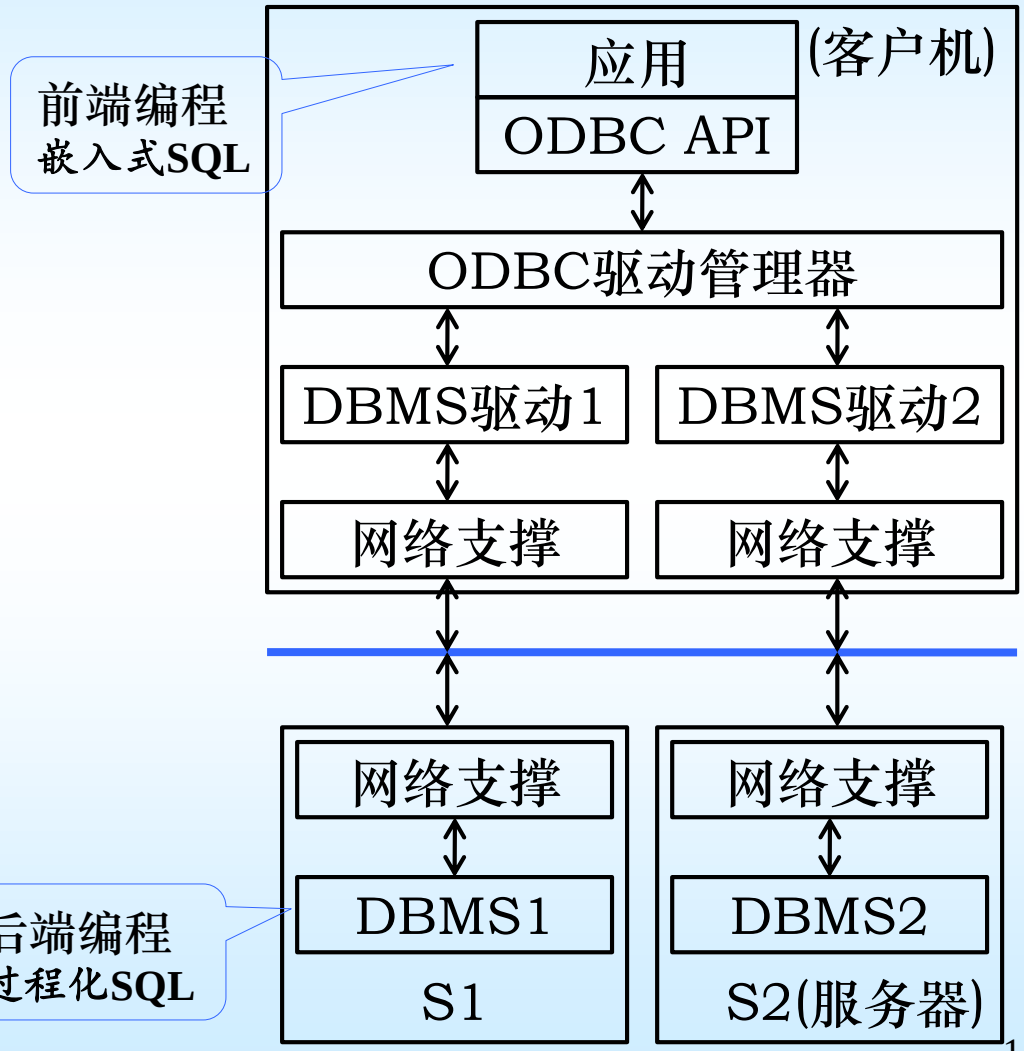
第八章 数据库编程

- 本章内容:
  - 8.1 嵌入式SQL
  - 8.2 过程化SQL
  - 8.3 存储过程和函数(触发器)
  - 8.4 ODBC编程
  - 8.5 OLE DB
  - 8.6 JDBC编程

访问数据库的统一API



第八章 数据库编程

- SQL的使用：
  - **交互式**

    基本的SQL是非过程化语言，不提供分枝判断、循环等过程化语句
  - **编程**（需要过程化功能）
    - **嵌入式SQL**（前端）

      把SQL语句嵌入到高级语言中，SQL语句进行数据库操作，高级语言完成过程化功能，借助高级语言的控制功能实现过程化
      - **静态嵌入式SQL**

        编译时（运行前）确定SQL语句
      - **动态嵌入式SQL**

        运行时确定SQL语句，根据需要临时“组装”SQL语句
    - **过程化SQL**（后端）

      SQL99对SQL自身进行了结构化扩展，使其具有基本的控制流程，功能几乎相当于通用程序设计语言

8.1 嵌入式SQL

- 什么是嵌入式SQL (*Embedded SQL*)?

  SQL语句嵌入高级语言中，这时高级语言被称为(宿)主语言

- 如何区分主语言与SQL语句?
  - C语言语法格式：**EXEC SQL <SQL语句>;**
  - Java语言（SQLJ）：**#SQL {<SQL语句>};**

  本课程默认使用C语言语法

  例：EXEC SQL DROP TABLE S ;

- SQL与主语言的交互

  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语句是非过程化的面向集合的语句，负责操纵数据库；高级语言语句是过程化的面向记录的语句，负责控制程序流程。这样，二者必然需要互相通信。通信时需要传递哪些信息？使用什么方法？

8.1 嵌入式SQL

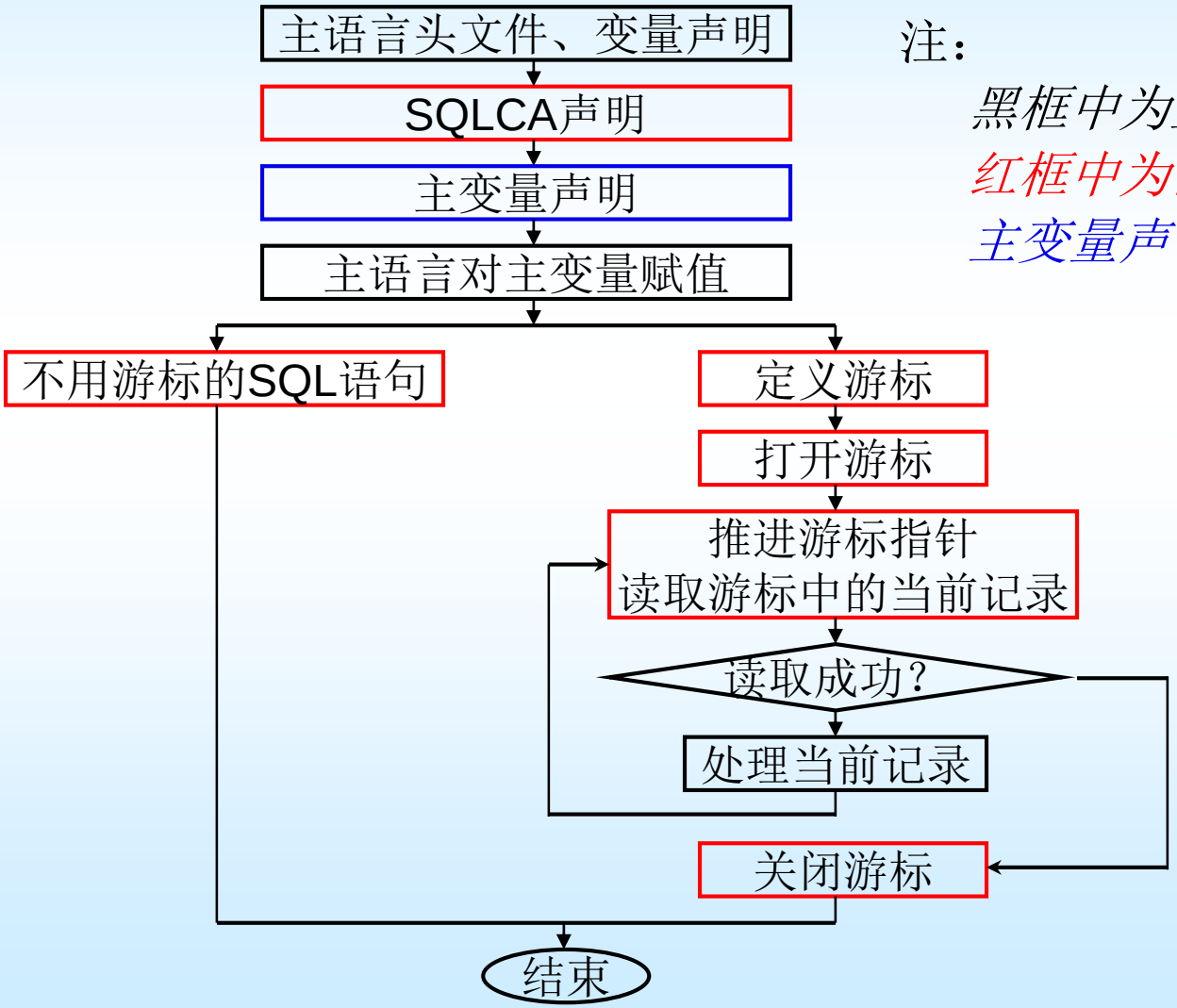
- SQL与主语言的交互
  - **信息**
    - **SQL→主语言：执行状态、查询结果**
    - **主语言→ SQL： 参数**
  - **方法**
    - **SQL通信区(SQLCA，SQL Communication Area)**
      - **向主语言传递SQL语句的执行状态信息**
      - **主语言能够据此控制程序流程**
    - **主变量：主语言的程序变量(指示变量: 主变量状态)**
      - **主语言向SQL语句提供参数**
      - **将SQL语句查询数据库的结果交主语言进一步处理**
    - **游标**

8.1 嵌入式SQL

- 什么是游标(Cursor)?
  - **游标是SQL查询语句向宿主语言提供查询结果集的一段公共缓冲区**
  - **用以解决集合性操作语言与过程性操作语言的不匹配**
  - **嵌入式SQL提供了逐条处理游标记录的功能**
    - 移动游标指针，将记录的各字段赋给主变量，交由宿主语言进一步处理

8.1 嵌入式SQL

☞ 含嵌入式SQL的程序执行流程：



注：

黑框中为主语言

红框中为嵌入式SQL语言

主变量声明是混合语言

8.1.1 不使用游标的SQL语句

- 说明性语句

  说明性语句是专为在嵌入式SQL中说明主变量、SQLCA等而设置的。

  - **说明主变量**

    1. **EXEC SQL BEGIN DECLARE SECTION;**
    2. **EXEC SQL END DECLARE SECTION;**

    这两条语句必须配对出现，相当于一个括号，两条语句中间是主变量的说明。

  - **说明SQLCA**

    3. **EXEC SQL INCLUDE SQLCA;**

8.1.1 不使用游标的SQL语句

- 数据定义语句

  [例1] 建立一个“学生”表S。

```
EXEC SQL CREATE TABLE S
                (Sno     CHAR(5) NOT NULL UNIQUE,
                 Sname   CHAR(20),
                 Ssex    CHAR(1),
                 Sage    INT,
                 Sdept   CHAR(15));
```

- 数据控制语句

  [例2] 把查询S表权限授给用户U1。

```
EXEC SQL GRANT SELECT ON TABLE S TO U1;
```

8.1.1 不使用游标的SQL语句

- 查询结果为单记录的SELECT语句

```
EXEC SQL SELECT  [ALL|DISTINCT]
                    <目标列表达式>[,<目标列表达式>]...
        INTO   <主变量>[<指示变量>]
                    [,<主变量>[<指示变量>]]...
        FROM   <表名或视图名>[,<表名或视图名>] ...
        [WHERE  <条件表达式>]
        [GROUP BY <列名1> [HAVING <条件表达式>]]
        [ORDER BY <列名2> [ASC|DESC]];
```

  - 把从数据库中找到的符合条件的记录，放到INTO子句指出的主变量中去

8.1.1 不使用游标的SQL语句

- 查询结果为单记录的SELECT语句
  - 使用注意事项:

1. 主变量的使用范围

- **INTO子句**
- **WHERE子句的条件表达式**
- **HAVING短语的条件表达式**

2. 使用指示变量

- **指示变量只能用于INTO子句中**
- **当SELECT子句的目标列返回空值时，系统不会给对应的主变量赋值(仍保持执行SQL语句之前的值)，而将其后的指示变量置为负值**
- **当发现指示变量值为负值时，不管主变量为何值，均应认为主变量值为NULL**

8.1.1 不使用游标的SQL语句

- 查询结果为单记录的SELECT语句
  - 使用注意事项:

3. 查询结果为空集

- 如果数据库中没有满足条件的记录，即查询结果为空，则DBMS将SQLCA .SQLCODE的值置为100

4. 查询结果为多条记录

- 程序出错，DBMS会在SQLCA中返回错误信息

8.1.1 不使用游标的SQL语句

[例3] 根据学生号码查询学生信息。

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

主变量givensno需要在此前由宿主语言赋值，且在SQL声明段中已经声明；主变量Hsno, Hname, Hsex, Hage和Hdept也均已在前面的声明段中声明过了。

此SQL语句执行后，INTO子句将查询的返回值赋给对应的五个主变量，可以由主语言进行过程化处理。解决了非过程化向过程化的转变。

8.1.1 不使用游标的SQL语句

[例4] 查询某个学生选修某门课程的成绩。

假设已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
         FROM SC
         WHERE Sno=:givensno AND Cno=:givencno;
```

当该学生成绩为空时，SQL语句执行后会给指示变量Gradeid置为负值，而主变量Hgrade仍保持原来的值(不正确)，这时主程序应根据指示变量的值判定主变量是否为有效值。

在开发应用程序时应慎用此句：随着数据的变动或查询条件的不同，返回记录可能不止一条，造成应用程序出错。

8.1.1 不使用游标的SQL语句

- 非CURRENT形式的UPDATE语句

[例5] 修改某个学生c1号课程的成绩。

假设该学生的学号已赋给givensno，新成绩已赋给newgrade。

```
EXEC SQL UPDATE SC
        SET Grade = :newgrade
        WHERE Cno= 'c1' AND Sno = :givensno ;
```

[例6] 将计算机系全体学生年龄置NULL值。

```
Sageid = -1;
EXEC SQL UPDATE S
        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，DBMS都会将CS系所有记录的年龄属性置空值。

8.1.1 不使用游标的SQL语句

- 非CURRENT形式的DELETE语句

[例7] 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 S
             WHERE Sname=:stdname);
```

8.1.1 不使用游标的SQL语句

- INSERT语句

[例8] 某个学生新选修了某门课程，将有关记录插入SC表。

假设学生的学号已赋给主变量stdno，课程号已赋给主变量couno。

```
gradeid=-1;
EXEC SQL INSERT
        INTO SC(Sno, Cno, Grade)
        VALUES(:stdno, :couno, :gr:gradeid);
```

由于该学生刚选修课程，尚未考试，因此成绩列为空。所以本例中用指示变量指示相应的主变量为空值。

8.1.2 使用游标的SQL语句

☞ 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

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

2. 打开游标

EXEC SQL OPEN <游标名>;

3. 移动游标指针，并取当前记录

EXEC SQL FETCH <游标名>
INTO <主变量>[<指示变量>]
[,<主变量>[<指示变量>]]...;

4. 关闭游标

EXEC SQL CLOSE <游标名>;

8.1.2 使用游标的SQL语句

- CURRENT形式的UPDATE语句和DELETE语句

  WHERE CURRENT OF <游标名>

使用游标的步骤:

(1) 说明游标

(2) 打开游标，把所有满足查询条件的记录从指定表取至缓冲区

(3) 推进游标指针，并把当前记录从缓冲区中取出来送至主变量

(4) 检查该记录是否需要处理(修改或删除)，是则处理之

(5) 重复第(3)和(4)步，用逐条取出结果集中的行进行判断和处理

(6) 关闭游标，释放结果集占用的缓冲区和其他资源

8.1.3 程序实例

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[20];
    char HSno[9];
    char HSname[20];
    char HSsex[2];
    int HSage;
    int NEWAGE;
EXEC SQL END DECLARE SECTION;          /*主变量说明结束*/
EXEC SQL INCLUDE sqlca;                /*定义SQL通信区*/

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*/
```

8.1.3 程序实例

```
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SX*/
      SELECT Sno, Sname, Ssex, Sage                  /*SX对应的查询语句*/
      FROM Student
      WHERE SDept = :deptname;                       /*传入参数*/
EXEC SQL OPEN SX;                 /*打开游标SX，指向查询结果的第一行*/
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，传出状态*/
         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*/
```

8.1.3 程序实例

```
    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');
    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更新语句*/
                     SET Sage = :NEWAGE
                     WHERE CURRENT OF SX ;
           }                                /*对当前游标指向的学生年龄进行更新*/
    }                                                        /*for结束*/
    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*/
}
```

8.1.4 动态SQL

- 前述嵌入式SQL语句在编译时已经确定，可变的只是使用主变量传递的值，其他部分运行时不可改变
- 动态SQL语句是在程序运行时构造并提交
  - 在程序运行过程中，以字符串的形式临时“组装”SQL语句(可能基于用户的输入)
  - 需传入的参数使用参数符号(?)表示

[例] 向TEST中插入元组，假设已创建表test(a int)。

```
EXEC SQL BEGIN DECLARE SECTION;
const char *stmt = "INSERT INTO test VALUES(?);";
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 */
```

作业

- 在嵌入式SQL中是如何协调SQL语言的集合处理方式和主语言的单记录处理方式的?

注：不交

8.2 过程化SQL

- 基本的SQL是非过程化语言。嵌入式SQL把SQL语句嵌入到高级语言中，以此实现过程化
- SQL99对SQL自身进行了结构化扩展，具有基本的控制流程，功能几乎相当于通用程序设计语言(但无图形界面功能)
  - 条件控制：**IF-THEN、CASE**
  - 循环控制：**WHILE、REPEAT、FOR**
  - 错误处理
  - 复合语句：**BEGIN…END**
  - 各个数据库产品的扩展语法差异较大

8.2.1 函数和存储过程

- SQL99允许定义**函数**和**存储过程**(Stored Procedure)，它们与其他编程语言中的函数和过程类似
  - 接受输入参数并以输出参数的格式向外层返回值
  - 包含一组用于在数据库中执行操作的SQL语句，这些语句作为一个单一的工作单元而执行
  - 向外层返回状态值，以指明成功或失败以及失败的原因

- 在使用函数和存储过程前，需用DDL语句创建它们
  - **CREATE FUCTION…**
  - **CREATE PROCEDURE…**

8.2.1 函数和存储过程

- 优点
  - 存储在数据库服务器，且已编译和优化，运行效率高
  - 具有安全特性（例如权限）
    - 用户可以被授予权限来执行函数或存储过程，而不必直接操纵其中引用的对象
  - 可以减少网络通信流量
    - 一个需要数百行SQL代码的操作可以通过一条语句来执行，从而不需要在网络中发送数百行代码
  - 允许模块化程序设计
    - 一旦创建，以后即可在多个程序中调用任意多次
    - 改进应用程序的可维护性和独立性
  - 方便实施企业规则
    - 在后端用函数或存储过程实现企业规则，当业务改变时，只需更改后端函数或存储过程，不用更改前端程序

存储过程实例

8.2.2 触发器

- 触发器(Triger)是数据库服务器中发生事件时(事件驱动)自动执行的特殊过程。
  - 用于满足特定条件时自动执行的任务
  - 功能很强大，可以实现数据库系统的很多重要功能，如完整性、安全性等，以及应用的业务流程和控制流程

8.2.2 触发器

- 触发器的定义

  CREATE TRIGGER <触发器名>

  {BEFORE | AFTER} <触发事件> ON <表名>

  FOR EACH {ROW | STATEMENT}

  [WHEN <触发条件>]

  <触发动作体>

  - 触发事件：**INSERT、DELETE、UPDATE**
  - 各个产品的语法差异较大

- 同一个表上的多个触发器遵循如下的执行顺序：

  （1）执行该表上的**BEFORE**触发器；

  （2）激活触发器的**SQL**语句；

  （3）执行该表上的**AFTER**触发器。

8.2.2 触发器

- 慎用触发器
  - 可能在不需要的时候执行，例如加载备份副本
  - 一个触发器的动作可以引发另外一个触发器，可能导致**无限触发链**
  - 实际上，触发器的很多应用可以用适当的存储过程取代
  - 在必须使用触发器的情况下使用