# 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 API

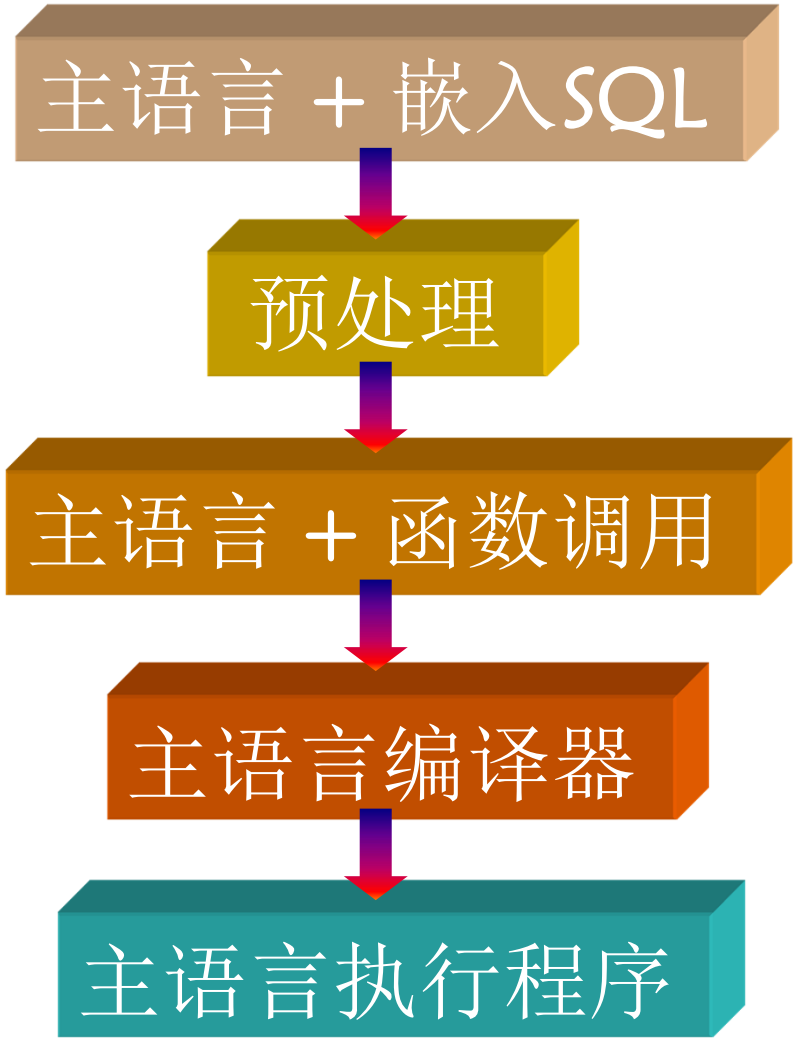
# 8.1嵌入式SQL

- 为什么使用嵌入式SQL？
- 嵌入式SQL执行过程
- 需要解决的几个问题

# 8.1.1为什么使用嵌入式SQL？

- 有些操作对于交互式SQL是不可能的任务
  - SQL的表达能力相比高级语言有一定的限制，有些数据访问要求单纯使用SQL无法完成。一方面，SQL在逐渐增强自己的表达能力，另一方面，太多的扩展会导致优化能力及执行效率的降低
- 非声明性动作
  - 实际的应用系统是非常复杂的，数据库访问只是其中一个部件。有些动作如与用户交互、图形化显示数据等只能用高级语言实现

# 8.1.2 嵌入式SQL执行过程



# 8.1 嵌入式SQL

- 嵌入sql示例

```
vage:=20;//宿主语言
vsno:="s1";
Exec sql update s
        set sage=:vage          where sno=:vsno;
…//宿主语言
```

- 预编译工作模式
  - 预编译将嵌入宿主语言的sql，编译成宿主语言的一段代码，执行这段代码，将完成相应sql调用执行；
- 预编译程序
  - 一般由DBMS供应商提供，如：oracle提供的Pro*c
  - 有些数据库应用开发工具，将预编译程序与主程序编译器合为一体，如：PowerBuilder

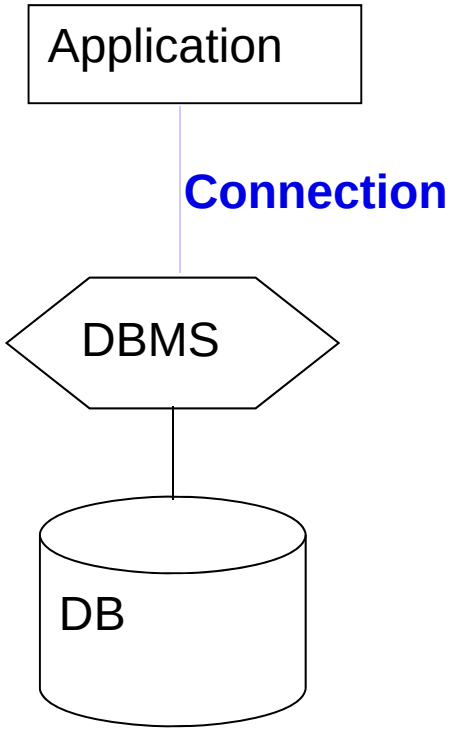
# 数据库连接

- 进行数据库访问必须基于数据库连接
- 数据库连接示例：

```
sqlca.servername="211.87.224.149"
sqlca.userid=…
sqlca.password=…
…
connect sqlca
```

- 撤销数据库连接：

```
disconnect sqlca
```




- 数据库连接建立与撤销的常用策略
  - 一般地，数据库连接可以在应用开始时完成
  - 相应地，在应用结束时撤销数据库连接
  - 应用执行过程中，可以保持数据库连接，以随时执行**sql**

# 区分SQL语句和宿主语句

- 区分SQL语句与C语言语句

  嵌入的SQL语句以EXEC SQL开始，以分号(;) 或END_EXEC结束

  *EXEC SQL　　delete from PROF*

  *where DNO = 10;*

# 数据传递

- 嵌入SQL语句与C语言之间的数据传递
  - 宿主变量
    - C变量，既可以用在C语句中，也可用在SQL语句中，用来在两者之间传递数据
  - 宿主变量的声明
    - 声明为通常的C变量，并将其放在下列标识语句之间

*EXEC SQL BEGIN DECLARE SECTION*

*EXEC SQL END DECLARE SECTION*

# 数据传递

```
EXEC SQL BEGIN DECLARE SECTION
        int   prof_no;
        char prof_name[30];
        int   salary;
EXEC SQL END DECLARE SECTION
```

# 数据传递

注：宿主变量出现于SQL语句中时，前面加（:）以区别列名

宿主变量可出现的地方：SQL的数据操纵语句中可出现常数的任何地方，select，fetch等语句的into子句中

示例：

```
EXEC SQL  select  PNAME , SAL
                  into    :prof_name , :salary
                  from    PROF
                  where   PNO = :prof_no ;
```

# 操作方式的协调

- SQL与主语言之间操作方式的协调
  - 执行方式的差别
    - SQL：一次一集合
    - C语言：一次一记录
  - 游标
    - 在查询结果的记录集合中移动的指针
    - 若一个SQL语句返回单个元组，则不用游标
    - 若一个SQL语句返回多个元组，则使用游标

# 操作方式的协调

- 不需要游标的数据操作
  - 结果是一个元组的select语句

```
EXEC SQL
    select  PNAME , SAL
    into  :prof_name, :salary
   from    PROF
   where  PNO = :prof_no ;
```

# 操作方式的协调

- insert语句

EXEC SQL insert into PROF values (:prof_no, :prof_name , :salary , :dept_no , : salary) ;

- delete语句

EXEC SQL delete from PROF
where PNO > :prof_no ;

- update语句

EXEC SQL update PROF set SAL = :salary
where PNO = : prof_no ;

# 操作方式的协调

- 需要游标的数据操作

  当select语句的结果中包含多个元组时，使用游标可以逐个存取这些元组

  活动集：select语句返回的元组的集合

  当前行：活动集中当前处理的那一行。游标即是指向当前行的指针

# 操作方式的协调

- 游标分类
  - **滚动游标**
    - 游标的位置可以来回移动，可在活动集中取任意元组
  - **非滚动游标**
    - 只能在活动集中顺序地取下一个元组
  - **更新游标**
    - 数据库对游标指向的当前行加锁，当程序读下一行数据时，本行数据解锁，下一行数据加锁

# 操作方式的协调

- 定义与使用游标的语句
  - declare

    定义一个游标，使之对应一个select语句

    declare 游标名 cursor for

    select语句[for update [of列表名]]

    for update：该游标可用于对当前行的修改删除
  - open

    打开游标，执行对应的查询，结果集为该游标的活动集

    open 游标名

# 操作方式的协调

- fetch

  在活动集中将游标移到特定的行，并取出该行数据放到相应的宿主变量中

  fetch [next | prior | first | last | current | relative n | absolute m]

  游标名 into [宿主变量表]

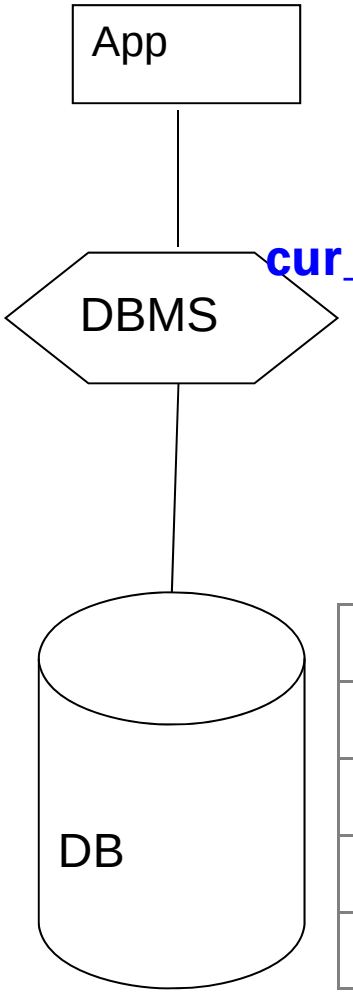
- close

  关闭游标，释放活动集及其所占资源。需要再使用该游标时，执行open语句

  close 游标名

# Cursor应用示例

```
Declare cur_s cursor for
    Select sno,sname from s
        where dept='计' ;
Open cur_s;
Fetch cur_s into :vsno,:vname;
    //get (s1,甲)
Fetch cur_s into :vsno,:vname;
    //get (s2,乙)
Fetch cur_s into :vsno,:vname;
    //no data found
Close cur_s;
```




| S1 | 甲 |
|---|---|
| S2 | 乙 |

S

| Sno | Sname | Dept |
|---|---|---|
| S1 | 甲 | 计 |
| S2 | 乙 | 计 |
| S3 | 丙 | 软 |
| S4 | 丁 | 数学 |

# SQLCA

- SQL语句执行信息反馈
  - 良好的应用程序必须提供对错误的处理，应用程序需要知道SQL语句是否正确执行了，发生错误时的错误代码，执行时遇到特殊情况时的警告信息。
  - SQL通讯域*SQLCA*是一结构,每一嵌入SQL语句的执行情况在其执行完成后写入SQLCA结构中的各变量中, 根据SQLCA中的内容可以获得每一嵌入SQL语句执行后的信息，应用程序就可以做相应的处理。

# SQLCA

- sqlca
  - **Sql Communicate Area**
  - 在宿主语言中宣布
  - 用于宿主语言和**sql**之间通讯
  - 是存储结构和函数的综合体
  - 在**PB**中，一个**SQL**语句执行结束后，**SQLCA.SQLCODE=0**，**100**，**-1**分别表示执行成功，未查找到符合条件的元组，执行出错

# 判定sql执行情况

- APP必须对每一条sql的执行情况进行判定
  - 网络问题、硬件问题、DB模式变更、数据异常、DBMS并发调度问题…，均可导致sql执行失败
  - APP无法保证sql一定能被DBMS正确执行
  - APP必须对每一条sql的执行情况进行判定
- 判定方法
  - 每条sql语句后，APP检查sqlca返回的执行报告
  - 示例：

```
select sname into :vname from s where …
if sqlca.sqlcode<>0 then
    报告错误，进行异常处理，必要时中止应用；
end if
```

# 动态sql

- 静态sql：程序编写时全部确定
- 动态sql：程序在运行时构造并提交执行的sql
- 动态sql示例(PowerBuilder)：

String vtablename,vusername,vstr,vsno,vsname

get vtablename,vusername

vstr=”grant select on ”+vtablename+“ to ”+vusername

execute immediate :vstr;

get vsno,vsname

vstr=“insert into s(sno,sname) values(“”;

vstr=vstr+vsno+”’,’”+vsname+”’)”;

//vstr=insert into s(sno,sname)values (‘s1’,’甲’ )

execute immediate :vstr;

# 8.2 存储过程（续）

1、存储过程综述

2、存储过程的基本操作

　　创建和执行带有参数的存储过程

3、存储过程重新编译

4、系统存储过程和扩展存储过程

# 8.2 存储过程（续）

存储过程综述

问题

要把完成某功能的SQL做成类似C语言的函数，供需要时调用，如何做？

1、存储过程概念

存储过程是一种数据库对象，是为了实现某个特定任务，将一组预编译的SQL语句以一个存储单元的形式存储在服务器上，供用户调用。存储过程在第一次执行时进行编译，然后将编译好的代码保存在高速缓存中以便以后调用，这样可以提高代码的执行效率。

# 8.2 存储过程（续）

- 存储过程同其他编程语言中的过程相似，有如下特点：
- 1）接受输入参数并以输出参数的形式将多个值返回至调用过程或批处理。
- 2）包含执行数据库操作（包括调用其它过程）的编程语句。
- 3）向调用过程或批处理返回状态值，以表明成功或失败（以及失败原因）。

# 8.2 存储过程（续）

基于以下几个方面考虑：

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。

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。

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。

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。

# 8.2 存储过程（续）

## 存储过程与视图的比较

| | 视图 | 存储过程 |
|---|---|---|
| 语句 | 只能是SELECT语句 | 可以包含程序流、逻辑以及SELECT语句 |
| 输入、返回结果 | 不能接受参数，只能返回结果集 | 可以有输入输出参数，也可以有返回值 |
| 典型应用 | 多个表格的连接查询 | 完成某个特定的较复杂的任务 |

# 存储过程的创建

- 用户自定义存储过程只能定义在当前数据库中。在管理存储过程之前，首先需要保证管理存储过程的用户已经获相应的系统权限。既可以通过OEM或SQL Developer等可视化工具管理存储过程，也可以通过PL/SQL管理存储过程。
- 创建存储过程时，在存储过程内可以包含各种PL/SQL语句，但以下语句除外：
  - CREATE VIEW
  - CREATE PROCEDURE
  - CREATE TRIGGER

# 创建存储过程——使用PL/SQL

- PL/SQL中的CREATE PROCEDURE命令用于创建存储过程：
- CREATE [ OR REPLACE ] PROCEDURE [用户方案.]<存储过程名>
- [ ( 参数1 参数模式 数据类型 [, … ] ) ]
- IS | AS
- [变量 数据类型[, …]]
- BEGIN
- PL/SQL语句
- [EXCEPTION 例外处理]
- END
- **参数模式指出参数的类型，有3种参数模式：**
  - IN（输入参数）；OUT（输出参数）；IN OUT（输入输出参数）；

# （1） 创建不带参数存储过程

- 例：创建不带参数存储过程 CNUM，该过程返回Student 表的行数 。
- CREATE OR REPLACE PROCEDURE CNUM
- AS
- CNUMBER INT;
- BEGIN
- SELECT COUNT(*) INTO CNUMBER FROM STUDENT;
- DBMS_OUTPUT.PUTLINE(‘STUDENT表共有‘ || CNUMBER || ’行记录 ‘);
- END CNUM;
- /

事实上，存储过程包含三个部分：声明部分 、执行部分 和异常处理部分 。

存储过程创建完后，可以通过查询数据字典USER_SOURCE来查看其名称和内容 。如查询过程“CNUM”的存储过程内容：

```
SELECT TEXT FROM USER_SOURCE
 WHERE TYPE='PROCEDURE' AND NAME='CNUM';
```

# （2）调用不带参数的存储过程

- ORACLE调用存储过程两种方式：
- （1）EXEC 存储过程名;
- （2）BEGIN
- 存储过程名；
- END;
- /

- 例：前面的 建立的存储过程 CNUM实例调用：
- （1）SET SERVEROUTPUT ON
- EXEC CNUM；
- 由于存储过程 定义中 需要在SQL*PLUS中输出字符串，因此在调用存储 过程前需要先打开 输出设置，即加上“SET SERVEROUTPUT ON”语句，否则无输出 。
- （2）BEGIN
- CNUM；
- END；
- /

# （3）创建带参数的存储过程

- 带参数的存储过程相当于其他程序设计语言中的函数，其可以从变量中获取具体值 并传递到参数中，参与存储过程的执行。
- 例：创建一个带参数的存储过程 GETCLASS，该存储过程获取STUDENT表的学号，输出对应的所在院系信息。

- CREATE OR REPLACE PROCEDURE GETCLASS(SNUM VARCHAR2)
- AS
- SCLASS STUDENT.SDEPT%TYPE;
- BEGIN
- SELECT SDEPT INTO SCLASS FROM STUDENT WHERE SNO=SNUM;
- DBMS_OUTPUT.PUT_LINE(SNUM || '的学生所在院系为' || SCLASS);
- EXCEPTION
- WHEN NO_DATA_FOUND THEN
- DBMS_OUTPUT.PUT_LINE('没有找到该学号 ');
- END;
- /

# （4）调用带参数的存储过程

- 与不带参数的存储过程不同，调用带参数的存储过程需要指定参数。ORACLE中调用带参数的存储过程3种方式：
- （1）EXEC 存储过程名（参数）；
- （2）CALL 存储过程名（参数）；
- （3）BEGIN
- 存储过程名（参数）；
- END;

- 例：调用建立的GETCLASS存储过程，并指定参数为“95001”。
- （1）EXEC GETCLASS (‘95001’);
- （2） CALL GETCLASS (‘95001’);
- （3）BEGIN
- GETCLASS(‘95001’);
- END;
- /

需要注意的是，如果传给一个存储过程的参数是变量时，必须使用上述第3种方式调用，并用DECLARE语句声明变量类型。例如，参数使用变量SNUM的调用实现语句：

```
DECLARE
SNUM STU.SNO%TYPE;
BEGIN
SNUM:='95001';
GETCLASS(SNUM);
END;
```

# （5）创建并调用带输出参数的存储过程

- 前面创建的带参数存储过程GETCLASS只含有一个参数，该参数为输入参数，即接收该参数进行存储过程进行操作。
- 事实上，参数有输入和输出之分，默认情况为输入参数，输出参数用于在存储过程外为其他语句提供输出。
- 例 ：创建含有输入和输出参数的存储过程GETAGE，该过程通过学生的学号查找对应年龄，并以输出参数返回。

- CREATE OR REPLACE GETAGE（SNU
  VARCHAR2，AGE OUT NUMBER）
- AS
- BEGIN
- SELECT SAGE INTO AGE FROM STUD
  SNO=SNUM；
- EXCEPTION
- WHEN NO_DATA_FOUND THEN
- DBMS_OUTPUT.PUT_LINE(‘没有找到该
- END；
- /

```
DECLARE
AGE
NUMBER；
BEGIN
GETAGE
（95001，
AGE）；
DBMS_OUTP
UT.PUT_LINE
(AGE);
END;
/
```

# 8.2 存储过程（续）

## 删除存储过程

**1、在 SQL Developer下 删除存储过程**

右击要删除的存储过程，从弹出的快捷菜单中选择“删除”命令，删除该存储过程。

**2、使用DROP PROCEDURE语句删除存储过程**

**DROP PROCEDURE**语句可以一次从当前数据库中将一个或多个存储过程或过程组删除，其语法格式如下：

**DROP PROCEDURE** 存储过程名称

例：删除存储过程**GETAGE**，其程序清单如下：

```
DROP PROCEDURE GETAGE
/
```

# 练习

- 以教材P56的学生数据库为例，student, course, sc三个表。
- 1、创建一个名为proc_1的存储过程，要求产生学生选课情况列表，包括学号，姓名，课程号，课程名称和成绩。
- 2、 创建一个名为proc_2的存储过程，要求产生某门课的学生选课情况列表，包括课程号，课程名称，学号，姓名，和成绩。（提供课程号作为参数）

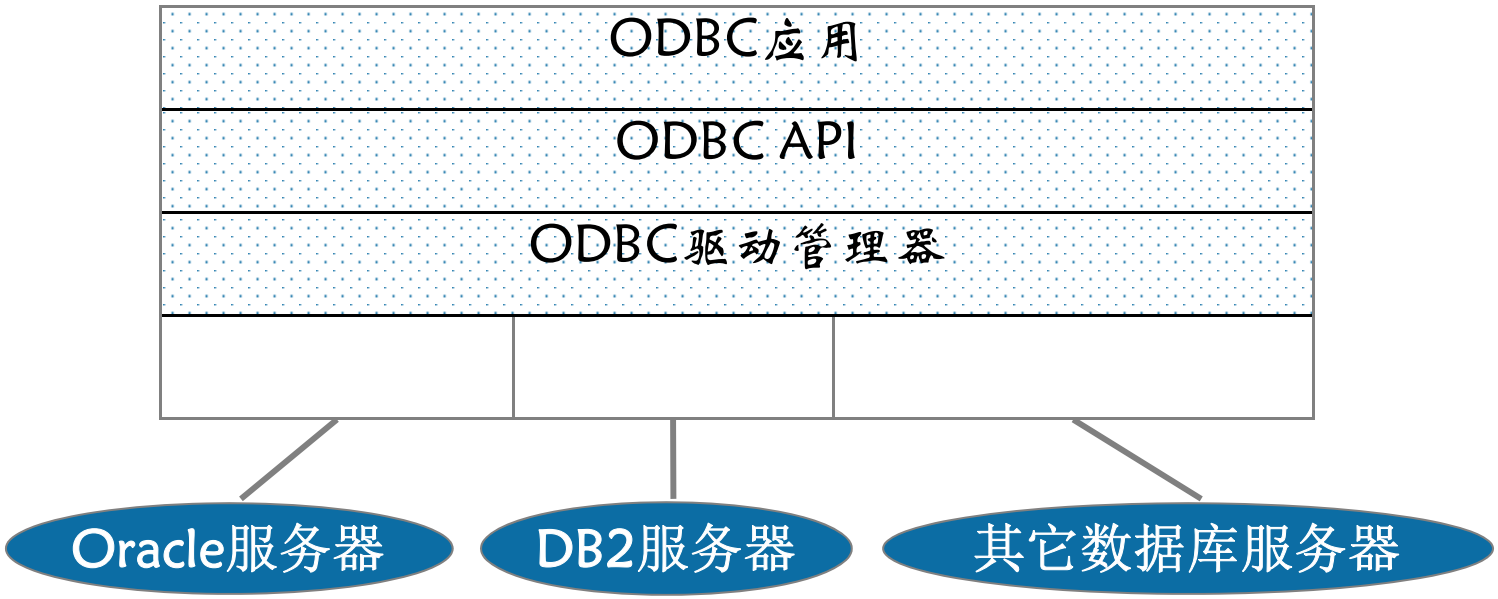
# 8.3 API

- 应用程序访问数据库的方法
  - Embedded sql
  - API
- API:应用程序接口
  - 提供若干函数，通过函数完成sql调用
    *f1*('select sno,sname from s…')
    *f2*(…)
    …
  - API只支持动态SQL,不支持静态SQL
- API的两个主要标准
  - ODBC
  - JDBC

# ODBC

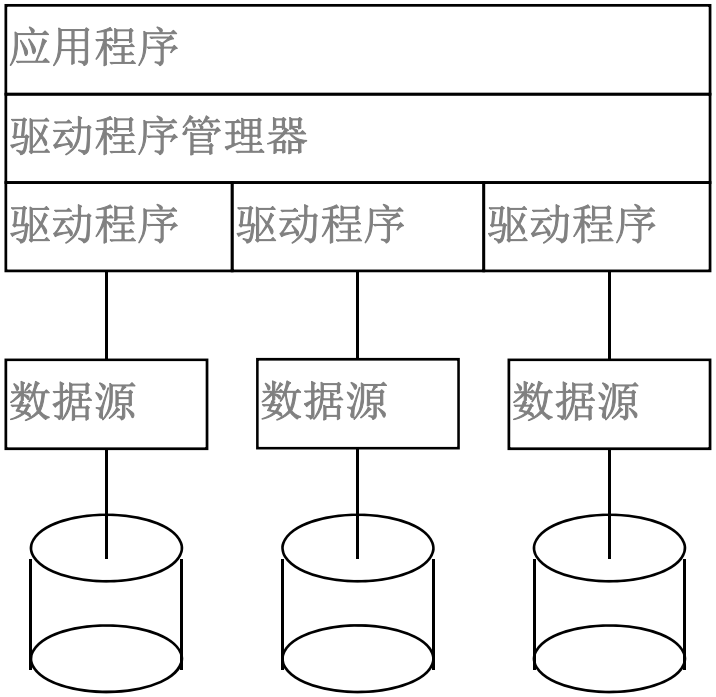
- ODBC（开放数据库互联标准）
  - 适用于客户-服务器体系结构，定义客户程序用以连接到数据库系统和发出**SQL**命令的**API**
  - 客户可以用同一**ODBC API**来连接到任何支持**ODBC**的数据库系统
  - **ODBC**允许用户同时连接到多个数据源并在这些数据源之间进行切换
  - 每个数据库系统必须提供一个驱动程序，受客户端的**ODBC**驱动程序管理器控制，负责与服务器连接和通讯以及进行所有必要的数据和查询格式转换
  - **ODBC API**定义一个**CLI(**调用层接口**)**、一个**SQL**语法定义以及关于允许的**CLI**调用序列的规则

# ODBC



# ODBC

- 开放数据库互连（ODBC）是为了实现异构数据库互连而由Microsoft公司推出的一种标准，它是一个单一的、公共的编程接口。ODBC提供不同的程序以存取不同的数据库，但只提供一种应用编程接口（API）给应用程序




ODBC的体系结构含有四个部件

·应用程序（Application）：执行ODBC函数的调用和处理，提交SQL语句并检索结果

·驱动程序管理器（Driver Manager）：为应用程序装载驱动程序

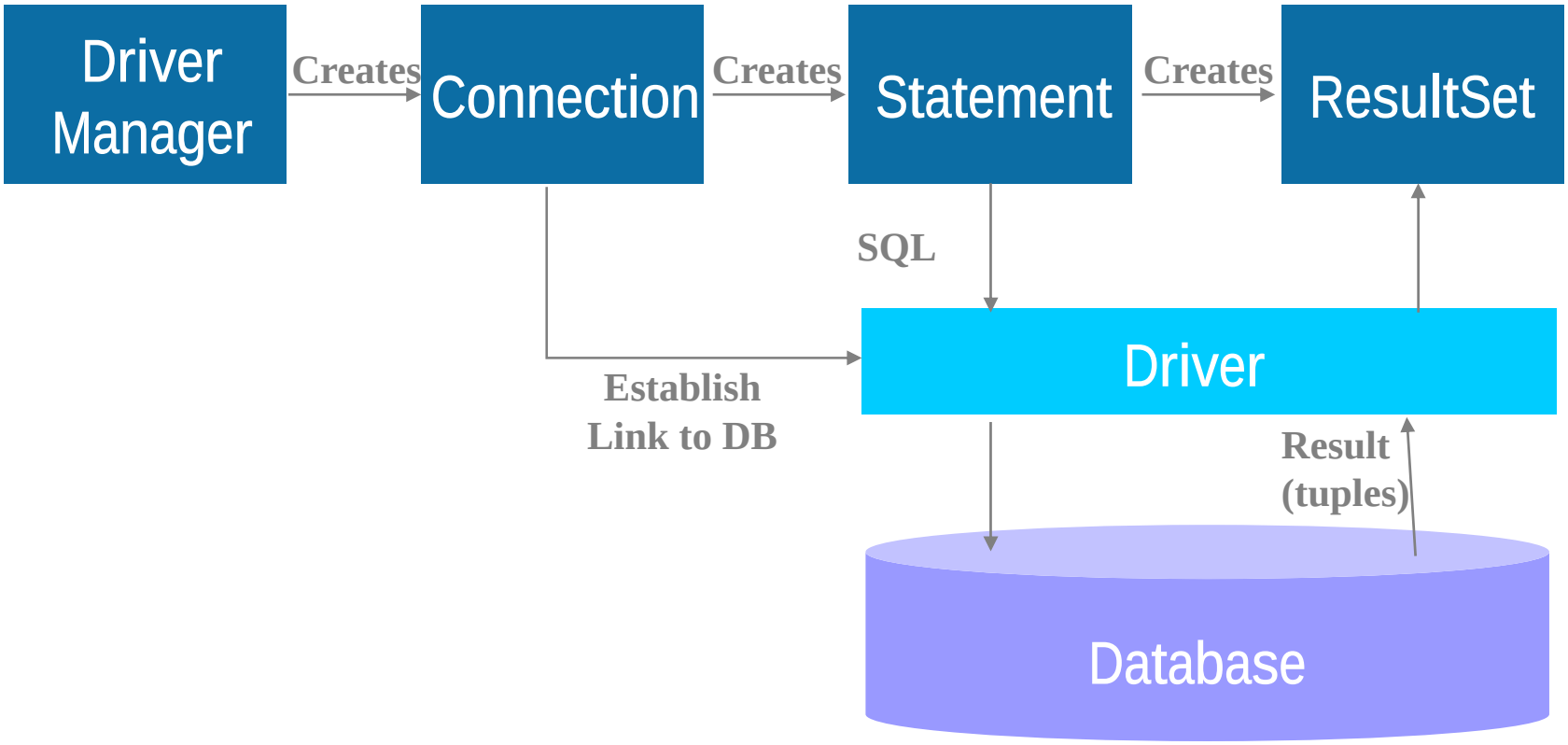
·驱动程序（Driver）：驱动程序是实现ODBC函数调用和同数据源交互作用的动态连接库，它执行ODBC函数调用，提交SQL请求到指定的数据源，并把结果返回给应用程序。如果需要，驱动程序也可改变应用程序的请求，以和特定的DBMS的语法匹配

·数据源（Data Source）：由用户需要存取的数据和与之相连的操作系统、DBMS及存取DBMS的网络平台组成

# JDBC

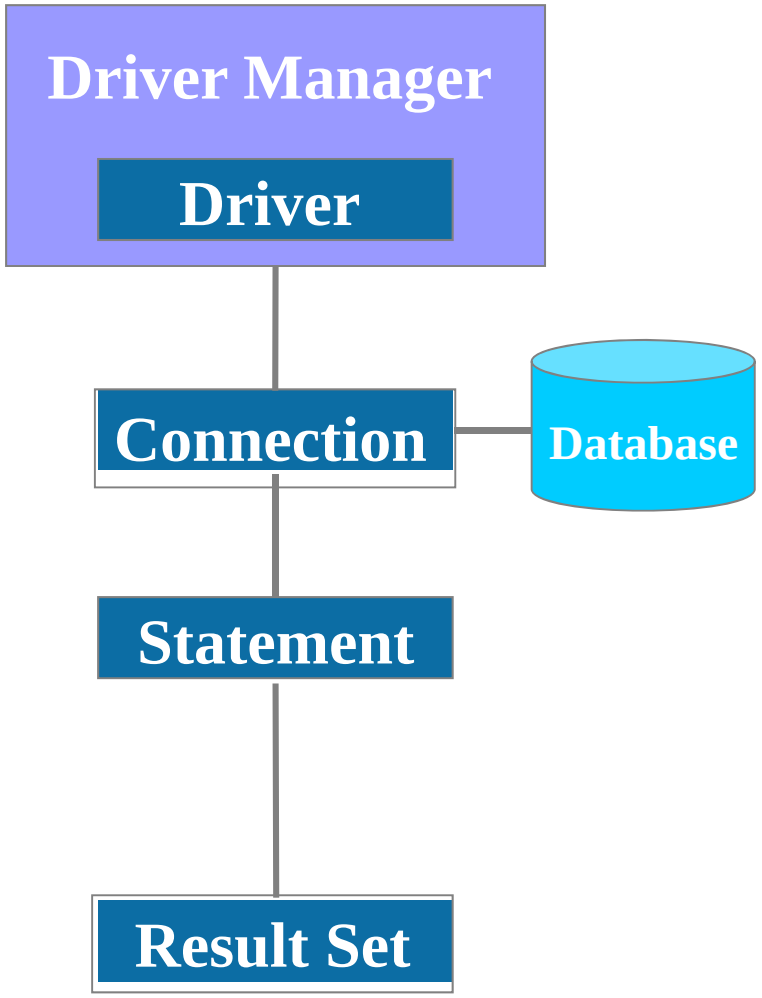
- JDBC
  - **Java DataBase Connectivity**
  - 面向**Java**的标准**API**
  - 其原理、体系结构、用法基本等价于**ODBC**

# JDBC—基本工作步骤


Driver Manager
Creates
Connection
Creates
Statement
Creates
ResultSet
SQL
Establish Link to DB
Driver
Result (tuples)
Database

# JDBC—基本工作步骤

- 1. Load the JDBC driver class
  - Class.forName(“driverName”);
- 2. Open a database connection
  - DriverManager.getConnection (“jdbc:xxx:datasource”);
- 3. Issue SQL statements
  - stmt = con.createStatement();
  - stmt.executeQuery (“Select * from myTable”);
- 4. Process result set
  - while (rs.next()) {
  - name = rs.getString(“name”);
  - amount = rs.getInt(“amt”); }



# JDBC与PowerBuilder对比

| Java | PowerBuilder |
| --- | --- |
| 4种方式加载驱动 | odbc,ado或厂商驱动 |
| Connection | Transaction-SQLCA |
| Statement | 执行各种Sql |
| 第一次Rs.next()==false | Sqlcode =100 |
| executUpdate()返回值 | sqlnrows |
| ResultSet遍历 | Cursor |

# JDBC要点

- sql执行情况判定
- 数据库连接的建立与关闭
- 立即执行vs预备语句
- 数据修改语句的执行
- 查询语句的执行
- 预备语句
- 元数据
- JDBC的事务管理

# JDBC:sql执行情况判定

- JDBC中sql执行情况判定
  - 统一使用**Java**的**Exception**机制
  - 提供类**SQLException**
  - 示例：

```
try
{…
 …
   }
catch (SQLException sqle)
{System.out.println(“SQLExeption:”+sqle);}
```

# JDBC数据库连接建立与关闭

- JDBC访问数据库，同样需要建立数据库连接
- 数据库连接建立与关闭

```
Class.forName("oracle.jdbc.driver.oracledriver");
        //定义驱动程序
Connection conn= DriverManager.getConnection
        (“jdbc:oracle:thin:@202.194.7.x:1000:student”,
“u1”,”pw1”);
        //建立数据库连接
```

- 关闭数据库连接

```
conn.close() ;
```

# JDBC:立即执行vs预备语句

- 立即执行
  - 使用**Statement**类
  - 将**sql**语句直接交给**DBMS**执行
  - 一次语句执行**DBMS**进行一次语句编译
- 使用预备语句执行
  - 使用**PreparedStatement**类
  - **sql**语句执行，首先进行编译，编译结果赋予**PreparedStatement**的对象
  - 预编译的结果可被反复多次执行
  - 同嵌入**sql**预编译不同**(**在编译程序时进行**)**，**JDBC**的预编译是在程序运行中进行的；
- 一个sql多次执行
  - 使用预备语句，仅编译一次；
  - 立即执行模式下，需多次编译
  - 在一**sql**多次执行时，使用预备语句比立即执行的速度快

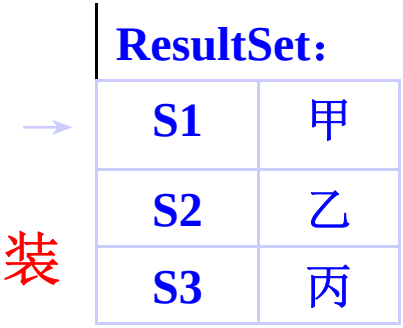
# 数据修改语句的立即执行

- JDBC中，数据查询语句和修改语句使用不同方法执行
- 数据修改语句的执行
  - 使用方法**executeUpdate()**
  - 适于执行**sql**语句**:insert,delete,update,DDL**
- 示例：

```
Statement stmt=conn.createStatement() ;
        //定义statement
stmt.executeUpdate(
        “insert into s(sno,snane) values (‘s1’, ‘甲’   )”);
        //执行statement
stmt.close() ;//释放statement
```

# 查询语句的立即执行

- 查询语句的执行
  - 使用方法**executeQuery()**
  - 返回结果集**ResultSet**
  - **ResultSet是JDBC对DBMS的Cursor的封装**
  - **ResultSet可以使用方法next()遍历**




- 示例：

```
Statement stmt=conn.createStatement();//创建statement
ResultSet rset;
rset =stmt.executeQuery("select sno,sname from s");
while (rset.next())
  {System.out.println(
       rset.getstring("sno")+":"+reset.getstring("sname"));
  }
rset.close();
stmt.close();
```

# JDBC：一个程序示例

```
try
    {Class.forName(“oracle.jdbc.driver.oracledriver”);//定义驱动程序
     Connection conn= DriverManager.getConnection
            ("jdbc:oracle:thin:@202.194.7.x:1000:student", "u1","pw1");
    Statement stmt=conn.createStatement();//定义statement
    try{//插入s(s1,甲)
        stmt.executeUpdate("insert into s(sno,snane) values (‘s1’, ‘甲’  )");
        }   catch (SQLException sqle)//错误处理
            {System.out.println(“could not insert:”+sqle);}
    try {//显示所有学生sno,sname
       ResultSet rset=stmt.executeQuery("select sno,sname from s");
       while (rset.next())
            {System.out.println(rset.getString("sno")+":"+reset.getString("sname"));}
       rset.close(); //释放rset
      }catch (SQLException sqle)//错误处理
            {System.out.println(“select sno,sname err:” +sqle);}
    stmt.close();//释放statement
    conn.close();//释放连接
    }catch (SQLException sqle)
            {System.out.println(“SQLException:”+sqle);}
```

# JDBC:使用预备语句执行

- 使用预备语句执行
  - 预先编译**sql**，编译结果赋予**PreparedStatement**类的对象
  - 预编译的结果可被反复多次执行
  - 一**sql**多次执行，仅编译一次，在多次执行时比立即执行的速度快
- 占位符
  - 预编译**sql**支持占位符“**?**”
  - 相当于宿主变量，只是没有名字，使用数字表示第几个“**?**”
  - **Sql**执行前，要对占位符赋值
  - 使用占位符可以增强防攻击的能力
- 示例：

```
PreparedStatement pstmt=conn.prepareStatement(
        "insert into s(sno,snane) values (?,? )”);
pstmt.setString(1,"s1") ;
pstmt.setString(2,"甲") ;
pstmt.excuteUpdate() ;
pstmt.setString(1,"s2") ;
pstmt.setString(2,"乙") ;
pstmt.excuteUpdate() ;
pstmt.close() ;
```

# 查询结果集元数据

- 元数据：描述数据的数据
- 查询结果集元数据
  - 描述查询结果集的属性类型(结果集的模式)
  - 对编程时不能确定结果集模式时非常有用
  - 示例，自由查询程序：用户输入查询sql语句，显示结果

**ResultSet:rset**

| Sno | Sname | Sage |
|---|---|---|
| S1 | 甲 | 20 |
| S3 | 丙 | 21 |

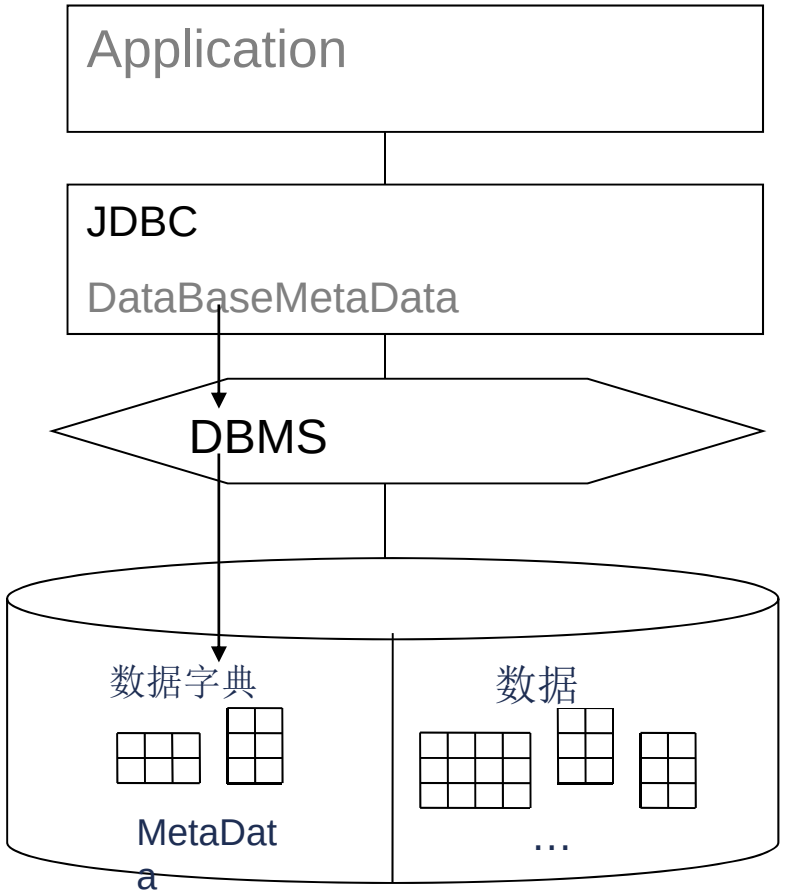
```
String vsqlstring=getSqlStringfromUser()
//假设用户输入：select sno,sname,sage from s where dept='计'
Statement stmt=conn.createStatement() ;//创建statement
ResultSet rsett=stmt.executeQuery(vsqlstring);
ResultSetMetaData rsmd=rset. getMetaData()
for(int i=1;i<=rsmd.getColumnCount();i++)
        {rsmd.getColumnName(i);
         rsmd.getColumnTypeName(i);
        …//根据结果集模式，建立显示结果的表格
        }
while (rset.next())
        {…}//根据结果集模式，显示一行结果数据
… //其它工作，释放Statement等
```

ResultSetMetaData:rsmd

| Column number | Column Name | Column Type | … |
|---|---|---|---|
| 1 | sno | char | … |
| 2 | sname | varchar | … |
| 3 | sage | int | … |

# 数据库模式元数据

- DataBaseMetaData
  - **JDBC**类
  - 对**DB**数据字典进行封装
  - 类方法可以读取数据字典元数据
  - 屏蔽了数据字典的具体实现模式
  - 对应用提供访问**DB**数据字典元数据的标准方法



# JDBC：事务管理

- JDBC事务管理
  - 自动提交
  - 非自动提交
- 自动提交事务
  - 每个**sql**语句作为一个独立事务管理
  - 每个**sql**语句执行后，事务自动提交
  - 设置事务管理模式为自动提交的命令：
    **conn.setAutoCommit(true);**
- 非自动提交
  - 提交事务：**conn.commit();**
  - 回滚事务：**conn.rollback();**
  - 提交/回滚：结束原事务，下一**sql**开始新事务
  - 设置非自动提交的命令：
    **conn.setAutoCommit(false);**

# 应用程序访问数据库方式比较

| 动态SQL vs 静态SQL、嵌入SQL vs API<br>对比图解 | | | |
|---|---|---|---|
| | | EmbeddedSQL<br>(Pro*c,SQLj,PB…) | API<br>(ODBC,JDBC) |
| 静态SQL（限DML）<br>（SQL在程序编译时一次性编译） | | √ | — |
| 动态SQL<br>（DML/DDL）<br>(SQL在程序运行时编译) | 准备(Prepare)<br>(一次编译，可反复执行) | √ | √ |
| | 立即(direct /Immediate)<br>(一次编译，一次执行) | √ | √ |