13. 服务器环境的 SQL 与嵌入式 SQL

13.1 服务器环境的 SQL

两层结构 (two-tier architecture)

- 客户端（前端，client）：写 SQL 语句，报告，分析工具等；
- 服务器端（后端，server）：处理查询，数据库访问等。

三层结构 (three-tier architecture)

- 前端 (front-end)：客户端应用，典型的有浏览器+Web服务器；
- 中间层 (middle-tier)：应用服务器，
- 后端 (back-end)：数据库服务器，执行应用服务器发送的查询。
  - **连接池**：同时维持被应用程序进程共享的多个开放的到数据库的连接。

SQL 环境：所有的数据库元素（表、视图、触发器等等）都在一个 SQL 环境中被定义。

- 数据库元素在 SQL 环境中被安排成层次结构：
  - 模式：一个永久的命名的对象集合；
  - 类别：一个命名的模式集合；
  - 集群：一个命名的类别集合。
- **模式**：包括基本表、视图、限制、断言、触发器等等，是最基本的组成单元，接近我们所说"数据库"的定义，但是比这个定义包含范围少。更多模式的元素包括：定义域与 UDT、字符集、校对（用于有序字符串）、权限以及储存的过程（程序）。
  - 模式的创建

    ```
    CREATE SCHEMA schema_name
      CREATE TABLE ...
      CREATE VIEW ...
      CREATE ASSERTION ...
      ...;
    ```

  - 模式的修改：`ALTER`；模式的删除：`DROP`。
  - 设定当前模式：

    ```
    SET SCHEMA schema_name;
    ```

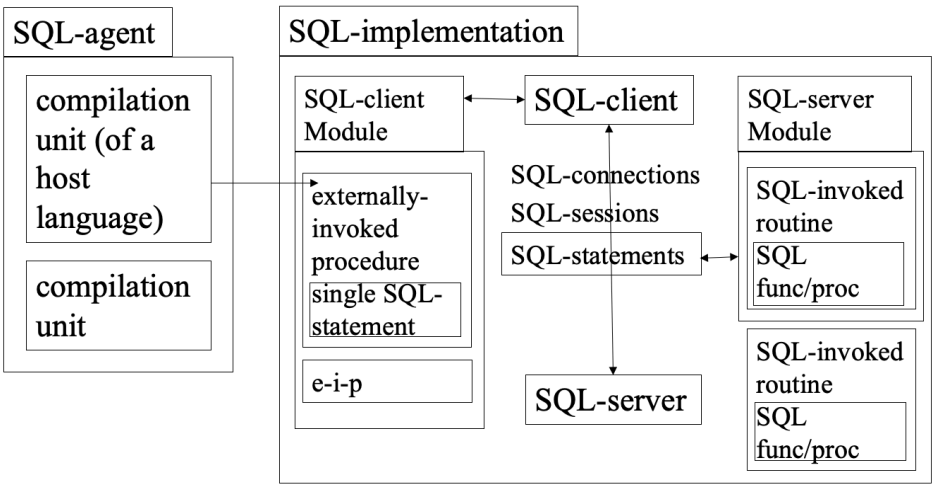
- **类别**：包括模式信息以及多个模式。
  - 模式中的对象可以用 `catalog1.schema2.obj3` 访问；
  - SQL 不定义类似的 `CREATE CATALOG` 指令，而是提供一个指令设置当前的类别：

    ```
    SET CATALOG catalog_name;
    ```

- **集群**：包括多个类别，在 SQL 中没有精确定义。每个用户有一个关联的集群，包含所有用户可访问的类别。一个集群是对于特定用户而言，数据库中可进行操作的最大范围。

SQL 客户端与 SQL 服务端：SQL 的具体实现是使用一个 SQL 服务端执行 SQL 代理 (SQL agent) 所发送的 SQL 指令。

- SQL 客户端 (SQL client)：一个建立自身与 SQL 服务端 SQL 连接的处理器；
- SQL 服务端 (SQL server)：一个管理 SQL 数据的处理器。
  - 同时管理 SQL 会话以及执行从 SQL 客户端接收的 SQL 指令。
- SQL 代理端 (SQL agent)：引起 SQL 语句执行的模块，一般包含源语言编译部分。



客户端与服务端的连接 (connection)

- 建立连接，形成会话：

```
CONNECT TO server_name [AS conn_name] AUTHORIZATION user and password
```

  - `server_name` 可以为默认值；
  - 直接执行 SQL 语句可能导致 SQL 客户端连接至默认服务器并进行 SQL 语句执行。
- 建立的连接可能是**活动的** (active) 或**睡眠的** (dormat)。一般来说，会有许多连接被创建，但是每个时刻只有一个连接是活动的，其余进程是睡眠的。可以通过如下命令切换：

```
SET CONNECTION conn_name
```

- 终止连接，会话结束：

```
DISCONNECT conn_name
```

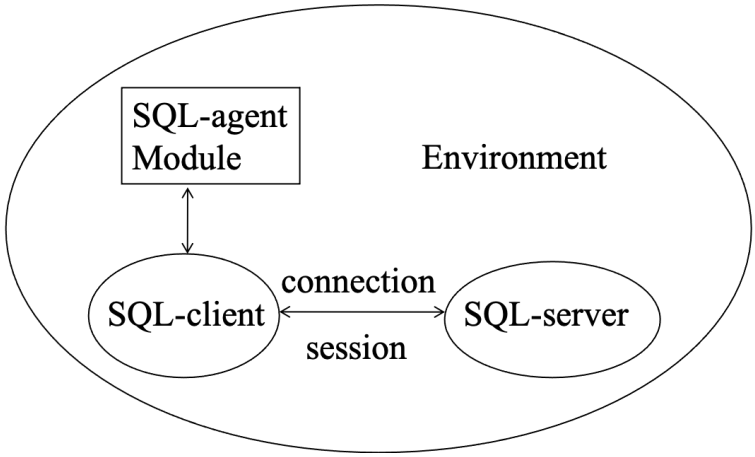
会话 (session)：SQL 服务端在连接是活动的时进行 SQL 操作，形成了一次**会话**。

- 会话与连接是同延的，一旦连接睡眠，会话也睡眠。
- 每个会话有：当前类别，当前模式以及一个授权用户。

模块：SQL 术语，表示一个应用程序。

- 有三种类别的模块：

    - 通用 SQL 接口：每个询问或其他命令本身是一个模块；
    - 嵌入式 SQL：编译后的源语言程序是一个模块；
    - 真正 SQL 程序：SQL 过程/函数的集合。
- **SQL 代理端** (SQL agent)：负责模块的执行，模块与 SQL agent 相当于程序与进程的关系。
    - 其中可能包含若干编译模块；
    - SQL 代理端一般是绑定在 SQL 客户端上的。



13.2 嵌入式 SQL

SQL 的两种模式

- 通用 SQL 接口：点对点查询，较少使用；
- 嵌入式 SQL：提供了调用层面的借口供用户端程序使用。
    - 用户端嵌入 SQL 的程序通过 DBMS 的预处理器将嵌入 SQL 编译为库函数以及相应的调用，然后再链接 SQL 库使用原语言进行编译，得到目标代码。

SQL/HL (Host Language) 接口

- 为了区分 SQL 与 HL 语句，我们加入如下指令表示 SQL 指令：

```
EXEC SQL sql statement
```

- 为方便 SQL 与 HL 的数据交换，使用 **共享变量**，在 HL 中用 `v` 而在 SQL 中用 `:v` 表示。
- 为方便 HL 得知 SQL 的执行情况，使用 `SQLSTATE` 储存 `SQL` 指令的返回值（执行情况），如 `00000` 表示执行成功；`02000` 表示没有数据，`40002` 表示由于 数据完整性约束导致的事务回滚等等。

共享变量

- 共享变量的声明：

```
EXEC SQL BEGIN DECLARE SECTION
  HL variable declarations
EXEC SQL END DECLARE SECTION
```

  其中只能定义 SQL 语言可以处理的类型；下面是一个 C 语言嵌入 SQL 的例子：

```
EXEC SQL BEGIN DECLARE SECTION;
  char snoInput[10], nameInput[20];
  char SQLSTATE[6];
EXEC SQL END DECLARE SECTION;
```

- 共享变量的使用：共享变量可以在 SQL 语句中当作“常量”使用。

```
snoInput = "007";
nameInput = "James Bond";
EXEC SQL INSERT INTO S(sno, name) VALUES(:snoInput, :nameInput);
if (strcmp(SQLSTATE, "00000") != 0)
  printf("INSERT failed.\n");
```

嵌入式 SQL 语句

- 嵌入式 SQL 语句必须是没有返回值的语句，如 `INSERT, DELETE, UPDATE, CREATE, DROP` 等等。
- 不能直接使用 `SELECT ... FROM`，由于其可能输出元组的集合；而 HL 不支持此类型。

单行 `SELECT`：`SELECT` 添加子句 `INTO` 则可以将取出的值存入共享变量：

```
SELECT A1, A2, ...
INTO :v1, :v2, ...
FROM ... WHERE ...
```

集合 `SELECT`

- **游标**：由于 SQL 的 `SELECT` 可能返回一个集合，故使用游标来逐个遍历结果的所有元组并取出，在 HL 中得到 SQL 的查询结果。
- **用法**

```
DECLARE cursor_name CURSOR FOR query
OPEN cursor_name
FETCH FROM cursor_name INTO shared variables
CLOSE cursor_name
```

  - `query` 可以是一个基本表或 `SELECT-FROM-WHERE` 语句；
  - `query` 在游标 `OPEN` 时执行，而不是在 `DECLARE` 时执行；
  - `OPEN` 初始化游标并准备取出第一个元组；
  - `FETCH` 将游标对应的下个元组取出并将游标下移一位；
  - 当游标关闭后，可以再次被打开。
- 典型的例子

```
EXEC SQL DECLARE c CURSOR FOR SELECT ... FROM ... WHERE ...;
EXEC SQL OPEN c;
EXEC SQL FETCH FROM c INTO :v;
while(strcmp(SQLSTATE, "02000")) {
  // ... process the tuple
  EXEC SQL FETCH FROM c INTO :v;
}
EXEC SQL CLOSE c;
```

- **通过游标修改数据**：对于一个基本表中遍历的游标，可以通过游标更新或删除当前元组。使用如下命令即可：

```
DELETE FROM ... WHERE CURRENT OF cursor_name
UPDATE ... SET ... WHERE CURRENT OF cursor_name
```

- **游标选项**：
  - 我们可能并不关心游标访问过程中关系的并发修改；不过我们关心这个过程，我们在定义时候加 `INSENSITIVE` 选项从而使得并发的改动对游标不可见（在一个游标的开关周期内）。

    ```
    DECLARE ... INSENSITIVE CURSOR FOR ...
    ```

  - 只读游标（优先级可能提升）：

    ```
    DECLARE ... CURSOR FOR ... FOR READ ONLY
    ```

- **游标的整体语法**

```
DECLARE cursor_name
  [SENSITIVE | INSENSITIVE | ASENSITIVE]
  [SCROLL | NO SCROLL] CURSOR
  [WITH HOLD | WITHOUT HOLD]
  [WITH RETURN | WITHOUT RETURN] FOR
  query
  [ORDER BY ...]
  [FOR {READ ONLY | UPDATE [OF column]}]
```

静态 SQL 与 动态 SQL

- 静态 SQL：所有嵌入 SQL 语句的细节都在编译时已知；
- 动态 SQL：SQL 语句在运行时才被建立、准备和执行。
  - 例如：用户输入 SQL 语句，用户提供完成 SQL 语句的必要信息。
  - 构造的 SQL 语句 `stmt_str` 在 HL 中以字符串形式呈现。

动态 SQL：所执行的 SQL 语句在编译时信息不完整，直到运行时才得到完整信息。

- **多次执行**：数据准备完成后，需要执行很多次。

```
EXEC SQL PREPARE stmt FROM :stmt_str
```

这个过程将会解释并生成一个询问执行计划，`stmt` 为 SQL 变量（仅在 SQL 语句中使用），`stmt_str` 为一个代表 SQL 语句的字符串，可以有部分参数为 "?" 表示缺失的信息。

```
EXEC SQL EXECUTE stmt
  [INTO HL-variables to receive results]
  [USING HL-variables to substitute for "?"]
```

注意，上述的查询结果必须是单行的，否则需要使用游标。

- **单次执行**：结合准备与执行过程，不保存询问执行计划。

```
EXEC SQL EXECUTE IMMEDIATE :stmt_str
```

  - 没有 `INTO` 语句：没有输出参数，也就是说 `stmt_str` 不能是一个 `SELECT ...` 语句；
  - 没有 `USING` 语句：由于我们只运行一次，不需要多次填入不同参数。
- **游标使用**：与静态 SQL 类似，在 `DECLARE` 前进行 `PREPARE` 即可。