第 2 章*Chapter 2*

## 客户端工具

本章将介绍 PostgreSQL 客户端工具，例如 pgAdmin 和 psql。pgAdmin 是一款功能丰富、开源免费的 PostgreSQL 图形化客户端工具，psql 是 PostgreSQL 自带的命令行客户端工具，功能全面，是 PostgreSQL 数据库工程师必须熟练掌握的命令行工具之一，本章将会详细介绍它的独特之处。

### 2.1 pgAdmin 4 简介

pgAdmin 是最流行的 PostgreSQL 图形化客户端工具，项目主页为：<https://www.pgadmin.org/>，由于 pgAdmin 4 工具使用比较简单，这里仅简单介绍。

#### 2.1.1 pgAdmin 4 安装

pgAdmin 支持 Linux、Unix、Mac OS X 和 Windows，由于编写此书时 pgAdmin 最新的大版本为 4，后面提到 pgAdmin 时我们都称为 pgAdmin 4，本节以在 Windows 7 上安装 pgAdmin 4 为例，简单介绍 pgAdmin 4。

安装包下载地址为：<https://www.postgresql.org/ftp/pgadmin/pgadmin4/v1.6/windows/>，下载完后根据提示安装即可，安装完打开 pgAdmin 4 的界面如图 2-1 所示。

#### 2.1.2 pgAdmin 4 使用

pgAdmin 4 的使用非常简单，这一小节将演示如何使用 pgAdmin 4 连接 PostgreSQL 数据库以及日常数据库操作。





图 2-1 pgAdmin 4 界面

## 1. pgAdmin 4 连接数据库

打开 pgAdmin 4 界面并创建服务，填写界面上的表单，如图 2-2 所示。

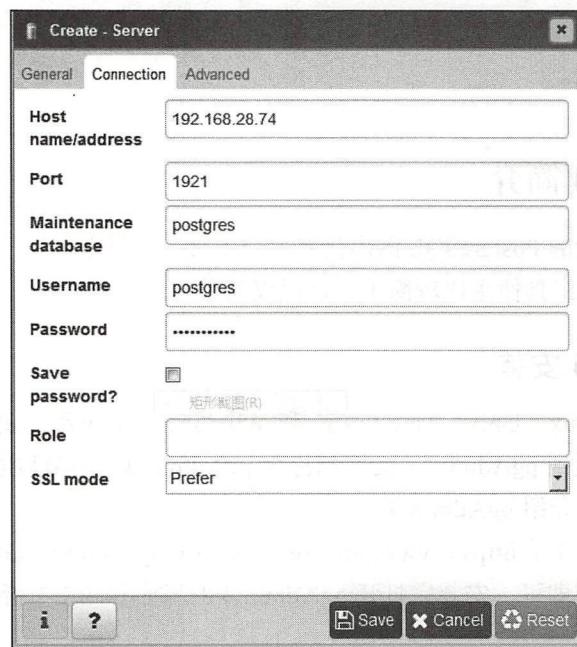


图 2-2 使用 pgAdmin 4 连接数据库

图 General 界面用于配置数据库连接别名，这里配置成 db1，Connection 配置页完成之后连接数据库如图 2-3 所示。



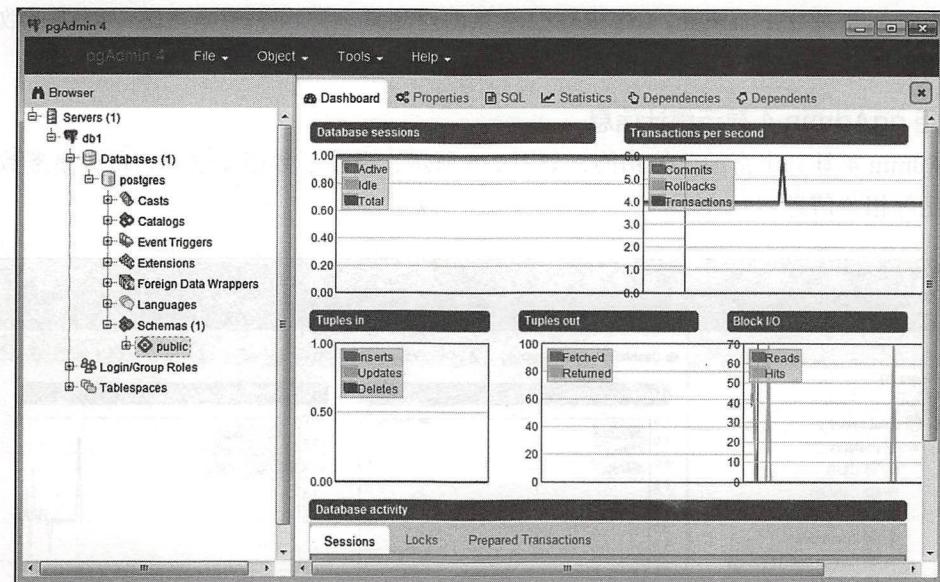


图 2-3 使用 pgAdmin 4 连接数据库

## 2. pgAdmin 4 查询工具的使用

在 pgAdmin 4 面板上点击 Tools 菜单中的 Query Tool，在弹出的窗口中可以进行日常的数据库 DDL、DML 操作，例如创建一张测试表，如图 2-4 所示。

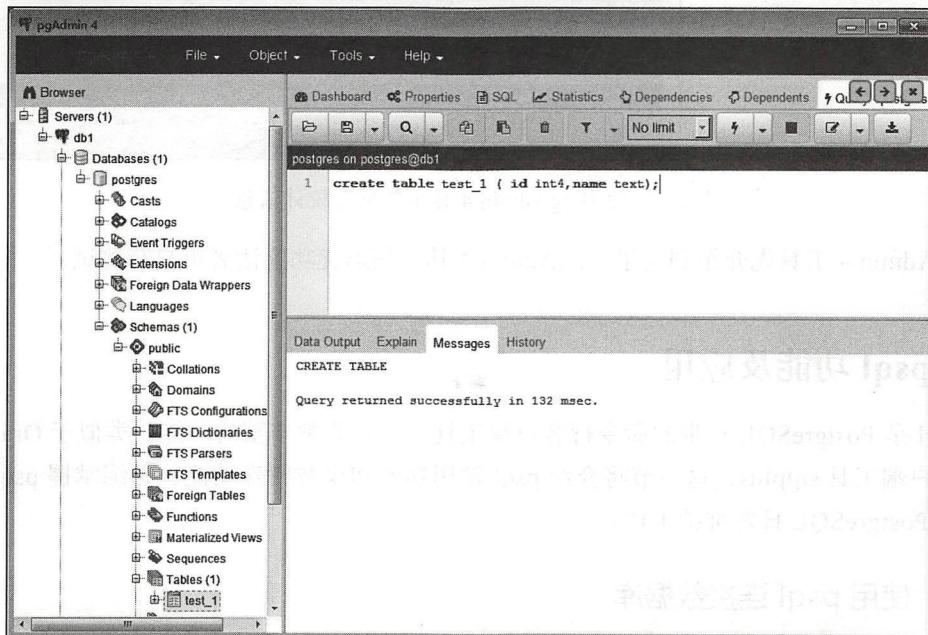


图 2-4 使用 pgAdmin 4 创建表



以上演示了使用 pgAdmin 4 连接数据库并创建测试表。由于篇幅有限，创建函数、序列、视图、DDL 等操作这里不再演示。

### 3. 用 pgAdmin 4 显示统计信息

pgAdmin 4 具有丰富的监控功能，如图 2-5 所示，显示了数据库进程、每秒事务数、记录数变化等相关信息。

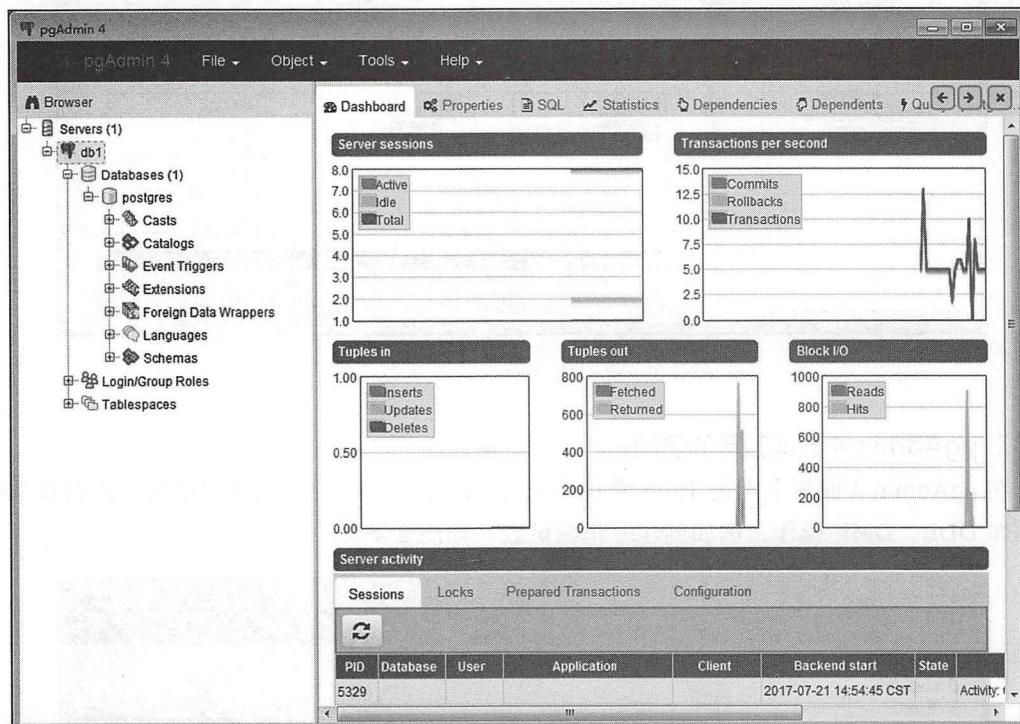


图 2-5 使用 pgAdmin 4 显示数据库统计信息

pgAdmin 4 工具先介绍到这里，pgAdmin 4 其他图形化功能读者可自行测试。

## 2.2 psql 功能及应用

psql 是 PostgreSQL 自带的命令行客户端工具，具有非常丰富的功能，类似于 Oracle 命令行客户端工具 sqlplus，这一节将介绍 psql 常用功能和少数特殊功能，熟练掌握 psql 能便捷处理 PostgreSQL 日常维护工作。

### 2.2.1 使用 psql 连接数据库

用 psql 连接数据库非常简单，可以在数据库服务端执行，也可以远程连接数据库，在





数据库服务端连接本地库示例如下所示：

```
[postgres@pghost1 ~]$ psql postgres postgres
psql (10.0)
Type "help" for help.
postgres=*
```

psql 后面的第一个 postgres 表示库名，第二个 postgres 表示用户名，端口号默认使用变量 \$PGPORT 配置的数据库端口号，这里是 1921 端口，为了后续演示方便，创建一个测试库 mydb，归属为用户 pguser，同时为 mydb 库分配一个新表空间 tbs\_mydb，如下所示：

```
--创建用户
postgres=# CREATE ROLE pguser WITH ENCRYPTED PASSWORD 'pguser';
CREATE ROLE

--创建表空间目录
[postgres@pghost1 ~]$ mkdir -p /database/pg10/pg_tbs/tbs_mydb

--创建表空间
postgres=# CREATE TABLESPACE tbs_mydb OWNER pguser LOCATION '/database/pg10/pg_
tbs/tbs_mydb';
CREATE TABLESPACE

--创建数据库
postgres=# CREATE DATABASE mydb
    WITH OWNER = pguser
        TEMPLATE = template0
        ENCODING = 'UTF8'
        TABLESPACE = tbs_mydb;
CREATE DATABASE

--赋权
GRANT ALL ON DATABASE mydb TO pguser WITH GRANT OPTION;
GRANT ALL ON TABLESPACE tbs_mydb TO pguser;
```

CREATE DATABASE 命令中的 OWNER 选项表示数据库属主，TEMPLATE 表示数据库模板，默认有 template0 和 template1 模板，也能自定义数据库模板，ENCODING 表示数据库字符集，这里设置成 UTF8，TABLESPACE 表示数据库的默认表空间，新建数据库对象将默认创建在此表空间上，通过 psql 远程连接数据库的语法如下所示：

```
psql [option...] [dbname [username]]
```

服务器 pgghost1 的 IP 为 192.168.28.74，pghost2 的 IP 为 192.168.28.75，在 pgghost2 主机上远程连接 pgghost1 上的 mydb 库命令如下：

```
[postgres@pghost2 ~]$ psql -h 192.168.28.74 -p 1921 mydb pguser
Password for user pguser:
psql (10.0)
Type "help" for help.
```





断开 psql 客户端连接使用 \q 元命令或 CTRL+D 快捷键即可，如下所示：

```
[postgres@pghost1 ~]$ psql mydb pguser
psql (10.0)
Type "help" for help.
mydb=> \q
```

下一小节将详细介绍 psql 支持的元命令。

## 2.2.2 psql 元命令介绍

psql 中的元命令是指以反斜线开头的命令，psql 提供丰富的元命令，能够便捷地管理数据库，比如查看数据库对象定义、查看数据库对象占用空间大小、列出数据库各种对象名称、数据导入导出等，比如查看数据库列表，如下所示：

```
postgres=# \l
          List of databases
   Name    |  Owner   | Encoding | Collate | Ctype | Access privileges
---+-----+-----+-----+-----+-----+
mydb   | postgres | UTF8    | C       | C     | =Tc/postgres      +
       |          |          |          |       | postgres=CTc/postgres+
       |          |          |          |       | pguser=C/postgres
postgres | postgres | UTF8    | C       | C     |
template0 | postgres | UTF8    | C       | C     | =c/postgres      +
template1 | postgres | UTF8    | C       | C     | =c/postgres      +
       |          |          |          |       | postgres=CTc/postgres
(4 rows)
```

### 1.\db 查看表空间列表

使用元命令 \db 查看表空间，如下所示：

```
postgres=# \db
          List of tablespaces
   Name    |  Owner   |           Location
---+-----+-----+
pg_default | postgres |
pg_global  | postgres |
tbs_mydb   | pguser   | /database/pg10/pg_tbs/tbs_mydb
(3 rows) 查看表定义
```

### 2.\d 查看表定义

先创建一张测试表，如下所示：

```
mydb=> CREATE TABLE test_1(id int4, name text,
        create_time timestamp without time zone default clock_timestamp());
CREATE TABLE

mydb=> ALTER TABLE test_1 ADD PRIMARY KEY (id);
ALTER TABLE
```





generate\_series 函数产生连续的整数，使用这个函数能非常方便地产生测试数据，查看表 test\_1 定义只需要执行元命令 \d 后接表名，如下所示：

```
mydb=> \d test_1
          Table "pguser.test_1"
  Column |          Type          | Collation | Nullable | Default
-----+-----+-----+-----+-----+
  id    | integer          |           | not null |
 name   | text             |           |           |
create_time| timestamp without time zone |           |           |
Indexes:
  "test_1_pkey" PRIMARY KEY , btree (id)
```

### 3. 查看表、索引占用空间大小

给测试表 test\_1 插入 500 万数据，如下所示：

```
mydb=> INSERT INTO test_1(id,name)
      SELECT n,n || '_francs'
      FROM generate_series(1,5000000) n;
INSERT 0 5000000
```

查看表大小执行 \dt+ 后接表名，如下所示：

```
mydb=> \dt+ test_1
          List of relations
 Schema | Name | Type | Owner | Size | Description
-----+-----+-----+-----+-----+
 pguser | test_1 | table | pguser | 287 MB |
(1 row)
```

查看索引大小执行 \di+ 后接索引名，如下所示：

```
mydb=> \di+ test_1_pkey
          List of relations
 Schema | Name | Type | Owner | Table | Size | Description
-----+-----+-----+-----+-----+-----+
 pguser | test_1_pkey | index | pguser | test_1 | 107 MB |
(1 row)
```

### 4. \sf 查看函数代码

元命令 \sf 后接函数名可查看函数定义，如下所示：

```
mydb=> \sf random_range
CREATE OR REPLACE FUNCTION pguser.random_range(integer, integer)
RETURNS integer
LANGUAGE sql
AS $function$
SELECT ($1 + FLOOR((\$2 - \$1 + 1) * random()) )::int4;
$function$
```

上述 \sf 命令后面可以只接函数的名称，或者函数名称及输入参数类型，例如 random\_





range(integer,integer)，PostgreSQL 支持名称相同但输入参数类型不同的函数，如果有同名函数，\sf 必须指定函数的参数类型。

### 5. \x 设置查询结果输出

使用 \x 可设置查询结果输出模式，如下所示：

```
mydb=> SELECT * FROM test_1 LIMIT 1;
          id |      name      |       create_time
-----+-----+-----
        1 | 1_pguser | 2017-07-22 11:16:15.97559
(1 row)

mydb=> \x
Expanded display is on.
mydb=> SELECT * FROM test_1 LIMIT 1;
-[ RECORD 1 ]-----
id          | 1
name        | 1_francs
create_time | 2017-07-22 11:16:15.97559
```

### 6. 获取元命令对应的 SQL 代码

psql 提供的元命令实质上向数据库发出相应的 SQL 查询，当使用 psql 连接数据库时，-E 选项可以获取元命令的 SQL 代码，如下所示：

```
[postgres@pghost1 ~]$ psql -E mydb pguser
psql (10.0)
Type "help" for help.

mydb=> \db
***** QUERY *****
SELECT spcname AS "Name",
pg_catalog.pg_get_userbyid(spcowner) AS "Owner",
pg_catalog.pg_tablespace_location(oid) AS "Location"
FROM pg_catalog.pg_tablespace
ORDER BY 1;
***** *****

          List of tablespaces
      Name   |   Owner   |           Location
-----+-----+-----
  pg_default | postgres |
  pg_global  | postgres |
tbs_mydb    | pguser    | /database/pg10/pg_tbs/tbs_mydb
(3 rows)
```

### 7. \? 元命令

PostgreSQL 支持的元命令很多，当忘记具体的元命令名称时可以查询手册，另一种便捷的方式是执行 \? 元命令列出所有的元命令，如下所示：



```
mydb=> \?

General
  \copyright           show PostgreSQL usage and distribution terms
  \crosstabview [COLUMNS] execute query and display results in crosstab
  \errverbose          show most recent error message at maximum verbosity
  \g [FILE] or ;      execute query (and send results to file or |pipe)
  \gexec               execute query, then execute each value in its result
  \gset [PREFIX]        execute query and store results in psql variables
  \gx [FILE]            as \g, but forces expanded output mode
  \q                  quit psql
  \watch [SEC]          execute query every SEC seconds

Help
  \? [commands]        show help on backslash commands
  \? options            show help on psql command-line options
  \? variables          show help on special variables
  \h [NAME]             help on syntax of SQL commands, * for all commands
```

\? 元命令可以迅速列出所有元命令以及这些元命令的说明及语法，给数据库维护管理带来很大的便利。

## 8. 便捷的 HELP 命令

psql 的 HELP 命令非常方便，使用元命令 \h 后接 SQL 命令关键字能将 SQL 命令的语法列出，对日常的数据库管理工作带来了极大的便利，例如 \h CREATE TABLESPACE 能显示此命令的语法，如下所示：

```
postgres=# \h CREATE TABLESPACE
Command:      CREATE TABLESPACE
Description: define a new tablespace
Syntax:
CREATE TABLESPACE tablespace_name
[ OWNER { new_owner | CURRENT_USER | SESSION_USER } ]
LOCATION 'directory'
[ WITH ( tablespace_option = value [, ... ] ) ]
```

\h 元命令后面不接任何 SQL 命令则会列出所有的 SQL 命令，为不完全记得 SQL 命令语法时提供了检索的途径。

### 2.2.3 psql 导入、导出表数据

psql 支持文件数据导入到数据库，也支持数据库表数据导出到文件中。COPY 命令和 \copy 命令都支持这两类操作，但两者有以下区别：

- 1) COPY 命令是 SQL 命令，\copy 是元命令。
- 2) COPY 命令必须具有 SUPERUSER 超级权限（将数据通过 stdin、stdout 方式导入导出情况除外），而 \copy 元命令不需要 SUPERUSER 权限。
- 3) COPY 命令读取或写入数据库服务端主机上的文件，而 \copy 元命令是从 psql 客户



端主机读取或写入文件。

4) 从性能方面看，大数据量导出到文件或大文件数据导入数据库，COPY 比 \copy 性能高。

### 1. 使用 COPY 命令导入导出数据

先来看看 COPY 命令如何将文本文件数据导入到数据库表中，首先在 mydb 库中创建测试表 test\_copy，如下所示：

```
mydb=> CREATE TABLE test_copy(id int4, name text);
CREATE TABLE
```

之后编写数据文件 test\_copy\_in.txt，字段分隔符用 TAB 键，也可设置其他分隔符，导入时再指定已设置的字段分隔符。test-copy-in.txt 文件如下所示：

```
[pg10@pghost1 script]$ cat test_copy_in.txt
1      a
2      b
3      c
```

之后以 postgres 用户登录 mydb 库，并将 test\_copy\_in.txt 文件中的数据导入到 test\_copy 表中。导入命令如下所示：

```
[pg10@pghost1 script]$ psql mydb postgres
psql (10.0)
Type "help" for help.

mydb=# COPY pguser.test_copy FROM '/home/postgres/script/test_copy_in.txt';
COPY 3
mydb=# SELECT * FROM pguser.test_copy ;
 id | name
----+---
 1  | a
 2  | b
 3  | c
(3 rows)
```

如果使用普通用户 pguser 导入文件数据，则报以下错误。

```
[pg10@pghost1 script]$ psql mydb pguser
psql (10.0)
Type "help" for help.

mydb=> COPY test_copy FROM '/home/postgres/script/test_copy_in.txt';
ERROR: must be superuser to COPY to or from a file
HINT: Anyone can COPY to stdout or from stdin. psql's \copy command also works for anyone.
```

报错信息示很明显，COPY 命令只有超级用户才能使用，而 \copy 元命令普通用户即可使用。接下来演示通过 COPY 命令将表 test\_copy 中的数据导出到文件，同样使用 postgres 用户登录到 mydb 库，如下所示。



```
[pg10@pghost1 script]$ psql mydb postgres
psql (10.0)
Type "help" for help.
mydb=# COPY pguser.test_copy TO '/home/postgres/test_copy.txt';
COPY 3
```

查看 test\_copy.txt 文件，如下所示：

```
[postgres@pghost1 ~]$ cat test_copy.txt
1      a
2      b
3      c
```

也可以将表数据输出到标准输出，而且不需要超级用户权限，如下所示：

```
[postgres@pghost1 ~]$ psql mydb pguser
psql (10.0)
Type "help" for help.

mydb=> COPY test_copy TO stdout;
1      a
2      b
3      c
```

也能从标准输入导入数据到表中，有兴趣的读者自行测试。

经常有运营或开发人员要求 DBA 提供生产库运营数据，为了显示方便，这时需要将数据导出到 csv 格式。

```
[postgres@pghost1 ~]$ psql mydb postgres
psql (10.0)
Type "help" for help.

mydb=# COPY pguser.test_copy TO '/home/postgres/test_copy.csv' WITH csv header;
COPY 4
```

上述命令中的 with csv header 是指导出格式为 csv 格式并且显示字段名称，以 csv 为后缀的文件可以使用 office excel 打开。以上数据导出示例都是基于全表数据导出的，如何仅导出表的一部分数据呢？如下代码仅导出表 test\_copy 中 ID 等于 1 的数据记录。

```
mydb=# COPY (SELECT * FROM pguser.test_copy WHERE id=1) TO '/home/postgres/1.txt';
COPY 1
mydb=# \q
[postgres@pghost1 ~]$ cat 1.txt
1      a
```

关于 COPY 命令更多说明详见手册 <https://www.postgresql.org/docs/10/static/sql-copy.html>。

## 2. 使用 \copy 元命令导入导出数据

COPY 命令是从数据库服务端主机读取或写入文件数据，而 \copy 元命令从 psql 客户端主机读取或写入文件数据，并且 \copy 元命令不需要超级用户权限，下面在 pghost2 主机



上以普通用户 pguser 远程登录 pghost1 主机上的 mydb 库，并且使用 \copy 元命令导出表 test\_copy 数据，如下所示：

```
[postgres@pghost2 ~]$ psql -h 192.168.28.74 -p 1921 mydb pguser  
Password for user pguser:  
pgsql (10.0)  
Type "help" for help.  
  
mydb=> \copy test_copy to '/home/postgres/test_copy.txt';  
COPY 3
```

查看 test\_copy.txt 文件，数据已导出，如下所示：

```
[postgres@pghost2 ~]$ cat test_copy.txt  
1      a  
2      b  
3      c
```

\copy 导入文件数据和 copy 命令类似，首先编写 test\_copy\_in.txt 文件，如下所示：

```
[postgres@pghost2 ~]$ cat test_copy_in.txt  
4      d
```

使用 \copy 命令导入文本 test\_copy\_in.txt 数据，如下所示：

```
[postgres@pghost2 ~]$ psql -h 192.168.28.74 -p 1921 mydb pguser  
Password for user pguser:  
pgsql (10.0)  
Type "help" for help.  
  
mydb=> \copy test_copy from '/home/postgres/test_copy_in.txt';  
COPY 1  
mydb=> SELECT * FROM test_copy WHERE id=4;  
 id | name  
----+---  
 4  | d  
(1 row)
```

没有超级用户权限的情况下，需要导出小表数据，通常使用 \copy 元命令，如果是大表数据导入操作，建议在数据库服务器主机使用 COPY 命令，效率更高。

## 2.2.4 psql 的语法和选项介绍

psql 连接数据库语法如下：

```
psql [option...] [dbname [username]]
```

其中 dbname 指连接的数据库名称，username 指登录数据库的用户名，option 有很多参数选项，这节列出重要的参数选项。



### 1. -A 设置非对齐输出模式

psql 执行 SQL 的输出默认是对齐模式，例如：

```
[postgres@phost1 ~]$ psql -c "SELECT * FROM user_ini WHERE id=1" mydb pguser
   id | user_id | user_name |          create_time
-----+-----+-----+-----
   1 | 186536 | KTU89H | 2017-08-05 15:59:25.359148+08
(1 row)
```

--注意返回结果，这里有空行

注意以上输出，格式是对齐的，psql 加上 -A 选项如下所示：

```
[postgres@phost1 ~]$ psql -A -c "SELECT * FROM user_ini WHERE id=1" mydb pguser
id|user_id|user_name|create_time
1|186536|KTU89H|2017-08-05 15:59:25.359148+08
(1 row)      --注意返回结果，没有空行
```

加上 -A 选项后以上输出的格式变成不对齐的了，并且返回结果中没有空行，接着看 -t 选项。

### 2. -t 只显示记录数据

另一个 psql 重要选项参数为 -t，-t 参数设置输出只显示数据，而不显示字段名称和返回的结果集行数，如下所示：

```
[postgres@phost1 ~]$ psql -t -c "SELECT * FROM user_ini WHERE id=1" mydb pguser
 1 | 186536 | KTU89H | 2017-08-05 15:59:25.359148+08
--注意返回结果，这里有空行
```

注意以上结果，字段名称不再显示，返回的结果集行数也没有显示，但尾部仍然有空行，因此 -t 参数通常和 -A 参数结合使用，这时仅返回数据本身，如下所示：

```
[postgres@phost1 ~]$ psql -At -c "SELECT * FROM user_ini WHERE id=1" mydb pguser
1|186536|KTU89H|2017-08-05 15:59:25.359148+08
```

以上结果仅返回了数据本身，在编写 shell 脚本时非常有效，特别是只取一个字段的时候，如下所示：

```
[postgres@phost1 ~]$ psql -At -c "SELECT user_name FROM user_ini WHERE id=1" mydb
pguser
KTU89H
```

### 3. -q 不显示输出信息

默认情况下，使用 psql 执行 SQL 命令时会返回多种消息，使用 -q 参数后将不再显示这些信息，下面通过一个例子进行演示，首先创建 test\_q.sql，并输入以下 SQL：

```
DROP TABLE if exists test_q;
CREATE TABLE test_q(id int4);
TRUNCATE TABLE test_q;
INSERT INTO test_q values (1);
INSERT INTO test_q values (2);
```

执行脚本 test\_q.sql，如下所示：

```
[postgres@pghost1 ~]$ psql mydb pguser -f test_q.sql
DROP TABLE
CREATE TABLE
TRUNCATE TABLE
INSERT 0 1
INSERT 0 1
```

执行脚本 test\_q.sql 后返回了大量信息，加上 -q 参数后，这些信息不再显示，如下所示：

```
[postgres@pghost1 ~]$ psql -q mydb pguser -f test_q.sql
--这里不再显示输出信息
```

-q 选项通常和 -c 或 -f 选项使用，在执行维护操作过程中，当输出信息不重要时，这个特性非常有用。

## 2.2.5 psql 执行 sql 脚本

psql 的 -c 选项支持在操作系统层面通过 psql 向数据库发起 SQL 命令，如下所示：

```
[postgres@pghost1 ~]$ psql -c "SELECT current_user;"
current_user
-----
postgres
(1 row)
```

-c 后接执行的 SQL 命令，可以使用单引号或双引号，同时支持格式化输出，如果想仅显示命令返回的结果，psql 加上 -At 选项即可，上一小节也有提到，如下所示：

```
[postgres@pghost1 ~]$ psql -At -c "SELECT current_user;"
```

上述内容演示了在操作系统层面通过 psql 执行 SQL 命令，那么如何导入数据库脚本文件呢？首先编写以下文件，文件名称为 test\_2.sql：

```
CREATE TABLE test_2(id int4);
INSERT INTO test_2 VALUES (1);
INSERT INTO test_2 VALUES (2);
INSERT INTO test_2 VALUES (3);
```

通过 -f 参数导入此脚本，命令如下：

```
[postgres@pghost1 ~]$ psql mydb pguser -f script/test_2.sql
CREATE TABLE
INSERT 0 1
INSERT 0 1
INSERT 0 1
```

以上命令的输出结果没有报错，表示文件中所有 SQL 正常导入。



**注意** psql 的 `-single-transaction` 或 `-1` 选项支持在一个事务中执行脚本，要么脚本中的所有 SQL 执行成功，如果其中有 SQL 执行失败，则文件中的所有 SQL 回滚。

## 2.2.6 psql 如何传递变量到 SQL

如何通过 psql 工具将变量传递到 SQL 中？例如以下 SQL：

```
SELECT * FROM table_name WHERE column_name = 变量;
```

下面演示两种传递变量的方式。

### 1.\set 元命令方式传递变量

\set 元子命令可以设置变量，格式如下所示，name 表示变量名称，value 表示变量值，如果不填写 value，变量值为空。

```
\set name value
```

test\_copy 表有四条记录，设置变量 `v_id` 值为 2，查询 id 值等于 2 的记录，如下所示：

```
mydb=>\set v_id 2
mydb=> SELECT * FROM test_copy WHERE id=:v_id;
      id | name
      ----+---
        2 | b
(1 row)
```

如果想取消之前变量设置的值，\set 命令后接参数名称即可，如下所示：

```
mydb=> \set v_id
```

通过 \set 元命令设置变量的一个典型应用场景是使用 pgbench 进行压力测试时使用 \set 元命令为变量赋值。

### 2. psql 的 -v 参数传递变量

另一种方法是通过 psql 的 `-v` 参数传递变量，首先编写 `select_1.sql` 脚本，脚本内容如下所示：

```
SELECT * FROM test_3 WHERE id=:v_id;
```

通过 psql 接 -v 传递变量，并执行脚本 `select_1.sql`，如下所示：

```
[postgres@pghost1 ~]$ psql -v v_id=1 mydb pguser -f select_1.sql
      id | name
      ---+---
        1 | a
(1 row)
```

以上设置变量 `v_id` 值为 1。

## 2.2.7 使用 psql 定制日常维护脚本

编写数据库维护脚本以提高数据库排障效率是 DBA 的工作职责之一，当数据库异常时，能够迅速发现问题并解决问题将为企业带来价值，当然数据库的健康检查和监控也要加强。这里主要介绍通过 psql 元命令定制日常维护脚本，预先将常用的数据库维护脚本配置好，数据库排障时直接使用，从而提高排障效率。

### 1. 定制维护脚本：查询活动会话

先来介绍 .psqlrc 文件，如果 psql 没有带 -X 选项，psql 尝试读取和执行用户 ~/.psqlrc 启动文件中的命令，结合这个文件能够方便地预先定制维护脚本，例如，查看数据库活动会话的 SQL 如下所示：

```
SELECT pid,username,datname,query,client_addr
FROM pg_stat_activity
WHERE pid <> pg_backend_pid() AND state='active' ORDER BY query;
```

pg\_stat\_activity 视图显示 PostgreSQL 进程信息，每一个进程在视图中存在一条记录，pid 指进程号，username 指数据库用户名，datname 指数据库名称，query 显示进程最近执行的 SQL，如果 state 值为 active 则 query 显示当前正在执行的 SQL，client\_addr 是进程的客户端 IP，state 指进程的状态，主要值为：

- ❑ active：后台进程正在执行 SQL。
- ❑ idle：后台进程为空闲状态，等待后续客户端发出命令。
- ❑ idle in transaction：后台进程正在事务中，并不是指正在执行 SQL。
- ❑ idle in transaction (aborted)：和 idle in transaction 状态类似，只是事务中的部分 SQL 异常。

关于此视图更多信息请参考手册 <https://www.postgresql.org/docs/10/static/monitoring-stats.html#pg-stat-activity-view>。

首先找到 ~/.psqlrc 文件，如果没有此文件则手工创建，编写以下内容，注意 \set 这行命令和 SQL 命令在一行中编写。

```
--查询活动会话
\set active_session 'select pid,username,datname,query,client_addr from pg_stat_
activity where pid <> pg_backend_pid() and state=\'active\' order by query,'
```

之后，重新连接数据库，执行 active\_session 命令，冒号后接变量名即可，如下所示：

```
postgres=# :active_session
      pid | username | datname |           query           | client_addr
-----+-----+-----+-----+
    14351 | pguser   | mydb   | update test_perl set create_time=now() WHERE id=$1; |
    14352 | pguser   | mydb   | update test_perl set create_time=now() WHERE id=$1; |
    14353 | pguser   | mydb   | update test_perl set create_time=now() WHERE id=$1; |
    14354 | pguser   | mydb   | update test_perl set create_time=now() WHERE id=$1; |
    14355 | pguser   | mydb   | update test_perl set create_time=now() WHERE id=$1; |
(5 rows)
```

通过以上设置，数据库排障时不需要临时手工编写查询活动会话的 SQL，只需输入：active\_session 即可，方便了日常维护操作。

## 2. 定制维护脚本：查询等待事件

PostgreSQL 也有等待事件的概念，对于问题诊断有较大参考作用，查询等待事件 SQL 如下所示：

```
SELECT pid,username,datname,query,client_addr,wait_event_type,wait_event
FROM pg_stat_activity
WHERE pid <> pg_backend_pid() AND wait_event is not null
ORDER BY wait_event_type;
```

同样，通过 \set 元命令将上述代码追加到 ~/.psqlrc 文件，注意 \set 命令和 SQL 命令在同一行中编写，如下所示：

```
--查看会话等待事件
\set wait_event 'select pid,username,datname,query,client_addr,wait_event_
type,wait_event from pg_stat_activity where pid <> pg_backend_pid() and wait_
event is not null order by wait_event_type;
```

之后，重新连接数据库，执行 wait\_event 命令，冒号后接变量名即可，如下所示：

```
postgres=# :wait_event
      pid | username | datname | query | client_addr | wait_event_type |      wait_event
-----+-----+-----+-----+-----+-----+-----+
    2652 |         |         |         |         | Activity       | AutoVacuumMain
  2655 | postgres |         |         |         | Activity       | LogicalLauncherMain
  2650 |         |         |         |         | Activity       | BgWriter Hibernate
  2649 |         |         |         |         | Activity       | CheckpointerMain
  2651 |         |         |         |         | Activity       | WalWriterMain
(5 rows)
```

以上介绍了查询活动会话和会话等待事件，其他维护脚本可根据实际情况定制，比如查看数据库连接数，如下所示：

```
--查看数据库连接数
\set connections 'select datname,username,client_addr,count(*) from pg_stat_
activity where pid <> pg_backend_pid() group by 1,2,3 order by 1,2,4 desc;'
```

通过元命令 \set 变量定制维护脚本只能在一定程度上方便数据库日常维护操作，工具化的监控工具不能少，比如 Zabbix 或 Nagios，这些监控工具可以非常方便地定制数据库各维度监控告警，并以图表形式展现性能数据。

## 2.2.8 psql 亮点功能

psql 还有其他非常突出的功能，比如显示 SQL 执行时间、反复执行当前 SQL、自动补全、历史命令上下翻动、客户端提示符等，这节主要介绍 psql 的这些常用的亮点功能。

### 1. \timing 显示 SQL 执行时间

\timing 元命令用于设置打开或关闭显示 SQL 的执行时间，单位为毫秒，例如：

```
mydb=> \timing
Timing is on.
mydb=> SELECT count(*) FROM user_ini;
      count
-----
      1000000
(1 row)

Time: 47.114 ms
```

以上显示 count 语句的执行时间为 47.114 毫秒，这个特性在调试 SQL 性能时非常有用，如果需要关闭这个选项，再次执行 \timing 元命令即可，如下所示：

```
mydb=> \timing
Timing is off.
```

### 2. \watch 反复执行当前 SQL

\watch 元命令会反复执行当前查询缓冲区的 SQL 命令，直到 SQL 被中止或执行失败，语法如下：

```
\watch [ seconds ]
```

seconds 表示两次执行间隔的时间，以秒为单位，默认为 2 秒，例如，每隔一秒反复执行 now() 函数查询当前时间：

```
mydb=> SELECT now();
      now
-----
      2017-08-14 11:20:02.157567+08
(1 row)

mydb=> \watch 1
Mon 14 Aug 2017 11:20:04 AM CST (every 1s)

      now
-----
      2017-08-14 11:20:04.299584+08
(1 row)

Mon 14 Aug 2017 11:20:05 AM CST (every 1s)

      now
-----
      2017-08-14 11:20:05.300991+08
```

以上设置是每秒执行一次 now() 命令。

### 3. Tab 键自动补全

psql 对 Tab 键自动补全功能的支持是一个很赞的特性，能够在没有完全记住数据库对象名称或者 SQL 命令语法的情况下使用，帮助用户轻松地完成各项数据库维护工作。例如，查询 mydb 库中某个 test 打头的表，但不记得具体表名，可以输入完 test 字符后按 Tab 键，psql 会提示以字符 test 打头的表，如下所示：

```
mydb=> SELECT * FROM test_
      test_1    test_2    test_copy  test_perl
mydb=> SELECT * FROM test_
```

DDL 也是支持 Tab 键自动补全的，如下所示：

```
mydb=> ALTER TABLE test_1 DROP CO
          COLUMN      CONSTRAINT
mydb=> ALTER TABLE test_1 DROP CO
```

### 4. 支持箭头键上下翻历史 SQL 命令

psql 支持箭头键上下翻历史 SQL 命令，非常方便，如下所示：

```
[postgres@pgghost1 ~]$ psql mydb pguser
psql (10.0)
Type "help" for help.
```

```
mydb=> SELECT count(*) FROM pg_stat_activity ; --这里使用箭头键上下翻历史命令
```

想要 psql 支持箭头键上下翻历史 SQL 命令，在编译安装 PostgreSQL 时需打开 readline 选项，这个选项在编译 PostgreSQL 时默认打开，也可以在编译时加上 --without-readline 选项关闭 readline，但不推荐。

### 5. psql 客户端提示符

以下命令显示了 psql 客户端提示符，“postgres=#”是默认的客户端提示符：

```
[postgres@pgghost1 ~]$ psql
psql (10.0)
Type "help" for help.
```

```
postgres=#
```

用户可根据喜好设置 psql 客户端提示符，psql 提供一系列选项供用户选择并设置，常用选项如下：

- %M：数据库服务器别名，不是指主机名，显示的是 psql 的 -h 参数设置的值；当连接建立在 Unix 域套接字上时则是 [local]。
- %>：数据库服务器的端口号。
- %n：数据库会话的用户名，在数据库会话期间，这个值可能会因为命令 SET SESSION AUTHORIZATION 的结果而改变。
- %/：当前数据库名称。

□ %#：如果是超级用户则显示“#”，其他用户显示“>”，在数据库会话期间，这个值可能会因为命令 SET SESSION AUTHORIZATION 的结果而改变。

□ %p：当前数据库连接的后台进程号。

□ %R：在 PROMPT1 中通常显示“=”，如果进程被断开则显示“!”。

上面仅介绍了主要的提示符，后面的演示示例可参考以上选项的解释。先来看 psql 客户端默认 prompt1 的配置，如下所示：

```
[postgres@phost1 ~]$ psql
psql (10.0)
Type "help" for help.

postgres=# \echo :PROMPT1
%/%R%#
```

元命令 \echo 是指显示变量值，PROMPT1 是系统提示符的变量，PROMPT1 是指当 psql 等待新命令发出时的常规提示符，它的默认设置为 %/%R%#，根据以上选项参数的解释很容易理解 %/ 指当前数据库名称 postgres，%R 指显示字符“=”，%# 显示字符“#”，下面看下 %M 选项，在数据库服务器主机通过 Unix 套接字连接，并设置 PROMPT1 值为 %M%R%#，如下所示：

```
[postgres@phost1 ~]$ psql
psql (10.0)
Type "help" for help.

postgres=# \set PROMPT1 '%M%R%#'
[local]=#
```

这时，psql 客户端显示 [local]，接下来在 phost2 主机上远程连接 phost1 上的库测试，并设置 PROMPT1 变量值为 “%M%R%#”，如下所示：

```
[postgres@phost2 ~]$ psql -h 192.168.28.74 mydb pguser -p 1921
Password for user pguser:
psql (10.0)
Type "help" for help.

mydb=> \set PROMPT1 '%M%R%#'
192.168.28.74=>
```

从上面看到，设置 PROMPT1 变量值为 “%M%R%#” 字符后，显示了 192.168.28.74 的 IP 地址，正好为 psql 参数 -h 的值。接下来演示稍复杂点的设置，如下所示：

```
[postgres@phost2 ~]$ psql -h 192.168.28.74 mydb pguser -p 1921
Password for user pguser:
psql (10.0)
Type "help" for help.

mydb=> \set PROMPT1 '%/@%M:@>%R%#'
mydb@192.168.28.74:1921=>
```



这里将 PROMPT1 设置成“%/@%M:%>%R%#”，“%>”是指数据库端口号，其他选项之前已介绍过，根据实践也非常好理解，设置好 PROMPT1 的格式后，可以将 PROMPT1 的设置命令写到客户端主机操作系统用户家目录的 .psqlrc 文件中，关于 .psqlrc 文件在 2.2.7 节中有详细介绍，在客户端主机操作系统用户家目录创建 .psqlrc 文件并写入以下代码，如下所示：

```
[postgres@pghost2 ~]$ touch .psqlrc
[postgres@pghost2 ~]$ vim .psqlrc
\set PROMPT1 '%/@%M:%>%R%#'
```

再次登录验证，代码已生效，如下所示：

```
[postgres@pghost2 ~]$ psql -h 192.168.28.74 mydb pguser -p 1921
Password for user pguser:
psql (10.0)
Type "help" for help.
```

mydb@192.168.28.74:1921=>

用户可根据自己的喜好设置 PROMPT1 并写入 .psqlrc 文件，psql 连接数据库时会读取 .psqlrc 文件并执行里面的命令。



提示 psql 默认有三个提示符：PROMPT1、PROMPT2、PROMPT3，PROMPT1 是指当 psql 等待新命令发出时的常规提示符，这个提示符使用得最多；PROMPT2 是指在命令输入过程中等待更多输入时发出的提示符，例如当命令没有使用分号终止或者引用没有被关闭时就会发出这个提示符，PROMPT2 的默认设置值与 PROMPT1 一样；PROMPT3 指在运行一个 SQL COPY FROM STDIN 命令并且需要在终端上输入一个行值时发出的提示符。

## 2.3 本章小结

本章介绍了 PostgreSQL 客户端连接工具 pgAdmin 4 和 psql 命令行工具，其中重点介绍了 psql 命令行工具的强大功能。通过学习本章内容，读者一方面了解到 pgAdmin 4 的基本用法，另一方面了解到 psql 工具的主要功能，比如元命令、数据导入导出、执行 SQL 脚本、带参数执行脚本、定制维护脚本等，熟练掌握 psql 能够提高数据库管理工作效率，对本书后续核心篇、进阶篇章节的阅读奠定了基础。



其...  
样对...  
adpa...  
入等...  
Chapter 3

### 第3章

## 数据类型

本章将介绍 PostgreSQL 的数据类型。PostgreSQL 的数据类型非常丰富，本章将介绍常规数据类型和一些非常规数据类型，比如常规数据类型中的数字类型、字符类型、日期/时间类型等，非常规数据类型中的布尔类型、网络地址类型、数组类型、范围类型、json/jsonb 类型等。介绍数据类型的同时也介绍数据类型相关操作符和函数，以及数据类型转换。

### 3.1 数字类型

PostgreSQL 支持的数字类型有整数类型、用户指定精度类型、浮点类型、serial 类型。

#### 3.1.1 数字类型列表

PostgreSQL 支持的数字类型如表 3-1 所示。

表 3-1 数字类型列表

类型名称	存储长度	描述	范围
smallint	2 字节	小范围整数类型	32 768 到 +32 767
integer	4 字节	整数类型	-2 147 483 648 到 +2 147 483 647
bigint	8 字节	大范围整数类型	-9 223 372 036 854 775 808 到 +9 223 372 036 854 775 807
decimal	可变	用户指定精度	小数点前 131 072 位；小数点后 16 383 位
numeric	可变	用户指定精度	小数点前 131 072 位；小数点后 16 383 位
real	4 字节	变长，不精确	6 位十进制精度



(续)

类型名称	存储长度	描述	范围
double precision	8字节	变长，不精确	15位十进制精度
smallserial	2字节	smallint 自增序列	1到32 767
serial	4字节	integer 自增序列	1到2 147 483 647
bigserial	8字节	bigint 自增序列	1到9 223 372 036 854 775 807

smallint、integer、bigint都是整数类型，存储一定范围的整数，超出范围将会报错。smallint存储2字节整数，字段定义时可写成int2，integer存储4字节整数，支持的数值范围比smallint大，字段定义时可写成int4，是最常用的整数类型，bigint存储8字节整数，支持的数值范围比integer大，字段定义时可写成int8。对于大多数使用整数类型的场景使用integer就够了，除非integer范围不够用的情况下才使用bigint。定义一张使用integer类型的表如下所示：

```
mydb=> CREATE TABLE test_integer (id1 integer,id2 int4);
```

decimal和numeric是等效的，可以存储指定精度的多位数据，比如带小数位的数据，适用于要求计算准确的数值运算，声明numeric的语法如下所示：

```
NUMERIC(precision, scale)
```

precision是指numeric数字里的全部位数，scale是指小数部分的数位数，例如18.222的precision为5，而scale为3；precision必须为正整数，scale可以是0或整数，由于numeric类型上的算术运算相比整数类型性能低，因此，如果两种数据类型都能满足业务需求，从性能上考虑不建议使用numeric数据类型。

real和double precision是指浮点数据类型，real支持4字节，double precision支持8字节，浮点数据类型在实际生产案例的使用相比整数类型会少些。

smallserial、serial和bigserial类型是指自增serial类型，严格意义上不能称之为一种数据类型，如下代码创建一张测试表，定义test\_serial表的id字段为serial类型：

```
mydb=> CREATE TABLE test_serial (id serial,flag text);
```

查看表test\_serial的表结构，如下所示：

```
mydb=> \d test_serial
          Table "pguser.test_serial"
   Column | Type    | Collation | Nullable | Default
-----+-----+-----+-----+
      id | integer |           | not null | nextval('test_serial_id_seq'::regclass)
     flag | text    |           |           |
```

以上显示id字段使用了序列test\_serial\_id\_seq，插入表数据时可以不指定serial字段名



称，将自动使用序列值填充，如下所示：

```
mydb=> INSERT INTO test_serial(flag) VALUES ('a');
INSERT 0 1
mydb=> INSERT INTO test_serial(flag) VALUES ('b');
INSERT 0 1
mydb=> INSERT INTO test_serial(flag) VALUES ('c');
INSERT 0 1
mydb=> SELECT * FROM test_serial;
 id | flag
----+-----
 1 | a
 2 | b
 3 | c
(3 rows)
```

### 3.1.2 数字类型操作符和数学函数

PostgreSQL 支持数字类型操作符和丰富的数学函数，例如支持加、减、乘、除、模取余操作符，如下所示：

```
mydb=> SELECT 1+2,2*3,4/2,8%3;
 ?column? | ?column? | ?column? | ?column?
-----+-----+-----+-----
 3 |      6 |      2 |      2
```

按模取余如下所示：

```
mydb=> SELECT mod(8,3);
mod
-----
 2
(1 row)
```

四舍五入函数如下所示：

```
mydb=> SELECT round(10.2),round(10.9);
 round | round
-----+-----
 10 |    11
(1 row)
```

返回大于或等于给出参数的最小整数，如下所示：

```
mydb=> SELECT ceil(3.6),ceil(-3.6);
 ceil | ceil
-----+-----
 4 |   -3
(1 row)
```

返回小于或等于给出参数的最大整数，如下所示：





```
mydb=> SELECT floor(3.6), floor(-3.6);
      floor | floor
-----+-----
      3  |   -4
(1 row)
```

## 3.2 字符类型

本节介绍 PostgreSQL 支持的字符类型，并且介绍常用的字符类型函数。

### 3.2.1 字符类型列表

PostgreSQL 支持的字符类型如表 3-2 所示。

表 3-2 字符数据类型列表

字符类型名称	描述
character varying(n), varchar(n)	变长，字符最大数有限制
character(n), char(n)	定长，字符数没达到最大值则使用空白填充
text	变长，无长度限制

character varying(n) 存储的是变长字符类型，n 是一个正整数，如果存储的字符串长度超出 n 则报错；如果存储的字符串长度比 n 小，character varying(n) 仅存储字符串的实际位数。character(n) 存储定长字符，如果存储的字符串长度超出 n 则报错；如果存储的字符串长度比 n 小，则用空白填充。为了验证此特性，下面做个实验，创建一张测试表，并插入一条测试数据，代码如下所示：

```
mydb=> CREATE TABLE test_char(col1 varchar (4), col2 character(4));
CREATE TABLE
mydb=> INSERT INTO test_char(col1,col2) VALUES ('a','a');
INSERT 0 1
```

表 test\_char 的字段 col1 类型为 character varying(4)，col2 类型为 character(4)，接下来计算两个字段值的字符串长度，代码如下所示：

```
mydb=> SELECT char_length(col1),char_length(col2) FROM test_char ;
      char_length | char_length
-----+-----
          1 |           1
(1 row)
```

char\_length(string) 显示字符串字符数，从上面结果可以看出字符串长度都为 1，接着查看两字段实际占用的物理空间大小，代码如下所示：

```
mydb=> SELECT octet_length(col1),octet_length(col2) FROM test_char ;
      octet_length | octet_length
```





```
-----+-----+-----+-----+
      |           |
1 |   4
-----+-----+-----+-----+
(1 row)
```

`octet_length(string)` 显示字符串占用的字节数, `col2` 字段占用了 4 个字节, 正好是 `col2` 字段定义的 `character` 长度。

值得一提的是 `character varying(n)` 类型如果不声明长度, 将存储任意长度的字符串, 而 `character(n)` 如果不声明长度则等效于 `character(1)`。

`text` 字符类型存储任意长度的字符串, 和没有声明字符长度的 `character varying` 字符类型几乎没有差别。

 提示 PostgreSQL 支持最大的字段大小为 1GB, 虽然文档上说没有声明长度的 `character varying` 和 `text` 都支持任意长度的字符串, 但仍受最大字段大小 1GB 的限制; 此外, 从性能上考虑这两种字符类型几乎没有差别, 只是 `character(n)` 类型当存储的字符串长度不够时会用空白填充, 这将带来存储空间一定程度的浪费, 使用时需注意。

### 3.2.2 字符类型函数

PostgreSQL 支持丰富的字符函数, 下面举例说明。

计算字符串中的字符数, 如下所示:

```
mydb=> SELECT char_length('abcd');
          +-----+
          | char_length |
          +-----+
          |          4 |
          +-----+
(1 row)
```

计算字符串占用的字节数, 如下所示:

```
mydb=> SELECT octet_length('abcd');
          +-----+
          | octet_length |
          +-----+
          |          4 |
          +-----+
(1 row)
```

指定字符在字符串的位置, 如下所示:

```
mydb=> SELECT position('a' in 'abcd');
          +-----+
          | position |
          +-----+
          |          1 |
          +-----+
(1 row)
```

提取字符串中的子串, 如下所示:

```
mydb=> SELECT substring('francs' from 3 for 4);
          +-----+
          | substring |
          +-----+
          |        ns |
          +-----+
(1 row)
```





```
ancs  
(1 row)
```

拆分字符串，`split_part` 函数语法如下：

```
split_part(string text, delimiter text, field int)
```

根据 `delimiter` 分隔符拆分字符串 `string`，并返回指定字段，字段从 1 开始，如下所示：

```
mydb=> SELECT split_part('abc@def1@nb','@',2);  
split_part  
-----  
def1  
(1 row)
```

## 3.3 时间 / 日期类型

PostgreSQL 对时间、日期数据类型的支持丰富而灵活，本节介绍 PostgreSQL 支持的时间、日期类型，及其操作符和常用函数。

### 3.3.1 时间 / 日期类型列表

PostgreSQL 支持的时间、日期类型如表 3-3 所示。

表 3-3 时间、日期数据类型列表

字符类型名称	存储长度	描述
<code>timestamp [ (p) ] [ without time zone ]</code>	8 字节	包括日期和时间，不带时区，简写成 <code>timestamp</code>
<code>timestamp [ (p) ] with time zone</code>	8 字节	包括日期和时间，带时区，简写成 <code>timestampz</code>
<code>date</code>	4 字节	日期，但不包含一天中的时间
<code>time [ (p) ] [ without time zone ]</code>	8 字节	一天中的时间，不包含日期，不带时区
<code>time [ (p) ] with time zone</code>	12 字节	一天中的时间，不包含日期，带时区
<code>interval [ fields ] [ (p) ]</code>	16 字节	时间间隔

我们通过一个简单的例子理解这几个时间、日期数据类型，先来看看系统自带的 `now()` 函数，`now()` 函数显示当前时间，返回的类型为 `timestamp [ (p) ] with time zone`，如下所示：

```
mydb=> SELECT now();  
now  
-----  
2017-07-29 09:44:25.493425+08  
(1 row)
```

这里提前介绍下类型转换，本章最后一节将专门介绍数据类型转换的常用方法，以下 SQL 中的两个冒号是指类型转换，转换成 `timestamp without time zone` 格式如下，注意返回的数据变化：





```
mydb=> SELECT now()::timestamp without time zone;  
now
```

```
-----  
2017-07-29 09:44:55.804403  
(1 row)
```

转换成 date 格式，如下所示：

```
mydb=> SELECT now()::date;  
now  
-----  
2017-07-29  
(1 row)
```

转换成 time without time zone，如下所示：

```
mydb=> SELECT now()::time without time zone;  
now  
-----  
09:45:49.390428  
(1 row)
```

转换成 time with time zone，如下所示：

```
mydb=> SELECT now()::  
time with time zone;  
now  
-----  
09:45:57.13139+08  
(1 row)
```

interval 指时间间隔，时间间隔单位可以是 hour、day、month、year 等，举例如下：

```
mydb=> SELECT now(),now()+interval'1 day';  
now | ?column?  
-----+  
2017-07-29 09:47:26.026418+08 | 2017-07-30 09:47:26.026418+08  
(1 row)
```

通过以上几个示例读者应该对时间、日期数据类型有了初步的了解，值得一提的是时间类型中的 (p) 是指时间精度，具体指秒后面小数点保留的位数，如果没声明精度默认值为 6，以下示例声明精度为 0：

```
mydb=> SELECT now(), now()::timestamp(0);  
now | now  
-----+  
2017-07-29 09:59:42.688445+08 | 2017-07-29 09:59:43  
(1 row)
```

### 3.3.2 时间 / 日期类型操作符

时间、日期数据类型支持的操作符有加、减、乘、除，下面举例说明。



日期相加，如下所示：

```
mydb=> SELECT date '2017-07-29' + interval'1 days';
?column?
-----
2017-07-30 00:00:00
(1 row)
```

日期相减，如下所示：

```
mydb=> SELECT date '2017-07-29' - interval'1 hour';
?column?
-----
2017-07-28 23:00:00
(1 row)
```

日期相乘，如下所示：

```
mydb=> SELECT 100* interval '1 second';
?column?
-----
00:01:40
(1 row)
```

日期相除，如下所示：

```
mydb=> SELECT interval '1 hour' / double precision '3';
?column?
-----
00:20:00
(1 row)
```

### 3.3.3 时间 / 日期类型常用函数

接下来演示时间、日期常用函数。

显示当前时间，如下所示：

```
mydb=> SELECT current_date, current_time;
current_date | current_time
-----
2017-07-29 | 10:53:10.375374+08
(1 row)
```

另一个非常重要的函数为 EXTRACT 函数，可以从日期、时间数据类型中抽取年、月、日、时、分、秒信息，语法如下所示：

```
EXTRACT(field FROM source)
```

field 值可以为 century、year、month、day、hour、minute、second 等，source 类型为 timestamp、time、interval 的值的表达式，例如取年份，代码如下所示：

```
mydb=> SELECT EXTRACT( year FROM now());
```



```
date_part
```

```
-----  
2017  
(1 row)
```

对于 timestamp 类型，取月份和月份里的第几天，代码如下所示：

```
mydb=> SELECT EXTRACT( month FROM now()),EXTRACT(day FROM now());  
date_part | date_part  
-----+-----  
7 | 29  
(1 row)
```

取小时、分钟，如下所示：

```
mydb=> SELECT EXTRACT( hour FROM now()), extract (minute FROM now());  
date_part | date_part  
-----+-----  
11 | 14  
(1 row)
```

取秒，如下所示：

```
mydb=> SELECT EXTRACT( second FROM now());  
date_part  
-----  
43.031366  
(1 row)
```

取当前日期所在年份中的第几周，如下所示：

```
mydb=> SELECT EXTRACT( week FROM now());  
date_part  
-----  
30  
(1 row)
```

当天属于当年的第几天，如下所示：

```
mydb=> SELECT EXTRACT( doy FROM now());  
date_part  
-----  
210  
(1 row)
```

## 3.4 布尔类型

前三小节介绍了 PostgreSQL 支持的数字类型、字符类型、时间日期类型，这些数据类型是关系型数据库的常规数据类型，此外 PostgreSQL 还支持很多非常规数据类型，比如布尔类型、网络地址类型、数组类型、范围类型、json/jsonb 类型等，从这一节开始将介绍 PostgreSQL 支持的非常规数据类型，本节介绍布尔类型，PostgreSQL 支持的布尔类



型如表 3-4 所示。

表 3-4 布尔数据类型

字符类型名称	存储长度	描述
boolean	1 字节	状态为 true 或 false

true 状态的有效值可以是 TRUE、t、true、y、yes、on、1；false 状态的有效值为 FALSE、f、false、n、no、off、0，首先创建一张表来进行演示，如下所示：

```
mydb=> CREATE TABLE test_boolean(cola boolean,colb boolean);
CREATE TABLE
mydb=> INSERT INTO test_boolean (cola,colb) VALUES ('true','false');
INSERT 0 1
mydb=> INSERT INTO test_boolean (cola,colb) VALUES ('t','f');
INSERT 0 1
mydb=> INSERT INTO test_boolean (cola,colb) VALUES ('TRUE','FALSE');
INSERT 0 1
mydb=> INSERT INTO test_boolean (cola,colb) VALUES ('yes','no');
INSERT 0 1
mydb=> INSERT INTO test_boolean (cola,colb) VALUES ('y','n');
INSERT 0 1
mydb=> INSERT INTO test_boolean (cola,colb) VALUES ('1','0');
INSERT 0 1
mydb=> INSERT INTO test_boolean (cola,colb) VALUES (null,null);
INSERT 0 1
```

查询表 test\_boolean 数据，尽管有多样的 true、false 状态输入值，查询表布尔类型字段时 true 状态显示为 t，false 状态显示为 f，并且可以插入 NULL 字符，查询结果如下所示：

```
mydb=> SELECT * FROM test_boolean ;
      cola | colb
-----+-----
      t    | f
      t    | f
      t    | f
      t    | f
      t    | f
      t    | f
      |
(7 rows)
```

## 3.5 网络地址类型

当有存储 IP 地址需求的业务场景时，对于 PostgreSQL 并不很熟悉的开发者可能会使用字符类型存储，实际上 PostgreSQL 提供用于存储 IPv4、IPv6、MAC 网络地址的专有网络地址数据类型，使用网络地址数据类型存储 IP 地址要优于字符类型，因为网络地址类型



一方面会对数据合法性进行检查，另一方面也提供了网络数据类型操作符和函数方便应用程序开发。

### 3.5.1 网络地址类型列表

PostgreSQL 支持的网络地址数据类型如表 3-5 所示。

表 3-5 网络地址数据类型列表

字符类型名称	存 储 长 度	描 述
cidr	7 或 19 字节	IPv4 和 IPv6 网络
inet	7 或 19 字节	IPv4 和 IPv6 网络
macaddr	6 字节	MAC 地址
macaddr8	8 字节	MAC 地址 (EUI-64 格式)

inet 和 cidr 类型存储的网络地址格式为 address/y，其中 address 表示 IPv4 或 IPv6 网络地址，y 表示网络掩码位数，如果 y 省略，则对于 IPv4 网络掩码为 32，对于 IPv6 网络掩码为 128，所以该值表示一台主机。

inet 和 cidr 类型都会对数据合法性进行检查，如果数据不合法会报错，如下所示：

```
mydb=> SELECT '192.168.2.1000'::inet;
ERROR: invalid input syntax for type inet: "192.168.2.1000"
LINE 1: select '192.168.2.1000'::inet;
```

inet 和 cidr 网络类型存在以下差别。

1) cidr 类型的输出默认带子网掩码信息，而 inet 不一定，如下所示：

```
mydb=> SELECT '192.168.1.100'::cidr;
          cidr
-----
          192.168.1.100/32
(1 row)

mydb=> SELECT '192.168.1.100/32'::inet;
          inet
-----
          192.168.1.100
(1 row)

mydb=> SELECT '192.168.0.0/16'::inet;
          inet
-----
          192.168.0.0/16
(1 row)
```

2) cidr 类型对 IP 地址和子网掩码合法性进行检查，而 inet 不会，如下所示：



```
mydb=> SELECT '192.168.2.0/8'::cidr;
ERROR: invalid cidr value: "192.168.2.0/8"
LINE 1: select '192.168.2.0/8'::cidr;
DETAILED: Value has bits set to right of mask.

mydb=> SELECT '192.168.2.0/8'::inet;
inet
-----
192.168.2.0/8
(1 row)

mydb=> SELECT '192.168.2.0/24'::cidr;
cidr
-----
192.168.2.0/24
(1 row)
```

因此，从这个层面来说 cidr 比 inet 网络类型更严谨。macaddr 和 macaddr8 存储 MAC 地址，这里不做介绍。

### 3.5.2 网络地址操作符

PostgreSQL 支持丰富的网络地址数据类型操作符，如表 3-6 所示。

表 3-6 网络地址数据类型操作符

操作符	描述	举例
<	小于	inet '192.168.1.5' < inet '192.168.1.6'
<=	小于等于	inet '192.168.1.5' <= inet '192.168.1.5'
=	等于	inet '192.168.1.5' = inet '192.168.1.5'
>=	大于等于	inet '192.168.1.5' >= inet '192.168.1.5'
>	大于	inet '192.168.1.5' > inet '192.168.1.4'
<>	不等于	inet '192.168.1.5' <> inet '192.168.1.4'
<<	被包含	inet '192.168.1.5' << inet '192.168.1/24'
<<=	被包含或等于	inet '192.168.1/24' <=> inet '192.168.1/24'
>>	包含	inet '192.168.1/24' >> inet '192.168.1/24'
>>=	包含或等于	inet '192.168.1/24' >>= inet '192.168.1/24'
&&	包含或被包含	inet '192.168.1/24' && inet '192.168.1.80/28'
~	按位取反	~ inet '192.168.1.6'
&	按位与	inet '192.168.1.6' & inet '0.0.0.255'
	按位或	inet '192.168.1.6'   inet '0.0.0.255'
+	加	inet '192.168.1.6' + 25
-	减	inet '192.168.1.43' - 36
-	减	inet '192.168.1.43' - inet '192.168.1.19'

### 3.5.3 网络地址函数

PostgreSQL 网络地址类型支持一系列内置函数，下面举例说明。

取 IP 地址，返回文本格式，如下所示：

```
mydb=> SELECT host(cidr '192.168.1.0/24');
          host
-----
        192.168.1.0
(1 row)
```

取 IP 地址和网络掩码，返回文本格式，如下所示：

```
mydb=> SELECT text(cidr '192.168.1.0/24');
          text
-----
        192.168.1.0/24
(1 row)
```

取网络地址子网掩码，返回文本格式，如下所示：

```
mydb=> SELECT netmask(cidr '192.168.1.0/24');
          netmask
-----
        255.255.255.0
```

## 3.6 数组类型

PostgreSQL 支持一维数组和多维数组，常用的数组类型为数字类型数组和字符型数组，也支持枚举类型、复合类型数组。

### 3.6.1 数组类型定义

先来看看数组类型的定义，创建表时在字段数据类型后面加方括号“[]”即可定义数组数据类型，如下所示：

```
CREATE TABLE test_array1 (
    id         integer,
    array_i    integer[],
    array_t    text[]
);
```

以上 integer[] 表示 integer 类型一维数组，text[] 表示 text 类型一维数组。

### 3.6.2 数组类型值输入

数组类型的插入有两种方式，第一种方式使用花括号方式，如下所示：

```
'{ val1 delim val2 delim ... }'
```

将数组元素值用花括号“{}”包围并用 delim 分隔符分开，数组元素值可以用双引号引用，delim 分隔符通常为逗号，如下所示：

```
mydb=> SELECT '{1,2,3}';
?column?
-----
{1,2,3}
(1 row)
```

往表 test\_array1 中插入一条记录的代码如下所示：

```
mydb=> INSERT INTO test_array1(id,array_i,array_t)
VALUES (1,'{1,2,3}', '{"a","b","c"}');
INSERT 0 1
```

数组类型插入的第二种方式为使用 ARRAY 关键字，例如：

```
mydb=> SELECT array[1,2,3];
array
-----
{1,2,3}
(1 row)
```

往 test\_array2 表中插入另一条记录，代码如下所示：

```
mydb=> INSERT INTO test_array1(id,array_i,array_t)
VALUES (2,ARRAY[4,5,6],ARRAY['d','e','f']);
INSERT 0 1
```

表 test\_array2 的数据如下所示：

```
mydb=> SELECT * FROM test_array1;
id | array_i | array_t
-----+-----+-----+
1 | {1,2,3} | {a,b,c}
2 | {4,5,6} | {d,e,f}
(2 rows)
```

### 3.6.3 查询数组元素

如果想查询数组所有元素值，只需查询数组字段名称即可，如下所示：

```
mydb=> SELECT array_i FROM test_array1 WHERE id=1;
array_i
-----
{1,2,3}
(1 row)
```

数组元素的引用通过方括号“[]”方式，数据下标写在方括号内，编号范围为 1 到 n，n 为数组长度，如下所示：

```
mydb=> SELECT array_i[1],array_t[3] FROM test_array1 WHERE id=1;
```

```
array_i | array_t
-----+-----
 1 | c
(1 row)
```

### 3.6.4 数组元素的追加、删除、更新

PostgreSQL 数组类型支持数组元素的追加、删除与更新操作，数组元素的追加使用 `array_append` 函数，用法如下所示：

```
array_append(anyarray, anyelement)
array_append 函数向数组末端追加一个元素，如下所示：
```

```
mydb=> SELECT array_append(array[1,2,3],4);
          array_append
-----+
{1,2,3,4}
(1 row)
```

数据元素追加到数组也可以使用操作符 `||`，如下所示：

```
mydb=> SELECT array[1,2,3] || 4;
          ?column?
-----+
{1,2,3,4}
(1 row)
```

数组元素的删除使用 `array_remove` 函数，`array_remove` 函数用法如下所示：

```
array_remove(anyarray, anyelement)
```

`array_remove` 函数将移除数组中值等于给定值的所有数组元素，如下所示：

```
mydb=> SELECT array[1,2,2,3],array_remove(array[1,2,2,3],2);
          array      | array_remove
-----+-----
{1,2,2,3}  | {1,3}
(1 row)
```

数组元素的修改代码如下所示：

```
mydb=> UPDATE test_array1 SET array_i[3]=4 WHERE id=1 ;
UPDATE 1
```

整个数组也能被更新，如下所示：

```
mydb=> UPDATE test_array1 SET array_i=array[7,8,9] WHERE id=1;
UPDATE 1
```

### 3.6.5 数组操作符

PostgreSQL 数组元素支持丰富操作符，如表 3-7 所示。

表 3-7 数组操作符

操作符	描述	举例	结果
=	等于	ARRAY[1,1,2,1,3,1]::int[] = ARRAY[1,2,3]	t
<>	不等于	ARRAY[1,2,3] <> ARRAY[1,2,4]	t
<	小于	ARRAY[1,2,3] < ARRAY[1,2,4]	t
>	大于	ARRAY[1,4,3] > ARRAY[1,2,4]	t
<=	小于等于	ARRAY[1,2,3] <= ARRAY[1,2,3]	t
>=	大于等于	ARRAY[1,4,3] >= ARRAY[1,4,3]	t
@>	包含	ARRAY[1,4,3] @> ARRAY[3,1]	t
<@	被包含	ARRAY[2,7] <@ ARRAY[1,7,4,2,6]	t
&&	重叠 (具有公共元素)	ARRAY[1,4,3] && ARRAY[2,1]	t
	数组和数组串接	ARRAY[1,2,3]    ARRAY[4,5,6]	{1,2,3,4,5,6}
	数组和数组串接	ARRAY[1,2,3]    ARRAY[[4,5,6],[7,8,9]]	{ {1,2,3}, {4,5,6}, {7,8,9} }
	元素和数组串接	3    ARRAY[4,5,6]	{3,4,5,6}
	数组和元素串接	ARRAY[4,5,6]    7	{4,5,6,7}

### 3.6.6 数组函数

PostgreSQL 支持丰富的数组函数，给数组添加元素或删除元素，如下所示：

```
mydb=> SELECT array_append(array[1,2],3),array_remove(array[1,2],2);
          array_append | array_remove
-----+-----
      {1,2,3}      | {1}
(1 row)
```

获取数组维度，如下所示：

```
mydb=> SELECT array_ndims(array[1,2]);
          array_ndims
-----
      1
(1 row)
```

获取数组长度，如下所示：

```
mydb=> SELECT array_length(array[1,2],1);
          array_length
-----
      2
(1 row)
```

返回数组中某个数组元素第一次出现的位置，如下所示：

```
mydb=> SELECT array_position(array['a','b','c','d'],'d');
          array_position
```

```
4
(1 row)
```

数组元素替换可使用函数 array\_replace，语法如下：

```
array_replace(anyarray, anyelement, anyelement)
```

函数返回值类型为 anyarray，使用第二个 anyelement 替换数组中的相同数组元素，如下所示：

```
mydb=> SELECT array_replace(array[1,2,5,4],5,10);
          array_replace
-----
{1,2,10,4}
(1 row)
```

将数组元素输出到字符串，可以使用 array\_to\_string 函数，语法如下：

```
array_to_string(anyarray, text [, text])
```

函数返回值类型为 text，第一个 text 参数指分隔符，第二个 text 表示将值为 NULL 的元素使用这个字符串替换，示例如下：

```
mydb=> SELECT array_to_string(array[1,2,null,3],',','10');
          array_to_string
-----
1,2,10,3
(1 row)
```

## 3.7 范围类型

范围类型包含一个范围内的数据，常见的范围数据类型有日期范围类型、整数范围类型等；范围类型提供丰富的操作符和函数，对于日期安排、价格范围应用场景比较适用。

### 3.7.1 范围类型列表

PostgreSQL 系统提供内置的范围类型如下：

- int4range——integer 范围类型
- int8range——bigint 范围类型
- numrange——numeric 范围类型
- tsrange——不带时区的 timestamp 范围类型
- tstzrange——带时区的 timestamp 范围类型
- daterange——date 范围类型

用户也可以通过 CREATE TYPE 命令自定义范围数据类型，integer 范围类型举例如下：

```
mydb=> SELECT int4range(1,5);
```



```

int4range
-----
[1,5)
(1 row)

mydb=> SELECT daterange('2017-07-01','2017-07-30');
daterange
-----
[2017-07-01,2017-07-30)

```

### 3.7.2 范围类型边界

每一个范围类型都包含下界和上界，方括号“[”表示包含下界，圆括号“(”表示排除下界，方括号“]”表示包含上界，圆括号“)”表示排除上界，也就是说方括号表示边界点包含在内，圆括号表示边界点不包含在内，范围类型值的输入有以下几种模式：

```

(lower-bound,upper-bound)
(lower-bound,upper-bound]
[lower-bound,upper-bound)
[lower-bound,upper-bound]
empty

```

注意 empty 表示空范围类型，不包含任何元素，看下面这个例子：

```

mydb=> SELECT int4range(4,7);
int4range
-----
[4,7)
(1 row)

```

以上表示包含 4、5、6，但不包含 7，标准的范围类型为下界包含同时上界排除，如下所示：

```

mydb=> SELECT int4range(1,3);
int4range
-----
[1,3)
(1 row)

```

以上没有指定数据类型边界模式，指定上界为“]”，如下所示：

```

mydb=> SELECT int4range(1,3,['']);
int4range
-----
[1,4)
(1 row)

```

虽然指定上界“]”，但上界依然显示为“)”，这是范围类型标准的边界模式，即下界包含同时上界排除，这点需要注意。



### 3.7.3 范围类型操作符

本节介绍常见的范围类型操作符。

包含元素操作符，如下所示：

```
mydb=> SELECT int4range(4,7) @> 4;
?column?
-----
t
(1 row)
```

包含范围操作符，如下所示：

```
mydb=> SELECT int4range(4,7) @>int4range(4,6);
?column?
-----
t
(1 row)
```

等于操作符，如下所示：

```
mydb=> SELECT int4range(4,7)=int4range(4,6,['']);
?column?
-----
t
(1 row)
```

其中“@>”操作符在范围数据类型中比较常用，常用来查询范围数据类型是否包含某个指定元素，由于篇幅关系，其他范围数据类型操作符这里不演示了。

### 3.7.4 范围类型函数

以下列举范围类型常用函数，例如，取范围下界，如下所示：

```
mydb=> SELECT lower(int4range(1,10));
lower
-----
1
(1 row)
```

取范围上界，如下所示：

```
mydb=> SELECT upper(int4range(1,10));
upper
-----
10
(1 row)
```

范围是否为空？示例如下：

```
mydb=> SELECT isempty(int4range(1,10));
isempty
-----

```



```
f
(1 row)
```

### 3.7.5 给范围类型创建索引

范围类型数据支持创建 GiST 索引，GiST 索引支持的操作符有“=”“&&”“<@”“@>”“<<”“>>”“-|”“&<”“&>”等，GiST 索引创建举例如下：

```
CREATE INDEX idx_ip_address_range ON ip_address USING gist (ip_range);
```

## 3.8 json/jsonb 类型

PostgreSQL 不只是一个关系型数据库，同时它还支持非关系数据类型 json (JavaScript Object Notation)，json 属于重量级的非常规数据类型，本节将介绍 json 类型、json 与 jsonb 差异、json 与 jsonb 操作符和函数，以及 jsonb 键值的追加、删除、更新。

### 3.8.1 json 类型简介

PostgreSQL 早在 9.2 版本已经提供了 json 类型，并且随着大版本的演进，PostgreSQL 对 json 的支持趋于完善，例如提供更多的 json 函数和操作符方便应用开发，一个简单的 json 类型例子如下：

```
mydb=> SELECT '{"a":1,"b":2}':>json;
          json
-----
 {"a":1,"b":2}
```

为了更好地演示 json 类型，接下来创建一张表，如下所示：

```
mydb=> CREATE TABLE test_json1 (id serial primary key, name json);
CREATE TABLE
```

以上示例定义字段 name 为 json 类型，插入表数据，如下所示：

```
mydb=> INSERT INTO test_json1 (name)
VALUES ('{"col1":1,"col2":"francs","col3":"male"}');
INSERT 0 1
```

```
mydb=> INSERT INTO test_json1 (name)
VALUES ('{"col1":2,"col2":"fp","col3":"female"}');
INSERT 0 1
```

查询表 test\_json1 数据，如下所示：

```
mydb=> SELECT * FROM test_json1;
      id   |           name
-----+-----
      1   | {"col1":1,"col2":"francs","col3":"male"}
      2   | {"col1":2,"col2":"fp","col3":"female"}
```



### 3.8.2 查询 json 数据

通过“->”操作符可以查询 json 数据的键值，如下所示：

```
mydb=> SELECT name -> 'col2' FROM test_json1 WHERE id=1;
?column?
-----
"francs"
(1 row)
```

如果想以文本格式返回 json 字段键值可以使用“->>”操作符，如下所示：

```
mydb=> SELECT name ->> 'col2' FROM test_json1 WHERE id=1;
?column?
-----
francs
(1 row)
```

### 3.8.3 jsonb 与 json 差异

PostgreSQL 支持两种 JSON 数据类型：json 和 jsonb，两种类型在使用上几乎完全相同，两者主要区别为以下：json 存储格式为文本而 jsonb 存储格式为二进制，由于存储格式的不同使得两种 json 数据类型的处理效率不一样，json 类型以文本存储并且存储的内容和输入数据一样，当检索 json 数据时必须重新解析，而 jsonb 以二进制形式存储已解析好的数据，当检索 jsonb 数据时不需要重新解析，因此 json 写入比 jsonb 快，但检索比 jsonb 慢，后面会通过测试验证两者读写性能的差异。

除了上述介绍的区别之外，json 与 jsonb 在使用过程中还存在差异，例如 jsonb 输出的键的顺序和输入不一样，如下所示：

```
mydb=> SELECT '{"bar": "baz", "balance": 7.77, "active":false}'::jsonb;
          jsonb
-----
 {"bar": "baz", "active": false, "balance": 7.77}
(1 row)
```

而 json 的输出键的顺序和输入完全一样，如下所示：

```
mydb=> SELECT '{"bar": "baz", "balance": 7.77, "active":false}'::json;
          json
-----
 {"bar": "baz", "balance": 7.77, "active":false}
(1 row)
```

另外，jsonb 类型会去掉输入数据中键值的空格，如下所示：

```
mydb=> SELECT '{"id":1, "name":"francs"}'::jsonb;
          jsonb
-----
 {"id": 1, "name": "francs"}
(1 row)
```



上例中 id 键与 name 键输入时是有空格的，输出显示空格键被删除，而 json 的输出和输入一样，不会删掉空格键：

```
mydb=> SELECT ' {"id":1,      "name":"francs"}'::json;
          json
-----
          {"id":1,      "name":"francs"}
(1 row)
```

另外，jsonb 会删除重复的键，仅保留最后一个，如下所示：

```
mydb=> SELECT ' {"id":1,
  "name":"francs",
  "remark":"a good guy!",
  "name":"test"
}'::jsonb;
          jsonb
-----
          {"id": 1, "name": "test", "remark": "a good guy!"}
(1 row)
```

上面 name 键重复，仅保留最后一个 name 键的值，而 json 数据类型会保留重复的键值。

在大多数应用场景下建议使用 jsonb，除非有特殊的需求，比如对 json 的键顺序有特殊的要求。

### 3.8.4 jsonb 与 json 操作符

以文本格式返回 json 类型的字段键值可以使用 “->>” 操作符，如下所示：

```
mydb=> SELECT name ->> 'col2' FROM test_json1 WHERE id=1;
          ?column?
-----
          frans
(1 row)
```

字符串是否作为顶层键值，如下所示：

```
mydb=> SELECT '{"a":1, "b":2}'::jsonb ? 'a';
          ?column?
-----
          t
(1 row)
```

删除 json 数据的键 / 值，如下所示：

```
mydb=> SELECT '{"a":1, "b":2}'::jsonb - 'a';
          ?column?
-----
          {"b": 2}
(1 row)
```



### 3.8.5 jsonb 与 json 函数

json 与 jsonb 相关的函数非常丰富，下面举例说明。

扩展最外层的 json 对象成为一组键 / 值结果集，如下所示：

```
mydb=> SELECT * FROM json_each('{"a":"foo", "b":"bar"}');
      key | value
-----+-----
      a   | "foo"
      b   | "bar"
(2 rows)
```

以文本形式返回结果，如下所示：

```
mydb=> SELECT * FROM json_each_text('{"a":"foo", "b":"bar"}');
      key | value
-----+-----
      a   | foo
      b   | bar
(2 rows)
```

一个非常重要的函数为 row\_to\_json() 函数，能够将行作为 json 对象返回，此函数常用来生成 json 测试数据，比如将一个普通表转换成 json 类型表，代码如下所示：

```
mydb=> SELECT * FROM test_copy WHERE id=1;
      id | name
-----+-----
      1  | a
(1 row)

mydb=> SELECT row_to_json(test_copy) FROM test_copy WHERE id=1;
      row_to_json
-----
      {"id":1,"name":"a"}
(1 row)
```

返回最外层的 json 对象中的键的集合，如下所示：

```
mydb=> SELECT * FROM json_object_keys('{"a":"foo", "b":"bar"}');
      json_object_keys
-----
      a
      b
(2 rows)
```

### 3.8.6 jsonb 键 / 值的追加、删除、更新

jsonb 键 / 值追加可通过 “||” 操作符，例如增加 sex 键 / 值，如下所示：

```
mydb=> SELECT '{"name":"francs", "age":"31"}'::jsonb ||
      '{"sex":"male"}'::jsonb;
```



```
?column?
-----
{"age": "31", "sex": "male", "name": "francs"}
(1 row)
```

jsonb 键 / 值的删除有两种方法，一种是通过操作符 “-” 删除，另一种通过操作符 “#-” 删除指定键 / 值，通过操作符 “-” 删除键 / 值的代码如下所示：

```
mydb=> SELECT '{"name": "James", "email": "james@localhost"}'::jsonb
      - 'email';
      ?column?
-----
 {"name": "James"}
(1 row)

mydb=> SELECT '['"red","green","blue"]'::jsonb - 0;
      ?column?
-----
 ["green", "blue"]
```

第二种方法是通过操作符 “#-” 删除指定键 / 值，通常用于有嵌套 json 数据删除的场景，如下代码删除嵌套 contact 中的 fax 键 / 值：

```
mydb=> SELECT '{"name": "James", "contact": {"phone": "01234 567890", "fax": "01987 543210"}}'::jsonb #- '{contact,fax}'::text[];
      ?column?
-----
 {"name": "James", "contact": {"phone": "01234 567890"}}
(1 row)
```

删除嵌套 aliases 中的位置为 1 的键 / 值，如下所示：

```
mydb=> SELECT '{"name": "James", "aliases": ["Jamie", "The Jamester", "J Man"]}'::jsonb
      #- '{aliases,1}'::text[];
      ?column?
-----
 {"name": "James", "aliases": ["Jamie", "J Man"]}
(1 row)
```

键 / 值的更新也有两种方式，第一种方式为 “||” 操作符，“||” 操作符可以连接 json 键，也可覆盖重复的键值，如下代码修改 age 键的值：

```
mydb=> SELECT '{"name": "francs", "age": "31"}'::jsonb || '{"age": "32"}'::jsonb;
      ?column?
-----
 {"age": "32", "name": "francs"}
(1 row)
```

第二种方式是通过 jsonb\_set 函数，语法如下：

```
jsonb_set(target jsonb, path text[], new_value jsonb[], create_missing boolean)
```





target 指源 jsonb 数据, path 指路径, new\_value 指更新后的键值, create\_missing 值为 true 表示如果键不存在则添加, create\_missing 值为 false 表示如果键不存在则不添加, 示例如下:

```
mydb=> SELECT jsonb_set('{"name":"francs","age":31})::jsonb,'{age}','"32"'::jsonb,false);
          jsonb_set
-----
  {"age": "32", "name": "francs"}
(1 row)

mydb=> SELECT jsonb_set('{"name":"francs","age":31})::jsonb,'{sex}','"male"'::jsonb,true);
          jsonb_set
-----
  {"age": "31", "sex": "male", "name": "francs"}
(1 row)
```

## 3.9 数据类型转换

前面几小节介绍了 PostgreSQL 常规数据类型和非常规数据类型, 这一小节将介绍数据类型转换, PostgreSQL 数据类型转换主要有三种方式: 通过格式化函数、CAST 函数、:: 操作符, 下面分别介绍。

### 3.9.1 通过格式化函数进行转换

PostgreSQL 提供一系列函数用于数据类型转换, 如表 3-8 所示。

表 3-8 数据类型转换函数

函 数	返 回 类 型	描 述	示 例
to_char(timestamp, text)	text	把时间戳转换成字符串	to_char(current_timestamp, 'HH12:MI:SS')
to_char(interval, text)	text	把间隔转换成字符串	to_char(interval '15h 2m 12s', 'HH24:MI:SS')
to_char(int, text)	text	把整数转换成字符串	to_char(125, '999')
to_char(numeric, text)	text	把数字转换成字符串	to_char(-125.8, '999D99S')
to_date(text, text)	date	把字符串转换成日期	to_date('05 Dec 2000', 'DD Mon YYYY')
to_number(text, text)	numeric	把字符串转换成数字	to_number('12,454.8', '99G999D9S')
to_timestamp(text, text)	timestamp with time zone	把字符串转换成时间戳	to_timestamp('05 Dec 2000', 'DD Mon YYYY')

### 3.9.2 通过 CAST 函数进行转换

将 varchar 字符类型转换成 text 类型, 如下所示:

```
mydb=> SELECT CAST(varchar'123' as text);
          text
-----
  123
```



```
mydb=> SELECT CAST(varchar'123' as int4);
          int4
-----
```

将 varchar 字符类型转换成 int4 类型，如下所示：

```
mydb=> SELECT CAST(varchar'123' as int4);
          int4
-----
```

123

### 3.9.3 通过 :: 操作符进行转换

以下例子转换成 int4 或 numeric 类型，如下所示：

```
mydb=> SELECT 1::int4, 3/2::numeric;
          int4 |      ?column?
-----+-----
          1 | 1.500000000000000
(1 row)
```

另一个例子，通过 SQL 查询给定表的字段名称，先根据表名在系统表 pg\_class 找到表的 OID，其中 OID 为隐藏的系统字段：

```
mydb=> SELECT oid,relname FROM pg_class WHERE relname='test_json1';
          oid |    relname
-----+-----
        16509 | test_json1
(1 row)
```

之后根据 test\_json1 表的 OID，在系统表 pg\_attribute 中根据 attrelid（即表的 OID）找到表的字段，如下所示：

```
mydb=> SELECT attname FROM pg_attribute WHERE attrelid='16509' AND attnum >0;
          attname
-----+
          id
          name
(2 rows)
```

上述操作需通过两步完成，但通过类型转换可一步到位，如下所示：

```
mydb=> SELECT attname
           FROM pg_attribute
           WHERE attrelid='test_json1'::regclass AND attnum >0;
          attname
-----+
          id
          name
(2 rows)
```

这节介绍了三种数据类型转换方法，第一种方法兼容性相对较好，第三种方法用法简捷。



 提示 pg\_class 系统表存储 PostgreSQL 对象信息，比如表、索引、序列、视图等，OID 是隐藏字段，唯一标识 pg\_class 中的一行，可以理解成 pg\_class 系统表的对象 ID；pg\_attribute 系统表存储表的字段信息，数据库表的每一个字段在这个视图中都有相应一条记录，pg\_attribute.attrelid 是指字段所属表的 OID，正好和 pgclass.oid 关联。

## 3.10 本章小结

本章介绍了 PostgreSQL 常规数据类型和非常规数据类型，常规数据类型如数字类型、字符类型、时间 / 日期类型，非常规数据类型如布尔类型、网络地址类型、数组类型、范围类型、json/jsonb 类型，同时介绍了数据类型相关函数、操作符，最后介绍数据类型转换的几种方法。此外 PostgreSQL 还支持 XML 类型、复合类型、对象标识类型等，由于篇幅关系，本书不做介绍，有兴趣读者可参考手册 <https://www.postgresql.org/docs/10/static/datatype.html>，了解 PostgreSQL 数据类型对于开发人员和 DBA 都非常重要。