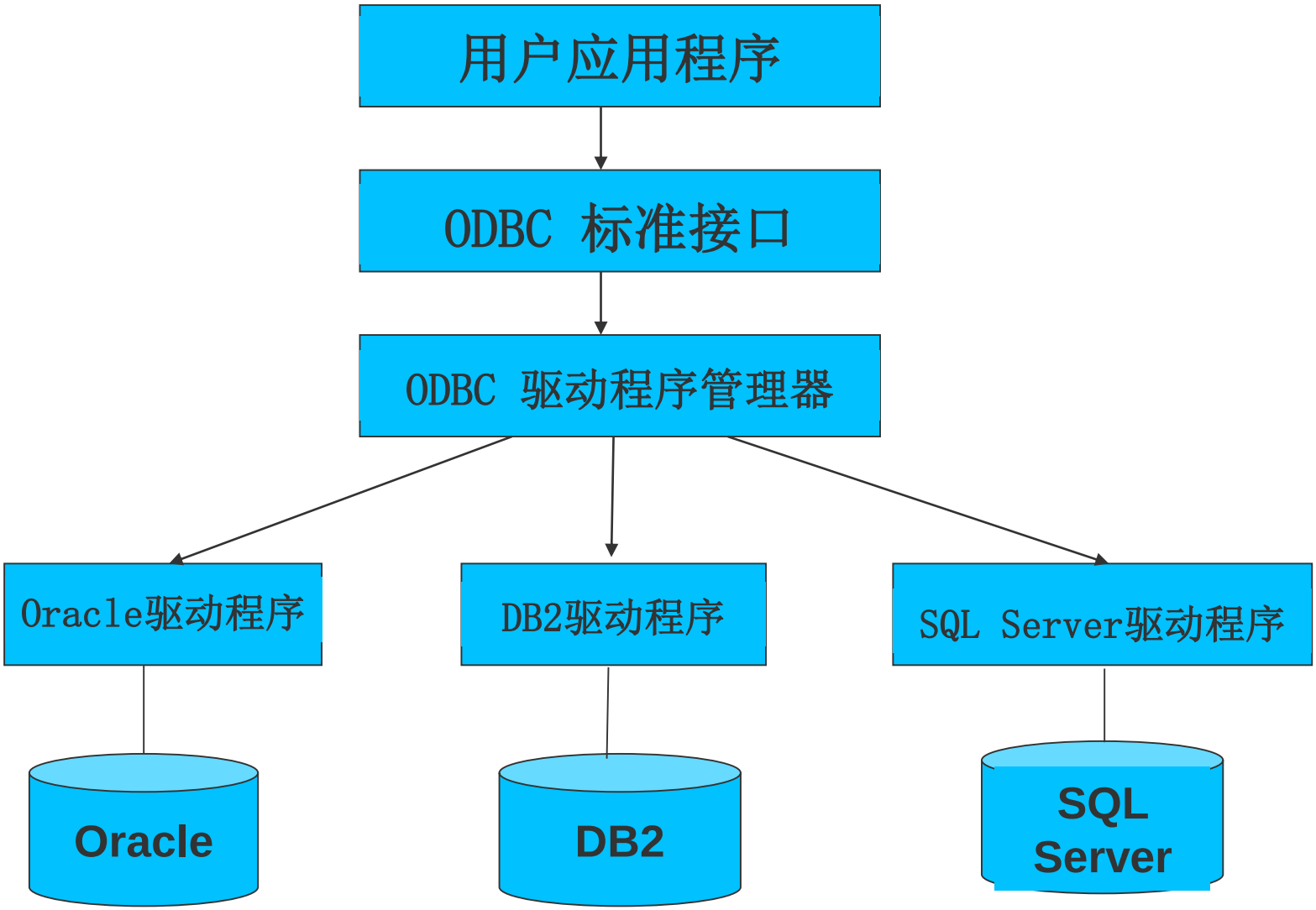
因为不同的数据库管理系统存在，系统之间有许多差异，在某个RDBMS下编写的应用程序不能在另一个RDBMS下运行。

- ODBC是微软公司推出的产品，为解决数据库系统的“开放”“互连”

# ODBC与SQL/CLI

- 1989年，Microsoft,Lotus,Sybase和DEC公司联合提出SQL Connectity访问接口规范。
- 1990年，数据库厂商集团SAG（SQL Access Group)接受SQL互连作为其CLI的基础
- 1992年，发布CLI 初步规范，被ANSI和ISO认可
- 同时，Microsoft发布了一个基于扩充的CLI的SDK,称为ODBC开发工具集
- 1995年，SQL/CLI被ISO正式批准，作为SQL-92的附件
- 目前， SQL/CLI已成为SQL:1999的一部分，且为多数RDBMS所支持

# 二、ODBC应用系统的体系结构



# 驱动程序管理器与数据库驱动程序

- 驱动程序管理器

  是ODBC的交通引导者。当一个应用程序请求对其数据源的连接，驱动程序管理器读取该数据源的描述，定位并加载适当的驱动程序，管理应用程序和驱动程序的连接。

- 数据库驱动程序

  通常由数据库厂商提供，实现必须的解释和通信所需要的功能，将ODBC SQL 语法翻译成服务器专用的SQL，将查询交付服务器并接收返回结果和状态信息。

# 三、ODBC数据源管理

- 数据源是最终用户需要访问的数据，包含了数据库位置和类型等信息，实际是数据连接的抽象。
- 在连接中，用数据源名代表用户名、服务器名、所连接的数据库名等，数据源对最终用户是透明的。
- 在开发ODBC数据库应用程序时首先要建立数据源并命名。

# 创建ODBC数据源

- “开始”菜单，“控制面板”，“管理工具”，“数据源ODBC”
- 点击“添加”，在列表框中选需安装的数据源驱动程序(如”Oracle in OraHome90”、”Adaptive Server Anywhere 8.0”等)
- 键入数据源名(student)及其他参数，不同的数据源参数不同

# 四、开发ODBC应用程序的典型步骤

(1)分别为程序所需的语句、连接、环境声明类型为SQLHSTMT、SQLHDBC、SQLHENV的**句柄变量**，声明SQLRETURN类型**返回变量**

(2)使用函数SQLAllocHandle在程序中**建立一个环境记录**

(3)使用函数SQLAllocHandle在程序中**建立一个连接记录**

(4)使用函数SQLConnect**建立一个到特定数据库服务器的连接**

# 开发ODBC应用程序的典型步骤-续

(5)使用函数SQLAllocHandle在程序中建立一个语句记录

(6)使用函数SQLPrepare准备语句

(7)在执行查询前，需要将各个参数通过函数SQLBindParameter和程序变量绑定

(8)使用SQLExecute函数执行由(5)中得到的句柄引用的SQL语句

(9)使用SQLBindCol函数将查询结果的各个列与C语言变量绑定

# 开发ODBC应用程序的典型步骤-续

(10)用SQLFetch函数将列值检索到C程序变量。如果查询结果是个元组的集合，每次SQLFetch调用将获取下一个元组，并将列值返回到所绑定的程序变量中

(11)最后，依次释放语句柄、连接柄和环境柄

例子：

下面的程序段读取一个部门编号，检索在该部门工作的雇员，并打印这些雇员的姓和工资。

# 开发ODBC应用程序的典型步骤-例

```
void printDepartmentsEmps(){
SQLHSTMT stmt1;  SQLHDBC con1;  SQLHENV env1;
SQLRETURN ret1,ret2,ret3,ret4;
ret1= SQLAllocHandle(SQL_HANDLE_ENV,
                              SQL_NULL_HANDLE,&env1);
If(! ret1) ret2=SQLAllocHandle(SQL_HANDLE_DBC,
                                   env1, &con1) else exit;
If(! ret2) ret3=SQLConnect(con1, ”dbs”, SQL_NTS,
      ”userID”, SQL_NTS,”passwd”, SQL_NTS) else exit;
If(! ret3) ret4=SQLAllocHandle(SQL_HANDLE_STMT,
                                   con1,&stmt1) else exit;
```

# 开发ODBC应用程序的典型步骤-例

```
SQLPrepare(stmt1,”select Lname,Salary from EMPLOYEE where
    Dno=?”,SQL_NTS);

prompt(“Enter the Department Number:”,dno);

SQLBindParameter(stmt1,1,SQL_INTEGER,&dno,4,&l1);

ret1=SQLExecute(stmt1);

If (!ret1) {

   SQLBindCol(stmt1,1,SQL_CHAR,&lname,15,&l2);

   SQLBindCol(stmt1,2,SQL_FLOAT,&salary,4,&l3);

    ret2=SQLFetch(stmt1);

    while (!ret2) {

        printf(lname,salary);   ret2=SQLFetch(stmt1)   }

}
```

# 小结

- ODBC目的：为了提高应用系统与数据库平台的独立性，使得应用系统的移植变得容易
- ODBC优点：
  - 使得应用系统的开发与数据库平台的选择、数据库设计等工作并行进行
  - 方便移植
  - 大大缩短整个系统的开发时间

# 第三节 JDBC

BEIJING JIAOTONG UNIVERSITY

# 第三节 JDBC

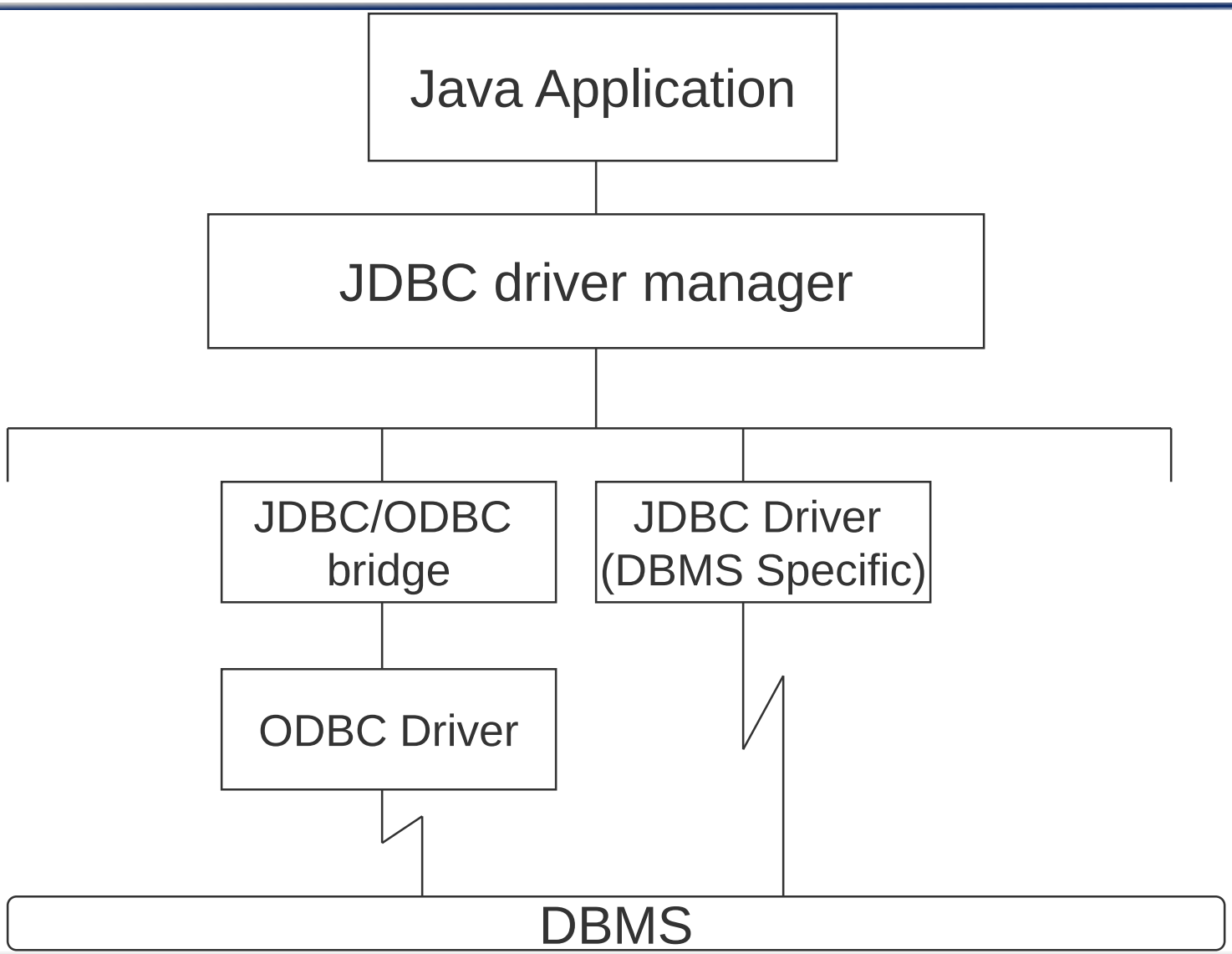
1 JDBC应用系统的体系结构

2 JDBC函数调用访问数据库的典型步骤

# JDBC

- 面向Java程序设计的SQL函数调用
- 许多RDBMS开发商都提供了JDBC驱动程序，从而可以通过Java程序访问它们的系统
- JDBC驱动程序是特定RDBMS开发商的JDBC应用编程接口中所指定的函数调用的实现

# 一、JDBC Architectures



# 二、JDBC函数调用访问数据库的典型步骤

(1)将JDBC**类库导入**到Java程序中,这些类称为java.sql.*

(2)**加载JDBC驱动程序**

若使用Jdbc:odbc桥，使用命令：
Class.forName(“sun.jdbc.odbc.jdbcOdbcDriver”)

若加载一个面向Oracle的JDBC驱动程序，使用命令：
Class.forName(“oracle.jdbc.driver.OracleDriver”)

(3)使用JDBC DriverManager类的getConnection函数**建立一个连接对象**

getConnection(url,dbaccount,password)

对于Jdbc:odbc桥，url为： jdbc:odbc:<数据源>

对于Oracle数据库，url为：Jdbc:oracle:thin@〈服务器〉:1521:<sid>

# JDBC函数调用访问数据库的典型步骤

(4)**创建语句对象**。一般情况下，一个查询要执行多次，就应使用PreparedStatement对象，因为它只需准备、检查、编译一次。

(5) 在执行查询前，需要**将各个参数与程序变量绑定。**对PreparedStatement对象应用诸如setString、setInteger、setDouble等函数来设置其参数。

(6)使用executeQuery函数**执行**由(4)中得到的对象引用的**SQL语句**

# JDBC函数调用访问数据库的典型步骤

JDBC中有一个通用函数execute,还有两个特化函数executeUpdate和executeQuery。

executeUpdate：用于SQL插入、删除或更新语句，并返回一个整型值以指示该操作所影响的元组数。

executeQuery：用于SQL检索语句，返回一个ResultSet类型对象。

(7) ResultSet类型对象r类似于嵌入式SQL中的游标，**r.next()移向r的下一个元组**，如果无对象，函数返回NULL。

(8) 根据各个属性的类型，通过使用各种get函数(getString、getInteger 、getDouble等),程序员可以引用当前元组中的各个属性。

# 连接数据库

```
try {
    Class.forName("sun.jdbc.odbc.
        JdbcOdbcDriver");
    String url = "jdbc:odbc:mydatabase";
    Connection con =
      DriverManager.getConnection(url,
          "login", "password");
  } catch (ClassNotFoundException e) {
  } catch (SQLException e) {
}
```

# 建表

```
try {
   Statement stmt = con.createStatement();
   stmt.executeUpdate( “create table
    mytable (col_a varchar(100), col_b
  integer, col_c float)");
   } catch (SQLException e) {
   }
```

# 插入记录

```
try {
    Statement stmt =
        con.createStatement();
    stmt.executeUpdate( “insert into
  mytable   value ('Patrick Chan', 123,
  1.23)");
} catch (SQLException e) {
   }
```

# 查询(使用属性名绑定)

```
try {
    Statement stmt = con.createStatement();
    ResultSet rs = stmt.executeQuery( “select * from   mytable");
     while (rs.next()) {
          String s = rs.getString( “col_a");
          int i = rs.getInt( “col_b");
          float f = rs.getFloat( “col_c");
          process(s, i, f);
     }
} catch (SQLException e) {
}
```

# 查询(使用属性号绑定)

```
try {
    Statement stmt = con.createStatement();
    ResultSet rs = stmt.executeQuery( “select * from   mytable");
     while (rs.next()) {
          String s = rs.getString(1);
          int i = rs.getInt(2);
          float f = rs.getFloat(3);
          process(s, i, f);
    }
} catch (SQLException e) {
}
```

# 修改

```
try {
   Statement stmt = con.createStatement();
   int numUpdated =
      stmt.executeUpdate( “update mytable
  set
      col_a = 'John Doe' where col_b = 123");
    } catch (SQLException e) {
}
```

# 动态绑定

```
try {
    PreparedStatement stmt =
      con.prepareStatement( “select *
      from mytable where col_a = ?");
    int column = 1;
    stmt.setString(column, "Patrick Chan");
    ResultSet rs = stmt.executeQuery();
} catch (SQLException e) {
}
```

# 动态绑定

```
try {
    PreparedStatement stmt =
  con.prepareStatement( “update
        mytable set col_a = ? where col_b = ?");
    stmt.setString(1, "John Doe");
    stmt.setInt(2, 123);
    int numUpdated = stmt.executeUpdate();
} catch (SQLException e) {
}
```

# 第四节 存储过程

1 存储过程概述

2 变量

3 批处理

4 控制结构

5 存储过程的创建、修改和删除

# 一、存储过程概述

存储过程是经编译和优化后存储在数据库服务器中的过程，优点是：

(1)模块化编程

创建一个存储过程存放在数据库中后，就可以被其他程序反复使用。

(2)快速执行

存储过程第一次被执行后，就驻留在内存中。以后执行就省去了重新分析、优化、编译的过程。

(3)减少网络通信量

有了存储过程后，在网络上只要一条语句就能执行一个存储过程。

(4)安全机制

通过隔离和加密的方法提高了数据库的安全性，通过授权可以让用户只能执行存储过程而不能直接访问数据库对象。

# 二、变量

## 变量定义

用DECLARE语句声明

```
DECLARE @i AS INT;
DECLARE @firstname AS VARCHAR(20),
        @lastname AS VARCHAR(40);
```

# 赋值

例1：

```
DECLARE @i AS INT;
SET @i =10;
```

例2：

```
DECLARE @empname AS VARCHAR(60);
SET @empname = (SELECT firstname + ‘ ‘+ lastname
                FROM HR.Employees
                WHERE empid=3);
```

例3：

```
DECLARE @firstname AS VARCHAR(20), @lastname AS
VARCHAR(40);

SELECT @firstname = firstname,

        @lastname = lastname

FROM HR.Employees

WHERE empid=3;
```

注：

如果查询返回多个满足条件的结果行，这段代码也不会失败，变量中保存的值是SQL Server随机访问到的最后一行中的值。

例4：

```
DECLARE @empname AS VARCHAR(60);

SET @empname = (SELECT firstname + ‘ ‘+
   lastname

                    FROM HR.Employees

                    WHERE mgrid=3);
```

注：

如果查询返回多个满足条件的结果行，这段代码会失败。 SET语句比赋值SELECT语句更安全。

# 三、批处理

批处理是从客户端应用程序发送到SQL Server的一组单条或多条T-SQL语句， SQL Server将批处理语句作为单个可执行的单元。

GO命令是客户端工具的命令，可以发出一批T-SQL语句结束的信号。

- 批处理是语句分析的单元

  如果分析成功，SQL Server会尝试执行批处理；如果存在语法错误，整个批处理就不会提交到SQL Server执行。

- 变量是属于定义它们的批处理的局部变量

```
DECLARE @i AS INT;
SET @i =10;
-- Succeeds
PRINT @i;
GO
-- Fails
PRINT @i;
```

# 四、控制结构-条件控制

1. IF *condition*
   *statement when true*
   ELSE
   *statement when false*
2. IF *condition*
   BEGIN
   *sequence of statements when true*
   END
   ELSE
   BEGIN
   *sequence of statements when false*
   END

3. IF语句可以嵌套

# CASE语句

- 语法1：

```
CASE 输入表达式
    WHEN 匹配值1 THEN 结果表达式1
    [WHEN 匹配值2 THEN 结果表达式2]
    …
    [ELSE 结果表达式]
END
```

例5：将pubs数据库中的图书信息表（titles)中的各种图书类型（type列）显示为全称。

```
USE pubs
SELECT title as 书名,
      图书类别 =
      CASE type
        WHEN  'popular_comp'  THEN  'Popular Computing'
        WHEN  'mod_cook'  THEN  'Modern Cooking'
        WHEN  'business'  THEN  'Business'
        WHEN  'trad_cook'  THEN  'Traditional Cooking'
        ELSE  'Not yet categorized'
      END,
      price as 价格
FROM titles
```

# CASE语句

- 语法2：

```
CASE
    WHEN 布尔表达式1 THEN 结果表达式1
    [WHEN 布尔表达式2 THEN 结果表达式2]
    …
    [ELSE 结果表达式]
END
```

# 例6：对pubs数据库中的各种价位的图书给予不同的提示。

```
USE pubs
SELECT title as 书名， price as 价格，
      价格类别 =
      CASE
        WHEN price IS NULL THEN  'Not yet priced'
        WHEN price<10 THEN  'Very Reasonable Title'
        WHEN price>=10 and price<20 THEN  'Coffee Table Title'
        ELSE  'Expensive book!'
      END
FROM titles
```

# 控制结构-循环控制

**WHILE** *condition*

<语句或语句块>

- 语句块由BEGIN开始和END结束
- 语句块可以包括CONTINUE或BREAK语句

CONTINUE使循环跳过该语句后面的语句，回到WHILE语句; BREAK语句将完全跳出循环，结束WHILE循环的执行。

# 五、存储过程分类

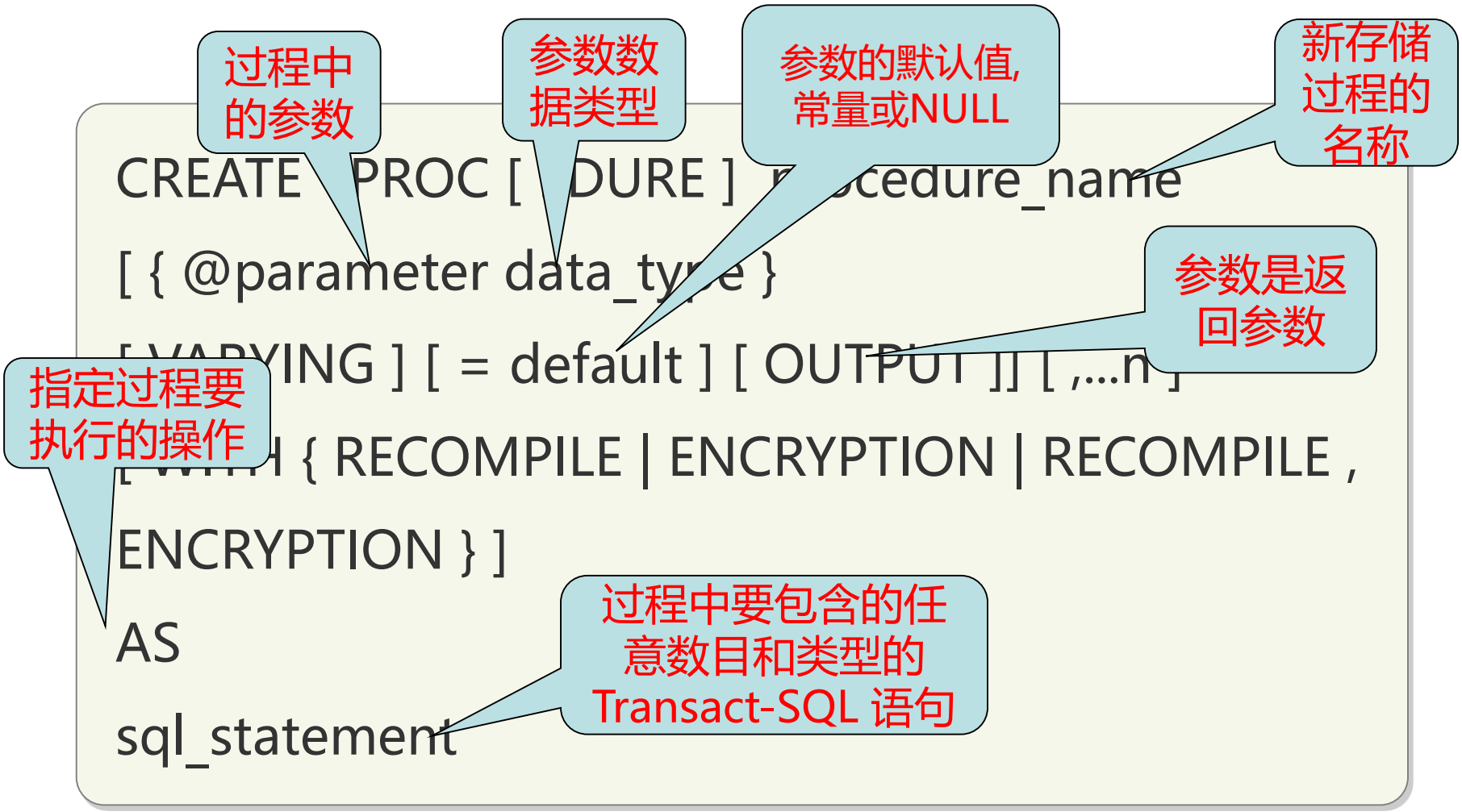
## 1. 系统存储过程

主要存储在master数据库中并以sp_为前缀，系统存储过程主要是从系统表中获取信息，从而为系统管理员管理SQL Server 提供支持。

## 2. 用户自定义存储过程

由用户创建并能完成某一特定功能（如查询用户所需数据信息）的存储过程。

# 存储过程的创建

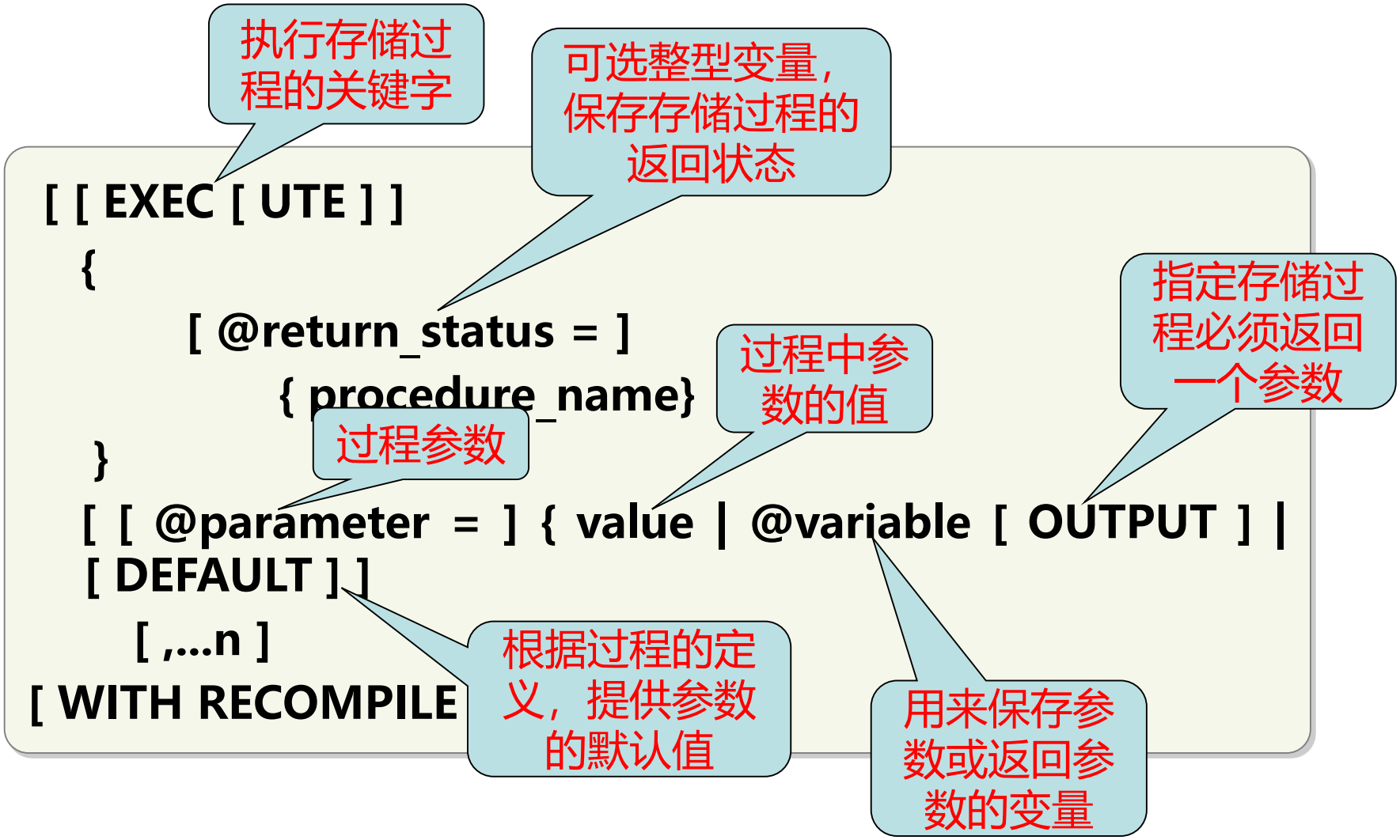

过程中的参数
参数数据类型
参数的默认值,常量或NULL
新存储过程的名称
参数是返回参数
指定过程要执行的操作
过程中要包含的任意数目和类型的Transact-SQL 语句
CREATE PROC [ EDURE ] procedure_name
[ { @parameter data_type }
[ VARYING ] [ = default ] [ OUTPUT ]] [ ,...n ]
[ WITH { RECOMPILE | ENCRYPTION | RECOMPILE ,
ENCRYPTION } ]
AS
sql_statement

- VARYING:仅适用于游标参数，指定作为输出参数支持的结果集
- 数据类型：所有数据类型均可以用作存储过程的参数。Cursor数据类型只能用于输出参数，因此，如果指定的数据类型为Cursor，也必须同时指定VARYING和OUTPUT关键字
- RECOMPILE:不保存执行计划，该存储过程将在运行时重新编译。
- ENCRYPTION：加密

例7：创建存储过程AvgScore,用于根据给定的院系和班级名称计算平均成绩，并使用输出参数返回结果

```
CREATE PROCEDURE AvgScore
   @org varchar(100),                    --院系名称，输入参数
   @class varchar(50),                   --班级名称，输入参数
   @score float OUTPUT                   --成绩，输出参数
   AS
   DECLARE @orgid int
   SET @orgid = 0
   --根据参数中指定的院系名称org，获取院系编号
   SELECT @orgid = 记录编号 FROM  院系 WHERE  院系名称= @org
   IF @orgid = 0
       BEGIN
       SET @ score = 0
       PRINT  '指定的院系记录不存在'
       END
   ELSE
       BEGIN
       SELECT @ score = AVG（入学成绩）
             FROM 学生 WHERE  所属院系= @orgid AND 班级= @class
             GROUP BY  所属院系, 班级
       END
```

# 存储过程的执行


执行存储过程的关键字
可选整型变量，保存存储过程的返回状态
指定存储过程必须返回一个参数
过程中参数的值
过程参数
[ [ EXEC [ UTE ] ]
{
[ @return_status = ]
{ procedure_name}
}
[ [ @parameter = ] { value | @variable [ OUTPUT ] |
[ DEFAULT ] ]
[ ,...n ]
[ WITH RECOMPILE
根据过程的定义，提供参数的默认值
用来保存参数或返回参数的变量

# 执行存储过程

- @return_status：可选的整型变量，保持存储过程的返回状态。
- 在使用格式“@parameter = value”时，参数名称和常量不一定按照存储过程定义时的顺序出现；但是，只要有一个参数使用该格式，其他参数也必须使用该格式。如果没有指定参数名，参数值必须按照存储过程定义时的顺序出现。
- WITH RECOMPILE：强制编译新的计划。建议尽量少使用该选项，因为它消耗较多系统资源，除非所提供的参数为非典型参数或者数据有很大的改变。

# 执行例7的存储过程

```
DECLARE @score float
EXEC AvgScore '软件系', '二班', @score OUTPUT
PRINT @score
```

# 存储过程的返回值

可以在存储过程中使用RETURN语句返回一个状态值，返回值只能是整数。

例8：创建存储过程AvgScore1,它的功能是根据给定的院系和班级名称计算平均成绩，并将结果使用输出参数返回。如果指定的院系存在，则返回1，否则返回0。

# 示例

```
CREATE PROCEDURE AvgScore1
    @org varchar(100),                    --院系名称，输入参数
    @class varchar(50),                   --班级名称，输入参数
    @score float OUTPUT                   --成绩，输出参数
    AS
    DECLARE @orgid int
    SET @orgid = 0
    --根据参数中指定的院系名称org，获取院系编号
    SELECT @orgid = 记录编号 FROM  院系 WHERE  院系名称= @org
    IF @orgid = 0
            RETURN 0
    ELSE
        BEGIN
        SELECT @ score = AVG（入学成绩）
              FROM 学生 WHERE  所属院系= @orgid AND 班级= @class
              GROUP BY  所属院系, 班级
        RETURN 1
        END
    GO
```

# 执行例8的存储过程

```
DECLARE @score float

DECLARE @result int

EXEC @result = AvgScore1 '软件系', '二班',
   @score OUTPUT

-- 检查返回值

IF @result = 1

PRINT @score

ELSE

PRINT '没有对应的记录'
```

# 存储过程的修改

```
ALTER   PROC [ EDURE ]  procedure_name
[ { @parameter data_type }
[ VARYING ] [ = default ] [ OUTPUT ]] [ ,...n ]
[ WITH { RECOMPILE | ENCRYPTION | RECOMPILE ,
ENCRYPTION } ]
AS
sql_statement
```

# 删除存储过程

DROP PROCEDURE procedure_name

例:

DROP PROCEDURE AvgScore

DROP PROCEDURE AvgScore1

# 小结

- 存储过程的优点
  - 经编译和优化后存储在数据库服务器中，运行效率高
  - 降低客户机和服务器之间的通信量
  - 有利于集中控制，方便维护

# 第五节 触发器

1 触发器概述

2 触发器的工作原理

3 触发器的创建

4 管理触发器

# 什么是触发器?

触发器是一种特殊类型的存储过程，它在指定表中的数据发生变化时自动执行。导致触发器生效的操作包括INSERT、UPDATE 和DELETE等。

# 为什么要使用触发器?

- 实现一些复杂的约束
  - 针对数据进行复杂的、仅通过普通的约束无法完成的检查

例如，当客户下订单时，检查其是否有足够的资金，或看看是否有足够的股票，如果任何一种检查失败，则返回错误信息，并对更新进行回滚。

- 基于原始被触发表中的更新，对另一张表进行更改

例如，在添加一个订单时，从股票中减少该订单所包含项目的数量，并创建自动的审核追踪，生成每条记录的更改历史。

# 触发器&存储过程

- 触发器主要是通过**事件**进行触发而被执行的，而存储过程可以通过存储过程**名字**而被直接调用。
- DBMS提供了两种主要机制用于维护数据的完整性：一种是约束，另一种就是触发器。触发器虽然是一种特殊的存储过程，但是它与表却是紧密联系的，离开了表它将不复存在（这点与约束十分类似）。
- 触发器的主要作用就是其能够实现复杂的完整性约束和保证数据的一致性。

# 触发器的类型

(1) AFTER触发器:

将在数据变动（INSERT、UPDATE和DELETE操作）完成后才被激发。对变动数据进行检查，如果发现错误，将拒绝或回滚变动的数据.(只能在表上定义.)

(2) INSTEAD OF 触发器:

将在数据变动以前被激发。并取代变动数据（INSERT、UPDATE和DELETE操作）的操作，转而去执行触发器定义的操作.(在表和视图上都可定义.)

# 触发器的工作原理

当修改表的数据而引发了触发器时，触发器将执行一系列T-SQL命令，在执行这些动作之前系统首先自动建立两个专用临时表：inserted表和deleted表。

1. 这两个表的结构与触发器表完全相同。而且只能由创建他们的触发器引用

2. 它们是临时的逻辑表,由系统来维护,不允许用户直接对它们进行修改，只能在触发器程序中查询表中的内容。

3. 它们存放在内存中,并不存放在数据库中,触发器执行完毕后，与该触发器相关的这两个表也会被删除。

# Insert、delete触发器的工作原理

## Insert触发器：

在向表中插入数据时,insert触发器自动执行,新增加的记录增加到触发器表中和inserted表中。

## delete触发器：

当在表中删除记录的时候，delete触发器自动执行，被删除的记录在deleted表中。

# update触发器工作原理

当修改表中的一条记录时，update触发器就会自动执行。执行一条update语句时，相当于先执行一个DELETE操作，再执行一个INSERT操作。所以旧的行被移动到DELETED表，而新的行同时插入INSERTED表。

触发器通过检查deleted表和inserted表和被修改的表来确定是否修改了行内容以及如何执行触发器操作。

# 创建触发器

创建触发器之前要注意如下几点：

（1）创建触发器所使用的语句CREATE TRIGGER必须是批处理中的第一个语句，随后的所有语句被解释为CREATE TRIGGER 语句定义的一部分。

（2）创建触发器的权限默认分配给表的所有者，且不能将该权限转移给其他用户。

（3）只能在当前数据库中创建触发器，但是触发器可以引用其他数据库的对象。

（4）如果已经给一个表的外键定义了级联删除或级联更新，则不能在该表上定义INSTEAD OF DELETE或INSTEAD OF UPDATE触发器。

（5）TRUNCATE TABLE不会引发DELETE触发器。

# 创建触发器的命令

```
CREATE TRIGGER 触发器名
ON {表名|视图名}
[ WITH ENCRYPTION ]
FOR|AFTER|INSTEAD OF
    { [INSERT],[UPDATE],[DELETE] }
AS
[ IF UPDATE(列名)[{AND|OR} UPDATE(列名)...] ]
SQL语句
```

# 参数说明

- WITH ENCRYPTION：加密syscomments表中包含CREATE TRIGGER语句文本的条目。
- FOR|AFTER:该选项的触发器会在底层数据被修改后运行其中的代码。如果表中存在针对级联更改的任何约束，这些级联操作会在触发器被触发之前完成。可以指定FOR，也可以指定AFTER。
- INSTEAD OF:执行触发器而不是执行触发SQL语句。在表或视图上，每个INSERT、UPDATE或DELETE语句最多可定义一个INSTEAD OF触发器。

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

  指定激活触发器的修改操作，必须至少指定一个选项。对于INSTEAD OF触发器，不允许在具有ON DELETE级联操作引用关系的表上使用DELETE选项，也不允许在具有ON UPDATE级联操作引用关系的表上使用UPDATE选项

- IF UPDATE(列名）:用于判断是否在指定的列上进行了INSERT或UPDATE操作，可以指定多列。

# 触发器举例

例1：在“学生”表中创建一个INSERT触发器，如果插入记录的院系编号值在“院系”表中存在下级单位（例如计算机学院包括软件系），则不执行插入操作，并提示用户.

# 触发器举例

```
CREATE TRIGGER insert_student ON 学生
FOR INSERT
AS
DECLARE @orgid int
DECLARE @orgname varchar(100)
-- 从插入的学生记录中提取“所属院系”值
SELECT @orgid =所属院系 FROM inserted
SELECT @orgname=院系名称 FROM 院系
                        WHERE   上级编号= @orgid
-- 如果存在下级单位
IF @orgname<>’’
   BEGIN
   PRINT ‘指定院系存在下级单位，请选择具体单位！’
   ROLLBACK TRANSACTION --回滚事务，撤消插入记录
   END
GO
```

# 回滚事务

如果在触发器中发出ROLLBACK TRANSACTION，系统将作如下处理：

- 回滚当前事务的所有修改，包括触发器所做的修改
- 触发器继续执行ROLLBACK语句之后的所有其余语句，如果这些语句中的任意语句修改数据，则不回滚这些修改。
- 批处理中，不执行所有位于激发触发器的语句之后的语句。

# 示例

例2.在“院系”表中创建一个DELETE触发器，如果删除记录的院系编号值在“院系”表中存在下级单位（例如，计算机学院包括软件系），则不执行删除操作，并提示用户。

```
CREATE TRIGGER delete_org ON 院系
INSTEAD OF DELETE
AS
DECLARE @orgid int
DECLARE @orgname varchar(100)
-- 从删除的院系记录中提取“所属院系”值
SELECT @orgid = 记录编号 FROM deleted
SELECT @orgname=院系名称 FROM 院系
                    WHERE   上级编号= @orgid
-- 如果存在下级单位
IF @orgname<>’’
   PRINT ‘指定院系存在下级单位，不允许被删除！’
ELSE
   DELETE FROM 院系 WHERE 记录编号= @orgid
GO
```

例3.在“学生”表中创建一个UPDATE触发器，如果修改记录的院系编号值在“院系”表中存在下级单位（例如，计算机学院包括软件系），则不执行修改操作，并提示用户。

```
CREATE TRIGGER update_student ON 学生
FOR UPDATE
AS
DECLARE @orgid int
DECLARE @orgname varchar(100)
IF UPDATE(所属院系)
BEGIN
    SELECT @orgid =所属院系 FROM inserted
    SELECT @orgname=院系名称 FROM 院系
                    WHERE   上级编号= @orgid
    IF @orgname<>''
   BEGIN
       PRINT '指定院系存在下级单位，请选择具体单位！'
       ROLLBACK TRANSACTION --回滚事务
   END
 END
 ELSE
PRINT '没有修改院系编号，无需触发器处理'
```

# 修改触发器

```
ALTER TRIGGER 触发器名
ON {表名|视图名}
[ WITH ENCRYPTION ]
FOR|AFTER|INSTEAD OF
    { [INSERT],[UPDATE],[DELETE] }
AS
[ IF UPDATE(列名)[ {AND|OR} UPDATE(列名)...] ]
SQL语句
```

# 删除触发器

**DROP TRIGGER 触发器名**

# 总结

- 存储过程与触发器是SQL Server中的两类数据库对象。它们都是由T-SQL语句编写而成的过程，所不同的是存储过程是由用户根据需要调用执行的，而触发器则是由某个动作（如删除或修改一条记录）引发执行的。

- 另外，存储过程可以不依附于表而单独存在，而触发器则必须依附于一个特定的表。

# 下课了。。。


休息一会儿。。。