# 数据库系统概论

# An Introduction to Database System

## 第八章 数据库编程

# 第八章 数据库编程

8.1 嵌入式SQL

8.2 存储过程

8.3 ODBC编程

# 8.1 嵌入式SQL

- SQL语言提供了两种不同的使用方式：
  - 交互式
  - 嵌入式
- 为什么要引入嵌入式SQL
  - SQL语言是非过程性语言
  - 事务处理应用需要高级语言
- 这两种方式细节上有差别，在程序设计的环境下，SQL语句要做某些必要的扩充

# 8.1 嵌入式SQL

8.1.1 嵌入式SQL的处理过程

8.1.2 嵌入式SQL语句与主语言之间的通信

8.1.3 不使用游标的SQL语句

8.1.4 使用游标的SQL语句

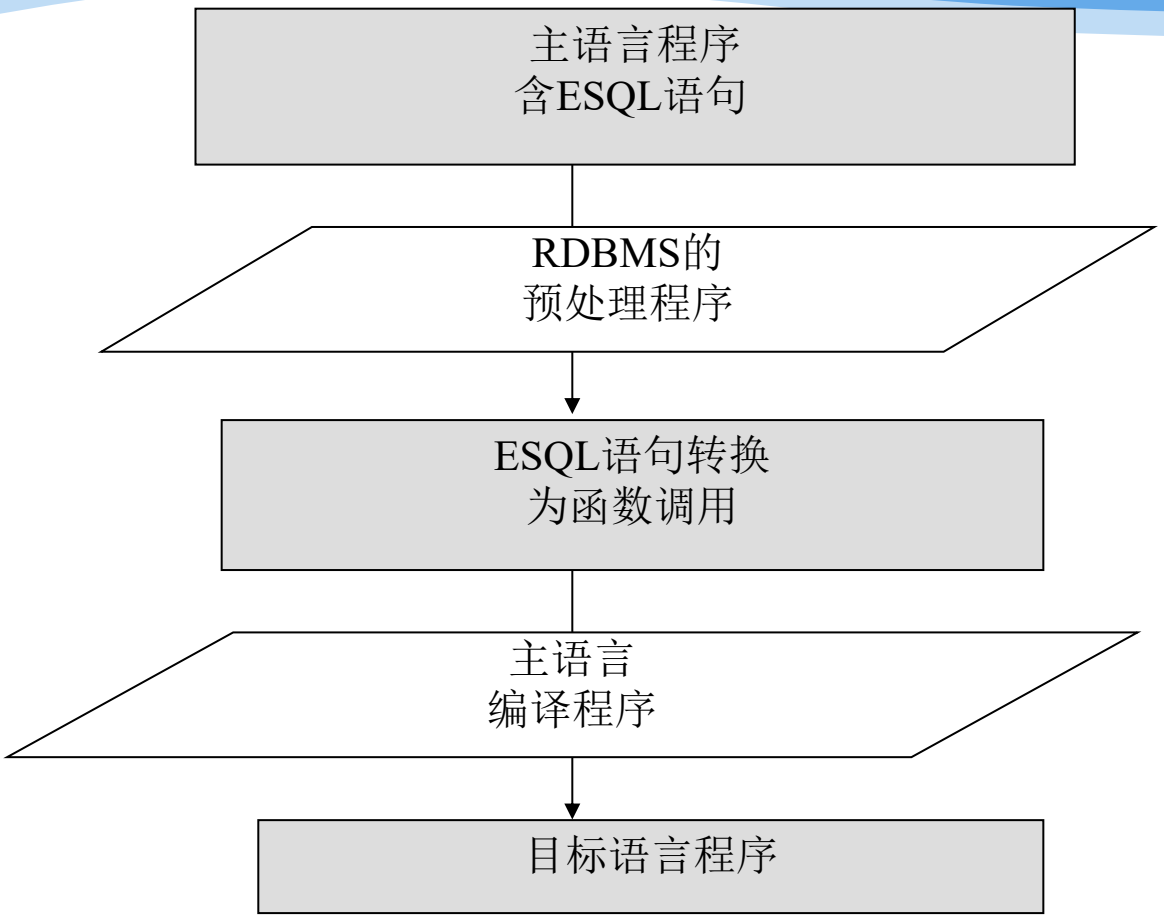
8.1.5 动态SQL

8.1.6 小结

# 8.1.1 嵌入式SQL的处理过程

- 主语言
  - 嵌入式SQL是将SQL语句嵌入程序设计语言中，被嵌入的程序设计语言，如C、C++、Java，称为宿主语言，简称主语言。
- 处理过程
  - 预编译方法

# 嵌入式SQL的处理过程（续）




ESQL基本处理过程

# 嵌入式SQL的处理过程（续）

- 为了区分SQL语句与主语言语句， 所有SQL语句必须加前缀EXEC SQL，以(;)结束:

  EXEC SQL <SQL语句>;

# 8.1 嵌入式SQL

8.1.1 嵌入式SQL的处理过程

8.1.2 嵌入式SQL与主语言的通信

8.1.3 不使用游标的SQL语句

8.1.4 使用游标的SQL语句

8.1.5 动态SQL

8.1.6 小结

# 8.1.2 嵌入式SQL语句与主语言之间的通信

- 将SQL嵌入到高级语言中混合编程，程序中会含有两种不同计算模型的语句
  - SQL语句
    - 描述性的面向集合的语句
    - 负责操纵数据库
  - 高级语言语句
    - 过程性的面向记录的语句
    - 负责控制程序流程
  - 它们之间应该如何通信？

# 嵌入式SQL语句与主语言之间的通信（续）

- 数据库工作单元与源程序工作单元之间的通信：
  - 1. SQL通信区
    - 向主语言传递SQL语句的执行状态信息
    - 使主语言能够据此控制程序流程
  - 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

# 一、SQL通信区

- SQLCA： SQL Communication Area
  - SQLCA是一个数据结构
- SQLCA的用途
  - SQL语句执行后，RDBMS反馈给应用程序信息
    - 描述系统当前工作状态
    - 描述运行环境
  - 这些信息将送到SQL通信区SQLCA中
  - 应用程序从SQLCA中取出这些状态信息，据此决定接下来执行的语句

# SQL通信区

- SQLCA使用方法：
  - 定义SQLCA
    - 用EXEC SQL INCLUDE SQLCA定义
  - 使用SQLCA
    - SQLCA中有一个存放每次执行SQL语句后返回代码的变量SQLCODE
    - 如果SQLCODE等于预定义的常量SUCCESS，则表示SQL语句成功，否则表示出错
    - 应用程序每执行完一条SQL 语句之后都应该测试一下SQLCODE的值，以了解该SQL语句执行情况并做相应处理

# 二、主变量

- 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
  - 在SQL语句中使用的主语言程序变量简称为主变量（Host Variable）

# 主变量（续）

- 主变量的类型
  - 输入主变量
  - 输出主变量
  - 一个主变量有可能既是输入主变量又是输出主变量

# 主变量（续）

- 指示变量：
  - 一个主变量可以附带一个指示变量（Indicator Variable）

- SQLCA(SQL Communication Area)为系统定义的全局变量，供应用程序与DBMS通信使用.
  - 使用时在应用程序中加:EXEC SQL INCLUDE SQLCA,不必再有用户说明
  - SQLCA有一个分量叫SQLCODE,可表示为:SQLCA.SQLCODE.
  - 每执行一条SQL语句,都要返回一个SQLCODE的值.
    - 0　　执行成功　　正数　　表示已执行,但有异常.
    - 100　表示无数据可以取.　负数　　表示出错未执行等.

# 主变量（续）

- 在SQL语句中使用主变量和指示变量的方法
  - 1) 说明主变量和指示变量

BEGIN DECLARE SECTION

.....

.....  (说明主变量和指示变量)

.....

END DECLARE SECTION

```
exec sql begin declare section;
    Char sno[7];
    Char cno[7];
    Float GRADE;
    Short Grade1;   指示变量,与GRADE连用才有意义.
exec sql end declare section;
```

# 主变量（续）

- 2) 使用主变量
  - 说明之后的主变量可以在SQL语句中任何一个能够使用表达式的地方出现
  - 为了与数据库对象名（表名、视图名、列名等）区别，SQL语句中的主变量名前要加冒号（:）作为标志
- 3) 使用指示变量
  - 指示变量前也必须加冒号标志
  - 必须紧跟在所指主变量之后

# 主变量（续）

- 在SQL语句之外(主语言语句中)使用主变量和指示变量的方法
  - 可以直接引用，不必加冒号
- **嵌入式SQL的可执行语句:**
  - **包括DDL,QL,DML,DCL**
  - **与sql基本一致,增加少许语法成分.**
  - **还包括进入数据库的语句connect和控制事物结束的语句.比如:EXEC SQL CONNECT :uid IDENTIFIED BY :pwd**

    **:uid 宿主变量,用户标示**

    **:pwd宿主变量,用户口令.**

    **又如: EXEC SQL INSERT INTO SC(SNO,CNO,GRADE)**

    **VALUES(:SNO,:CNO,:GRADE);**

# 三、游标（cursor）

- 为什么要使用游标
  - SQL语言与主语言具有不同数据处理方式
  - 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
  - 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

# 游标（续）

- 游标
  - 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
  - 每个游标区都有一个名字
  - 用户可以用SQL语句逐一从游标中获取记录，并赋给主变量，交由主语言进一步处理
    - **说明游标语句：**

**exec sql declare <游标名> cursor for**
**Select …**
**From …**
**Where …;**

- **打开游标语句: exec sql open <游标名> ;**
- **取数语句:**

**EXEC SQL FETCH <游标名> INTO :hv1,:hv2,…;**
**一般用在循环语句里,当无数可取时,sqlcode取100.**

- **关闭游标语句:**

**EXEC SQL CLOSE <游标名>;**
**使用示例:p136 success:取数成功与否**

# 四、建立和关闭数据库连接

- 建立数据库连接

  EXEC SQL CONNECT TO *target* [AS *connection-name*] [USER *user-name*];

  *target*是要连接的数据库服务器：
  - 常见的服务器标识串，如<dbname>@<hostname>:<port>
  - 包含服务器标识的SQL串常量
  - DEFAULT

  *connect-name*是可选的连接名，连接必须是一个有效的标识符
  在整个程序内只有一个连接时可以不指定连接名

- 关闭数据库连接

  EXEC SQL DISCONNECT [*connection*];

- 程序运行过程中可以修改当前连接：

  EXEC SQL SET CONNECTION *connection-name* | DEFAULT;

# 五、程序实例

[例1]依次检查某个系的学生记录，交互式更新某些学生年龄。

```
EXEC SQL BEGIN DECLARE SECTION;  /*主变量说明开始*/
    char deptname[64];
    char HSno[64];
    char HSname[64];
    char HSsex[64];
    int    HSage;
    int    NEWAGE;
EXEC SQL END DECLARE SECTION;     /*主变量说明结束*/
long SQLCODE;
EXEC SQL INCLUDE sqlca;              /*定义SQL通信区*/
```

# 程序实例（续）

```
int main(void)                          /*C语言主程序开始*/
{
    int    count = 0;
    char  yn;                           /*变量yn代表yes或no*/
    printf("Please choose the department name(CS/MA/IS): ");
    scanf("%s", deptname);                  /*为主变量deptname赋值*/
    EXEC SQL CONNECT TO TEST@localhost:54321 USER
     "SYSTEM" /"MANAGER";          /*连接数据库TEST*/
    EXEC SQL DECLARE SX CURSOR FOR   /*定义游标*/
          SELECT Sno, Sname, Ssex, Sage   /*SX对应语句的执行结果*/
          FROM Student
          WHERE SDept = :deptname;
    EXEC SQL OPEN SX;          /*打开游标SX便指向查询结果的第一行*/
```

# 程序实例（续）

```
for ( ; ; )                          /*用循环结构逐条处理结果集中的记录*/
{
    EXEC SQL FETCH SX INTO :HSno, :HSname, :HSsex,:HSage;
                                     /*推进游标，将当前数据放入主变量*/
    if (sqlca.sqlcode != 0)          /* sqlcode != 0,表示操作不成功*/
          break;                     /*利用SQLCA中的状态信息决定何时退出循环*/
    if(count++ == 0)                 /*如果是第一行的话，先打出行头*/
     printf("\n%-10s %-20s %-10s %-10s\n", "Sno", "Sname", "Ssex", "Sage");
     printf("%-10s %-20s %-10s %-10d\n", HSno, HSname, HSsex, HSage);
                                              /*打印查询结果*/
     printf("UPDATE AGE(y/n)?"); /*询问用户是否要更新该学生的年龄*/
    do{
          scanf("%c",&yn);
     }
     while(yn != 'N' && yn != 'n' && yn != 'Y' && yn != 'y');
```

# 程序实例（续）

```
    if (yn == 'y' || yn == 'Y')              /*如果选择更新操作*/
          {
           printf("INPUT NEW AGE:");
           scanf("%d",&NEWAGE);          /*用户输入新年龄到主变量中*/
           EXEC SQL UPDATE Student                    /*嵌入式SQL*/
                    SET Sage = :NEWAGE
                    WHERE CURRENT OF SX ;
          }              /*对当前游标指向的学生年龄进行更新*/
}

    EXEC SQL CLOSE SX;          /*关闭游标SX不再和查询结果对应*/
    EXEC SQL COMMIT WORK;                      /*提交更新*/
    EXEC SQL DISCONNECT TEST;                  /*断开数据库连接*/
}
```

# 8.1 嵌入式SQL

8.1.1 嵌入式SQL的处理过程

8.1.2 嵌入式SQL语句与主语言之间的通信

8.1.3 不使用游标的SQL语句

8.1.4 使用游标的SQL语句

8.1.5 动态SQL

8.1.6 小结

# 8.1.3 不用游标的SQL语句

- 不用游标的SQL语句的种类
  - 说明性语句
  - 数据定义语句
  - 数据控制语句
  - 查询结果为单记录的SELECT语句
  - 非CURRENT形式的增删改语句

# 不用游标的SQL语句（续）

- 一、查询结果为单记录的SELECT语句
- 二、非CURRENT形式的增删改语句

# 一、查询结果为单记录的SELECT语句

- 这类语句不需要使用游标，只需要用INTO子句指定存放查询结果的主变量

[例2] 根据学生号码查询学生信息。假设已经把要查询的学生的学号赋给了主变量givensno。

```
EXEC SQL SELECT Sno，Sname，Ssex，Sage，Sdept
                INTO  :Hsno，  : Hname ，:Hsex，:Hage，:Hdept
                FROM  Student
                WHERE Sno=:givensno；
```

# 查询结果为单记录的SELECT语句（续）

(1) INTO子句、WHERE子句和HAVING短语的条件表达式中均可以使用主变量

(2)查询返回的记录中，可能某些列为空值NULL。

(3) 如果查询结果实际上并不是单条记录，而是多条记录，则程序出错，RDBMS会在SQLCA中返回错误信息

# 查询结果为单记录的SELECT语句（续）

[例3] 查询某个学生选修某门课程的成绩。假设已经把将要查询的学生的学号赋给了主变量givensno，将课程号赋给了主变量givencno。

```
EXEC SQL SELECT Sno，Cno，Grade
                INTO :Hsno，:Hcno，:Hgrade:Gradeid
                                    /*指示变量Gradeid*/
                FROM  SC
                WHERE Sno=:givensno AND Cno=:givencno；
```

如果**Gradeid < 0**，不论**Hgrade**为何值，均认为该学生成绩为空值。

# 二、非CURRENT形式的增删改语句

- 在UPDATE的SET子句和WHERE子句中可以使用主变量，SET子句还可以使用指示变量

[例4] 修改某个学生选修1号课程的成绩。

```
EXEC SQL UPDATE SC
        SET Grade=:newgrade      /*修改的成绩已赋给主变量*/
        WHERE Sno=:givensno;     /*学号赋给主变量givensno*/
```

# 非CURRENT形式的增删改语句（续）

[例5] 将计算机系全体学生年龄置NULL值。

```
Sageid=-1；
EXEC SQL UPDATE Student
                SET Sage=:Raise :Sageid
                WHERE Sdept= ‘CS’；
```

将指示变量Sageid赋一个负值后，无论主变量Raise为何值，RDBMS都会将CS系所有学生的年龄置空值 。

等价于：

```
EXEC SQL UPDATE Student
                SET Sage=NULL
                WHERE Sdept= 'CS'；
```

# 非CURRENT形式的增删改语句（续）

[例6] 某个学生退学了，现要将有关他的所有选课记录删除掉。假设该学生的姓名已赋给主变量stdname。

```
EXEC SQL DELETE
        FROM SC
        WHERE Sno=
            (SELECT Sno
             FROM Student
             WHERE Sname=:stdname);
```

# 非CURRENT形式的增删改语句（续）

[例7] 某个学生新选修了某门课程，将有关记录插入SC表中。假设插入的学号已赋给主变量stdno，课程号已赋给主变量couno。

```
gradeid=-1；              /*用作指示变量，赋为负值*/
EXEC SQL INSERT
        INTO SC(Sno，Cno，Grade)
        VALUES(:stdno，:couno，:gr :gradeid)；
```

**由于该学生刚选修课程，成绩应为空，所以要把指示变量赋为负值**

# 8.1 嵌入式SQL

- 8.1.1 嵌入式SQL的处理过程
- 8.1.2 嵌入式SQL语句与主语言之间的通信
- 8.1.3 不使用游标的SQL语句
- 8.1.4 使用游标的SQL语句
- 8.1.5 动态SQL
- 8.1.6 小结

# 8.1.4 使用游标的SQL语句

- 必须使用游标的SQL语句
  - 查询结果为多条记录的SELECT语句
  - CURRENT形式的UPDATE语句
  - CURRENT形式的DELETE语句

# 使用游标的SQL语句（续）

- 一、 查询结果为多条记录的SELECT语句
- 二、CURRENT形式的UPDATE和DELETE语句

# 一、 查询结果为多条记录的SELECT语句

- 使用游标的步骤

1. 说明游标
2. 打开游标
3. 推进游标指针并取当前记录
4. 关闭游标

# 1. 说明游标

- 使用DECLARE语句
- 语句格式

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

- 功能
  - 是一条说明性语句，这时DBMS并不执行SELECT指定的查询操作。

# 2. 打开游标

- 使用OPEN语句
- 语句格式

  EXEC SQL OPEN <游标名>;

- 功能
  - 打开游标实际上是执行相应的SELECT语句，把所有满足查询条件的记录从指定表取到缓冲区中
  - 这时游标处于活动状态，指针指向查询结果集中第一条记录

# 3.推进游标指针并取当前记录

- 使用FETCH语句
- 语句格式

EXEC SQL FETCH [[NEXT|PRIOR|

FIRST|LAST] FROM] <游标名>

INTO <主变量>[<指示变量>][,<主变量>[<指示变量>]]...;

# 推进游标指针并取当前记录（续）

- 功能
  - 指定方向推动游标指针，然后将缓冲区中的当前记录取出来送至主变量供主语言进一步处理
  - NEXT|PRIOR|FIRST|LAST：指定推动游标指针的方式
    - NEXT：向前推进一条记录
    - PRIOR：向回退一条记录
    - FIRST：推向第一条记录
    - LAST：推向最后一条记录
    - 缺省值为NEXT

# 4. 关闭游标

- 使用CLOSE语句
- 语句格式

  EXEC SQL CLOSE <游标名>;

- 功能
  - 关闭游标，释放结果集占用的缓冲区及其他资源
- 说明
  - 游标被关闭后，就不再和原来的查询结果集相联系
  - 被关闭的游标可以再次被打开，与新的查询结果相联系

# 二、CURRENT形式的UPDATE语句和DELETE语句

- CURRENT形式的UPDATE语句和DELETE语句的用途
  - 面向集合的操作
  - 一次修改或删除所有满足条件的记录

# CURRENT形式的UPDATE语句和DELETE语句(续)

- 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
  - ➢从中进一步找出要修改或删除的记录
  - ➢用CURRENT形式的UPDATE语句和DELETE语句修改或删除之
  - ➢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

# CURRENT形式的UPDATE语句和DELETE语句(续)

- 不能使用CURRENT形式的UPDATE语句和DELETE语句：
  - 当游标定义中的SELECT语句带有UNION或ORDER BY子句
  - 该SELECT语句相当于定义了一个不可更新的视图

# 嵌 入 式 SQL

- 8.1.1 嵌入式SQL的处理过程
- 8.1.2 嵌入式SQL语句与主语言之间的通信
- 8.1.3 不使用游标的SQL语句
- 8.1.4 使用游标的SQL语句
- 8.1.5 动态SQL
- 8.1.6 小结

# 8.1.5 动态SQL

- 静态嵌入式SQL
  - 静态嵌入式SQL语句能够满足一般要求
  - 无法满足要到执行时才能够确定要提交的SQL语句
- 动态嵌入式SQL
  - 允许在程序运行过程中临时“组装”SQL语句
  - 支持动态组装SQL语句和动态参数两种形式

# 动态SQL简介（续）

- 一、使用SQL语句主变量
- 二、动态参数

# 一、使用SQL语句主变量

- SQL语句主变量：
  - 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
  - SQL语句主变量在程序执行期间可以设定不同的SQL语句，然后立即执行

# 使用SQL语句主变量（续）

[例9] 创建基本表TEST

```
EXEC SQL BEGIN DECLARE SECTION;
const char *stmt = "CREATE TABLE test(a int);";
                /* SQL语句主变量 */
EXEC SQL END DECLARE SECTION;
... ...
EXEC SQL EXECUTE IMMEDIATE :stmt;
                /* 执行语句 */
```

- 动态SQL:
  - 在嵌入式SQL中,提供动态构造SQL功能
  - 分为直接执行/带动态参数/查询类 3种类型
- 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主变量来存放要执行SQL语句.
  - SQL语句固定部分由程序直接附给主变量.
  - SQL的可变部分由程序提示用户输入
  - 最后,用exec sql EXECUTE IMMEDIATE语句执行字符串主变量中的SQL语句.

```
exec sql begin declare section;
   Char sqlstring[200];
exec sql end declare section;
   char cond[150];
/*添入固定的部分*/
Strcpy (sqlstring,”DELETE FROM S WHERE”)
/*提示用户输入查询条件*/
printf(“Enter search condition:”);
Scanf(“%s”,cond);
Strcat (sqlstring,cond);
EXEC SQL EXECUTE IMMEDIATE:sqlstring;
….
```

# 二、动态参数

- 动态参数
  - SQL语句中的可变元素
  - 使用参数符号(?)表示该位置的数据在运行时设定
- 和主变量的区别
  - 动态参数的输入不是编译时完成绑定
  - 而是通过 (prepare)语句准备主变量和执行(execute)时绑定数据或主变量来完成

# 动态参数（续）

- 使用动态参数的步骤：

1.声明SQL语句主变量。

2.准备SQL语句(PREPARE)。

EXEC SQL PREPARE <语句名> FROM <SQL语句主变量>;

# 动态参数（续）

- 使用动态参数的步骤（续）：

3.执行准备好的语句(EXECUTE)

EXEC SQL EXECUTE <语句名> [INTO <主变量表>] [USING < 主变量或常量>];

# 动态参数（续）

[例10]向TEST中插入元组。

```
EXEC SQL BEGIN DECLARE SECTION;
const char *stmt = "INSERT INTO test VALUES(?);";
                    /*声明SQL主变量 */
EXEC SQL END DECLARE SECTION;
... ...
EXEC SQL PREPARE mystmt FROM :stmt;   /* 准备语句 */
... ...
EXEC SQL EXECUTE mystmt USING 100; /* 执行语句 */
EXEC SQL EXECUTE mystmt USING 200; /* 执行语句 */
```

# 8.1 嵌入式SQL

8.1.1 嵌入式SQL的处理过程

8.1.2 嵌入式SQL语句与主语言之间的通信

8.1.3 不使用游标的SQL语句

8.1.4 使用游标的SQL语句

8.1.5 动态SQL

8.1.6 小结

# 8.1.6 小 结

- 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）

- 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
- 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

# 第八章 数据库编程

8.1 嵌入式SQL

8.2 存储过程

8.3 ODBC编程

# 8.2 存储过程

- SQL-invoked routines：
  - 存储过程(SQL-invoked procedure)
  - 函数(SQL-invoked function)

# 8.2 存储过程

8.2.1 PL/SQL的块结构

8.2.2 变量常量的定义

8.2.3 控制结构

8.2.4 存储过程

8.2.5 小结

# 8.2.1 PL/SQL的块结构

- PL/SQL：
  - SQL的扩展
  - 增加了过程化语句功能
  - 基本结构是块
    - 块之间可以互相嵌套
    - 每个块完成一个逻辑操作

# PL/SQL的块结构（续）

- **PL/SOL块的基本结构：**

**1.定义部分**

DECLARE

------变量、常量、游标、异常等

- 定义的变量、常量等只能在该基本块中使用
- 当基本块执行结束时，定义就不再存在

# PL/SQL的块结构（续）

- **PL/SOL块的基本结构(续)：**

**2.执行部分**

BEGIN

------SQL语句、PL/SQL的流程控制语句

EXCEPTION

------异常处理部分

END;

# 8.2 存储过程

8.2.1 PL/SQL的块结构

8.2.2 变量常量的定义

8.2.3 控制结构

8.2.4 存储过程

8.2.5 小结

# 8.2.2 变量常量的定义

1. **PL/SQL中定义变量的语法形式是:**

变量名 数据类型 ［ ［NOT NULL］:=初值表达式］或

变量名 数据类型 ［ ［NOT NULL］初值表达式］

2. **常量的定义类似于变量的定义:**

常量名 数据类型 CONSTANT :=常量表达式

常量必须要给一个值，并且该值在存在期间或常量的作用域内不能改变。如果试图修改它，PL/SQL将返回一个异常。

3. **赋值语句**

变量名称:=表达式

# 8.2 存储过程

8.2.1 PL/SQL的块结构

8.2.2 变量常量的定义

8.2.3 控制结构

8.2.4 存储过程

8.2.5 小结

# 8.2.3 控制结构

- **PL/SQL 功能：**
  - 一、条件控制语句
  - 二、循环控制语句
  - 三、错误处理

# 控制结构（续）

## 一、 条件控制语句

**IF-THEN， IF-THEN-ELSE和嵌套的IF语句**

1. IF *condition* THEN
   *Sequence_of_statements*;
   END IF
2. IF condition THEN
   *Sequence_of_statements1*;
   ELSE
   *Sequence_of_statements2;*
   END IF;
3. 在THEN和ELSE子句中还可以再包括IF语句，即IF语句可以嵌套

# 控制结构（续）

## 二、循环控制语句

**LOOP， WHILE-LOOP和FOR-LOOP**

**1.最简单的循环语句LOOP**

```
LOOP
    Sequence_of_statements;
END LOOP;
```

多数数据库服务器的PL/SQL都提供EXIT、BREAK或LEAVE等循环结束语句，保证LOOP语句块能够结束。

# 控制结构（续）

## 二、循环控制语句（续）

### 2. WHILE-LOOP

WHILE condition LOOP

*Sequence_of_statements;*

END LOOP;

- 每次执行循环体语句之前，首先对条件进行求值
- 如果条件为真，则执行循环体内的语句序列。
- 如果条件为假，则跳过循环并把控制传递给下一个语句

### 3. FOR-LOOP

FOR count IN ［REVERSE］*bound1* … *bound2* LOOP

*Sequence_of_statements*;

END LOOP;

# 控制结构（续）

## 三、错误处理：

- 如果PL/SQL在执行时出现异常，则应该让程序在产生异常的语句处停下来，根据异常的类型去执行异常处理语句
- SQL标准对数据库服务器提供什么样的异常处理做出了建议，要求PL/SQL管理器提供完善的异常处理机制

# 8.2 存储过程

- 8.2.1 PL/SQL的块结构
- 8.2.2 变量常量的定义
- 8.2.3 控制结构
- 8.2.4 存储过程
- 8.2.5 小结

# 8.2.4 存储过程

- PL/SQL块类型：
  - 命名块：编译后保存在数据库中，可以被反复调用，运行速度较快。存储过程和函数是命名块
  - 匿名块：每次执行时都要进行编译，它不能被存储到数据库中，也不能在其他的PL/SQL块中调用

# 存储过程（续）

- 一、 存储过程的优点
- 二、 存储过程的用户接口
- 三、 游标

# 存储过程（续）

- 存储过程：由PL/SQL语句书写的过程，经编译和优化后存储在数据库服务器中，使用时只要调用即可。
- 一、存储过程的优点：

1. 运行效率高
2. 降低了客户机和服务器之间的通信量
3. 方便实施企业规则

# 存储过程（续）

- 二、 存储过程的用户接口：

1. 创建存储过程
2. 执行存储过程
3. 删除存储过程

# 二、 存储过程的用户接口

## 1. 创建存储过程：

CREATE Procedure 过程名（［参数1，参数2，...］）

AS

<PL/SQL块>；

- 过程名：数据库服务器合法的对象标识
- 参数列表：用名字来标识调用时给出的参数值，必须指定值的数据类型。参数也可以定义输入参数、输出参数或输入/输出参数。默认为输入参数。
- 过程体：是一个<PL/SQL块>。包括声明部分和可执行语句部分

# 存储过程的用户接口（续）

［例11］ 利用存储过程来实现下面的应用: 从一个账户转指定数额的款项到另一个账户中。

```
CREATE PROCEDURE TRANSFER(inAccount INT， outAccount
 INT， amount FLOAT)
  AS DECLARE
       totalDeposit FLOAT;
  BEGIN                          /* 检查转出账户的余额 */
     SELECT total INTO totalDeposit
     FROM ACCOUNT WHERE ACCOUNTNUM=outAccount;
     IF totalDeposit IS NULL THEN    /* 账户不存在或账户中没有存款 */
        ROLLBACK;
        RETURN;
     END IF;
```

# 存储过程的用户接口（续）

```
IF totalDeposit < amount THEN          /* 账户账户存款不足 */
    ROLLBACK;
    RETURN;
END IF;
UPDATE account SET total=total-amount
WHERE ACCOUNTNUM=outAccount;
                                       /* 修改转出账户，减去转出额 */
UPDATE account SET total=total + amount WHERE
ACCOUNTNUM=inAccount;
                                       /* 修改转入账户，增加转出额 */
COMMIT;                                /* 提交转账事务 */
END;
```

# 存储过程的用户接口（续）

- 重命名存储过程

ALTER Procedure 过程名1 RENAME TO 过程名2;

# 存储过程的用户接口（续）

## ❖2. 执行存储过程：

CALL/PERFORM Procedure 过程名(［参数1，参数2，...］)；

- 使用CALL或者PERFORM等方式激活存储过程的执行。
- 在PL/SQL中，数据库服务器支持在过程体中调用其他存储过程

［例12］从账户01003815868转一万元到01003813828账户中。

CALL Procedure TRANSFER(01003813828，01003815868，10000)；

# 存储过程的用户接口（续）

3. 删除存储过程

DROP PROCEDURE 过程名（）；

# 三、游 标

- 在PL/SQL中，如果SELECT语句只返回一条记录，可以将该结果存放到变量中。
- 当查询返回多条记录时，就要使用游标对结果集进行处理
- 一个游标与一个SQL语句相关联。
- PL/SQL中的游标由PL/SQL引擎管理

# 8.2 存储过程

- 8.2.1 PL/SQL的块结构
- 8.2.2 变量常量的定义
- 8.2.3 控制结构
- 8.2.4 存储过程
- 8.2.5 小结

# 8.2.5 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

# 第八章 数据库编程

8.1 嵌入式SQL

8.2 存储过程

8.3 ODBC编程

# 8.3 ODBC编程

- ODBC优点：
  - 移植性好
  - 能同时访问不同的数据库
  - 共享多个数据资源

# 8.3 ODBC编程

- 8.3.1 数据库互连概述
- 8.3.2 ODBC工作原理概述
- 8.3.3 ODBC API 基础
- 8.3.4 ODBC的工作流程
- 8.3.5 小结

# 8.3.1 数据库互连概述

- **ODBC产生的原因：**
  - 由于不同的数据库管理系统的存在，在某个RDBMS下编写的应用程序就不能在另一个RDBMS下运行
  - 许多应用程序需要共享多个部门的数据资源，访问不同的RDBMS

# 数据库互连概述（续）

- ODBC：
  - 是微软公司开放服务体系(Windows Open Services Architecture，WOSA)中有关数据库的一个组成部分
  - 提供了一组访问数据库的标准API
- ODBC约束力：
  - 规范应用开发
  - 规范RDBMS应用接口

# 8.3 ODBC编程

- 8.3.1 数据库互连概述
- 8.3.2 ODBC工作原理概述
- 8.3.3 ODBC API 基础
- 8.3.4 ODBC的工作流程
- 8.3.5 小结

# 8.3.2 ODBC工作原理概述

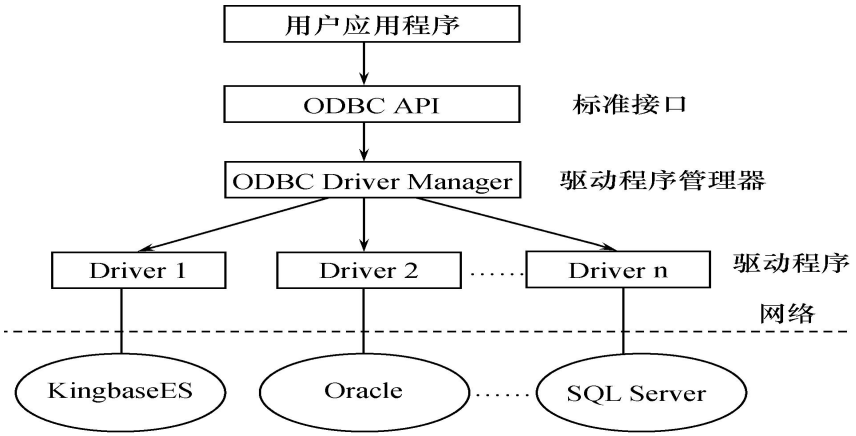
- ODBC应用系统的体系结构：

一、 用户应用程序

二、 驱动程序管理器

三、 数据库驱动程序

四、 **ODBC数据源管理**



# 一、 应用程序

- **ODBC应用程序包括的内容:**
  - 请求连接数据库；
  - 向数据源发送SQL语句；
  - 为SQL语句执行结果分配存储空间，定义所读取的数据格式；
  - 获取数据库操作结果，或处理错误；
  - 进行数据处理并向用户提交处理结果；
  - 请求事务的提交和回滚操作；
  - 断开与数据源的连接。

# 二、驱动程序管理器

- 驱动程序管理器：用来管理各种驱动程序
  - 包含在ODBC32.DLL中
  - 管理应用程序和驱动程序之间的通信
  - 建立、配置或删除数据源并查看系统当前所安装的数据库ODBC驱动程序
  - 主要功能：
    - 装载ODBC驱动程序
    - 选择和连接正确的驱动程序
    - 管理数据源
    - 检查ODBC调用参数的合法性
    - 记录ODBC函数的调用等

# 三、数据库驱动程序

- ODBC通过驱动程序来提供应用系统与数据库平台的独立性
- ODBC应用程序不能直接存取数据库
  - 其各种操作请求由驱动程序管理器提交给某个RDBMS的ODBC驱动程序
  - 通过调用驱动程序所支持的函数来存取数据库。
  - 数据库的操作结果也通过驱动程序返回给应用程序。
  - 如果应用程序要操纵不同的数据库，就要动态地链接到不同的驱动程序上。

# 数据库驱动程序（续）

- ODBC驱动程序类型：
  - 单束
    - 数据源和应用程序在同一台机器上
    - 驱动程序直接完成对数据文件的I/O操作
    - 驱动程序相当于数据管理器
  - 多束
    - 支持客户机/服务器、客户机/应用服务器/数据库服务器等网络环境下的数据访问
    - 由驱动程序完成数据库访问请求的提交和结果集接收
    - 应用程序使用驱动程序提供的结果集管理接口操纵执行后的结果数据

# 四、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数据库类型等信息，是一种数据连接的抽象
- 数据源对最终用户是透明的
  - ODBC给每个被访问的数据源指定唯一的数据源名（Data Source Name，简称DSN）,并映射到所有必要的、用来存取数据的低层软件
  - 在连接中，用数据源名来代表用户名、服务器名、所连接的数据库名等
  - 最终用户无需知道DBMS或其他数据管理软件、网络以及有关ODBC驱动程序的细节

# ODBC数据源管理（续）

例如，假设某个学校在MS SQL Server和KingbaseES上创建了两个数据库：学校人事数据库和教学科研数据库。

- 学校的信息系统要从这两个数据库中存取数据
- 为方便与两个数据库连接，为学校人事数据库创建一个数据源名PERSON，为教学科研数据库创建一个名为EDU的数据源。
- 当要访问每一个数据库时，只要与PERSON和EDU连接即可,不需要记住使用的驱动程序、服务器名称、数据库名

# 8.3 ODBC编程

- 8.3.1 数据库互连概述
- 8.3.2 ODBC工作原理概述
- 8.3.3 ODBC API 基础
- 8.3.4 ODBC的工作流程
- 8.3.5 小结

# 8.3.3 ODBC API 基础

- ODBC 应用程序接口的一致性
  - API一致性
    - API一致性级别有核心级、扩展1级、扩展2级
  - 语法一致性
    - 语法一致性级别有最低限度SQL语法级、核心SQL语法级、扩展SQL语法级

# ODBC API 基础（续）

- 一、 函数概述
- 二、 句柄及其属性
- 三、 数据类型

# 一、 函数概述

- **ODBC 3.0 标准提供了76个函数接口：**
  - 分配和释放环境句柄、连接句柄、语句句柄；
  - 连接函数（SQLDriverconnect等）；
  - 与信息相关的函数（如获取描述信息函数SQLGetinfo、SQLGetFuction）；
  - 事务处理函数（如SQLEndTran）；
  - 执行相关函数（SQLExecdirect、SQLExecute等）；
  - 编目函数，ODBC 3.0提供了11个编目函数如SQLTables、SQLColumn等，应用程序可以通过对编目函数的调用来获取数据字典的信息如权限、表结构等

# 函数概述（续）

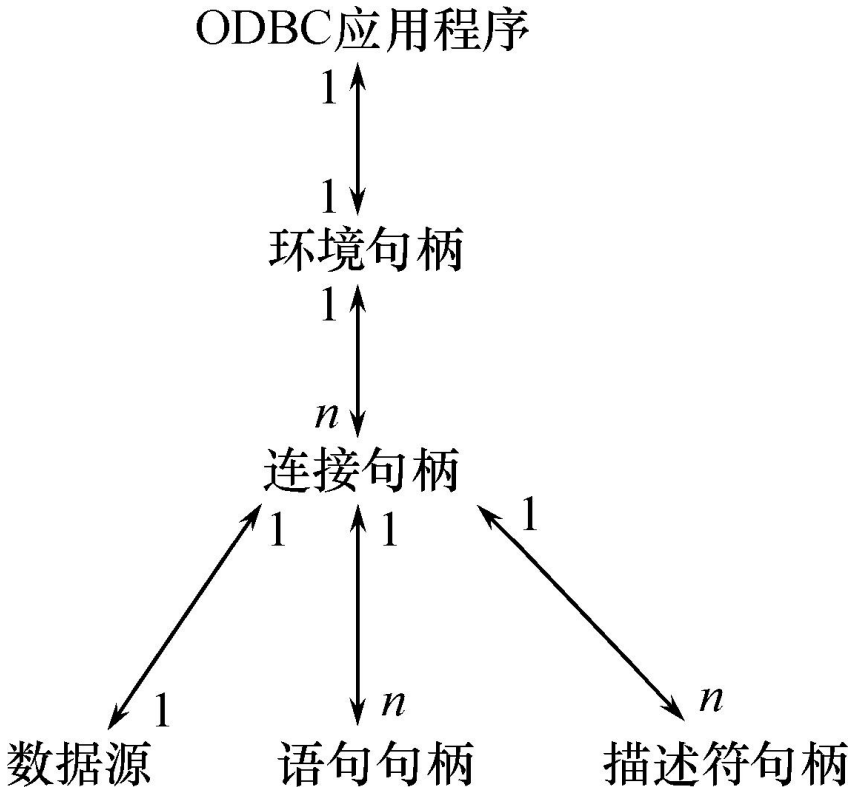
- ODBC 1.0和ODBC 2.x、ODBC 3.x函数使用上有很多差异
- MFC ODBC对较复杂的ODBC API进行了封装，提供了简化的调用接口

# 二、 句柄及其属性

- 句柄是32位整数值，代表一个指针
- ODBC 3.0中句柄分类：
  - 环境句柄
  - 连接句柄
  - 语句句柄
  - 描述符句柄

# 句柄及其属性（续）

- 应用程序句柄之间的关系




应用程序句柄之间的关系

# 句柄及其属性（续）

1. 每个ODBC应用程序需要建立一个ODBC环境，分配一个环境句柄，存取数据的全局性背景如环境状态、当前环境状态诊断、当前在环境上分配的连接句柄等；
2. 一个环境句柄可以建立多个连接句柄，每一个连接句柄实现与一个数据源之间的连接；

# 句柄及其属性（续）

3. 在一个连接中可以建立多个语句句柄，它不只是一个SQL语句，还包括SQL语句产生的结果集以及相关的信息等；

4. 在ODBC 3.0中又提出了描述符句柄的概念，它是描述SQL语句的参数、结果集列的元数据集合。

# 三、 数据类型

- ODBC数据类型：
  - SQL数据类型：用于数据源
  - C数据类型 ：用于应用程序的C代码

- 应用程序可以通过SQLGetTypeInfo来获取不同的驱动程序对于数据类型的支持情况

# 数据类型（续）

SQL数据类型和C数据类型之间的转换规则

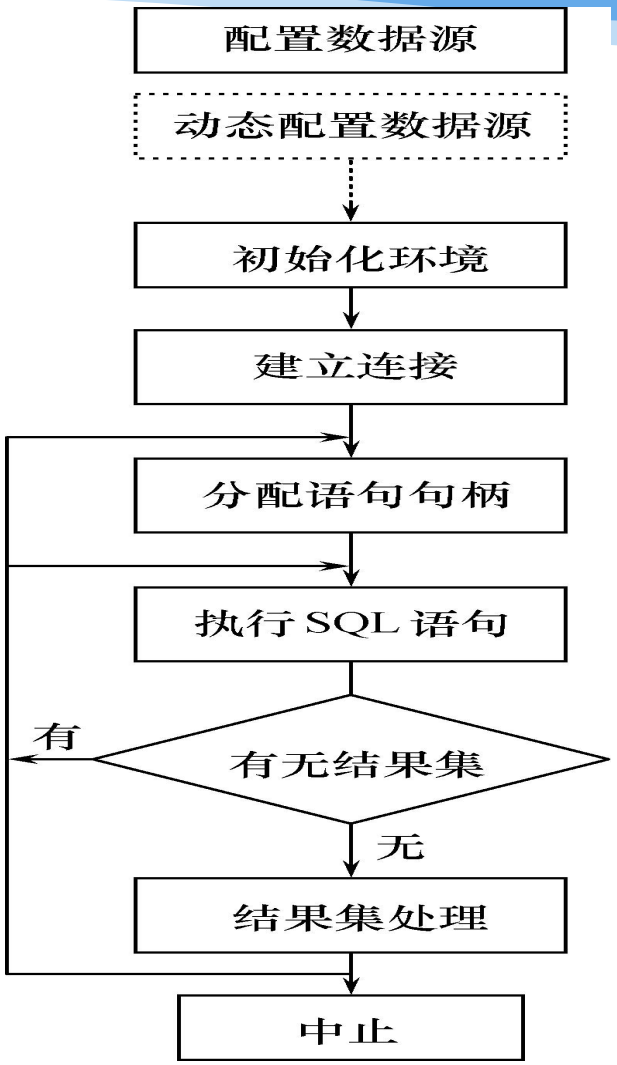
| | SQL<br>数据类型 | C数据类型 |
|---|---|---|
| SQL数据类型 | 数据源之间转换 | 应用程序变量传送到语句参数（SQLBindparameter） |
| C数据类型 | 从结果集列中返回到应用程序变量（SQLBindcol） | 应用程序变量之间转换 |

# 8.3 ODBC编程

- 8.3.1 数据库互连概述
- 8.3.2 ODBC工作原理概述
- 8.3.3 ODBC API 基础
- 8.3.4 ODBC的工作流程
- 8.3.5 小结

# 8.3.4 ODBC的工作流程

- ODBC的工作流程：



# ODBC的工作流程（续）

［例13］将KingbaseES数据库中Student表的数据备份到SQL SERVER数据库中。

- 该应用涉及两个不同的RDBMS中的数据源
- 使用ODBC来开发应用程序，只要改变应用程序中连接函数（SQLConnect）的参数，就可以连接不同RDBMS的驱动程序，连接两个数据源

# ODBC的工作流程（续）

- 在应用程序运行前，已经在KingbaseES和SQL SERVER中分别建立了STUDENT关系表

```
CREATE TABLE Student
    （ Sno   CHAR（9） PRIMARY KEY，
      Sname  CHAR（20） UNQUE
      Ssex   CHAR（2），
      Sage   SMALLINT，
      Sdept  CHAR（20）
    ）；
```

# ODBC的工作流程（续）

- 应用程序要执行的操作是：
  - 在KingbaseES上执行SELECT * FROM STUDENT；
  - 把获取的结果集，通过多次执行

    INSERT INTO STUDENT （Sno，Sname，Ssex，Sage，Sddept） VALUES （?， ?， ?， ?， ?）；
  - 插入到SQL SERVER的STUDENT表中

# ODBC的工作流程（续）

- 操作步骤：

  一、 配置数据源

  二、 初始化环境

  三、 建立连接

  四、 分配语句句柄

  五、 执行SQL语句

  六、 结果集处理

  七、 中止处理

# 一、 配置数据源

- 配置数据源两种方法：

  ⑴运行数据源管理工具来进行配置；

  ⑵使用Driver Manager 提供的ConfigDsn函数来增加、修改或删除数据源

- 在［例13］中，采用了第一种方法创建数据源。因为要同时用到KingbaseES和SQL Server，所以分别建立两个数据源，将其取名为KingbaseES ODBC和SQLServer。

# 配置数据源（续）

［例13］创建数据源的详细过程

```
#include <stdlib.h>
#include <stdio.h>
#include <windows.h>
#include <sql.h>
#include <sqlext.h>
#include <Sqltypes.h>
#define SNO_LEN 30
#define NAME_LEN 50
#define DEPART_LEN 100
#define SSEX_LEN 5
```

# 配置数据源（续）

［例13］创建数据源---第一步：定义句柄和变量

```
int main()
{
    /* Step 1 定义句柄和变量 */
    //以king开头的表示的是连接KingbaseES的变量
    //以server开头的表示的是连接SQLSERVER的变量
    SQLHENV    kinghenv，  serverhenv;                    //环境句柄
    SQLHDBC                kinghdbc，serverhdbc;                  //连接句柄
    SQLHSTMT               kinghstmt，serverhstmt;     //语句句柄
    SQLRETURN              ret;
    SQLCHAR  sName［NAME_LEN］，sDepart［DEPART_LEN］，
    sSex［SSEX_LEN］，  sSno［SNO_LEN］;
    SQLINTEGER   sAge;
    SQLINTEGER  cbAge=0，  cbSno=SQL_NTS，  cbSex=SQL_NTS，
    cbName=SQL_NTS，cbDepart=SQL_NTS;
```

# 二、初始化环境

- 没有和具体的驱动程序相关联，由Driver Manager来进行控制，并配置环境属性
- 应用程序通过调用连接函数和某个数据源进行连接后，Driver Manager才调用所连的驱动程序中的SQLAllocHandle，来真正分配环境句柄的数据结构

# 初始化环境代码

［例13］创建数据源---第二步：初始化环境

```
/* Step 2 初始化环境 */
    ret=SQLAllocHandle(SQL_HANDLE_ENV，SQL_NULL_HANDLE，
    &kinghenv);
    ret=SQLAllocHandle(SQL_HANDLE_ENV，SQL_NULL_HANDLE，
    &serverhenv);
     ret=SQLSetEnvAttr(kinghenv，        SQL_ATTR_ODBC_VERSION，
                    (void*)SQL_OV_ODBC3，  0);
    ret=SQLSetEnvAttr(serverhenv，      SQL_ATTR_ODBC_VERSION，
                    (void*)SQL_OV_ODBC3，  0);
```

# 三、 建立连接

- 应用程序调用SQLAllocHandle分配连接句柄，通过SQLConnect、SQLDriverConnect或SQLBrowseConnect与数据源连接
- SQLConnect连接函数，输入参数为：
  - 配置好的数据源名称
  - 用户ID
  - 口令
- [例13]中KingbaseES ODBC为数据源名字，SYSTEM为用户名，MANAGER为用户密码

# 建立连接代码

［例13］创建数据源---第三步：建立连接

```
/* Step 3 :建立连接 */
ret=SQLAllocHandle(SQL_HANDLE_DBC， kinghenv， &kinghdbc);
ret=SQLAllocHandle(SQL_HANDLE_DBC， serverhenv，
    &serverhdbc);
ret=SQLConnect(kinghdbc，"KingbaseES ODBC"，SQL_NTS，
    "SYSTEM"，SQL_NTS， "MANAGER"，SQL_NTS);
if (!SQL_SUCCEEDED(ret))//连接失败时返回错误值
    return-1;
ret=SQLConnect(serverhdbc， "SQLServer"， SQL_NTS， "sa"，
    SQL_NTS，"sa"，SQL_NTS);
if (!SQL_SUCCEEDED(ret) )                    //连接失败时返回错误值
    return -1;
```

# 四、分配语句句柄

- 处理任何SQL语句之前，应用程序还需要首先分配一个语句句柄
- 语句句柄含有具体的SQL语句以及输出的结果集等信息
- [例13]中分配了两个语句句柄：
  - 一个用来从KingbaseES中读取数据产生结果集（kinghstmt）
  - 一个用来向SQLSERVER插入数据（serverhstmt）
- 应用程序还可以通过SQLtStmtAttr来设置语句属性（也可以使用默认值）
- [例13]中结果集绑定的方式为按列绑定

# 分配语句句柄代码

［例13］创建数据源---第四步

```
/* Step 4 :初始化语句句柄 */
ret=SQLAllocHandle(SQL_HANDLE_STMT， kinghdbc， &kinghstmt);
ret=SQLSetStmtAttr(kinghstmt，SQL_ATTR_ROW_BIND_TYPE，
    (SQLPOINTER)
   SQL_BIND_BY_COLUMN，SQL_IS_INTEGER );
ret=SQLAllocHandle(SQL_HANDLE_STMT， serverhdbc，
    &serverhstmt);
```

# 五、 执行SQL语句

- 应用程序处理SQL语句的两种方式：
  - 预处理（SQLPrepare、SQLExecute适用于语句的多次执行）
  - 直接执行（SQLExecdirect）
- 如果SQL语句含有参数，应用程序为每个参数调用SQLBindParameter，并把它们绑定至应用程序变量
- 应用程序可以直接通过改变应用程序缓冲区的内容从而在程序中动态的改变SQL语句的具体执行

# 执行SQL语句（续）

- 应用程序根据语句的类型进行的处理
  - 有结果集的语句（select或是编目函数），则进行结果集处理。
  - 没有结果集的函数，可以直接利用本语句句柄继续执行新的语句或是获取行计数（本次执行所影响的行数）之后继续执行。
- 在[例13]中，使用SQLExecdirect获取KingbaseES中的结果集，并将结果集根据各列不同的数据类型绑定到用户程序缓冲区
- 在插入数据时，采用了预编译的方式，首先通过SQLPrepare来预处理SQL语句，将每一列绑定到用户缓冲区
- 应用程序可以直接修改结果集缓冲区的内容

# 程序源码

［例13］创建数据源---第五步：执行SQL语句

```
/* Step 5 :两种方式执行语句 */
/* 预编译带有参数的语句 */
ret=SQLPrepare(serverhstmt，"INSERT INTO STUDENT(SNO，SNAME，SSEX，
    SAGE，SDEPT) VALUES (?， ?， ?， ?， ?)"， SQL_NTS);
if (ret==SQL_SUCCESS || ret==SQL_SUCCESS_WITH_INFO)
{
ret=SQLBindParameter(serverhstmt，1，SQL_PARAM_INPUT，SQL_C_CHAR，
    SQL_CHAR， SNO_LEN， 0， sSno， 0， &cbSno);
ret=SQLBindParameter(serverhstmt，2，SQL_PARAM_INPUT， SQL_C_CHAR，
    SQL_CHAR， NAME_LEN， 0， sName， 0， &cbName);
ret=SQLBindParameter(serverhstmt，3，SQL_PARAM_INPUT，SQL_C_CHAR，
    SQL_CHAR， 2， 0， sSex， 0， &cbSex);
ret=SQLBindParameter(serverhstmt，4，SQL_PARAM_INPUT，
SQL_C_LONG，SQL_INTEGER， 0， 0， &sAge， 0， &cbAge);
```

# 程序源码

```
ret=SQLBindParameter(serverhstmt，5，SQL_PARAM_INPUT，SQL_C_CHAR，
SQL_CHAR， DEPART_LEN， 0， sDepart， 0， &cbDepart);
}
  /*执行SQL语句*/
ret=SQLExecDirect(kinghstmt，"SELECT * FROM STUDENT"，SQL_NTS);
if (ret==SQL_SUCCESS || ret==SQL_SUCCESS_WITH_INFO)
{
    ret=SQLBindCol(kinghstmt， 1， SQL_C_CHAR，sSno，SNO_LEN，
      &cbSno);
    ret=SQLBindCol(kinghstmt， 2， SQL_C_CHAR，sName，NAME_LEN，
      &cbName);
    ret=SQLBindCol(kinghstmt， 3， SQL_C_CHAR， sSex， SSEX_LEN，
      &cbSex);
    ret=SQLBindCol(kinghstmt， 4， SQL_C_LONG， &sAge， 0， &cbAge);
    ret=SQLBindCol(kinghstmt， 5， SQL_C_CHAR， sDepart，
      DEPART_LEN， &cbDepart);
}
```

# 六、结果集处理

- 应用程序可以通过SQLNumResultCols来获取结果集中的列数
- 通过SQLDescribeCol或是SQLColAttrbute函数来获取结果集每一列的名称、数据类型、精度和范围

# 结果集处理（续）

- ODBC中使用游标来处理结果集数据
- ODBC中游标类型：
  - forward-only游标，是ODBC的默认游标类型
  - 可滚动（scroll）游标：
    - 静态（static）
    - 动态（dynamic）
    - 码集驱动（keyset-driven）
    - 混合型（mixed）

# 结果集处理（续）

- 结果集处理步骤：
  - ODBC游标打开方式不同于嵌入式SQL，不是显式声明而是系统自动产生一个游标（Cursor），当结果集刚刚生成时，游标指向第一行数据之前
  - 应用程序通过SQLBindCol，把查询结果绑定到应用程序缓冲区中，通过SQLFetch或是SQLFetchScroll来移动游标获取结果集中的每一行数据
  - 对于如图像这类特别的数据类型当一个缓冲区不足以容纳所有的数据时，可以通过SQLGetdata分多次获取
  - 最后通过SQLClosecursor来关闭游标

# 程序源码代码

［例13］创建数据源---第六步：结果集处理

```
/* Step 6 ： 处理结果集并执行预编译后的语句*/
 while ( (ret=SQLFetch(kinghstmt) ) !=SQL_NO_DATA_FOUND)
 {
     if(ret==SQL_ERROR)  printf("Fetch error \n");
     else
             ret=SQLExecute(serverhstmt);
}
```

# 七、中止处理

- 应用程序中止步骤：
  - 首先释放语句句柄
  - 释放数据库连接
  - 与数据库服务器断开
  - 释放ODBC环境

# 中止处理代码

［例13］创建数据源---第七步：中止处理

```
    /* Step 7 中止处理*/
    SQLFreeHandle(SQL_HANDLE_STMT，kinghstmt);
    SQLDisconnect(kinghdbc);
    SQLFreeHandle(SQL_HANDLE_DBC， kinghdbc);
    SQLFreeHandle(SQL_HANDLE_ENV，kinghenv);
    SQLFreeHandle(SQL_HANDLE_STMT， serverhstmt);
    SQLDisconnect(serverhdbc);
    SQLFreeHandle(SQL_HANDLE_DBC， serverhdbc);
    SQLFreeHandle(SQL_HANDLE_ENV， serverhenv);
    return 0;
}
```

# 8.3 ODBC编程

- 8.3.1 数据库互连概述
- 8.3.2 ODBC工作原理概述
- 8.3.3 ODBC API 基础
- 8.3.4 ODBC的工作流程
- 8.3.5 小结

# 8.3.5 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

# 下课了。。。


休息一会儿。。。