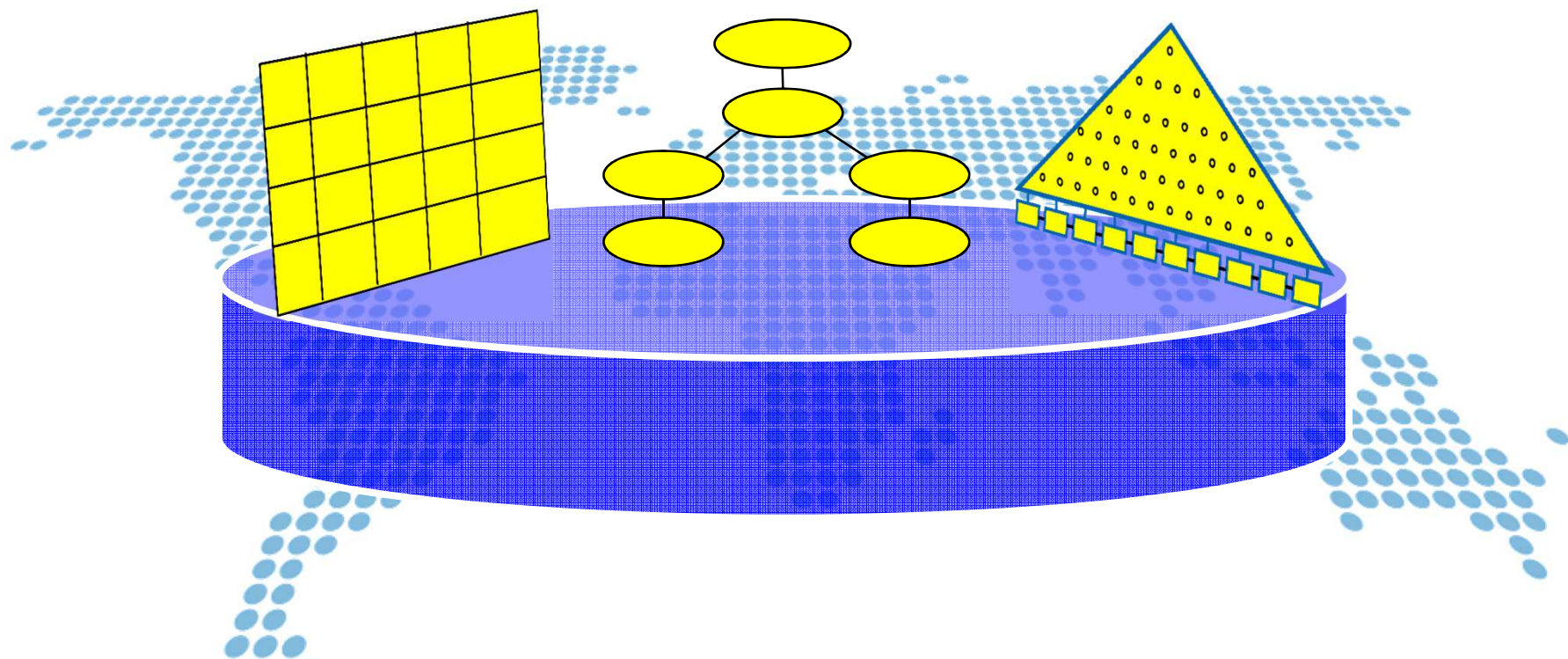


数据库系统

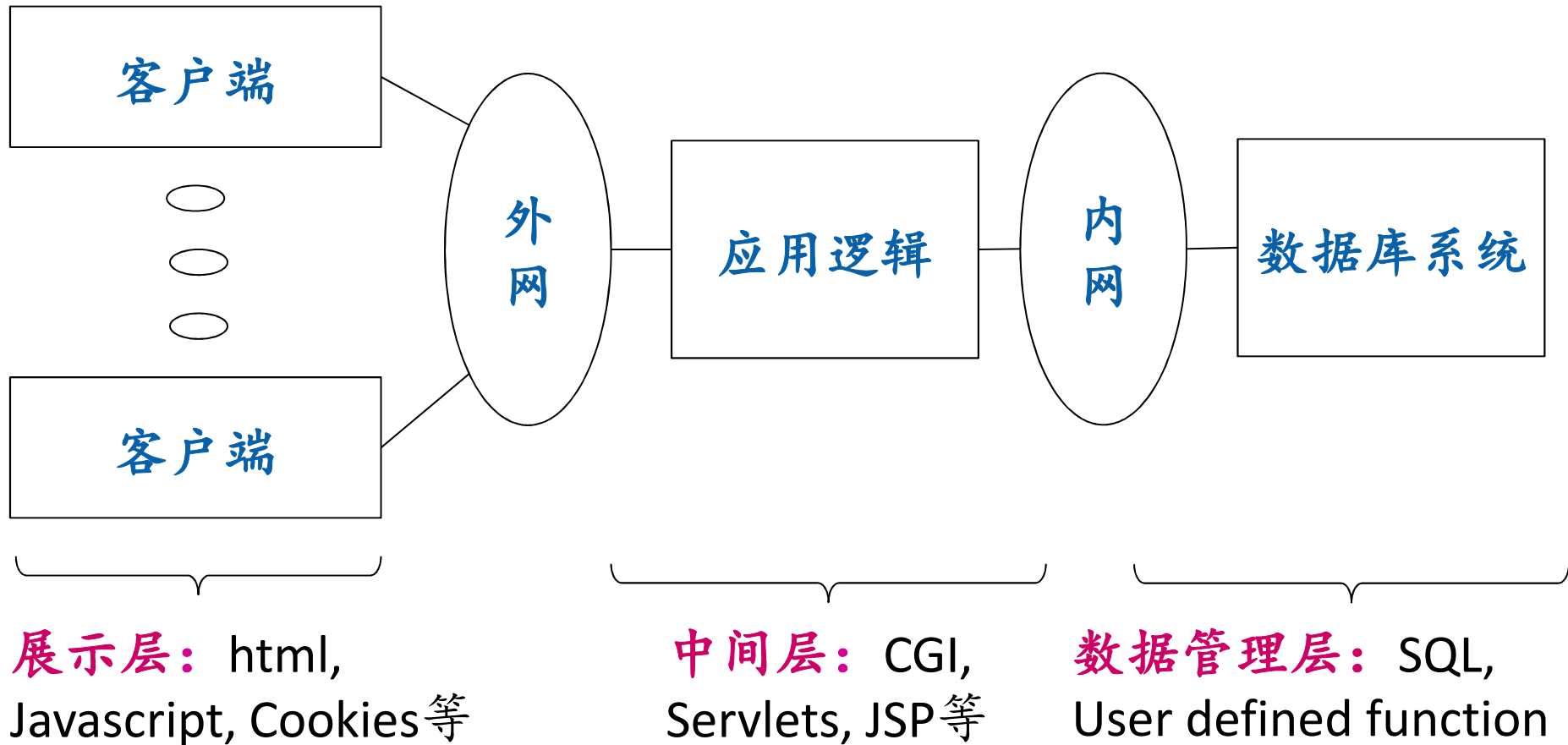
# 数据库程序设计

陈世敏

(中科院计算所)



# Web应用的三层结构



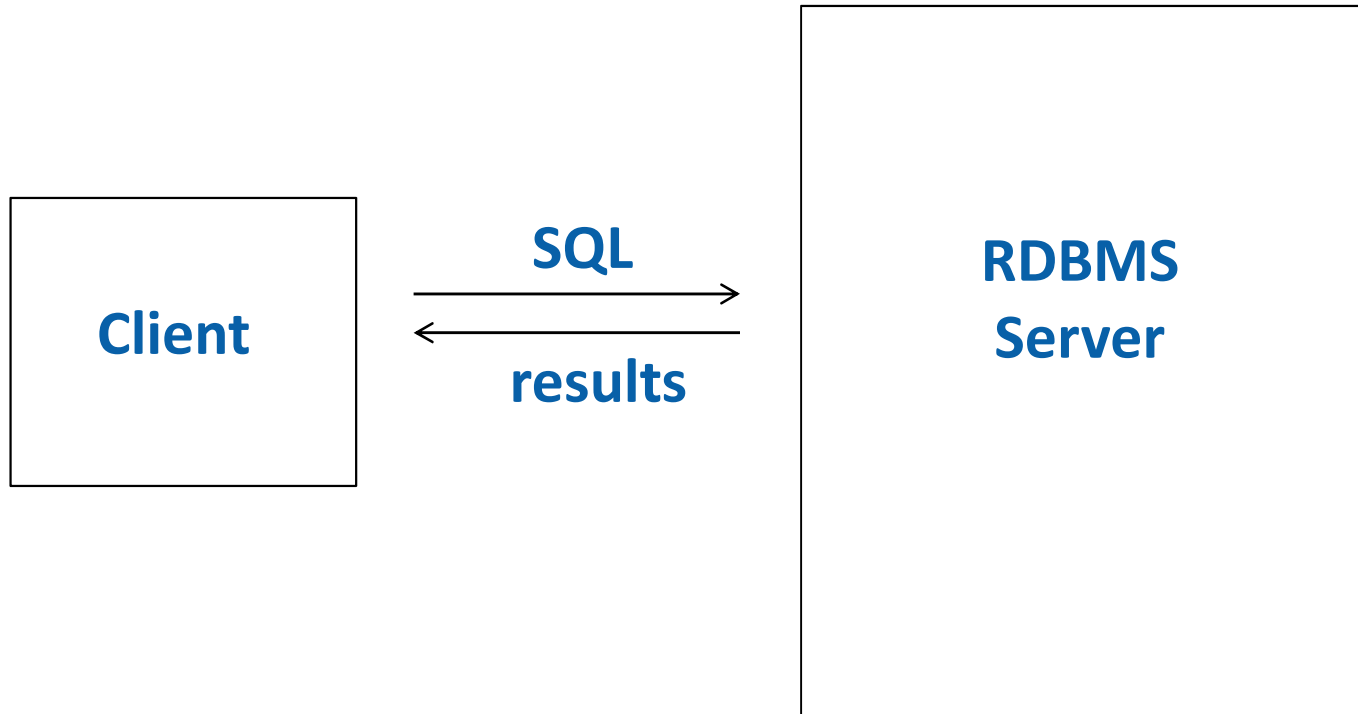
# 实验2安排

- 内容：开发一个基于数据库的Web应用
  - 熟悉Web应用的三层结构
  - 实践需求分析、概念设计、逻辑设计、模式细化等
  - 学习数据库编程
- 本堂课：10月13日
  - 数据库编程
- 下堂课：10月20日
  - Web编程介绍
  - 实验2具体要求
- 11月24日：验收实验2

# Outline

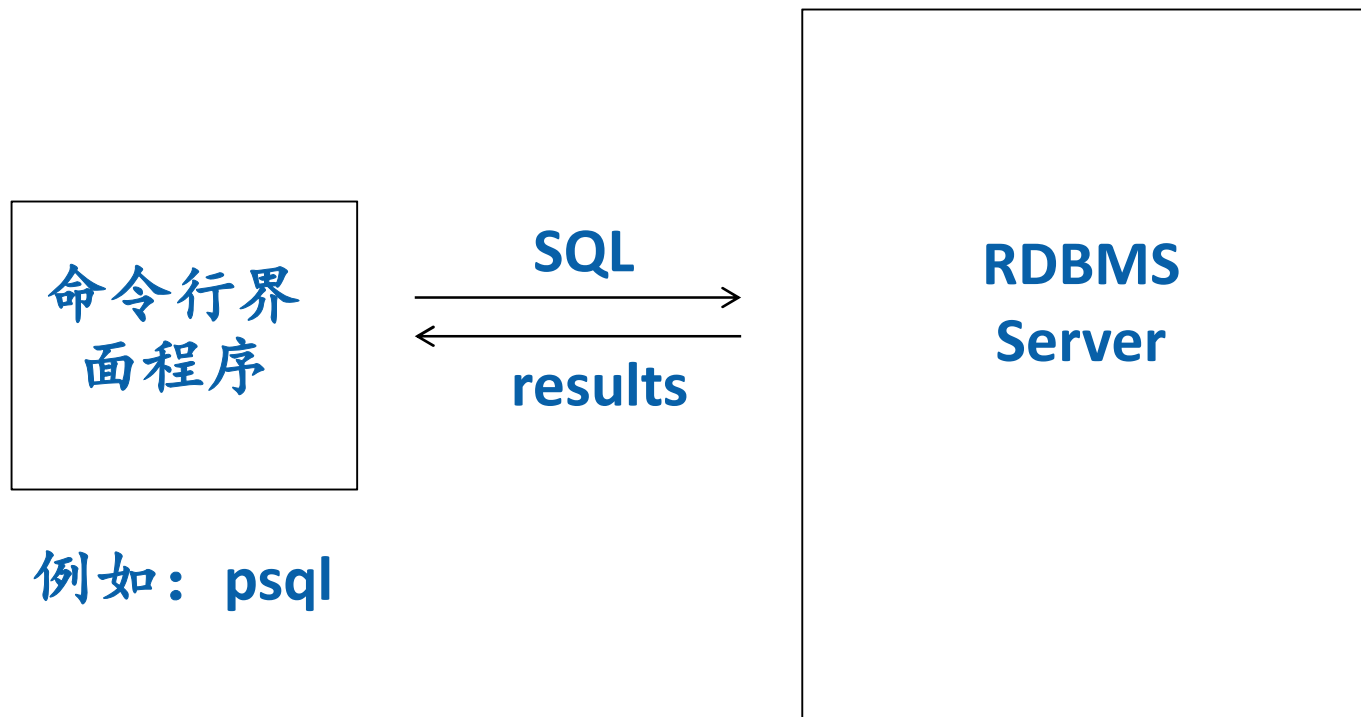
- 数据库编程简介
- 程序执行的基础知识
- 数据库客户端编程
- 数据库系统内部的扩展编程

# 数据库系统为典型的Client / Server



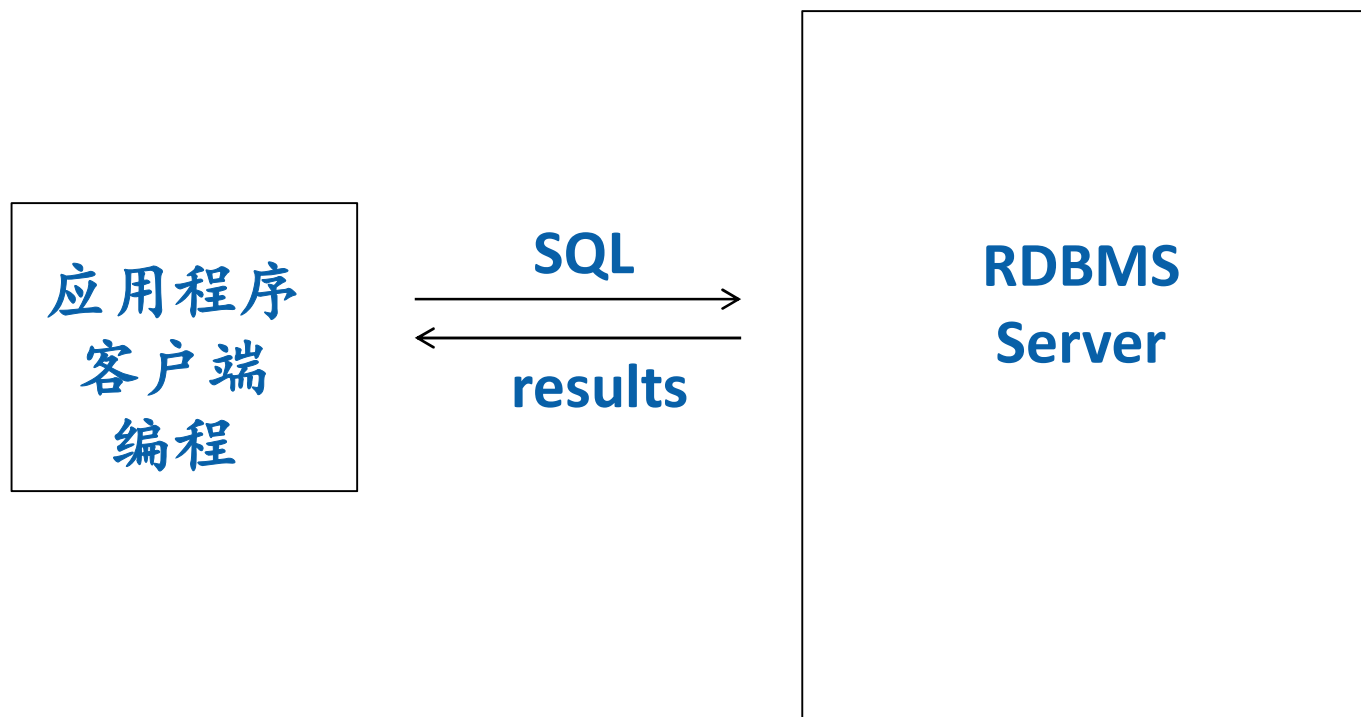
# 使用命令行界面

- 在实验1中，我们已经接触了这种用法



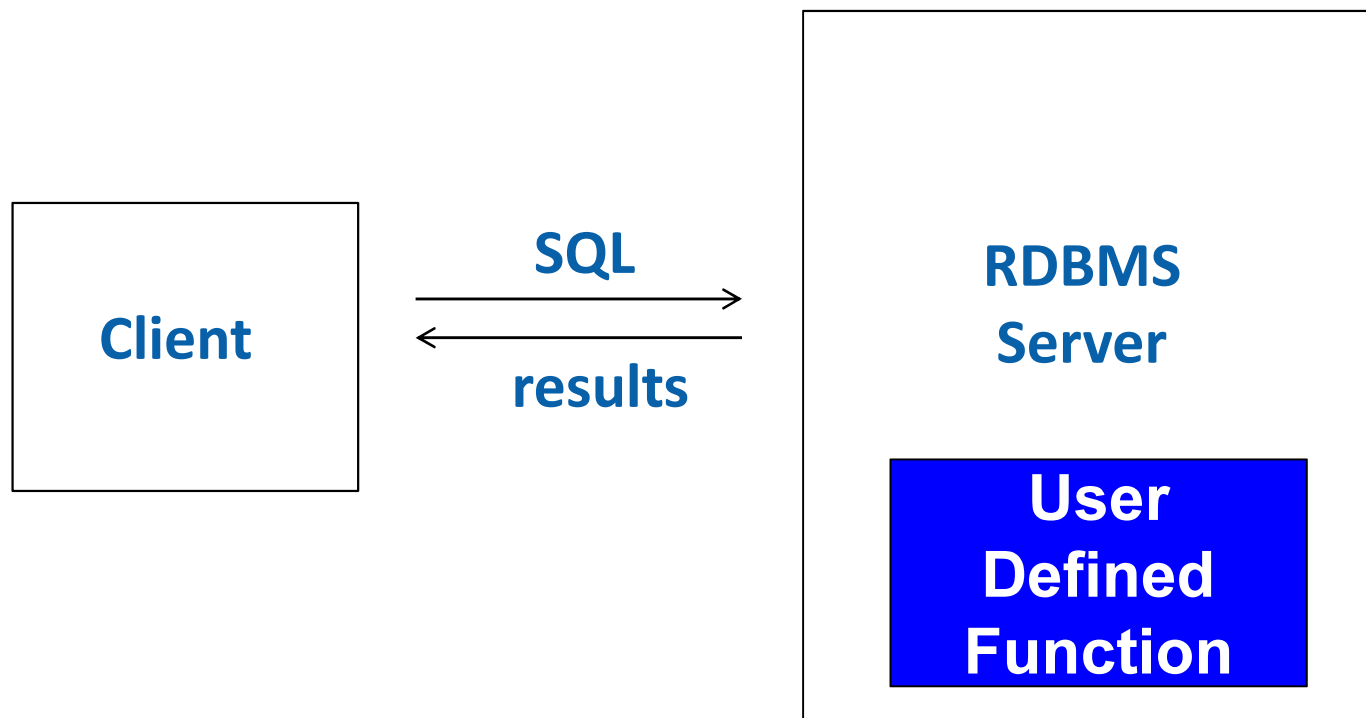
通常完成大部分数据库的管理工作，如create table, create index, create view等

# 在程序中访问数据库系统



- 数据库系统提供的运行库：libpq
- 嵌入式SQL
- JDBC或ODBC

# 数据库系统扩展编程





# PostgreSQL开发环境

- 修改/etc/apt/sources.list

```
deb http://cn.archive.ubuntu.com/ubuntu/ trusty main restricted universe multiverse
deb-src http://cn.archive.ubuntu.com/ubuntu/ trusty main restricted universe multiverse

deb http://cn.archive.ubuntu.com/ubuntu/ trusty-updates main restricted universe multiverse
deb-src http://cn.archive.ubuntu.com/ubuntu/ trusty-updates main restricted universe multiverse

deb http://cn.archive.ubuntu.com/ubuntu/ trusty-backports main restricted universe multiverse
deb-src http://cn.archive.ubuntu.com/ubuntu/ trusty-backports main restricted universe multiverse

deb http://security.ubuntu.com/ubuntu trusty-security main restricted
deb-src http://security.ubuntu.com/ubuntu trusty-security main restricted
deb http://security.ubuntu.com/ubuntu trusty-security universe
deb-src http://security.ubuntu.com/ubuntu trusty-security universe
deb http://security.ubuntu.com/ubuntu trusty-security multiverse
deb-src http://security.ubuntu.com/ubuntu trusty-security multiverse

deb http://extras.ubuntu.com/ubuntu/ trusty main
deb-src http://extras.ubuntu.com/ubuntu trusty main
```

# PostgreSQL开发环境

- 更新安装包列表

```
$ sudo apt-get update
```

- 根据更新的列表检查当前安装的包

```
$ sudo apt-get -f install
```

# PostgreSQL开发环境

- 安装libpq-dev开发相关文件

```
$ sudo apt-get install libpq-dev
```

- 安装嵌入式SQL开发相关文件

```
$ sudo apt-get install libecpg-dev
```

- 安装数据库系统服务端开发相关文件

```
$ sudo apt-get install postgresql-server-dev-9.3
```

- 安装了什么，可以用下述命令看

```
$ dpkg-query -L libpq-dev
```

□ 头文件/usr/include/postgresql/\*.h, 库文件/usr/lib/下

# Outline

- 数据库编程简介
- 程序执行的基础知识
- 数据库客户端编程
- 数据库系统内部的扩展编程

# 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

```
#include <stdio.h>

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

# 写源程序

- 用编辑器写源程序的文本

□ 例如, vim

```
#include <stdio.h>

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

头文件: `stdio.h`, 尖括号是指到系统默认目录 `/usr/include` 下面去找这个文件

# 写源程序

- 用编辑器写源程序的文本

□ 例如, vim

```
#include <stdio.h>

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

Main调用了c标准输出printf



# 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

# 编译器gcc

- 实际上gcc不仅完成了编译，它包括下述步骤
  - C的预处理：#include, #define等
  - C程序编译成为.o文件
  - 链接
- 不同语言有不同的编译器
  - C++： g++
- 当然有些语言是解释执行的，例如许多脚本语言
  - Perl, python, bash shell

# 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

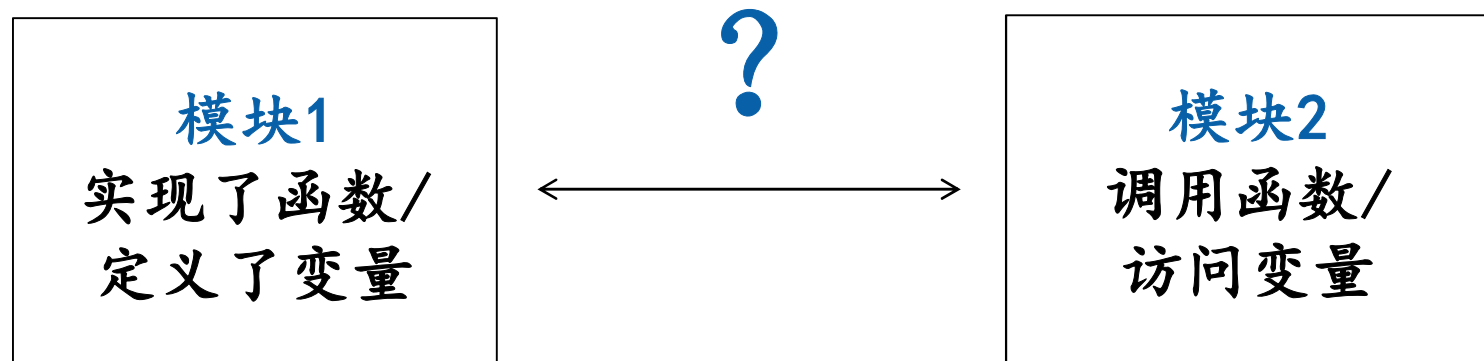
```
$ ls -l /usr/lib/x86_64-linux-gnu/libc.*  
-rw-r--r-- 1 root root 5040492 Feb 26 2015 /usr/lib/x86_64-linux-gnu/libc.a  
-rw-r--r-- 1 root root 298 Feb 26 2015 /usr/lib/x86_64-linux-gnu/libc.so
```

- 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

# 多个模块：外部函数/变量

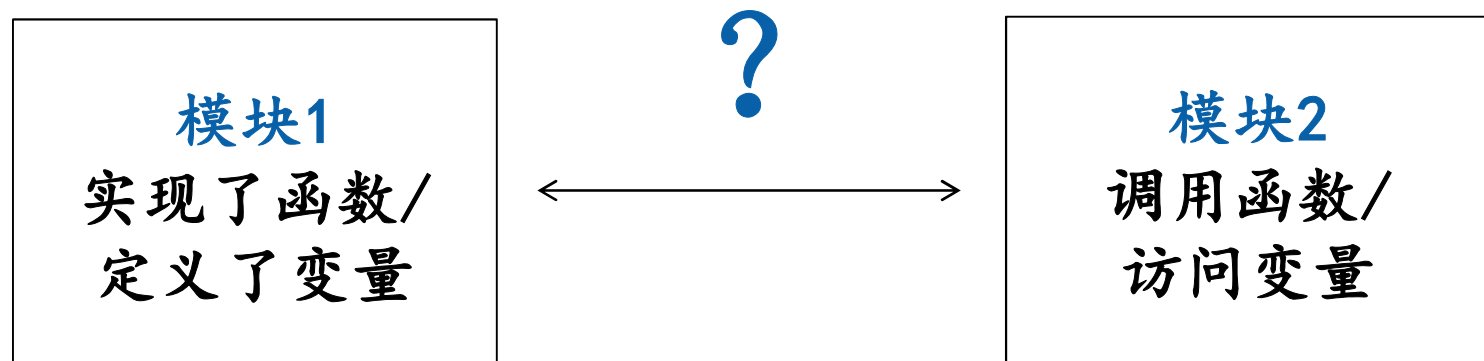


例如：libc中定义了  
printf()

例如：HelloWorld.c  
调用了printf()

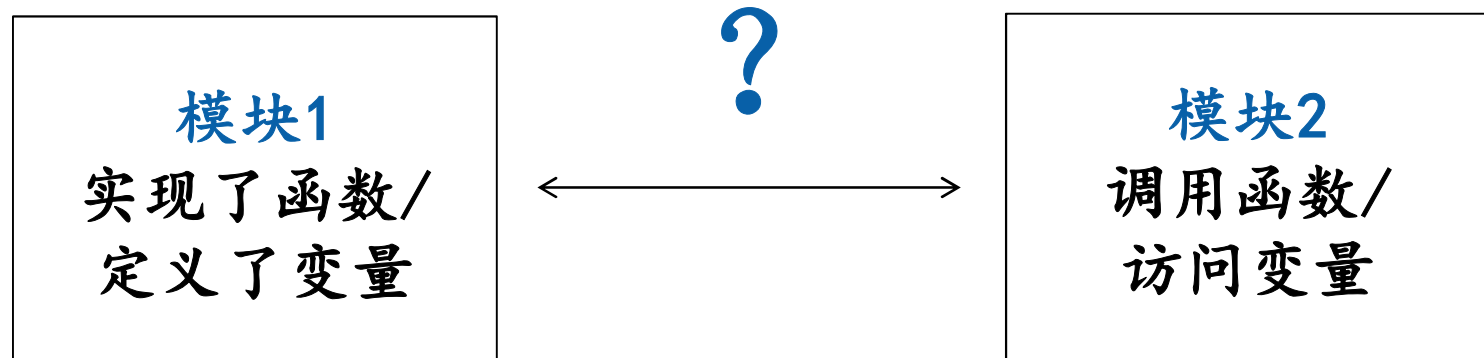
- 编译：找到正确的类型定义，生成正确的代码
- 链接：找到正确的代码入口地址，变量地址

# 多个模块：外部函数/变量



- 编译：找到正确的类型定义，生成正确的代码
  - extern 变量声明，类型定义，函数原型定义
  - #include <stdio.h>
  - 生成的代码中，对应的外部函数和变量的地址是未知的，会进行标记

# 多个模块：外部函数/变量



- **链接：**找到正确的代码入口地址，变量地址

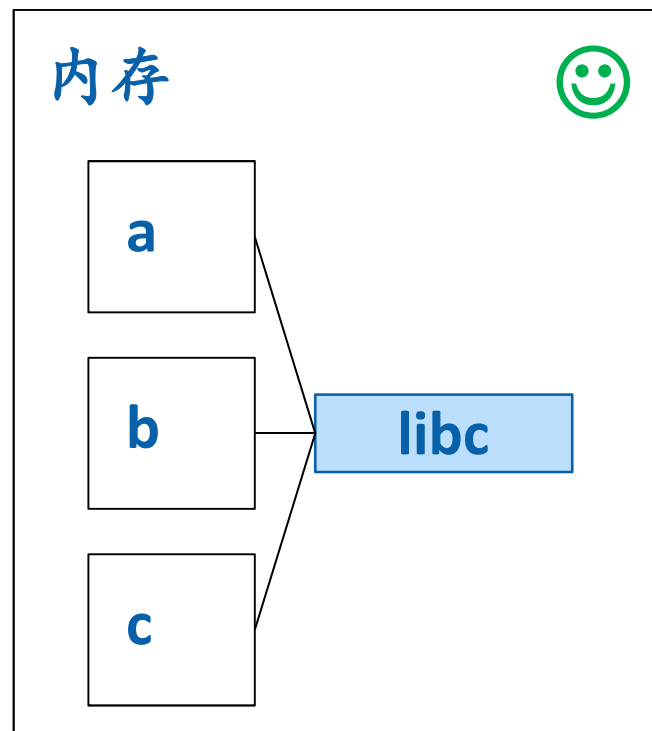
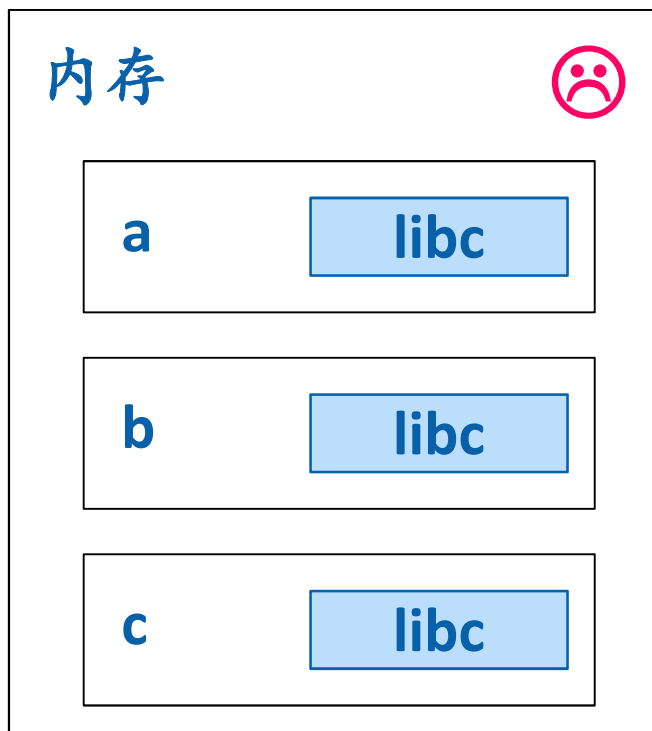
- 静态链接

- 链接器可以看到函数的实现代码和变量的位置
    - 找到标记，回填地址

- 动态链接：在加载中完成

# 为什么要动态链接？

- 同一个库可能为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



# 虚拟内存 (Virtual Memory)

- 大家在计算机原理、操作系统课会仔细学习
- 物理地址 (Physical Address)
  - 内存条的地址，取决于内存插槽、地址线等
  - CPU进行内存访问，必须指定物理地址
- 虚拟地址 (Virtual Address)
  - 程序在运行中，访问的内存地址
    - 例如，指针就是虚拟地址
  - 虚拟地址从0开始，到0x7fff ffff ffff ffff，是用户空间 (x86-64 linux)
  - 不同的程序在运行中的虚拟内存是互相隔离的，各有各各自的虚存空间
- 虚拟地址➡物理地址：由硬件+操作系统完成
  - 操作系统对每个程序有一个虚存页表，记录虚存➡物理空间的映射
  - 硬件对每次内存访问都进行地址转换



# 我们看一下HelloWorld执行的虚存

```
$ cat /proc/4925/maps
```

```
00400000-00401000 r-xp 00000000 08:01 962596 /home/dbms/db-programming/hello/HelloWorld
00600000-00601000 r--p 00000000 08:01 962596 /home/dbms/db-programming/hello/HelloWorld
00601000-00602000 rw-p 00001000 08:01 962596 /home/dbms/db-programming/hello/HelloWorld
7f77d22b7000-7f77d2472000 r-xp 00000000 08:01 1053454 /lib/x86_64-linux-gnu/libc-2.19.so
7f77d2472000-7f77d2671000 ---p 001bb000 08:01 1053454 /lib/x86_64-linux-gnu/libc-2.19.so
7f77d2671000-7f77d2675000 r--p 001ba000 08:01 1053454 /lib/x86_64-linux-gnu/libc-2.19.so
7f77d2675000-7f77d2677000 rw-p 001be000 08:01 1053454 /lib/x86_64-linux-gnu/libc-2.19.so
7f77d2677000-7f77d267c000 rw-p 00000000 00:00 0
7f77d267c000-7f77d269f000 r-xp 00000000 08:01 1053430 /lib/x86_64-linux-gnu/ld-2.19.so
7f77d2885000-7f77d2888000 rw-p 00000000 00:00 0
7f77d289a000-7f77d289e000 rw-p 00000000 00:00 0
7f77d289e000-7f77d289f000 r--p 00022000 08:01 1053430 /lib/x86_64-linux-gnu/ld-2.19.so
7f77d289f000-7f77d28a0000 rw-p 00023000 08:01 1053430 /lib/x86_64-linux-gnu/ld-2.19.so
7f77d28a0000-7f77d28a1000 rw-p 00000000 00:00 0
7fffd1ee3000-7fffd1f04000 rw-p 00000000 00:00 0 [stack]
7fffd1fa5000-7fffd1fa7000 r--p 00000000 00:00 0 [vvar]
7fffd1fa7000-7fffd1fa9000 r-xp 00000000 00:00 0 [vdso]
ffffffff600000-ffffffff601000 r-xp 00000000 00:00 0 [vsyscall]
```

# 我们看一下HelloWorld执行的虚存

```
$ cat /proc/4925/maps
```

```
00400000-00401000 r-xp 00000000 08:01 962596 /home/dbms/db-programming/hello/HelloWorld
00600000-00601000 r--p 00000000 08:01 962596 /home/dbms/db-programming/hello/HelloWorld
00601000-00602000 rw-p 00001000 08:01 962596 /home/dbms/db-programming/hello/HelloWorld
7f77d22b7000-7f77d2472000 r-xp 00000000 08:01 1053454 /lib/x86_64-linux-gnu/libc-2.19.so
7f77d2472000-7f77d2671000 ---p 001bb000 08:01 1053454 /lib/x86_64-linux-gnu/libc-2.19.so
7f77d2671000-7f77d2675000 r--p 001ba000 08:01 1053454 /lib/x86_64-linux-gnu/libc-2.19.so
7f77d2675000-7f77d2677000 rw-p 001be000 08:01 1053454 /lib/x86_64-linux-gnu/libc-2.19.so
7f77d2677000-7f77d267c000 rw-p 00000000 00:00 0
7f77d267c000-7f77d269f000 r-xp 00000000 08:01 1053430 /lib/x86_64-linux-gnu/ld-2.19.so
7f77d2885000-7f77d2888000 rw-p 00000000 00:00 0
7f77d289a000-7f77d289e000 rw-p 00000000 00:00 0
7f77d289e000-7f77d289f000 r--p 00022000 08:01 1053430 /lib/x86_64-linux-gnu/ld-2.19.so
7f77d289f000-7f77d28a0000 rw-p 00023000 08:01 1053430 /lib/x86_64-linux-gnu/ld-2.19.so
7f77d28a0000-7f77d28a1000 rw-p 00000000 00:00 0
7fffd1ee3000-7fffd1f04000 rw-p 00000000 00:00 0 [stack]
7fffd1fa5000-7fffd1fa7000 r--p 00000000 00:00 0 [vvar]
7fffd1fa7000-7fffd1fa9000 r-xp 00000000 00:00 0 [vdso]
ffffffff600000-ffffffff601000 r-xp 00000000 00:00 0 [vsyscall]
```

HelloWorld的代码、全局常量、全局/静态变量



# 我们看一下HelloWorld执行的虚存

```
$ cat /proc/4925/maps
```

```
00400000-00401000 r-xp 00000000 08:01 962596 /home/dbms/db-programming/hello/HelloWorld
00600000-00601000 r--p 00000000 08:01 962596 /home/dbms/db-programming/hello/HelloWorld
00601000-00602000 rw-p 00001000 08:01 962596 /home/dbms/db-programming/hello/HelloWorld
7f77d22b7000-7f77d2472000 r-xp 00000000 08:01 1053454 /lib/x86_64-linux-gnu/libc-2.19.so
7f77d2472000-7f77d2671000 ---p 001bb000 08:01 1053454 /lib/x86_64-linux-gnu/libc-2.19.so
7f77d2671000-7f77d2675000 r--p 001ba000 08:01 1053454 /lib/x86_64-linux-gnu/libc-2.19.so
7f77d2675000-7f77d2677000 rw-p 001be000 08:01 1053454 /lib/x86_64-linux-gnu/libc-2.19.so
7f77d2677000-7f77d267c000 rw-p 00000000 00:00 0
7f77d267c000-7f77d269f000 r-xp 00000000 08:01 1053430 /lib/x86_64-linux-gnu/ld-2.19.so
7f77d2885000-7f77d2888000 rw-p 00000000 00:00 0
7f77d289a000-7f77d289e000 rw-p 00000000 00:00 0
7f77d289e000-7f77d289f000 r--p 00022000 08:01 1053430 /lib/x86_64-linux-gnu/ld-2.19.so
7f77d289f000-7f77d28a0000 rw-p 00023000 08:01 1053430 /lib/x86_64-linux-gnu/ld-2.19.so
7f77d28a0000-7f77d28a1000 rw-p 00000000 00:00 0
7fffd1ee3000-7fffd1f04000 rw-p 00000000 00:00 0 [stack]
7fffd1fa5000-7fffd1fa7000 r--p 00000000 00:00 0 [vvar]
7fffd1fa7000-7fffd1fa9000 r-xp 00000000 00:00 0 [vdso]
ffffffff600000-ffffffff601000 r-xp 00000000 00:00 0 [vsyscall]
```

libc的代码（共享）、全局常量（共享）、全局/静态变量（这部分是每个程序特有的）

# 我们看一下HelloWorld执行的虚存

```
$ cat /proc/4925/maps
```

```
00400000-00401000 r-xp 00000000 08:01 962596 /home/dbms/db-programming/hello/HelloWorld
00600000-00601000 r--p 00000000 08:01 962596 /home/dbms/db-programming/hello/HelloWorld
00601000-00602000 rw-p 00001000 08:01 962596 /home/dbms/db-programming/hello/HelloWorld
7f77d22b7000-7f77d2472000 r-xp 00000000 08:01 1053454 /lib/x86_64-linux-gnu/libc-2.19.so
7f77d2472000-7f77d2671000 ---p 001bb000 08:01 1053454 /lib/x86_64-linux-gnu/libc-2.19.so
7f77d2671000-7f77d2675000 r--p 001ba000 08:01 1053454 /lib/x86_64-linux-gnu/libc-2.19.so
7f77d2675000-7f77d2677000 rw-p 001be000 08:01 1053454 /lib/x86_64-linux-gnu/libc-2.19.so
7f77d2677000-7f77d267c000 rw-p 00000000 00:00 0
7f77d267c000-7f77d269f000 r-xp 00000000 08:01 1053430 /lib/x86_64-linux-gnu/ld-2.19.so
7f77d2885000-7f77d2888000 rw-p 00000000 00:00 0
7f77d289a000-7f77d289e000 rw-p 00000000 00:00 0
7f77d289e000-7f77d289f000 r--p 00022000 08:01 1053430 /lib/x86_64-linux-gnu/ld-2.19.so
7f77d289f000-7f77d28a0000 rw-p 00023000 08:01 1053430 /lib/x86_64-linux-gnu/ld-2.19.so
7f77d28a0000-7f77d28a1000 rw-p 00000000 00:00 0
7fffd1ee3000-7fffd1f04000 rw-p 00000000 00:00 0 [stack]
7fffd1fa5000-7fffd1fa7000 r--p 00000000 00:00 0 [vvar]
7fffd1fa7000-7fffd1fa9000 r-xp 00000000 00:00 0 [vdso]
ffffffff600000-ffffffff601000 r-xp 00000000 00:00 0 [vsyscall]
```

ld是程序加载器



# 我们看一下HelloWorld执行的虚存

```
$ cat /proc/4925/maps
```

```
00400000-00401000 r-xp 00000000 08:01 962596 /home/dbms/db-programming/hello/HelloWorld
00600000-00601000 r--p 00000000 08:01 962596 /home/dbms/db-programming/hello/HelloWorld
00601000-00602000 rw-p 00001000 08:01 962596 /home/dbms/db-programming/hello/HelloWorld
7f77d22b7000-7f77d2472000 r-xp 00000000 08:01 1053454 /lib/x86_64-linux-gnu/libc-2.19.so
7f77d2472000-7f77d2671000 ---p 001bb000 08:01 1053454 /lib/x86_64-linux-gnu/libc-2.19.so
7f77d2671000-7f77d2675000 r--p 001ba000 08:01 1053454 /lib/x86_64-linux-gnu/libc-2.19.so
7f77d2675000-7f77d2677000 rw-p 001be000 08:01 1053454 /lib/x86_64-linux-gnu/libc-2.19.so
7f77d2677000-7f77d267c000 rw-p 00000000 00:00 0
7f77d267c000-7f77d269f000 r-xp 00000000 08:01 1053430 /lib/x86_64-linux-gnu/ld-2.19.so
7f77d2885000-7f77d2888000 rw-p 00000000 00:00 0
7f77d289a000-7f77d289e000 rw-p 00000000 00:00 0
7f77d289e000-7f77d289f000 r--p 00022000 08:01 1053430 /lib/x86_64-linux-gnu/ld-2.19.so
7f77d289f000-7f77d28a0000 rw-p 00023000 08:01 1053430 /lib/x86_64-linux-gnu/ld-2.19.so
7f77d28a0000-7f77d28a1000 rw-p 00000000 00:00 0
7fffd1ee3000-7fffd1f04000 rw-p 00000000 00:00 0 [stack]
7fffd1fa5000-7fffd1fa7000 r--p 00000000 00:00 0 [vvar]
7fffd1fa7000-7fffd1fa9000 r-xp 00000000 00:00 0 [vdso]
ffffffff600000-ffffffff601000 r-xp 00000000 00:00 0 [vsyscall]
```

程序栈：每个函数的动态变量，返回地址等

# 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

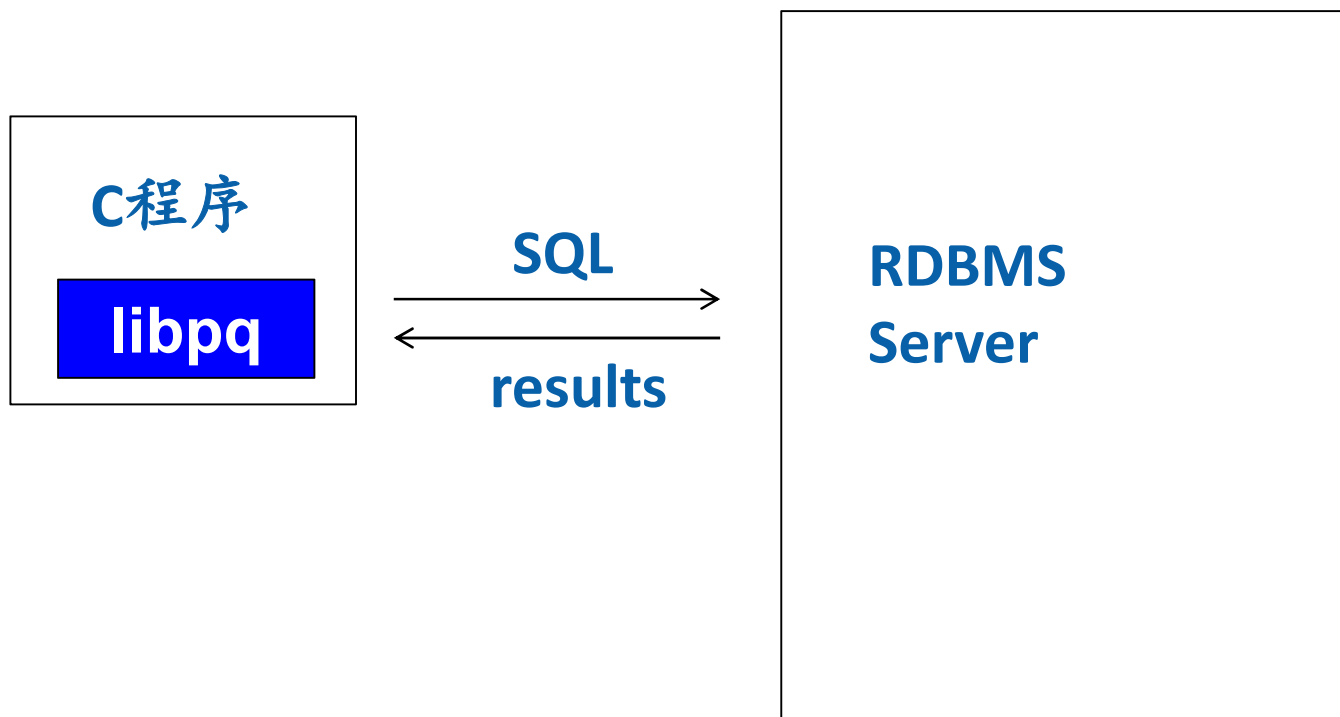
- 跳转到程序的开始位置
- 什么时候运行结束？
  - 从main返回
  - 实际上是调用了系统函数exit

# Outline

- 数据库编程简介
- 程序执行的基础知识
- 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



# 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

```
1 #include <stdio.h>
2 #include <postgresql/libpq-fe.h>
3
4 int main(int argc, char *argv[])
5 {
6     PGconn* conn = PQconnectdb("dbname=tpch");
7
8     if (PQstatus(conn) != CONNECTION_OK) {
9         fprintf(stderr, "Connection to db failed: %s",
10                PQerrorMessage(conn));
11         PQfinish(conn);
12         return 1;
13     }
14
15     printf("Connection is successful!\n");
16     PQfinish(conn);
17     return 0;
18 }
```

# myexample1.c

```
1 #include <stdio.h>
2 #include <postgresql/libpq-fe.h>
3
4 int main(int argc, char *argv[])
5 {
6     PGconn* conn = PQconnectdb("dbname=tpch");
7
8     if (PQstatus(conn) != CONNECTION_OK) {
9         fprintf(stderr, "Connection to db failed: %s",
10                PQerrorMessage(conn));
11         PQfinish(conn);
12         return 1;
13     }
14
15     printf("Connection is successful!\n");
16     PQfinish(conn);
17     return 0;
18 }
```

#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

```
1 #include <stdio.h>
2 #include <postgresql/libpq-fe.h>
3
4 int main(int argc, char *argv[])
5 {
6     PGconn* conn = PQconnectdb("dbname=tpch");
7
8     if (PQstatus(conn) != CONNECTION_OK) {
9         fprintf(stderr, "Connection to db failed: %s",
10                PQerrorMessage(conn));
11         PQfinish(conn);
12         return 1;
13     }
14
15     printf("Connection is successful!\n");
16     PQfinish(conn);
17     return 0;
18 }
```

**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

```
1 #include <stdio.h>
2 #include <postgresql/libpq-fe.h>
3
4 int main(int argc, char *argv[])
5 {
6     PGconn* conn = PQconnectdb("dbname=tpch");
7
8     if (PQstatus(conn) != CONNECTION_OK) {
9         fprintf(stderr, "Connection to db failed: %s",
10                PQerrorMessage(conn));
11         PQfinish(conn);
12         return 1;
13     }
14
15     printf("Connection is successful!\n");
16     PQfinish(conn);
17     return 0;
18 }
```

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

```
1 #include <stdio.h>
2 #include <postgresql/libpq-fe.h>
3
4 int main(int argc, char *argv[])
5 {
6     PGconn* conn = PQconnectdb("dbname=tpch");
7
8     if (PQstatus(conn) != CONNECTION_OK) {
9         fprintf(stderr, "Connection to db failed: %s",
10                PQerrorMessage(conn));
11         PQfinish(conn);
12         return 1;
13     }
14
15     printf("Connection is successful!\n");
16     PQfinish(conn);
17     return 0;
18 }
```

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

```
1 #include <stdio.h>
2 #include <postgresql/libpq-fe.h>
3
4 int main(int argc, char *argv[])
5 {
6     PGconn* conn = PQconnectdb("dbname=tpch");
7
8     if (PQstatus(conn) != CONNECTION_OK) {
9         fprintf(stderr, "Connection to db failed: %s",
10                PQerrorMessage(conn));
11         PQfinish(conn);
12         return 1;
13     }
14
15     printf("Connection is successful!\n");
16     PQfinish(conn);
17     return 0;
18 }
```

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!
```



# 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
<code>char *PQhost (const PGconn *conn);</code>	机器名
<code>char *PQport(const PGconn *conn);</code>	端口号
<code>char *PQdb(const PGconn *conn);</code>	数据库名
<code>char *PQuser(const PGconn *conn);</code>	用户名
<code>char *PQpass(const PGconn *conn);</code>	密码
<code>int PQserverVersion(const PGconn *conn);</code>	Postgresql的版本号
<code>char *PQerrorMessage(const PGconn *conn);</code>	出错信息

# PQfinish: 关闭连接

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

- 结束关闭连接

# 回到myexample1.c，再看一下

```
1 #include <stdio.h>
2 #include <postgresql/libpq-fe.h>
3
4 int main(int argc, char *argv[])
5 {
6     PGconn* conn = PQconnectdb("dbname=tpch");
7
8     if (PQstatus(conn) != CONNECTION_OK) {
9         fprintf(stderr, "Connection to db failed: %s",
10                PQerrorMessage(conn));
11         PQfinish(conn);
12         return 1;
13     }
14
15     printf("Connection is successful!\n");
16     PQfinish(conn);
17     return 0;
18 }
```

向localhost上运行的Postgresql数据库系统建立连接，打开tpch数据库，其它连接参数为默认值，如果成功写成功信息，最后关闭连接

# myexample2.c

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <postgresql/libpq-fe.h>
4
5 void exitNicely(PGconn* conn)
6 { PQfinish(conn); exit(1); }
7
8 int main(int argc, char *argv[])
9 {
10     PGconn* conn = PQconnectdb("dbname=tpch");
11
12     if (PQstatus(conn) != CONNECTION_OK) {
13         fprintf(stderr, "Connection to db failed: %s\n", PQerrorMessage(conn));
14         exitNicely(conn);
15     }
16
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);
30
31     PQfinish(conn);
32     return 0;
33 }
```

# myexample2.c

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <postgresql/libpq-fe.h>
4
5 void exitNicely(PGconn* conn)
6 { PQfinish(conn); exit(1);}
7
8 int main(int argc, char *argv[])
9 {
10     PGconn* conn = PQconnectdb("dbname=tpch");
11
12     if (PQstatus(conn) != CONNECTION_OK) {
13         fprintf(stderr, "Connection to db failed: %s\n", PQerrorMessage(conn));
14         exitNicely(conn);
15     }
16
31     PQfinish(conn);
32     return 0;
33 }
```

前部和尾部是与myexample1.c是一致的

注意：exitNicely函数调用exit( )直接退出了



# 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);
```

**PQexec**对于打开的数据库连接，执行一个SQL语句



# 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);
```

**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**exitNicely**

# 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);
```

**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$
```

# 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

## myexample2.c

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <postgresql/libpq-fe.h>
4
5 void exitNicely(PGconn* conn)
6 { PQfinish(conn); exit(1); }
7
8 int main(int argc, char *argv[])
9 {
10     PGconn* conn = PQconnectdb("dbname=tpch");
11
12     if (PQstatus(conn) != CONNECTION_OK) {
13         fprintf(stderr, "Connection to db failed: %s\n", PQerrorMessage(conn));
14         exitNicely(conn);
15     }
16
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);
30
31     PQfinish(conn);
32     return 0;
33 }
```

# myexample2-1.c: 我们试一下html输出

```
24
25     printf("<html><body>\n");
26
27     PQprintOpt po = {0};
28     po.header=1; po.align=1; po.fieldSep="|";
29     po.html3=1; po.caption="Region Information"; po.tableOpt="border=1";
30     PQprint(stdout, res, &po);
31
32     printf("</body></html>\n");
33
```

- 使用html格式输出: html3=1
- 设置了html table标题: caption
- 设置了html table的属性: tableOpt

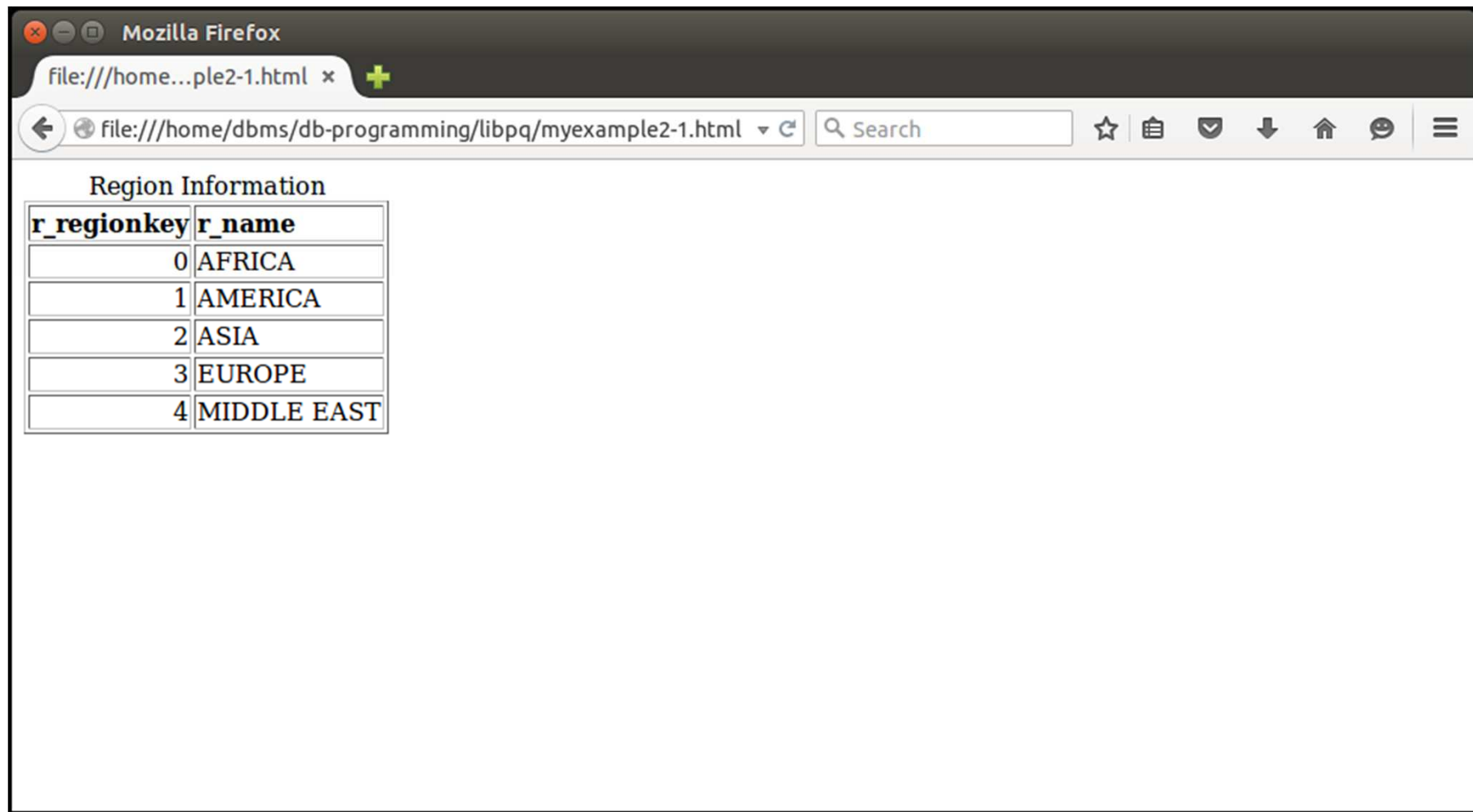
# 编译运行myexample2-1.c

## • 运行

```
dbms@ubuntu:~/db-programming/libpq$ ./myexample2-1 > myexample2-1.html
dbms@ubuntu:~/db-programming/libpq$ cat myexample2-1.html
<html><body>
<table border=1><caption align="top">Region Information</caption>
<tr><th align="right">r_regionkey</th><th align="left">r_name</th></tr>
<tr><td align="right">0</td><td align="left">AFRICA</td></tr>
<tr><td align="right">1</td><td align="left">AMERICA</td></tr>
<tr><td align="right">2</td><td align="left">ASIA</td></tr>
<tr><td align="right">3</td><td align="left">EUROPE</td></tr>
<tr><td align="right">4</td><td align="left">MIDDLE EAST</td></tr>
</table>
</body></html>
dbms@ubuntu:~/db-programming/libpq$
```

- 把stdout重定向写入了一个文件（> 文件名）
- 这里table部分是PQprint输出的，注意tableOpt和caption的位置
- html和body部分是我们自己printf的

# 我们用浏览器看一下产生的html





## myexample3.c 获取返回的行列值

```
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     }
```



# 获得res具体信息的函数

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

# myexample3.c 获取返回的行列值

```
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

# PQgetvalue

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

- 行和列都是从0开始

# myexample3.c 获取返回的行列值

```
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讲解内容

- 前面讲了
  - 建立连接，关闭连接
  - 执行SQL，获得结果，打印结果
- 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);
```

# 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
<code>paramFormat[k]</code>	0	1	忽略
<code>paramVal[k]</code>	指向以0结尾的串	指向二进制值	NULL
<code>paramLen[k]</code>	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 REGPROCID 24
#define TEXTOID 25
```

# 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
```

# 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

- 其它内容

- <https://www.postgresql.org/docs/9.3/static/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



# Outline

- 数据库编程简介
- 程序执行的基础知识
- 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

# 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



# 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;
```

# 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

# 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;

# 关闭连接

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

- 如果不写connection, 那么默认为当前的



## 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 }
```

## 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 }
```

需要在SQL中用到的  
变量定义

## 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 }
```

连接数据库

## 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 }
```

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



## 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

# 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!
```

# 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];
```

- ecpg会产生类似如下的代码

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

# 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

☞ 如何处理多个返回记录？

# 游标 (Cursor)

## 定义游标

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

## 打开游标

```
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

### 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 }
```

## 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

```
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 }
```

循环读取游标直至  
完毕NOT FOUND



# 编译运行myexample6.pgc



- 运行

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

# 游标的FETCH语句

FETCH [方向] cursor\_name

- 方向

- NEXT (默认) , PRIOR, FORWARD, BACKWARD
- 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

# 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

# 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;
```

用 Prepare  
准备, 然后  
打开 cursor

```
EXEC SQL WHENEVER NOT FOUND DO BREAK;
```

```
while (1) {
```

```
    EXEC SQL FETCH mycur INTO :v1, :v2;
```

```
    .....
```

```
}
```

需要事先知道返回多少列,  
每列的类型是什么 😞

```
EXEC SQL CLOSE mycur;
```

```
EXEC SQL COMMIT;
```



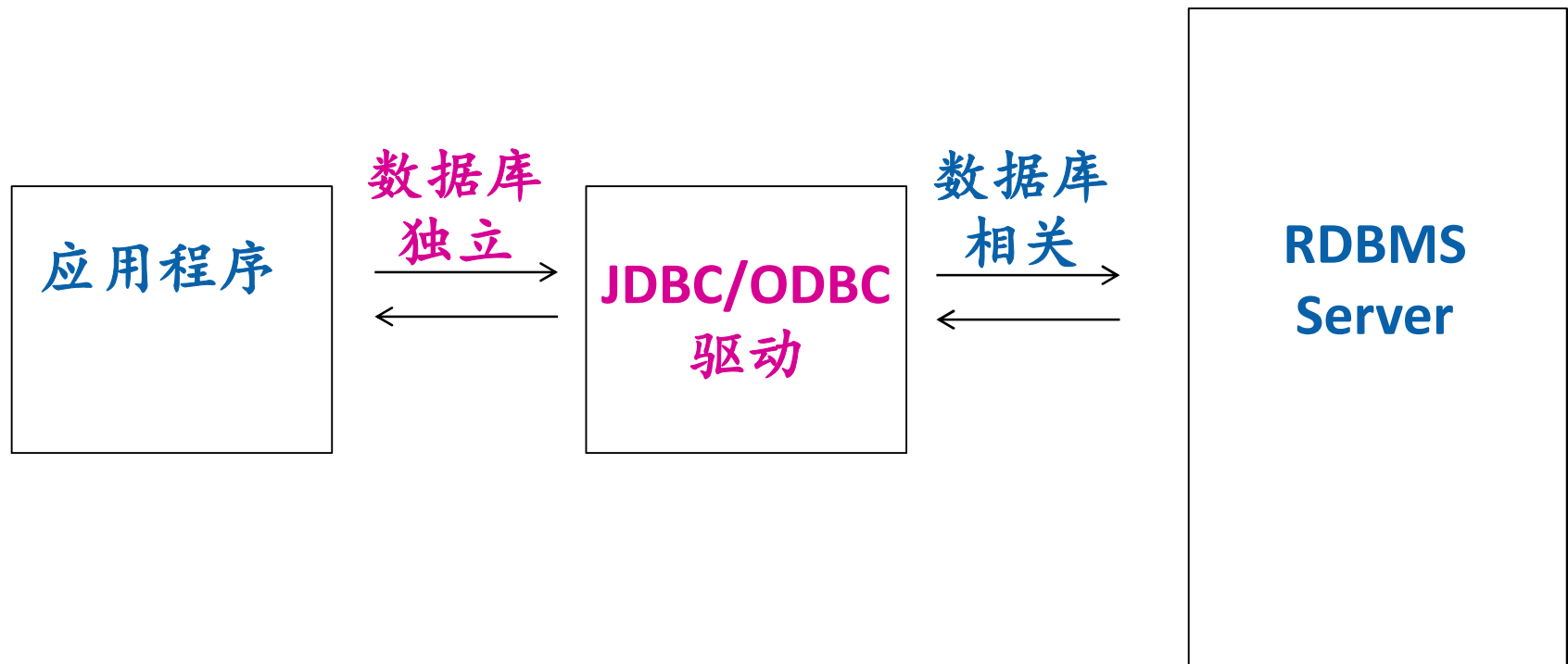
# Outline

- 数据库编程简介
- 程序执行的基础知识
- 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

# 为什么要JDBC和ODBC?

- 前述方法都是和特定数据库相关的
  - 即使嵌入式SQL的语法是标准的
  - 但是，仍然需要使用特定数据库的工具进行编译
  - 产生的代码只可以访问特定数据库系统
- 如果希望产生的代码与数据库独立，不需要重新编译就可以访问不同数据库系统
- 那么就使用JDBC或ODBC
  - JDBC（Java Database Connectivity）：基于Java，是由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

# JDBC/ODBC



# 功能

- 与libpq和嵌入式SQL大同小异

- 建立连接
- 关闭连接
- 执行语句
- 获取结果
- 等等

- JDBC

- <http://docs.oracle.com/javase/tutorial/jdbc/index.html>

- ODBC

- <http://www.unixodbc.org/>
- <http://www.easysoft.com/developer/languages/c/index.html>

# Outline

- 数据库编程简介
- 程序执行的基础知识
- 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
  - 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};

- 上述是主要部分，更多的选项参见：  
<https://www.postgresql.org/docs/9.3/static/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

# 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

## 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

## 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 IMMUTABLE STRICT;
```

- 是否读取和修改数据库状态

- **IMMUTABLE**：纯计算，不依赖于数据库内容，输入相同输出总是一样的

- **STABLE**：读数据库内容，不写数据库内容，如果数据库内容不变，那么相同输入有相同输出

- **VOLATILE**：默认，可能写数据库内容，无法优化

## 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 IMMUTABLE STRICT;
```

- 是否处理NULL参数

- **STRICT**: 输入为NULL时，系统不调用，而是返回NULL
- **CALLED ON NULL INPUT**: 默认，函数可以处理NULL参数

# 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2”： 纯计算的例子

```
CREATE FUNCTION sub2(x integer, y integer)
RETURNS integer AS $$
```

```
    select $1 - $2 as answer;
```

```
$$ LANGUAGE SQL IMMUTABLE STRICT;
```

可以用\$*k*来引用对应的函数参数



## 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(name varchar(20))  
RETURNS TABLE(sname varchar(20), major  
varchar(20), gpa float)  
AS $$
```

```
    select S.name, S.major, S.gpa  
    from Student S  
    where S.name= $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

# Outline

- 数据库编程简介
- 程序执行的基础知识
- 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
  - 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

```
1 #include <string.h>
2 #include "postgres.h"
3 #include "fmgr.h"
4
5 #ifdef PG_MODULE_MAGIC
6 PG_MODULE_MAGIC;
7 #endif
8
9 PG_FUNCTION_INFO_V1(sub_xy);
10 Datum sub_xy(PG_FUNCTION_ARGS)
11 {
12     int32 x = PG_GETARG_INT32(0);
13     int32 y = PG_GETARG_INT32(1);
14
15     PG_RETURN_INT32(x - y);
16 }
17
18 PG_FUNCTION_INFO_V1(concat_text);
19 Datum concat_text(PG_FUNCTION_ARGS)
20 {
21     text *arg1 = PG_GETARG_TEXT_P(0);
22     text *arg2 = PG_GETARG_TEXT_P(1);
23     int32 new_text_size = VARSIZE(arg1) + VARSIZE(arg2) - VARHDRSZ;
24     text *new_text = (text *) palloc(new_text_size);
25
26     SET_VARSIZE(new_text, new_text_size);
27     memcpy(VARDATA(new_text), VARDATA(arg1), VARSIZE(arg1) - VARHDRSZ);
28     memcpy(VARDATA(new_text) + (VARSIZE(arg1) - VARHDRSZ),
29           VARDATA(arg2), VARSIZE(arg2) - VARHDRSZ);
30     PG_RETURN_TEXT_P(new_text);
31 }
```

## 包含头文件

```
1 #include <string.h>
2 #include "postgres.h"
3 #include "fmgr.h"
4
5 #ifdef PG_MODULE_MAGIC
6 PG_MODULE_MAGIC;
7 #endif
8
9 PG_FUNCTION_INFO_V1(sub_xy);
10 Datum sub_xy(PG_FUNCTION_ARGS)
11 {
12     int32 x = PG_GETARG_INT32(0);
13     int32 y = PG_GETARG_INT32(1);
14
15     PG_RETURN_INT32(x - y);
16 }
17
18 PG_FUNCTION_INFO_V1(concat_text);
19 Datum concat_text(PG_FUNCTION_ARGS)
20 {
21     text *arg1 = PG_GETARG_TEXT_P(0);
22     text *arg2 = PG_GETARG_TEXT_P(1);
23     int32 new_text_size = VARSIZE(arg1) + VARSIZE(arg2) - VARHDRSZ;
24     text *new_text = (text *) palloc(new_text_size);
25
26     SET_VARSIZE(new_text, new_text_size);
27     memcpy(VARDATA(new_text), VARDATA(arg1), VARSIZE(arg1) - VARHDRSZ);
28     memcpy(VARDATA(new_text) + (VARSIZE(arg1) - VARHDRSZ),
29           VARDATA(arg2), VARSIZE(arg2) - VARHDRSZ);
30     PG_RETURN_TEXT_P(new_text);
31 }
```



```
1 #include <string.h>
2 #include "postgres.h"
3 #include "fmgr.h"
4
5 #ifdef PG_MODULE_MAGIC
6 PG_MODULE_MAGIC;
7 #endif
8
9 PG_FUNCTION_INFO_V1(sub_xy);
10 Datum sub_xy(PG_FUNCTION_ARGS)
11 {
12     int32 x = PG_GETARG_INT32(0);
13     int32 y = PG_GETARG_INT32(1);
14
15     PG_RETURN_INT32(x - y);
16 }
17
18 PG_FUNCTION_INFO_V1(concat_text);
19 Datum concat_text(PG_FUNCTION_ARGS)
20 {
21     text *arg1 = PG_GETARG_TEXT_P(0);
22     text *arg2 = PG_GETARG_TEXT_P(1);
23     int32 new_text_size = VARSIZE(arg1) + VARSIZE(arg2) - VARHDRSZ;
24     text *new_text = (text *) palloc(new_text_size);
25
26     SET_VARSIZE(new_text, new_text_size);
27     memcpy(VARDATA(new_text), VARDATA(arg1), VARSIZE(arg1) - VARHDRSZ);
28     memcpy(VARDATA(new_text) + (VARSIZE(arg1) - VARHDRSZ),
29           VARDATA(arg2), VARSIZE(arg2) - VARHDRSZ);
30     PG_RETURN_TEXT_P(new_text);
31 }
```

PostgreSQL UDF模块必须包括

```
1 #include <string.h>
2 #include "postgres.h"
3 #include "fmgr.h"
4
5 #ifdef PG_MODULE_MAGIC
6 PG_MODULE_MAGIC;
7 #endif
```

## UDF函数头定义

注意：除函数名外其它是不变的

```
8
9 PG_FUNCTION_INFO_V1(sub_xy);
10 Datum sub_xy(PG_FUNCTION_ARGS)
```

```
11 {
12     int32 x = PG_GETARG_INT32(0);
13     int32 y = PG_GETARG_INT32(1);
14
15     PG_RETURN_INT32(x - y);
16 }
```

```
17
18 PG_FUNCTION_INFO_V1(concat_text);
19 Datum concat_text(PG_FUNCTION_ARGS)
```

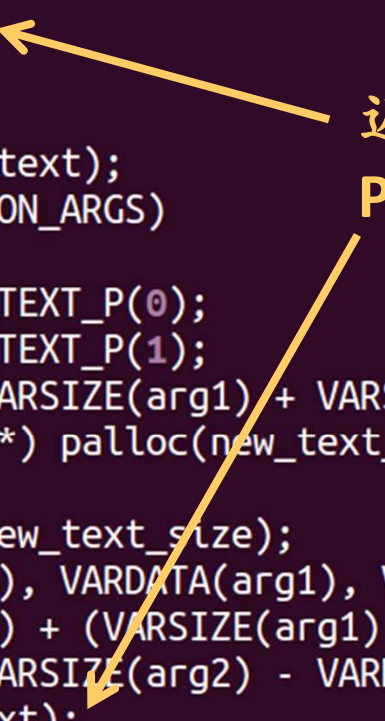
```
20 {
21     text *arg1 = PG_GETARG_TEXT_P(0);
22     text *arg2 = PG_GETARG_TEXT_P(1);
23     int32 new_text_size = VARSIZE(arg1) + VARSIZE(arg2) - VARHDRSZ;
24     text *new_text = (text *) palloc(new_text_size);
25
26     SET_VARSIZE(new_text, new_text_size);
27     memcpy(VARDATA(new_text), VARDATA(arg1), VARSIZE(arg1) - VARHDRSZ);
28     memcpy(VARDATA(new_text) + (VARSIZE(arg1) - VARHDRSZ),
29           VARDATA(arg2), VARSIZE(arg2) - VARHDRSZ);
30     PG_RETURN_TEXT_P(new_text);
31 }
```

```
1 #include <string.h>
2 #include "postgres.h"
3 #include "fmgr.h"
4
5 #ifdef PG_MODULE_MAGIC
6 PG_MODULE_MAGIC;
7 #endif
8
9 PG_FUNCTION_INFO_V1(sub_xy);
10 Datum sub_xy(PG_FUNCTION_ARGS)
11 {
12     int32 x = PG_GETARG_INT32(0);
13     int32 y = PG_GETARG_INT32(1);
14
15     PG_RETURN_INT32(x - y);
16 }
17
18 PG_FUNCTION_INFO_V1(concat_text);
19 Datum concat_text(PG_FUNCTION_ARGS)
20 {
21     text *arg1 = PG_GETARG_TEXT_P(0);
22     text *arg2 = PG_GETARG_TEXT_P(1);
23     int32 new_text_size = VARSIZE(arg1) + VARSIZE(arg2) - VARHDRSZ;
24     text *new_text = (text *) palloc(new_text_size);
25
26     SET_VARSIZE(new_text, new_text_size);
27     memcpy(VARDATA(new_text), VARDATA(arg1), VARSIZE(arg1) - VARHDRSZ);
28     memcpy(VARDATA(new_text) + (VARSIZE(arg1) - VARHDRSZ),
29           VARDATA(arg2), VARSIZE(arg2) - VARHDRSZ);
30     PG_RETURN_TEXT_P(new_text);
31 }
```

获取输入参数用  
**PG\_GETARG\_XXX**



```
1 #include <string.h>
2 #include "postgres.h"
3 #include "fmgr.h"
4
5 #ifdef PG_MODULE_MAGIC
6 PG_MODULE_MAGIC;
7 #endif
8
9 PG_FUNCTION_INFO_V1(sub_xy);
10 Datum sub_xy(PG_FUNCTION_ARGS)
11 {
12     int32 x = PG_GETARG_INT32(0);
13     int32 y = PG_GETARG_INT32(1);
14
15     PG_RETURN_INT32(x - y);
16 }
17
18 PG_FUNCTION_INFO_V1(concat_text);
19 Datum concat_text(PG_FUNCTION_ARGS)
20 {
21     text *arg1 = PG_GETARG_TEXT_P(0);
22     text *arg2 = PG_GETARG_TEXT_P(1);
23     int32 new_text_size = VARSIZE(arg1) + VARSIZE(arg2) - VARHDRSZ;
24     text *new_text = (text *) palloc(new_text_size);
25
26     SET_VARSIZE(new_text, new_text_size);
27     memcpy(VARDATA(new_text), VARDATA(arg1), VARSIZE(arg1) - VARHDRSZ);
28     memcpy(VARDATA(new_text) + (VARSIZE(arg1) - VARHDRSZ),
29           VARDATA(arg2), VARSIZE(arg2) - VARHDRSZ);
30     PG_RETURN_TEXT_P(new_text);
31 }
```



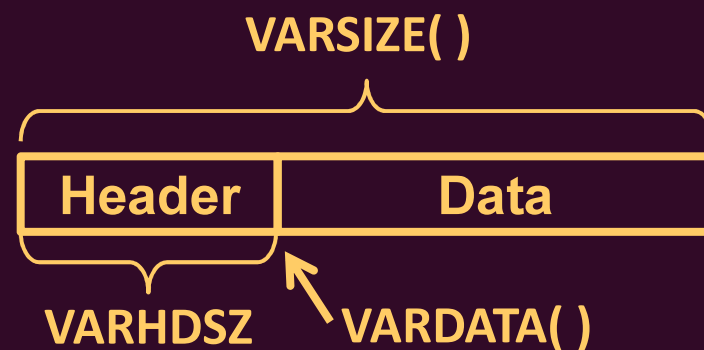
返回值用

**PG\_RETURN\_XXX**

```
1 #include <string.h>
2 #include "postgres.h"
3 #include "fmgr.h"
4
5 #ifdef PG_MODULE_MAGIC
6 PG_MODULE_MAGIC;
7 #endif
8
9 PG_FUNCTION_INFO_V1(sub_xy);
10 Datum sub_xy(PG_FUNCTION_ARGS)
11 {
12     int32 x = PG_GETARG_INT32(0);
13     int32 y = PG_GETARG_INT32(1);
14
15     PG_RETURN_INT32(x - y);
16 }
17
18 PG_FUNCTION_INFO_V1(concat_text);
19 Datum concat_text(PG_FUNCTION_ARGS)
20 {
21     text *arg1 = PG_GETARG_TEXT_P(0);
22     text *arg2 = PG_GETARG_TEXT_P(1);
23     int32 new_text_size = VARSIZE(arg1) + VARSIZE(arg2) - VARHDRSZ;
24     text *new_text = (text *) palloc(new_text_size);
25
26     SET_VARSIZE(new_text, new_text_size);
27     memcpy(VARDATA(new_text), VARDATA(arg1), VARSIZE(arg1) - VARHDRSZ);
28     memcpy(VARDATA(new_text) + (VARSIZE(arg1) - VARHDRSZ),
29           VARDATA(arg2), VARSIZE(arg2) - VARHDRSZ);
30     PG_RETURN_TEXT_P(new_text);
31 }
```

获取输入的int32,  
然后相减

## 变长类型的结构:



```
1 #include <string.h>
2 #include "postgres.h"
3 #include "fmgr.h"
4
5 #ifdef PG_MODULE_MAGIC
6 PG_MODULE_MAGIC;
7 #endif
8
9 PG_FUNCTION_INFO_V1(sub_xy);
10 Datum sub_xy(PG_FUNCTION_ARGS)
11 {
12     int32 x = PG_GETARG_INT32(0);
13     int32 y = PG_GETARG_INT32(1);
14
15     PG_RETURN_INT32(x - y);
16 }
17
18 PG_FUNCTION_INFO_V1(concat_text);
19 Datum concat_text(PG_FUNCTION_ARGS)
20 {
21     text *arg1 = PG_GETARG_TEXT_P(0);
22     text *arg2 = PG_GETARG_TEXT_P(1);
23     int32 new_text_size = VARSIZE(arg1) + VARSIZE(arg2) - VARHDRSZ;
24     text *new_text = (text *) palloc(new_text_size);
25
26     SET_VARSIZE(new_text, new_text_size);
27     memcpy(VARDATA(new_text), VARDATA(arg1), VARSIZE(arg1) - VARHDRSZ);
28     memcpy(VARDATA(new_text) + (VARSIZE(arg1) - VARHDRSZ),
29           VARDATA(arg2), VARSIZE(arg2) - VARHDRSZ);
30     PG_RETURN_TEXT_P(new_text);
31 }
```

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

# Outline

- 数据库编程简介
- 程序执行的基础知识
- 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
  - 过程SQL函数

# FUNCTION的编程语言

- SQL

- C语言

- 更多内容参见

- <https://www.postgresql.org/docs/9.3/static/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？

**PL/pgSQL**

过程型语言结构，  
如循环，if语句

普通SQL

# 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 程序定义

LANGUAGE plpgsql;

☞ 程序定义部分

# PL/pgSQL程序定义

\$\$

[DECLARE

declarations]

} 变量声明

BEGIN

statements

} 程序语句

END;

\$\$

- 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：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
[ ELIF 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

## 举例：返回所有小写的region名

```
create function all_region()  
returns table(lower_name char(25)) as $$  
declare  
    a_row record;  
begin  
    for a_row in select * from region loop  
        lower_name := lower(a_row.r_name);  
        return next;  
    end loop;  
    return;  
end;  
$$ language plpgsql;
```



# 执行一下

```
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=#
```

# 小结

- 数据库编程简介
- 程序执行的基础知识
- 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