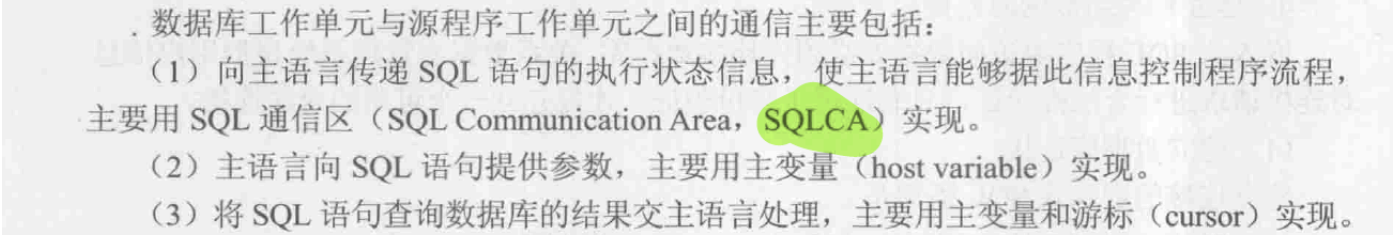
嵌入式SQL

与主语言之间的通信



．数据库工作单元与源程序工作单元之间的通信主要包括：

（1）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，主要用 SQL 通信区（SQL Communication Area，SQLCA）实现。

（2）主语言向 SQL 语句提供参数，主要用主变量（host variable）实现。

（3）将 SQL 语句查询数据库的结果交主语言处理，主要用主变量和游标（cursor）实现。

SQLCA

SQL 语句执行后,系统要反馈给应用程序若干信息,主要包括描述系统当前工作状态和运行环境的各种数据。这些信息将送到SQL通信区中,应用程序从SQL通信区中取出这些状态信息,据此决定接下来执行的语句。

SQL通信区在应用程序中用EXEC SQL INCLUDE SQLCA加以定义。SQL通信区中有一个**变量SQLCODE,用来存放每次执行SQL语句后返回的代码**。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。

如果SQLCODE 等于预定义的常量SUCCESS,则表示 SQL语句成功,否则在SQLCODE存放错误代码。程序员可以根据错误代码查找问题。

主编量

```
Host variable—主变量
indicator variable—指示变量
```

在嵌入式SQL中，主变量（Host variables）是非常有用的工具，它们允许嵌入式SQL语句与主编程语言之间有着直接的数据交换。通过这种方式，可以将SQL查询的结果直接赋值给编程语言中的变量，或者可以将编程语言中的变量值用于SQL查询。这提高了编程语言与数据库交互的灵活性和效率。

输入主变量 通常用于SQL语句中的查询条件，由应用程序对其赋值，SQL语句再引用这个值。例如，在进行数据库查询时，可以使用输入主变量作为查询条件的一部分。

输出主变量 通常用于存储SQL查询的结果。SQL语句执行后，将结果赋值或设置状态信息到输出主变量中，然后返回给应用程序。

指示变量（Indicator variables）与主变量一起使用，用来处理特殊情况，如NULL值、值被截断等。它是一个整型变量，能够“指示”所指主变量的值或条件的状态。

假设我们有一个数据库表 `Employees`，它有两列：`EmpID`(员工ID)和 `Name`(员工姓名)。

我们想要通过员工ID查询员工的姓名。在C语言中使用嵌入式SQL来完成这个任务的方法可能如下：

```
#include <sqlca.h>

/* BEGIN DECLARE SECTION */
EXEC SQL BEGIN DECLARE SECTION;
int emp_id_input;
char emp_name_output[20];
int emp_name_indicator = -1;  // 指示变量，用来检测是否返回空值
EXEC SQL END DECLARE SECTION;
/* END DECLARE SECTION */

/* 将emp_id_input赋值为想要查询的员工ID */
emp_id_input = 101;

/* 执行查询 */
EXEC SQL SELECT name INTO :emp_name_output INDICATOR :emp_name_indicator FROM
Employees WHERE EmpID = :emp_id_input;

if(sqlca.sqlcode == 0) { // SQL 语句执行成功
    if(emp_name_indicator >= 0) { // 处理返回结果
        printf("员工姓名: %s\n", emp_name_output);
    } else {
        printf("没有找到员工或姓名字段为 NULL。\n");
    }
} else { // SQL 语句执行失败
    printf("查询失败。\n");
}
```

在这个例子中：

- `emp_id_input`是一个输入主变量，用于在`WHERE`子句中指定查询条件。
- `emp_name_output`是一个输出主变量，用于存储查询到的员工姓名。
- `emp_name_indicator`是一个指示变量，用于指示`emp_name_output`的值是否为NULL（如果员工不存在或姓名字段为NULL）。

执行查询后，通过检查`sqlca.sqlcode`（一个由SQL通信区（SQLCA）提供的状态变量）来确定SQL语句是否成功执行。然后，通过检查指示变量`emp_name_indicator`来判断是否找到了员工和员工姓名是否为NULL。

游标

为啥有游标：解决sql面向集合，but主语言面向记录，解决这两种语言的差异

> To use a cursor（使用游标的步骤）：declare（说明游标）、open（打开游标）、fetch（推进游标指针并取当前记录）、close（关闭游标）

游标的使用分为四个主要步骤：declare, open, fetch, 和 close。

1. Declare（声明游标）

声明游标是定义游标的第一步，它涉及到指定游标的名字以及对应的SELECT语句，这告诉数据库系统你打算如何使用游标来获取数据。声明时，游标并没有真正的打开或取得数据，它仅仅定义了一个可用来稍后操作的游标。

2. Open（打开游标）

打开游标的步骤实际上初始化了游标所对应的结果集。在这个步骤中，数据库系统执行与游标相关联的SELECT语句，并将结果集设置为游标可以访问的状态。此时，游标指向结果集的第一条记录之前。

3. Fetch（获取数据）

当游标打开后，可以通过fetch操作来逐条获取结果集中的记录。每次fetch操作都会移动游标到下一条记录，并将那条记录的数据赋值给事先声明的变量。如果到达结果集末尾，fetch操作会通知没有更多的记录可以取得。

4. Close（关闭游标）

关闭游标的操作是在完成数据的遍历之后进行的。执行close操作将释放游标占用的资源和锁定，并将游标的状态置为关闭。这是一个良好的数据库编程习惯，可以防止内存泄漏和其他潜在的问题。

比如说我们有一个数据库表 `Orders`，包括订单号 `order_id`和订单金额 `amount`。要逐行访问某个特定客户的所有订单，我们可以使用如下的游标步骤：

```
-- 声明游标
DECLARE orders_cursor CURSOR FOR SELECT order_id, amount FROM Orders WHERE
customer_id = 'C001';

-- 打开游标
OPEN orders_cursor;

-- 循环，逐条取得并处理每个订单
FETCH NEXT FROM orders_cursor INTO @order_id_variable, @amount_variable;
WHILE @@FETCH_STATUS = 0
BEGIN
    -- 在这里，可以对每一条订单记录 @order_id_variable 和 @amount_variable 进行处理
    PRINT 'Order ID: ' + CAST(@order_id_variable AS VARCHAR) + ', Amount: ' +
CAST(@amount_variable AS VARCHAR);

    -- 为获取下一条记录，再次推进游标
    FETCH NEXT FROM orders_cursor INTO @order_id_variable, @amount_variable;
END

-- 关闭游标，释放资源
CLOSE orders_cursor;
DEALLOCATE orders_cursor; -- 可选的，彻底删除游标定义
```

优化

9.3.2 查询树的启发式优化

本节讨论应用启发式规则（heuristic rules）的代数优化。这是对关系代数表达式的查询树进行优化的方法。典型的启发式规则有：

（1）选择运算应尽可能先做。在优化策略中这是最重要、最基本的一条。它常常可使执行代价节约几个数量级，因为选择运算一般使计算的中间结果大大变小。

（2）把投影运算和选择运算同时进行。如有若干投影和选择运算，并且它们都对同一个关系操作，则可以在扫描此关系的同时完成所有这些运算以避免重复扫描关系。

（3）把投影同其前或后的双目运算结合起来，没有必要为了去掉某些字段而扫描一遍关系。

（4）把某些选择同在它前面要执行的笛卡儿积结合起来成为一个连接运算，连接（特别是等值连接）运算要比同样关系上的笛卡儿积省很多时间。

（5）找出公共子表达式。如果这种重复出现的子表达式的结果不是很大的关系，并且从外存中读入这个关系比计算该子表达式的时间少得多，则先计算一次公共子表达式并把结果写入中间文件是合算的。当查询的是视图时，定义视图的表达式就是公共子表达式的情况。