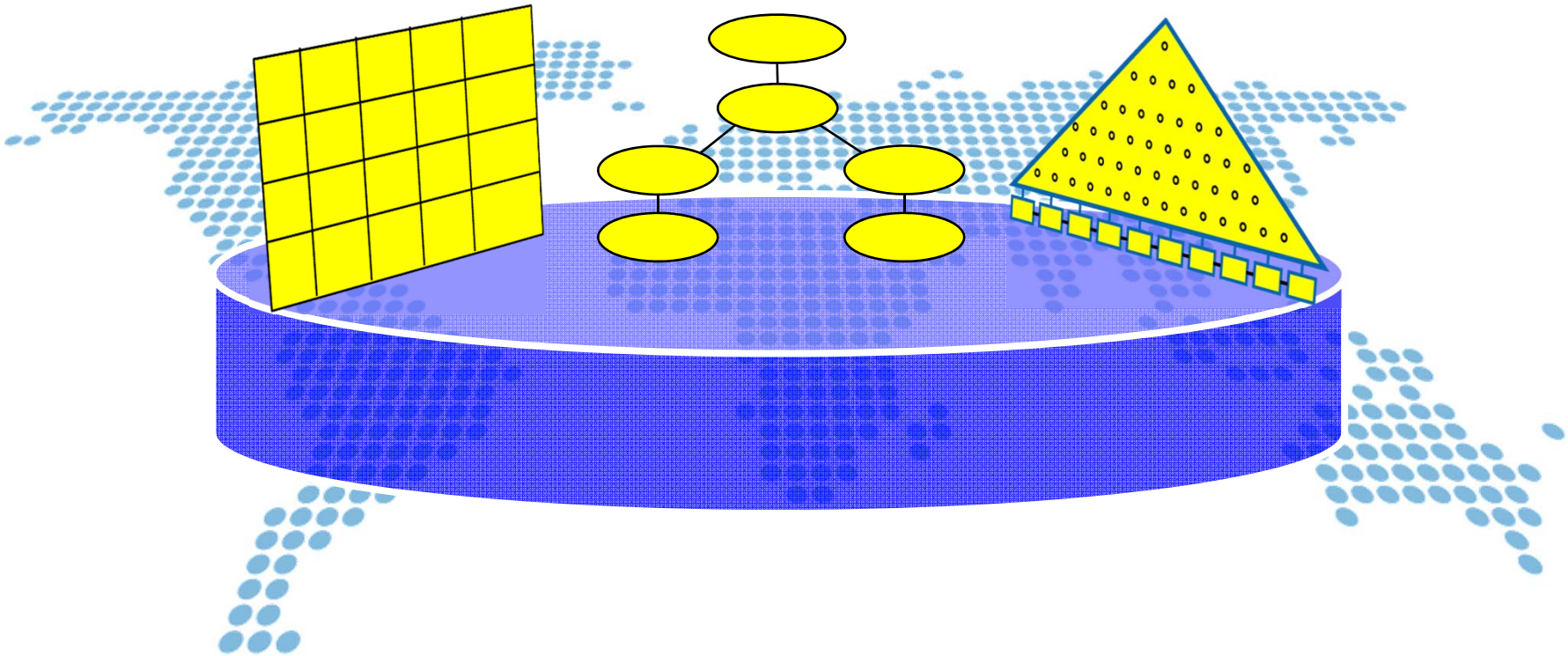
数据库系统

数据库程序设计

陈世敏

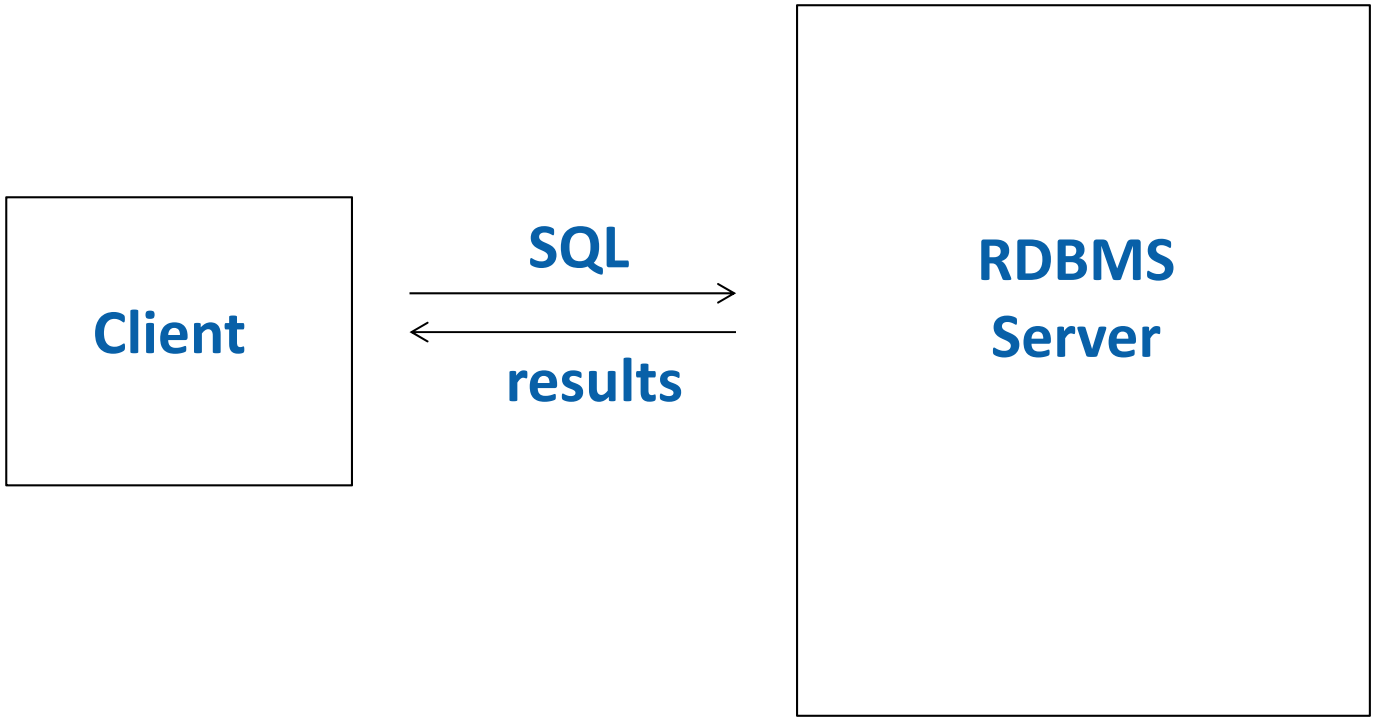
（中科院计算所）



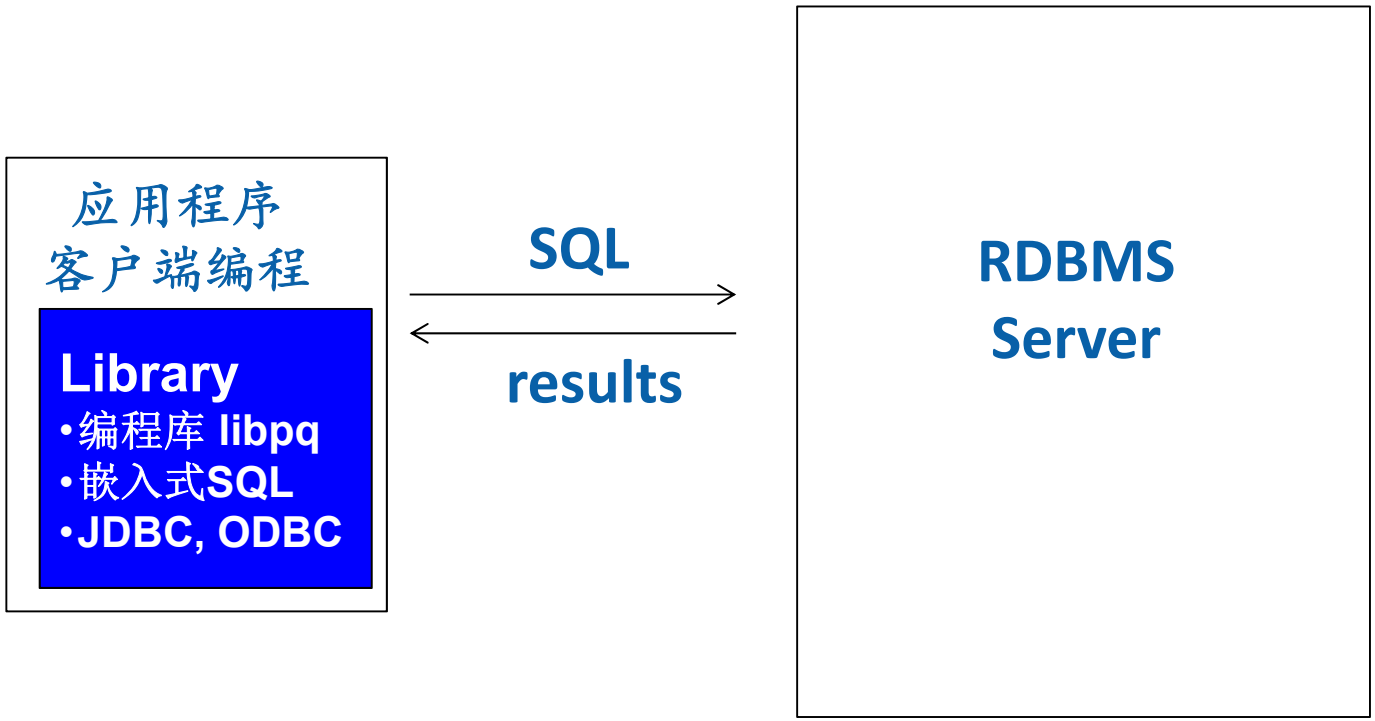
Outline

- 数据库编程简介
- 数据库客户端编程
- 数据库系统内部的扩展编程

数据库系统为典型的Client / Server

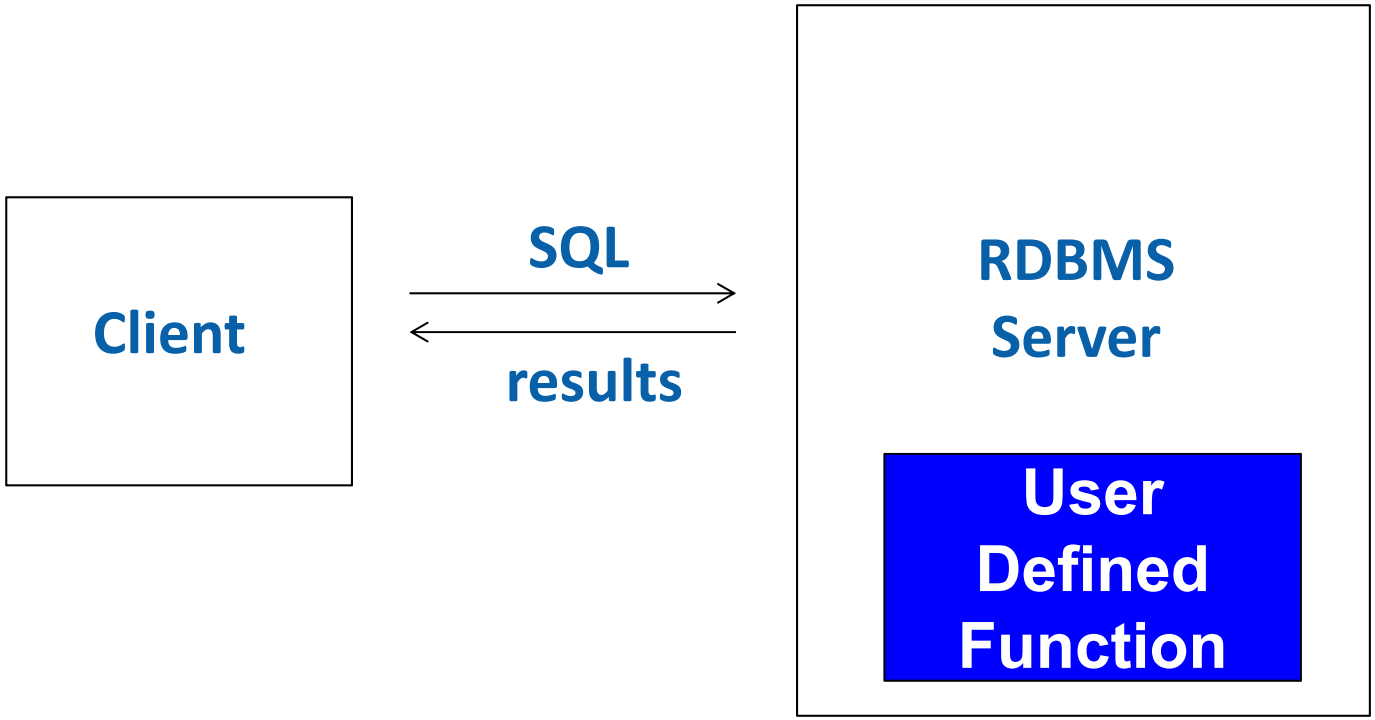


方法1：应用程序➔数据库系统



- Psql实际上可看作是一个应用程序，就是如此实现

方法2：数据库系统内部运行应用程序



PostgreSQL开发环境

```
# apt update
# apt install libpq-dev libecpg-dev postgresql-server-dev-10
```

- libpq开发相关文件： libpq-dev
- 嵌入式SQL开发相关文件： libecpg-dev
- 数据库系统服务端开发相关文件：postgresql-server-dev-10

```
# service postgresql restart
```

- 安装了什么，可以用下述命令看

```
$ dpkg-query -L 安装包的名称
```

Outline

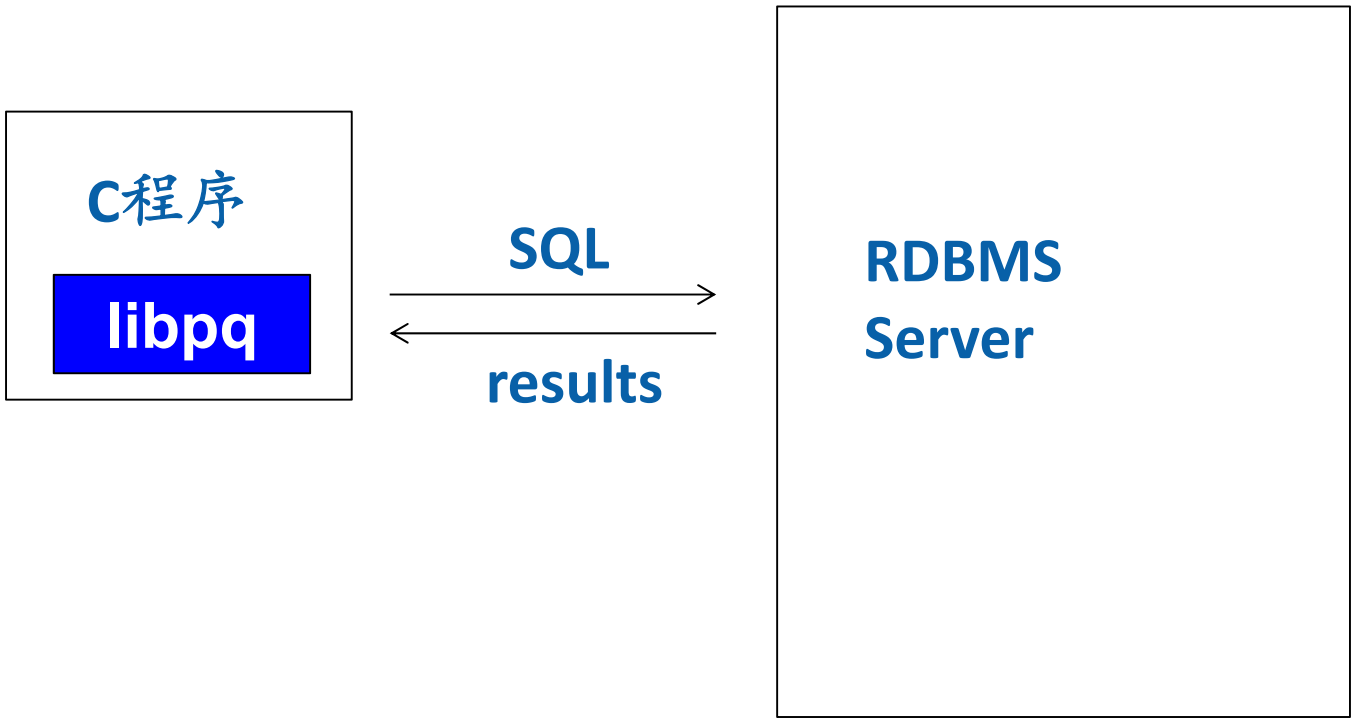
- 数据库编程简介
- 数据库编程
  - Libpq
  - 嵌入式SQL
  - JDBC和ODBC
- 数据库系统内部的扩展编程

总思路

- 应用编程方式都需要提供下述功能
  ① 连接数据库系统
  ② 执行SQL语句
  ③ 获得执行结果

- 下面以libpq为例，其他的方式大家可以课后自己学习，具体用时，可以仔细地看讲义、书、编程手册

libpq



- C程序调用libpq提供的函数
- Libpq与后端数据库系统进行连接，完成要求的操作

myexample1.c

```c
#include <stdio.h>
#include <postgresql/libpq-fe.h>

int main(int argc, char *argv[])
{
    PGconn* conn = PQconnectdb("dbname=tpch");

    if (PQstatus(conn) != CONNECTION_OK) {
      fprintf(stderr, "Connection to db failed: %s",
                      PQerrorMessage(conn));
      PQfinish(conn);
      return 1;
    }

    printf("Connection is successful!\n");
    PQfinish(conn);
    return 0;
}
```

源文件在/home/dbms/setup/db-programming/libpq

myexample1.c

```c
#include <stdio.h>
#include <postgresql/libpq-fe.h>

int main(int argc, char *argv[])
{
    PGconn* conn = PQconnectdb("dbname=tpch");

    if (PQstatus(conn) != CONNECTION_OK) {
      fprintf(stderr, "Connection to db failed: %s",
                      PQerrorMessage(conn));
      PQfinish(conn);
      return 1;
    }

    printf("Connection is successful!\n");
    PQfinish(conn);
    return 0;
}
```

#include 头文件，注意因为头文件在/usr/include/postgresql下面，必须使用相对路径

myexample1.c

建立数据库连接

```c
#include <stdio.h>
#include <postgresql/libpq-fe.h>

int main(int argc, char *argv[])
{
    PGconn* conn = PQconnectdb("dbname=tpch");

    if (PQstatus(conn) != CONNECTION_OK) {
      fprintf(stderr, "Connection to db failed: %s",
                      PQerrorMessage(conn));
      PQfinish(conn);
      return 1;
    }

    printf("Connection is successful!\n");
    PQfinish(conn);
    return 0;
}
```

PQconnectdb建立向后端数据库系统的连接，这里打开的数据库是tpch（是我们在实验1中创建的），返回PGconn*

myexample1.c

```c
#include <stdio.h>
#include <postgresql/libpq-fe.h>

int main(int argc, char *argv[])
{
    PGconn* conn = PQconnectdb("dbname=tpch");

    if (PQstatus(conn) != CONNECTION_OK) {
      fprintf(stderr, "Connection to db failed: %s",
                      PQerrorMessage(conn));
      PQfinish(conn);
      return 1;
    }

    printf("Connection is successful!\n");
    PQfinish(conn);
    return 0;
}
```

PQstatus对返回的连接conn检查状态

CONNECTION_OK是一个状态常量，表示一切正常

myexample1.c

```c
#include <stdio.h>
#include <postgresql/libpq-fe.h>

int main(int argc, char *argv[])
{
    PGconn* conn = PQconnectdb("dbname=tpch");

    if (PQstatus(conn) != CONNECTION_OK) {
        fprintf(stderr, "Connection to db failed: %s",
                        PQerrorMessage(conn));
        PQfinish(conn);
        return 1;
    }

    printf("Connection is successful!\n");
    PQfinish(conn);
    return 0;
}
```

如果不正常，那么就向标准错误输出stderr写错误信息

PQerrorMessage返回连接conn错误信息

myexample1.c

结束数据库连接

```c
#include <stdio.h>
#include <postgresql/libpq-fe.h>

int main(int argc, char *argv[])
{
    PGconn* conn = PQconnectdb("dbname=tpch");

    if (PQstatus(conn) != CONNECTION_OK) {
      fprintf(stderr, "Connection to db failed: %s",
                      PQerrorMessage(conn));
      PQfinish(conn);
      return 1;
    }

    printf("Connection is successful!\n");
    PQfinish(conn);
    return 0;
}
```

PQfinish结束连接conn

程序结束时返回0通常认为是正常，非0认为是出错

myexample1.c编译运行

- 编译

```
$ gcc -g -Wall -o myexample1 myexample1.c –lpq
```

注意-lpq链接了libpq动态链接库

- 运行

```
dbms@ubuntu:~/db-programming/libpq$ ./myexample1
Connection is successful!
```

myexample2.c

```c
#include <stdio.h>
#include <stdlib.h>
#include <postgresql/libpq-fe.h>

void exitNicely(PGconn* conn)
{ PQfinish(conn); exit(1);}

int main(int argc, char *argv[])
{
    PGconn* conn = PQconnectdb("dbname=tpch");

    if (PQstatus(conn) != CONNECTION_OK) {
      fprintf(stderr, "Connection to db failed: %s\n", PQerrorMessage(conn));
      exitNicely(conn);
    }

    PGresult *res = PQexec(conn, "select r_regionkey, r_name from region");

    ExecStatusType status= PQresultStatus(res);
    if ((status!=PGRES_TUPLES_OK)&&(status!=PGRES_SINGLE_TUPLE)){
      fprintf(stderr, "failed execution: %s\n", PQresultErrorMessage(res));
      exitNicely(conn);
    }

    PQprintOpt po = {0};
    po.header=1; po.align=1; po.fieldSep="|";
    PQprint(stdout, res, &po);

    PQclear(res);

    PQfinish(conn);
    return 0;
}
```

myexample2.c：中间部分

进行SQL查询

```
17    PGresult *res = PQexec(conn, "select r_regionkey, r_name from region");
18
19    ExecStatusType status= PQresultStatus(res);
20    if ((status!=PGRES_TUPLES_OK)&&(status!=PGRES_SINGLE_TUPLE)){
21      fprintf(stderr, "failed execution: %s\n", PQresultErrorMessage(res));
22      exitNicely(conn);
23    }
24
25    PQprintOpt po = {0};
26    po.header=1; po.align=1; po.fieldSep="|";
27    PQprint(stdout, res, &po);
28
29    PQclear(res);
```

PQexec对于打开的数据库连接，执行一个SQL语句

myexample2.c：中间部分

获取查询结果状态

```
17    PGresult *res = PQexec(conn, "select r_regionkey, r_name from region");
18
19    ExecStatusType status= PQresultStatus(res);
20    if ((status!=PGRES_TUPLES_OK)&&(status!=PGRES_SINGLE_TUPLE)){
21      fprintf(stderr, "failed execution: %s\n", PQresultErrorMessage(res));
22      exitNicely(conn);
23    }
24
25    PQprintOpt po = {0};
26    po.header=1; po.align=1; po.fieldSep="|";
27    PQprint(stdout, res, &po);
28
29    PQclear(res);
```

PQresultStatus检查执行结果是否正确，若不正确，输出PQresultErrorMessage错误信息，然后调用exieNicely

myexample2.c：中间部分

打印查询结果

```
17    PGresult *res = PQexec(conn, "select r_regionkey, r_name from region");
18
19    ExecStatusType status= PQresultStatus(res);
20    if ((status!=PGRES_TUPLES_OK)&&(status!=PGRES_SINGLE_TUPLE)){
21      fprintf(stderr, "failed execution: %s\n", PQresultErrorMessage(res));
22      exitNicely(conn);
23    }
24
25    PQprintOpt po = {0};
26    po.header=1; po.align=1; po.fieldSep="|";
27    PQprint(stdout, res, &po);
28
29    PQclear(res);
```

PQprint打印执行结果

输出列名和记录条数（head=1），列对齐（align=1），分隔符为|（fieldSep=“|”）

myexample2.c：中间部分

```
17      PGresult *res = PQexec(conn, "select r_regionkey, r_name from region");
18
19      ExecStatusType status= PQresultStatus(res);
20      if ((status!=PGRES_TUPLES_OK)&&(status!=PGRES_SINGLE_TUPLE)){
21        fprintf(stderr, "failed execution: %s\n", PQresultErrorMessage(res));
22        exitNicely(conn);
23      }
24
25      PQprintOpt po = {0};
26      po.header=1; po.align=1; po.fieldSep="|";
27      PQprint(stdout, res, &po);
28
29      PQclear(res);
```

PQclear释放res占用的空间

myexample2.c编译运行

- 编译

```
$ gcc -g -Wall -o myexample2 myexample2.c -lpq
```

- 运行

```
dbms@ubuntu:~/db-programming/libpq$ ./myexample2
r_regionkey|r_name
-----------+-------------------------
          0|AFRICA
          1|AMERICA
          2|ASIA
          3|EUROPE
          4|MIDDLE EAST
(5 rows)

dbms@ubuntu:~/db-programming/libpq$
```

myexample3.c 获取查询结果的行列值

```
25      int num_rows = PQntuples(res);
26      int num_cols = PQnfields(res);
27      int r, i;
28      char *val;
29
30      for (i=0; i<num_cols; i++) {
31         printf("%s", PQfname(res, i));
32         printf("%c", (i<num_cols-1)?',':'\n');
33      }
34
35      for (r=0; r<num_rows; r++) {
36         for (i=0; i<num_cols; i++) {
37            val= PQgetvalue(res, r, i);
38            printf("%s", val);
39            printf("%c", (i<num_cols-1)?',':'\n');
40         }
41      }
```

myexample3.c 获取查询结果的行列值

```
25      int num_rows = PQntuples(res);
26      int num_cols = PQnfields(res);
27      int r, i;
28      char *val;
29
30      for (i=0; i<num_cols; i++) {
31          printf("%s", PQfname(res, i));
32          printf("%c", (i<num_cols-1)?',':'\n');
33      }
34
35      for (r=0; r<num_rows; r++) {
36          for (i=0; i<num_cols; i++) {
37              val= PQgetvalue(res, r, i);
38              printf("%s", val);
39              printf("%c", (i<num_cols-1)?',':'\n');
40          }
41      }
```

行数、列数

列名

myexample3.c 获取查询结果的行列值

```
25      int num_rows = PQntuples(res);
26      int num_cols = PQnfields(res);
27      int r, i;
28      char *val;
29
30      for (i=0; i<num_cols; i++) {
31          printf("%s", PQfname(res, i));
32          printf("%c", (i<num_cols-1)?',':'\n');
33      }
34
35      for (r=0; r<num_rows; r++) {
36          for (i=0; i<num_cols; i++) {
37              val= PQgetvalue(res, r, i);
38              printf("%s", val);
39              printf("%c", (i<num_cols-1)?',':'\n');
40          }
41      }
```

得到r行i列的值

- PQexec默认的返回值是文本类型的

编译运行myexample3.c

- 运行

```
dbms@ubuntu:~/db-programming/libpq$ ./myexample3
r_regionkey,r_name
0,AFRICA
1,AMERICA
2,ASIA
3,EUROPE
4,MIDDLE EAST
dbms@ubuntu:~/db-programming/libpq$ _
```

- 注意我们用了逗号作为分隔符

Libpq：反复执行同一个SQL语句？

- 如果要反复执行一个SQL语句，但是有不同参数

  一个例子：

```
select n_name
from region, nation
where n_regionid = r_regionid
   and r_name=$1;
```

  又一个例子：

```
insert into region(r_regionid, r_name)
values ($1, $2);
```

myexample4.c 采用模板

```
17      PGresult *res = PQprepare(conn, "query_temp",
18                "select n_name from region, nation "
19                "where n_regionkey = r_regionkey and r_name=$1",
20                1, NULL);
21      if (PQresultStatus(res) != PGRES_COMMAND_OK) {
22        fprintf(stderr, "failed execution: %s\n", PQresultErrorMessage(res));
23        exitNicely(conn);
24      }
25      PQclear(res);
26
27      char *param_val[1]= {"ASIA"};
28      int   param_len[1]= {0};
29      int   param_format[1]= {0};
30
31      res = PQexecPrepared(conn, "query_temp", 1, (const char **)param_val,
32                           param_len, param_format, 0);
```

注：其它部分与myexample3.c相同。

myexample4.c 采用模板

```
17    PGresult *res = PQprepare(conn, "query_temp",
18              "select n_name from region, nation "
19              "where n_regionkey = r_regionkey and r_name=$1",
20              1, NULL);
21    if (PQresultStatus(res) != PGRES_COMMAND_OK) {
22      fprintf(stderr, "failed execution: %s\n", PQresultErrorMessage(res));
23      exitNicely(conn);
24    }
25    PQclear(res);
26
27    char *param_val[1]= {"ASIA"};
28    int   param_len[1]= {0};
29    int   param_format[1]= {0};
30
31    res = PQexecPrepared(conn, "query_temp", 1, (const char **)param_val,
32                         param_len, param_format, 0);
```

PQprepare准备模板，检查结果，然后释放结果空间

myexample4.c 采用模板

```
17      PGresult *res = PQprepare(conn, "query_temp",
18                "select n_name from region, nation "
19                "where n_regionkey = r_regionkey and r_name=$1",
20                1, NULL);
21      if (PQresultStatus(res) != PGRES_COMMAND_OK) {
22        fprintf(stderr, "failed execution: %s\n", PQresultErrorMessage(res));
23        exitNicely(conn);
24      }
25      PQclear(res);
26
27      char *param_val[1]= {"ASIA"};
28      int   param_len[1]= {0};
29      int   param_format[1]= {0};
30
31      res = PQexecPrepared(conn, "query_temp", 1, (const char **)param_val,
32                           param_len, param_format, 0);
```

PQexePrepared执行模板，注意参数的设置

编译运行myexample4.c

- 运行

```
dbms@ubuntu:~/db-programming/libpq$ ./myexample4
n_name
INDIA
INDONESIA
JAPAN
CHINA
VIETNAM
```

嵌入式SQL

- SQL标准定义
- 在C程序中增加特殊的标记
  EXEC SQL 嵌入式SQL语句

  - 有些像C的预处理语句（#开头）
  - 也是通过预处理器转化为C代码

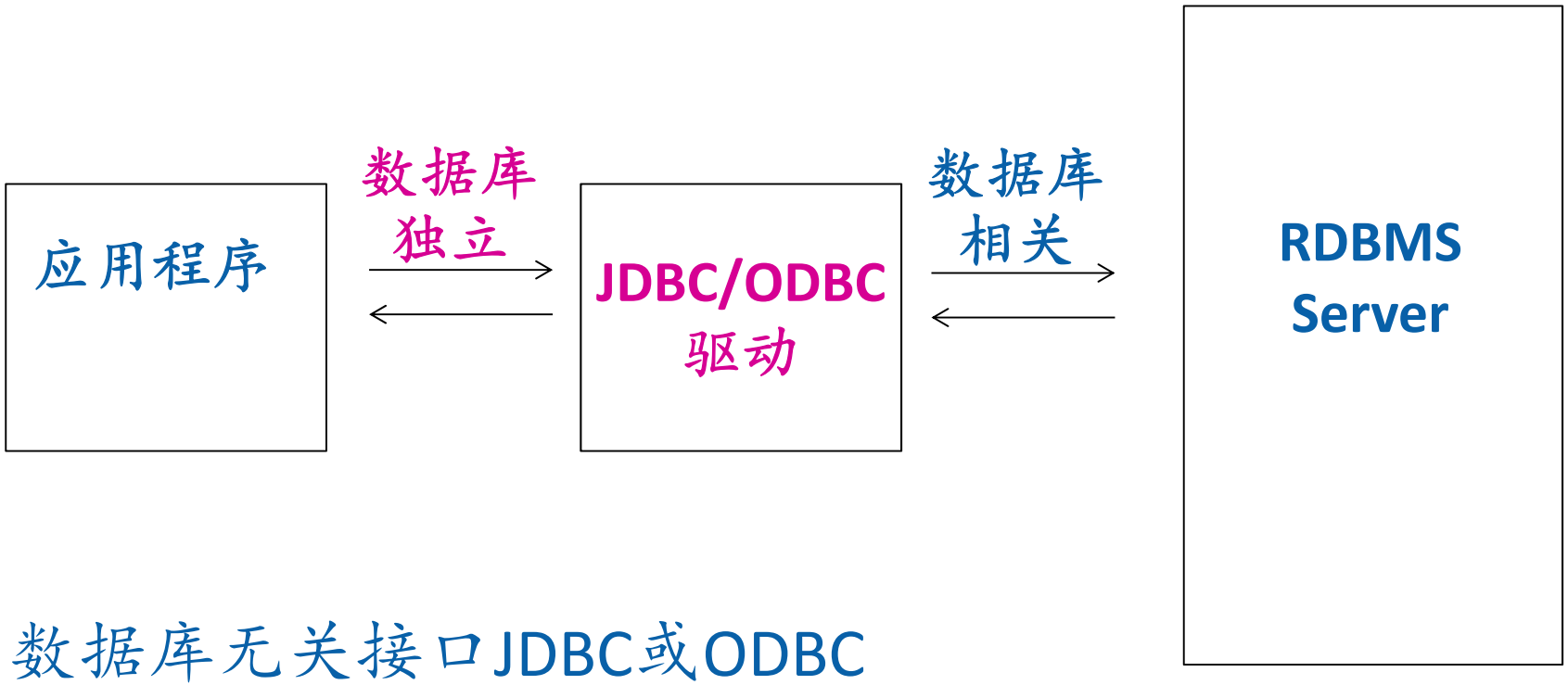
myexample5.pgc

```c
#include <stdio.h>

int main(int argc, char *argv[])
{
  EXEC SQL BEGIN DECLARE SECTION;
  int  key;
  char name[30];
  EXEC SQL END DECLARE SECTION;

  EXEC SQL CONNECT TO tpch;

  while (1) {
    printf("region key: ");
    if (scanf("%d", &key) != 1) {perror("scanf"); return 1;}
    if (key < 0) {printf("bye!\n"); break;}

    EXEC SQL select r_name into :name from region where r_regionkey=:key;

    printf("region name: %s\n", name);
  }

  EXEC SQL DISCONNECT;
  return 0;
}
```

在C程序中增加特殊的标记
EXEC SQL 嵌入式SQL语句

PostgreSQL中的嵌入式SQL

JDBC/ODBC



- 数据库无关接口JDBC或ODBC
  - JDBC（Java Database Connectivity）：基于Java
    由Sun公司发布的，现在由Oracle维护
  - ODBC（Open Database Connectivity）：
    由Microsoft主推，现在Unix-like系统上unixODBC

Outline

- 数据库编程简介
- 数据库编程
- 数据库系统内部的扩展编程
  - user defined function (也称为stored procedure)
  - SQL函数
  - C函数
  - 过程SQL函数

create function 创建用户定义的函数

```
CREATE FUNCTION
函数名([[参数模式] [参数名] 参数类型 [, ...]])
 [RETURNS 返回值类型|
  RETURNS TABLE(列名 列的类型 [, ...] )]
 {LANGUAGE 语言名 |
  AS 程序定义};
```

- 参数模式
  - IN：默认，是输入参数
  - OUT：输出
  - INOUT：即是输入又是输出
  - VARIADIC：不定个数的参数，都是相同类型
  - 注意：使用了OUT和INOUT，就不能RETURNS TABLE

- 支持的语言有：SQL（默认），C，plpgsql等

例2：纯计算的例子

```
CREATE FUNCTION sub2(x integer, y integer)
RETURNS integer AS $$
    select x-y as answer;
$$ LANGUAGE SQL;
```

说明：两个IN参数，返回整数值

使用

1. 对应位置提供参数值：

```
select sub2(20, 10);
```

2. 参数名和值：顺序不重要

```
select sub2(x:=20, y:=10);
select  sub2(y:=10, x:=20);
```

例5：返回Table

```
CREATE FUNCTION getInfo(major varchar(20))
RETURNS TABLE(sname varchar(20), gpa float)
AS $$
    select S.name, S.gpa
    from Student S
    where S.major= $1;
$$ LANGUAGE SQL;
```

TABLE 返回一个表，列的类型在括号中声明

这样，函数就可以用于from语句

小结

- 数据库编程简介
- 数据库编程
  - Libpq
  - 嵌入式SQL
  - JDBC和ODBC
- 数据库系统内部的扩展编程
  - user defined function (也称为stored procedure)
  - SQL函数
  - C函数
  - PL/pgSQL函数

Backup Slides

Backup Slides 1: 程序执行的基础知识

程序运行的过程

- 写源程序
- 编译（Compile）
- 链接（Link）
- 加载（Load）
- 运行（Run）

写源程序

- 用编辑器写源程序的文本
  - 例如，vim

```c
#include <stdio.h>

int main(int argc, char *argv[])
{
    printf("Hello world!\n");
    return 0;
}
```

编译（Compile）

- 把源程序文本➔目标代码

```
$ gcc -O2 -Wall -o HelloWorld HelloWorld.c
```

- gcc：编译器，gcc是GNU的编译器
- -O2：O代表Optimization，编译优化等级
  - 通常在调试时，不写
  - 最高O3
- -Wall：W代表Warning，all是指输出所有的警告
- -g：在生成的代码中嵌入调试信息，包括目标代码到源程序的对应关系

链接（Link）

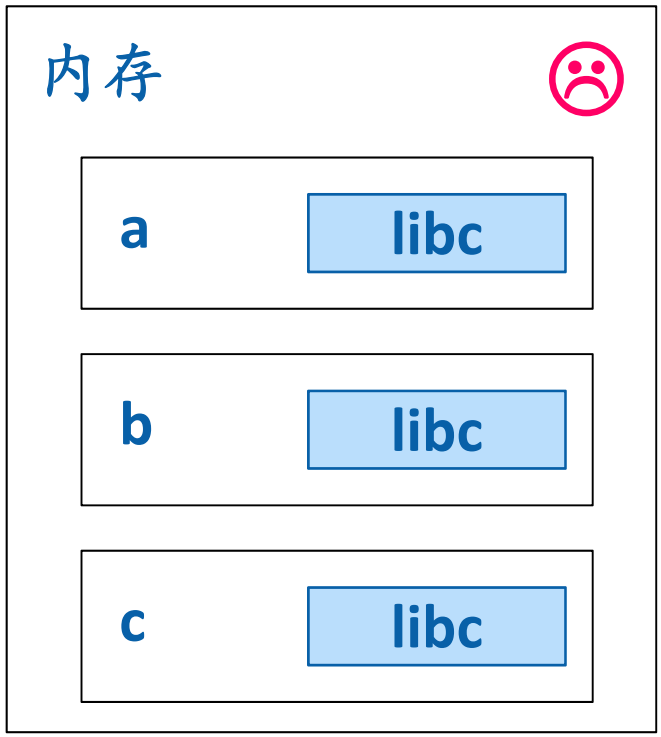
- 一个程序可能包括多个模块
  - 多个源程序的目标代码
    - HelloWorld.c ➔ HelloWorld.o
  - 库函数
    - 默认的库： libc，也就是默认-lc，在系统默认库目录下

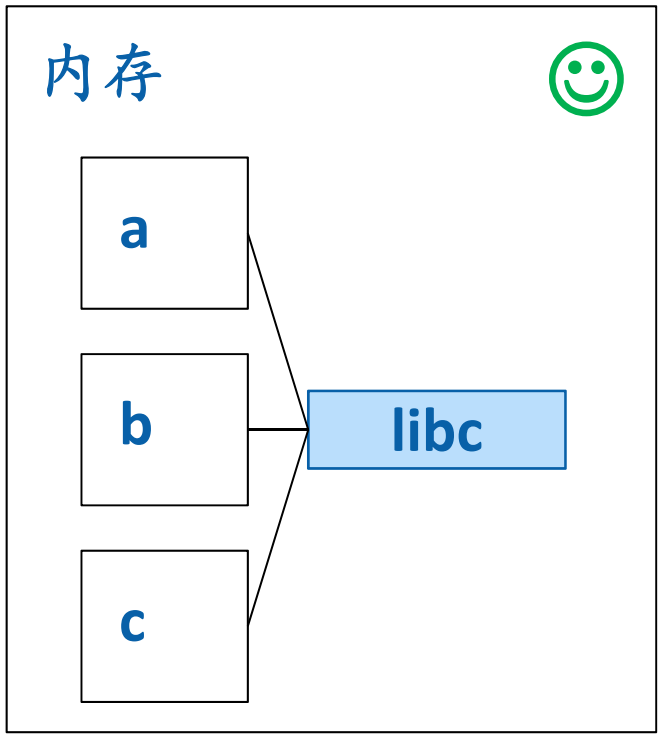
```
$ ls -l /usr/lib/x86_64-linux-gnu/libc.*
-rw-r--r-- 1 root root 5040492 Feb 26  2015 /usr/lib/x86_64-linux-gnu/libc.a
-rw-r--r-- 1 root root     298 Feb 26  2015 /usr/lib/x86_64-linux-gnu/libc.so
```

    - 这里libc.a是静态库，libc.so是动态链接库
    - 除了默认的库，一些函数调用可能需要显式链接库，例如数学库libm，这时在编译器命令行增加：-l<库名>，例如-lm
- 链接就是把多个模块的目标代码合成在一起的过程

动态链接？

- 同一个库可能被很多正在运行的程序使用
  - 例如libc
  - 为了避免冗余，希望在内存中对这个库只保留一份





加载（Load）

- 动态加载的过程
  - 从操作系统分配内存，装入代码、全局常量、全局/静态变量等
  - 找到动态共享库（Shared Library），如果之前没有那么加载，如果有那么建立虚存映射
  - 完成地址回填

运行

- 跳转到程序的开始位置

- 什么时候运行结束？
  - 从main返回
  - 实际上是调用了系统函数exit

Backup Slides 2: libpq

libpq讲解内容

- 我们讲了基础部分
  - 建立连接，关闭连接
  - 执行SQL，获得结果，打印结果
  - 使用模板
  - 参见：backup slides中关于函数的说明
- 其它内容
  - https://www.postgresql.org/docs/10/libpq.html
  - 异步请求、异步通知
  - 环境变量
  - LDAP, SSL等
- 其它语言的PostgreSQL接口、嵌入式SQL等都是基于libpq实现的

PQconnectdb：建立连接

```
PGconn *PQconnectdb(const char *conninfo);
```

- conninfo是参数设置
  - 形式： keyword=value，多个设置由空格分开
  - 主要参数
    - host=机器名　　默认为localhost
    - port=端口号
    - dbname=数据库名
    - user=用户名　　默认为当前用户
    - password=密码　　如果postgresql设置了密码
    - connection_timeout=多少秒
    - client_encoding=当前语言编码　　可以查看系统LC_CTYPE
- 返回的PGconn*是分配连接对象的指针
  - 下面的操作都要基于这个指针来进行

PQstatus：检查连接状态

```
ConnStatusType PQstatus(const PGconn *conn);
```

- 返回连接状态
  - CONNECTION_OK
  - CONNECTION_BAD

一系列获得conn具体信息的函数

函数原型	功能
char *PQhost (const PGconn *conn);	机器名
char *PQport(const PGconn *conn);	端口号
char *PQdb(const PGconn *conn);	数据库名
char *PQuser(const PGconn *conn);	用户名
char *PQpass(const PGconn *conn);	密码
int PQserverVersion(const PGconn *conn);	Postgresql的版本号
char *PQerrorMessage(const PGconn *conn);	出错信息

PQfinish：关闭连接

```
void PQfinish(PGconn *conn);
```

- 结束关闭连接

PQexec：执行SQL语句

```
PGresult *PQexec(
            PGconn *conn,
            const char *command);
```

- 执行command中的一个或多个SQL语句
- 返回最后一个语句的执行结果

PQresultStatus：检查语句返回结果

```
ExecStatusType PQresultStatus(
                    const PGresult *res);
```

- res是PQprepare等的返回指针
- PQresultStatus的结果
  - PGRES_TUPLES_OK和PGRES_SINGLE_TUPLE
    - Select语句成功
  - PGRES_COMMAND_OK
    - 无返回结果的语句成功，例如insert, delete, update
  - PGRES_FATAL_ERROR、PGRES_BAD_RESPONSE、PGRES_EMPTY_QUERY
    - 出错：执行错误、返回响应错误、发送的查询为空

进一步获得ResultStatus的错误信息

```
char *PQresultErrorMessage(
        const PGresult *res);
```

- 获得错误信息的文本描述

PQprint：打印select所有结果

```
void PQprint(FILE *fout,       // 输出文件流，例如 stdout
             const PGresult *res,
             const PQprintOpt *po);

typedef struct {
    pqbool  header;      // 是1/否0输出列名和行数
    pqbool  align;       // 是1/否0列对齐
    pqbool  standard;    // 不要用
    pqbool  html3;       // 是1/否0输出html的table形式
    pqbool  expanded;    // expand tables
    pqbool  pager;       // use pager for output if needed
    char    *fieldSep;   // 必须设为分隔符，不能为NULL
    char    *tableOpt;   // 若html table element的属性
    char    *caption;    // 若html table的标题
    char    **fieldName;// 可以替代默认的列名
} PQprintOpt;
```

PQclear：释放PGresult空间

```
void PQclear(PGresult *res);
```

- 释放PGresult中的所有资源
- 一个PGresult* res的内容可以一直使用，直至PQclear

获得res具体信息的函数

函数原型	功能
int PQntuples(const PGresult *res);	结果记录数
int PQnfields(const PGresult *res);	每条记录的列数
char *PQfname(const PGresult *res, int column_number);	返回列名，列从0开始
int PQfformat(const PGresult *res, int column_number);	列的表达形式，0文本，1二进制
Oid PQftype(const PGresult *res, int column_number);	列的类型

PQgetvalue

```
char *PQgetvalue(
        const PGresult *res,
        int row_number,
        int column_number);
```

- 行和列都是从0开始

PQprepare：准备语句模板

```
PGresult *PQprepare(
            PGconn *conn,
            const char *stName,     // 模板名
            const char *query,      // 单个含参数SQL语句
            int nParams,            // 参数个数
            const Oid paramTypes[]// 参数类型
);
```

- 准备一个查询模板，为了可以反复使用，降低开销
- Query是一个语句，不可以有多个语句
- 语句中包含未知参数：$1, $2, …,
  - 用$加数字表示参数，对应于nParams, paramTypes[ ]

PQexecPrepared：执行模板

```
PGresult *PQexecPrepared(
            PGconn *conn,
            const char *stName,        // 模板名
            int nParams,               // 参数个数
            const char *paramVal[],
            const int paramLen[],
            const int paramFormat[],
            int resultFormat);
```

参数值表达形式	文本	二进制	空值
paramFormat[k]	0	1	忽略
paramVal[k]	指向以0结尾的串	指向二进制值	NULL
paramLen[k]	忽略	二进制值的长度	忽略

- resultFormat: 0文本，1二进制

文本和二进制

- 文本
  - 所有的类型都被转化为文本
  - 当resultFormat=0时
- 二进制（整数是bigendian的，用ntohl转化）
  - 类型的Oid在postgresql源代码中
    postgresql-9.3.14/src/include/catalog/pg_type.h

```
#define   BOOLOID   16
#define   BYTEAOID   17
#define   CHAROID   18
#define   NAMEOID   19
#define   INT8OID   20
#define   INT2OID   21
#define   INT2VECTOROID   22
#define   INT4OID   23
#define   REGPROCOID   24
#define   TEXTOID   25
```

Backup Slides 3: 嵌入式SQL

嵌入式SQL

- 什么是嵌入式SQL
- 基础使用
  - 连接数据库
  - 声明变量
  - 执行语句
  - 关闭连接
- 使用游标处理多个结果记录
- 动态SQL

嵌入式SQL

- SQL标准定义
- 在C程序中增加特殊的标记
  EXEC SQL 嵌入式SQL语句

  - 有些像C的预处理语句（#开头）
  - 也是通过预处理器转化为C代码

PostgreSQL中的嵌入式SQL

myexample5.pgc

```c
#include <stdio.h>

int main(int argc, char *argv[])
{
  EXEC SQL BEGIN DECLARE SECTION;
  int  key;
  char name[30];
  EXEC SQL END DECLARE SECTION;

  EXEC SQL CONNECT TO tpch;

  while (1) {
    printf("region key: ");
    if (scanf("%d", &key) != 1) {perror("scanf"); return 1;}
    if (key < 0) {printf("bye!\n"); break;}

    EXEC SQL select r_name into :name from region where r_regionkey=:key;

    printf("region name: %s\n", name);
  }

  EXEC SQL DISCONNECT;
  return 0;
}
```

myexample5.pgc

```c
#include <stdio.h>

int main(int argc, char *argv[])
{
  EXEC SQL BEGIN DECLARE SECTION;
  int  key;
  char name[30];
  EXEC SQL END DECLARE SECTION;

  EXEC SQL CONNECT TO tpch;

  while (1) {
    printf("region key: ");
    if (scanf("%d", &key) != 1) {perror("scanf"); return 1;}
    if (key < 0) {printf("bye!\n"); break;}

    EXEC SQL select r_name into :name from region where r_regionkey=:key;

    printf("region name: %s\n", name);
  }

  EXEC SQL DISCONNECT;
  return 0;
}
```

需要在SQL中用到的变量定义

声明变量

```
EXEC SQL BEGIN DECLARE SECTION;
// C变量定义
int a;
char name[30];
EXEC SQL END DECLARE SECTION;
```

- 然后在语句中可以使用这些变量
  - 格式为:变量名
  - 例如:a, :name等

细节：数据类型对应关系

PostgreSQL data type	Host variable type
smallint	short
integer	int
bigint	long long
decimal	decimal (见pgtypes_numeric.h)
numeric	numeric (见pgtypes_numeric.h)
real	float
double precision	double
oid	unsigned int
character(n), varchar(n), text	char[n+1], VARCHAR[n+1] (见下页)
timestamp	timestamp(见pgtypes_timestamp.h)
date	date (见pgtypes_timestamp.h)
boolean	bool

VARCHAR

- 对于

```
VARCHAR str[100];
```

- ecpq会产生类似如下的代码

```
struct varchar_1  {
    int len;
    char arr[100];
} str;
```

myexample5.pgc

```c
#include <stdio.h>

int main(int argc, char *argv[])
{
  EXEC SQL BEGIN DECLARE SECTION;
  int  key;
  char name[30];
  EXEC SQL END DECLARE SECTION;

  EXEC SQL CONNECT TO tpch;

  while (1) {
    printf("region key: ");
    if (scanf("%d", &key) != 1) {perror("scanf"); return 1;}
    if (key < 0) {printf("bye!\n"); break;}

    EXEC SQL select r_name into :name from region where r_regionkey=:key;

    printf("region name: %s\n", name);
  }

  EXEC SQL DISCONNECT;
  return 0;
}
```

连接数据库

建立连接

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

- 例子：

```
EXEC SQL CONNECT TO mydb@sql.mydomain.com AS myconnection
USER john;

EXEC SQL BEGIN DECLARE SECTION;
const char *target = "mydb@sql.mydomain.com";
const char *user = "john";
const char *passwd = "secret";
EXEC SQL END DECLARE SECTION;
EXEC SQL CONNECT TO :target USER :user USING :passwd;
```

myexample5.pgc

```c
#include <stdio.h>

int main(int argc, char *argv[])
{
  EXEC SQL BEGIN DECLARE SECTION;
  int  key;
  char name[30];
  EXEC SQL END DECLARE SECTION;

  EXEC SQL CONNECT TO tpch;

  while (1) {
    printf("region key: ");
    if (scanf("%d", &key) != 1) {perror("scanf"); return 1;}
    if (key < 0) {printf("bye!\n"); break;}

    EXEC SQL select r_name into :name from region where r_regionkey=:key;

    printf("region name: %s\n", name);
  }

  EXEC SQL DISCONNECT;
  return 0;
}
```

执行 select语句，
这里的key是由scanf输入的，
select只产生一条记录

执行单个SQL语句

EXEC SQL 单个SQL语句;

- 通常为不返回结果的语句
  - 例如： insert, delete, update, create/drop等
- 或者为返回单一记录的Select语句
  - 例如：

```
select r_name into :name
from  region
where r_regionkey = 1;
```

myexample5.pgc

```
1  #include <stdio.h>
2
3  int main(int argc, char *argv[])
4  {
5    EXEC SQL BEGIN DECLARE SECTION;
6    int  key;
7    char name[30];
8    EXEC SQL END DECLARE SECTION;
9
10   EXEC SQL CONNECT TO tpch;
11
12   while (1) {
13     printf("region key: ");
14     if (scanf("%d", &key) != 1) {perror("scanf"); return 1;}
15     if (key < 0) {printf("bye!\n"); break;}
16
17     EXEC SQL select r_name into :name from region where r_regionkey=:key;
18
19     printf("region name: %s\n", name);
20   }
21
22   EXEC SQL DISCONNECT;
23   return 0;
24 }
```

关闭数据库连接

关闭连接

```
EXEC SQL DISCONNECT [connection];
```

- 如果不写connection，那么默认为当前的

编译运行myexample5.pgc

- 编译

```
$ ecpg myexample5.pgc
$ gcc -g -Wall -I/usr/include/postgresql -o myexample5 myexample5.c -lecpg
```

- 运行

```
dbms@ubuntu:~/db-programming/embedded$ ./myexample5
region key: 3
region name: EUROPE
region key: 2
region name: ASIA
region key: 4
region name: MIDDLE EAST
region key: 0
region name: AFRICA
region key: 1
region name: AMERICA
region key: -1
bye!
```

嵌入式SQL

- 什么是嵌入式SQL
- 基础使用
  - 连接数据库
  - 声明变量
  - 执行语句
  - 关闭连接
- 使用游标处理多个结果记录
- 动态SQL

myexample6.pgc
用游标输出多条结果

```
1  #include <stdio.h>
2
3  int main(int argc, char *argv[])
4  {
5    EXEC SQL BEGIN DECLARE SECTION;
6    char name[30];
7    EXEC SQL END DECLARE SECTION;
8
9    EXEC SQL CONNECT TO tpch;
10
11   EXEC SQL DECLARE mycur CURSOR FOR select r_name from region;
12   EXEC SQL OPEN mycur;
13
14   EXEC SQL WHENEVER NOT FOUND DO BREAK;
15   while(1) {
16     EXEC SQL FETCH mycur INTO :name;
17     printf("%s\n", name);
18   }
19
20   EXEC SQL CLOSE mycur;
21   EXEC SQL COMMIT;
22
23   EXEC SQL DISCONNECT;
24   return 0;
25 }
```

游标（Cursor）

定义和打开游标

```
EXEC SQL DECLARE mycur CURSOR FOR
     select r_name from region;
EXEC SQL OPEN mycur;
```

读一行数据

```
EXEC SQL FETCH mycur INTO :name;
```

关闭游标

```
EXEC SQL CLOSE mycur;
EXEC SQL COMMIT;
```

myexample6.pgc

```
1  #include <stdio.h>
2
3  int main(int argc, char *argv[])
4  {
5    EXEC SQL BEGIN DECLARE SECTION;
6    char name[30];
7    EXEC SQL END DECLARE SECTION;
8
9    EXEC SQL CONNECT TO tpch;
10
11   EXEC SQL DECLARE mycur CURSOR FOR select r_name from region;
12   EXEC SQL OPEN mycur;
13
14   EXEC SQL WHENEVER NOT FOUND DO BREAK;
15   while(1) {
16     EXEC SQL FETCH mycur INTO :name;
17     printf("%s\n", name);
18   }
19
20   EXEC SQL CLOSE mycur;
21   EXEC SQL COMMIT;
22
23   EXEC SQL DISCONNECT;
24   return 0;
25 }
```

定义和打开游标

myexample6.pgc

```
1  #include <stdio.h>
2
3  int main(int argc, char *argv[])
4  {
5    EXEC SQL BEGIN DECLARE SECTION;
6    char name[30];
7    EXEC SQL END DECLARE SECTION;
8
9    EXEC SQL CONNECT TO tpch;
10
11   EXEC SQL DECLARE mycur CURSOR FOR select r_name from region;
12   EXEC SQL OPEN mycur;
13
14   EXEC SQL WHENEVER NOT FOUND DO BREAK;
15   while(1) {
16     EXEC SQL FETCH mycur INTO :name;
17     printf("%s\n", name);
18   }
19
20   EXEC SQL CLOSE mycur;
21   EXEC SQL COMMIT;
22
23   EXEC SQL DISCONNECT;
24   return 0;
25 }
```

关闭游标

myexample6.pgc

```c
#include <stdio.h>

int main(int argc, char *argv[])
{
  EXEC SQL BEGIN DECLARE SECTION;
  char name[30];
  EXEC SQL END DECLARE SECTION;

  EXEC SQL CONNECT TO tpch;

  EXEC SQL DECLARE mycur CURSOR FOR select r_name from region;
  EXEC SQL OPEN mycur;

  EXEC SQL WHENEVER NOT FOUND DO BREAK;
  while(1) {
    EXEC SQL FETCH mycur INTO :name;
    printf("%s\n", name);
  }

  EXEC SQL CLOSE mycur;
  EXEC SQL COMMIT;

  EXEC SQL DISCONNECT;
  return 0;
}
```

循环读取游标直至完毕NOT FOUND

游标的FETCH语句

FETCH [方向] cursor_name

- 方向
  - NEXT（默认）, PRIOR
  - FIRST, LAST
  - 等等

WHENEVER语句

EXEC SQL WHENEVER <条件> <动作>;

- 条件
  - SQLERROR
  - SQLWARNING
  - NOT FOUND
- 动作
  - GOTO label　　产生一条C的 goto语句
  - SQLPRINT　　在stderr上打印错误信息
  - STOP　　exit(1)
  - DO BREAK　　break;
  - CALL func(args)　　调用C函数func(args)

错误代码

- 嵌入式SQL定义了一个全局变量sqlca
  - sqlca是一个struct,
  - 包含多个与错误相关的属性
- sqlca.sqlcode
  - 错误代码为数字
  - deprecated
- sqlca.sqlstate
  - 错误代码为长度为5的字符串
  - “00000”代表正确

编译运行myexample6.pgc

- 运行

```
dbms@ubuntu:~/db-programming/embedded$ ./myexample6
AFRICA
AMERICA
ASIA
EUROPE
MIDDLE EAST
```

动态SQL

- SQL语句不是在编译时就确定了
- 而是在运行中产生的
  - 例如，由用户输入

执行一个没有返回结果的语句

```
EXEC SQL BEGIN DECLARE SECTION;
char stmt [256];
EXEC SQL END DECLARE SECTION;
```

动态产生stmt

```
EXEC SQL EXECUTE IMMEDIATE :stmt;
```

获取结果

动态产生了stmt

```
EXEC SQL PREPARE myst FROM :stmt;
EXEC SQL DECLARE mycur CURSOR FOR myst;
EXEC SQL OPEN mycur;

EXEC SQL WHENEVER NOT FOUND DO BREAK;
while (1) {
  EXEC SQL FETCH mycur INTO :v1,:v2;
  ......
}

EXEC SQL CLOSE mycur;
EXEC SQL COMMIT;
```

用Prepare准备，然后打开cursor

需要事先知道返回多少列，每列的类型是什么☹

Backup Slides 4:
数据库系统内部的扩展编程

Outline

- 数据库系统内部的扩展编程
  - user defined function (也称为stored procedure)
  - SQL函数
  - C函数
  - 过程SQL函数

例1：纯计算的例子

```
CREATE FUNCTION one() RETURNS integer AS $$
    select 1 as result;
$$ LANGUAGE SQL;
```

无参数，返回整数值

$$ … $$ 把SQL语句括起来

调用：select one();

例2：纯计算的例子

```
CREATE FUNCTION sub2(x integer, y integer)
RETURNS integer AS $$
    select x-y as answer;
$$ LANGUAGE SQL;
```

两个IN参数，返回整数值

调用纯计算的函数

1. 对应位置提供参数值：

```
select sub2(20, 10);
```

2. 参数名和值：顺序不重要

```
select sub2(x:=20, y:=10);
select  sub2(y:=10, x:=20);
```

例3：修改数据库

```
CREATE FUNCTION
incSalary(id integer, percent real)
RETURNS numeric(15,2) AS $$

    update faculty
    set salary = salary*(1.0+percent)
    where fid= id;

    select salary from faculty where fid=id;

$$ LANGUAGE SQL;
```

多个SQL语句，返回最后一个语句的结果

例4：使用OUT参数

```
CREATE FUNCTION sum_prod (x int, y int, OUT sum
int, OUT prod int)
AS $$ SELECT x + y, x * y $$
LANGUAGE SQL;

SELECT * FROM sum_prod(7,8);
 sum | prod
-----+-----
  15 |   56
(1 row)
```

没有RETURNS，两个OUT参数

例5：返回Table

```
CREATE FUNCTION getInfo(major varchar(20))
RETURNS TABLE(sname varchar(20), gpa float)
AS $$
    select S.name, S.gpa
    from Student S
    where S.major= $1;
$$ LANGUAGE SQL;
```

TABLE 返回一个表，列的类型在括号中声明

这样，函数就可以用于from语句

create function 创建用户定义的函数

```
CREATE FUNCTION
函数名([[参数模式] [参数名] 参数类型 [, ...]])
 [RETURNS 返回值类型|
  RETURNS TABLE(列名 列的类型 [, ...] )]
 {LANGUAGE 语言名 |
  AS 程序定义};
```

- 上述是主要部分，更多的选项参见：
  https://www.postgresql.org/docs/10/sql-createfunction.html

- create function的具体语法在不同系统上有差异，在使用时需要参照相应的使用手册

create function 创建用户定义的函数

```
CREATE FUNCTION
函数名([[参数模式] [参数名] 参数类型 [, ...]])
 [RETURNS 返回值类型|
  RETURNS TABLE(列名 列的类型 [, ...] )]
 {LANGUAGE 语言名 |
  AS 程序定义};
```

- 参数模式
  - IN：默认，是输入参数
  - OUT：输出
  - INOUT：即是输入又是输出
  - VARIADIC：不定个数的参数，都是相同类型
- 注意：使用了OUT和INOUT，就不能RETURNS TABLE

create function 创建用户定义的函数

```
CREATE FUNCTION
函数名([[参数模式] [参数名] 参数类型 [, ...]])
 [RETURNS 返回值类型|
  RETURNS TABLE(列名 列的类型 [, ...] )]
 {LANGUAGE 语言名 |
  AS 程序定义};
```

- 支持的语言有：SQL（默认），C，plpgsql等

Outline

- 数据库系统内部的扩展编程
  - user defined function (也称为stored procedure)
  - SQL函数
  - C函数
  - 过程SQL函数

用户定义函数的语言

- 上述Function都是SQL的
- C语言
  - 大量内部提供的函数是这种方式实现的
  - 动态加载目标代码模块，然后执行其中的函数
  - 我们举例简介一下
- 过程语言：除了SQL和C之外的语言
  - PostgreSQL本身只支持SQL和C，不知道其它语言
  - 每种其它的语言是由加载的一个动态模块提供的支持
    - 语法解析、运行相应语言的程序
  - 我们主要介绍一下PL/pgSQL：它和Oracle的语法一致

myexample7.c
user defined function in C

```c
#include <string.h>
#include "postgres.h"
#include "fmgr.h"

#ifdef PG_MODULE_MAGIC
PG_MODULE_MAGIC;
#endif

PG_FUNCTION_INFO_V1(sub_xy);
Datum sub_xy(PG_FUNCTION_ARGS)
{
    int32 x = PG_GETARG_INT32(0);
    int32 y = PG_GETARG_INT32(1);

    PG_RETURN_INT32(x - y);
}

PG_FUNCTION_INFO_V1(concat_text);
Datum concat_text(PG_FUNCTION_ARGS)
{
    text *arg1 = PG_GETARG_TEXT_P(0);
    text *arg2 = PG_GETARG_TEXT_P(1);
    int32 new_text_size = VARSIZE(arg1) + VARSIZE(arg2) - VARHDRSZ;
    text *new_text = (text *) palloc(new_text_size);

    SET_VARSIZE(new_text, new_text_size);
    memcpy(VARDATA(new_text), VARDATA(arg1), VARSIZE(arg1) - VARHDRSZ);
    memcpy(VARDATA(new_text) + (VARSIZE(arg1) - VARHDRSZ),
           VARDATA(arg2), VARSIZE(arg2) - VARHDRSZ);
    PG_RETURN_TEXT_P(new_text);
}
```

```c
#include <string.h>
#include "postgres.h"
#include "fmgr.h"

#ifdef PG_MODULE_MAGIC
PG_MODULE_MAGIC;
#endif

PG_FUNCTION_INFO_V1(sub_xy);
Datum sub_xy(PG_FUNCTION_ARGS)
{
    int32 x = PG_GETARG_INT32(0);
    int32 y = PG_GETARG_INT32(1);

    PG_RETURN_INT32(x - y);
}

PG_FUNCTION_INFO_V1(concat_text);
Datum concat_text(PG_FUNCTION_ARGS)
{
    text *arg1 = PG_GETARG_TEXT_P(0);
    text *arg2 = PG_GETARG_TEXT_P(1);
    int32 new_text_size = VARSIZE(arg1) + VARSIZE(arg2) - VARHDRSZ;
    text *new_text = (text *) palloc(new_text_size);

    SET_VARSIZE(new_text, new_text_size);
    memcpy(VARDATA(new_text), VARDATA(arg1), VARSIZE(arg1) - VARHDRSZ);
    memcpy(VARDATA(new_text) + (VARSIZE(arg1) - VARHDRSZ),
           VARDATA(arg2), VARSIZE(arg2) - VARHDRSZ);
    PG_RETURN_TEXT_P(new_text);
}
```

包含头文件

```c
#include <string.h>
#include "postgres.h"
#include "fmgr.h"

#ifdef PG_MODULE_MAGIC
PG_MODULE_MAGIC;
#endif

PG_FUNCTION_INFO_V1(sub_xy);
Datum sub_xy(PG_FUNCTION_ARGS)
{
    int32 x = PG_GETARG_INT32(0);
    int32 y = PG_GETARG_INT32(1);

    PG_RETURN_INT32(x - y);
}

PG_FUNCTION_INFO_V1(concat_text);
Datum concat_text(PG_FUNCTION_ARGS)
{
    text *arg1 = PG_GETARG_TEXT_P(0);
    text *arg2 = PG_GETARG_TEXT_P(1);
    int32 new_text_size = VARSIZE(arg1) + VARSIZE(arg2) - VARHDRSZ;
    text *new_text = (text *) palloc(new_text_size);

    SET_VARSIZE(new_text, new_text_size);
    memcpy(VARDATA(new_text), VARDATA(arg1), VARSIZE(arg1) - VARHDRSZ);
    memcpy(VARDATA(new_text) + (VARSIZE(arg1) - VARHDRSZ),
           VARDATA(arg2), VARSIZE(arg2) - VARHDRSZ);
    PG_RETURN_TEXT_P(new_text);
}
```

PostgreSQL UDF模块必须包括

UDF函数头定义

注意：除函数名外其它是不变的

```c
#include <string.h>
#include "postgres.h"
#include "fmgr.h"

#ifdef PG_MODULE_MAGIC
PG_MODULE_MAGIC;
#endif

PG_FUNCTION_INFO_V1(sub_xy);
Datum sub_xy(PG_FUNCTION_ARGS)
{
    int32 x = PG_GETARG_INT32(0);
    int32 y = PG_GETARG_INT32(1);

    PG_RETURN_INT32(x - y);
}

PG_FUNCTION_INFO_V1(concat_text);
Datum concat_text(PG_FUNCTION_ARGS)
{
    text *arg1 = PG_GETARG_TEXT_P(0);
    text *arg2 = PG_GETARG_TEXT_P(1);
    int32 new_text_size = VARSIZE(arg1) + VARSIZE(arg2) - VARHDRSZ;
    text *new_text = (text *) palloc(new_text_size);

    SET_VARSIZE(new_text, new_text_size);
    memcpy(VARDATA(new_text), VARDATA(arg1), VARSIZE(arg1) - VARHDRSZ);
    memcpy(VARDATA(new_text) + (VARSIZE(arg1) - VARHDRSZ),
           VARDATA(arg2), VARSIZE(arg2) - VARHDRSZ);
    PG_RETURN_TEXT_P(new_text);
}
```

```c
#include <string.h>
#include "postgres.h"
#include "fmgr.h"

#ifdef PG_MODULE_MAGIC
PG_MODULE_MAGIC;
#endif

PG_FUNCTION_INFO_V1(sub_xy);
Datum sub_xy(PG_FUNCTION_ARGS)
{
    int32 x = PG_GETARG_INT32(0);
    int32 y = PG_GETARG_INT32(1);

    PG_RETURN_INT32(x - y);
}

PG_FUNCTION_INFO_V1(concat_text);
Datum concat_text(PG_FUNCTION_ARGS)
{
    text *arg1 = PG_GETARG_TEXT_P(0);
    text *arg2 = PG_GETARG_TEXT_P(1);
    int32 new_text_size = VARSIZE(arg1) + VARSIZE(arg2) - VARHDRSZ;
    text *new_text = (text *) palloc(new_text_size);

    SET_VARSIZE(new_text, new_text_size);
    memcpy(VARDATA(new_text), VARDATA(arg1), VARSIZE(arg1) - VARHDRSZ);
    memcpy(VARDATA(new_text) + (VARSIZE(arg1) - VARHDRSZ),
           VARDATA(arg2), VARSIZE(arg2) - VARHDRSZ);
    PG_RETURN_TEXT_P(new_text);
}
```

获取输入参数用
PG_GETARG_XXX

```c
#include <string.h>
#include "postgres.h"
#include "fmgr.h"

#ifdef PG_MODULE_MAGIC
PG_MODULE_MAGIC;
#endif

PG_FUNCTION_INFO_V1(sub_xy);
Datum sub_xy(PG_FUNCTION_ARGS)
{
    int32 x = PG_GETARG_INT32(0);
    int32 y = PG_GETARG_INT32(1);

    PG_RETURN_INT32(x - y);
}

PG_FUNCTION_INFO_V1(concat_text);
Datum concat_text(PG_FUNCTION_ARGS)
{
    text *arg1 = PG_GETARG_TEXT_P(0);
    text *arg2 = PG_GETARG_TEXT_P(1);
    int32 new_text_size = VARSIZE(arg1) + VARSIZE(arg2) - VARHDRSZ;
    text *new_text = (text *) palloc(new_text_size);

    SET_VARSIZE(new_text, new_text_size);
    memcpy(VARDATA(new_text), VARDATA(arg1), VARSIZE(arg1) - VARHDRSZ);
    memcpy(VARDATA(new_text) + (VARSIZE(arg1) - VARHDRSZ),
           VARDATA(arg2), VARSIZE(arg2) - VARHDRSZ);
    PG_RETURN_TEXT_P(new_text);
}
```

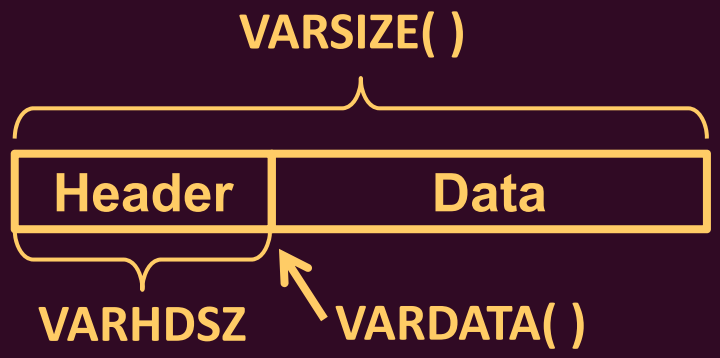
返回值用
PG_RETURN_XXX

```c
#include <string.h>
#include "postgres.h"
#include "fmgr.h"

#ifdef PG_MODULE_MAGIC
PG_MODULE_MAGIC;
#endif

PG_FUNCTION_INFO_V1(sub_xy);
Datum sub_xy(PG_FUNCTION_ARGS)
{
    int32 x = PG_GETARG_INT32(0);
    int32 y = PG_GETARG_INT32(1);

    PG_RETURN_INT32(x - y);
}

PG_FUNCTION_INFO_V1(concat_text);
Datum concat_text(PG_FUNCTION_ARGS)
{
    text *arg1 = PG_GETARG_TEXT_P(0);
    text *arg2 = PG_GETARG_TEXT_P(1);
    int32 new_text_size = VARSIZE(arg1) + VARSIZE(arg2) - VARHDRSZ;
    text *new_text = (text *) palloc(new_text_size);

    SET_VARSIZE(new_text, new_text_size);
    memcpy(VARDATA(new_text), VARDATA(arg1), VARSIZE(arg1) - VARHDRSZ);
    memcpy(VARDATA(new_text) + (VARSIZE(arg1) - VARHDRSZ),
           VARDATA(arg2), VARSIZE(arg2) - VARHDRSZ);
    PG_RETURN_TEXT_P(new_text);
}
```

获取输入的int32，
然后相减

```c
#include <string.h>
#include "postgres.h"
#include "fmgr.h"

#ifdef PG_MODULE_MAGIC
PG_MODULE_MAGIC;
#endif

PG_FUNCTION_INFO_V1(sub_xy);
Datum sub_xy(PG_FUNCTION_ARGS)
{
    int32 x = PG_GETARG_INT32(0);
    int32 y = PG_GETARG_INT32(1);

    PG_RETURN_INT32(x - y);
}

PG_FUNCTION_INFO_V1(concat_text);
Datum concat_text(PG_FUNCTION_ARGS)
{
    text *arg1 = PG_GETARG_TEXT_P(0);
    text *arg2 = PG_GETARG_TEXT_P(1);
    int32 new_text_size = VARSIZE(arg1) + VARSIZE(arg2) - VARHDRSZ;
    text *new_text = (text *) palloc(new_text_size);

    SET_VARSIZE(new_text, new_text_size);
    memcpy(VARDATA(new_text), VARDATA(arg1), VARSIZE(arg1) - VARHDRSZ);
    memcpy(VARDATA(new_text) + (VARSIZE(arg1) - VARHDRSZ),
           VARDATA(arg2), VARSIZE(arg2) - VARHDRSZ);
    PG_RETURN_TEXT_P(new_text);
}
```

变长类型的结构：



用palloc代替malloc

编译myexample7.c

- 要生成动态库

```
$ gcc -fpic -shared -I/usr/include/postgresql/9.3/server
-o myexample7.so ./myexmaple7.c
```

- 注意
  - -fpic –shared 用于生成动态库
  - -I/usr/include/postgresql/9.3/server 头文件位置
  - -o myexample7.so 注意文件后缀为 .so
  - 可以有优化等级等其它选项

创建C语言的UDF和运行

```
$ psql -t tpch
psql (9.3.14)
Type "help" for help.

tpch=# create function sub_xy(integer, integer) returns integer
tpch-# as '/home/dbms/db-programming/udf/myexample7', 'sub_xy'
tpch-# language C strict;
CREATE FUNCTION
tpch=# select sub_xy(100, 10);
     90

tpch=# create function concat_text(text, text) returns text
tpch-# as '/home/dbms/db-programming/udf/myexample7', 'concat_text'
tpch-# language C strict;
CREATE FUNCTION
tpch=# select concat_text('abc', '123');
 abc123
```

- 注意：其中的as后面是目标代码（省略.so后缀）和C函数名

FUNCTION的编程语言

- SQL
- C语言
  - 更多内容参见 https://www.postgresql.org/docs/10/xfunc-c.html
- 过程语言：除了SQL和C之外的语言
  - PostgreSQL本身只支持SQL和C，不知道其它语言
  - 每种其它的语言是由加载的一个动态模块提供的支持
    - 语法解析、运行相应语言的程序
  - 我们主要介绍一下PL/pgSQL：它和Oracle的语法一致

PL/pgSQL与普通的SQL有什么不同？

create function 创建PL/pgSQL函数

```
CREATE FUNCTION
函数名([[参数模式] [参数名] 参数类型 [, ...]])
 [RETURNS 返回值类型|
  RETURNS TABLE(列名 列的类型 [, ...] )]
  AS $$
   [DECLARE
       declarations]
    BEGIN
        statements
    END;
  $$
  LANGUAGE plpgsql;
```

变量声明

程序语句

- 语句由分号结束
- --注释以两个减号开头
- 标识符不区分大小写

举例：返回所有小写的region名

```
tpch=# create function all_region()
tpch-# returns table(lower_name char(25)) as $$
tpch$# declare
tpch$#   a_row record;
tpch$# begin
tpch$#   for a_row in select * from region loop
tpch$#      lower_name := lower(a_row.r_name);
tpch$#      return next;
tpch$#   end loop;
tpch$#   return;
tpch$# end;
tpch$# $$ language plpgsql;
CREATE FUNCTION
tpch=# select all_region();
 africa
 america
 asia
 europe
 middle east

tpch=#
```

举例：求圆的面积

```
create function circle_area(radius float)
returns float as $$
declare
  pi float := 3.1415926;
begin
  return pi * radius * radius;
end;
$$ language plpgsql;
```

变量声明

变量名　类型　{:= 初始值};

- 类型
  - 任何SQL数据类型
  - RECORD类型
  - 等
- 例如

```
user_id integer;
url varchar :='http://www.carch.ac.cn/~chensm/';
a_row RECORD;
```

注：　使用发现=与:=都可以

语句

- 注释

```
--This is a comment line
```

- 任何SQL语句
- 赋值语句 :=

```
x := y * 1.2;
```

- 控制语句
  - IF-THEN-ELSE
  - CASE
  - FOR, WHILE, LOOP, CONTINUE, EXIT
  - RETURN

IF-THEN-ELSE

```
IF boolean-expression THEN
    statements
[ ELSIF boolean-expression THEN
    statements ]
...
[ ELSE
    statements ]
END IF;
```

CASE：比照C语言中的switch

```
CASE search-expression
  WHEN expr [, expr ...] THEN
       statements
  WHEN expr [, expr ...] THEN
       statements
  [ELSE
       statements ]
END CASE;
```

FOR循环：整数

```
FOR var IN start .. end BY step LOOP
    statements
END LOOP;
```

循环变量var从start开始，每次循环增加step，直至大于end为止，step默认为1

例如：

```
FOR i IN 1 .. 10 LOOP
    -- i 取值1,2,3,4,5,6,7,8,9,10
END LOOP;
```

FOR循环：查询结果

```
FOR recordvar IN query LOOP
    statements
END LOOP;
```

循环变量是RECORD类型，每次循环为下一个结果行

例如：

```
FOR a_row IN select * from Student LOOP
    ...
END LOOP;
```

循环的其它语句

```
WHILE boolean-expression LOOP
    statements
END LOOP;

LOOP
    statements
END LOOP;

EXIT;       比照C语言的break
CONTINUE;   比照C语言的continue
```

注：EXIT和CONTINUE可以跳出多重循环，具体见
https://www.postgresql.org/docs/9.3/static/plpgsql-control-structures.html

RETURN

函数返回单个值

```
RETURN expression;
```

函数返回TABLE

```
RETURN NEXT;
RETURN QUERY query;
RETURN;
```

RETURN NEXT和RETURN QUERY产生返回结果

RETURN无参，真正结束返回