POSTGRESQL 中文学习手册

2012 年 10 月 10 日整理

本手册由 WEVVV 制作

目　　录

PostgreSQL 学习手册(数据表)

一、表的定义：

对于任何一种关系型数据库而言，表都是数据存储的最核心、最基础的对象单元。现在就让我们从这里起步吧。

1. 创建表：

```
CREATE TABLE products (
    product_no integer,
    name text,
    price numeric
);
```

2. 删除表：

```
DROP TABLE products;
```

3. 创建带有缺省值的表：

```
CREATE TABLE products (
    product_no integer,
    name text,
    price numeric DEFAULT 9.99 --DEFAULT 是关键字，其后的数值 9.99 是字段 price 的默认值。
);

CREATE TABLE products (
    product_no SERIAL,          --SERIAL 类型的字段表示该字段为自增字段，完全等同于 Oracle 中的 Sequence。
    name text,
    price numeric DEFAULT 9.99
);
```

输出为：

NOTICE: CREATE TABLE will create implicit sequence "products_product_no_seq" for serial column "products.product_no"

4. 约束：

检查约束是表中最为常见的约束类型，它允许你声明在某个字段里的数值必须满足一个布尔表达式。不仅如此，我们也可以声明表级别的检查约束。

```
CREATE TABLE products (
    product_no integer,
    name text,
    --price 字段的值必须大于 0，否则在插入或修改该字段值是，将引发违规错误。还需要说明的是，该检查约束
    --是匿名约束，即在表定义时没有显示命名该约束，这样 PostgreSQL 将会根据当前的表名、字段名和约束类型，
    --为该约束自动命名，如：products_price_check。
    price numeric CHECK (price > 0)
);

CREATE TABLE products (
    product_no integer,
    name text,
    --该字段的检查约束被显示命名为 positive_price。这样做好处在于今后维护该约束时，可以根据该名进行直接操作。
```

```
    price numeric CONSTRAINT positive_price CHECK (price > 0)
);
```

下面的约束是非空约束，即约束的字段不能插入空值，或者是将已有数据更新为空值。

```
CREATE TABLE products (
    product_no integer NOT NULL,
    name text NOT NULL,
    price numeric
);
```

如果一个字段中存在多个约束，在定义时可以不用考虑约束的声明顺序。

```
CREATE TABLE products (
    product_no integer NOT NULL,
    name text NOT NULL,
    price numeric NOT NULL CHECK (price > 0)
);
```

唯一性约束，即指定的字段不能插入重复值，或者是将某一记录的值更新为当前表中的已有值。

```
CREATE TABLE products (
    product_no integer UNIQUE,
    name text,
    price numeric
);
```

```
CREATE TABLE products (
    product_no integer,
    name text,
    price numeric,
    UNIQUE (product_no)
);
```

为表中的多个字段定义联合唯一性。

```
CREATE TABLE example (
    a integer,
    b integer,
    c integer,
    UNIQUE (a, c)
);
```

为唯一性约束命名。

```
CREATE TABLE products (
    product_no integer CONSTRAINT must_be_different UNIQUE,
    name text,
    price numeric
);
```

在插入数据时，空值(NULL)之间被视为不相等的数据，因此对于某一唯一性字段，可以多次插入空值。然而需要注意的是，这一规则并不是被所有数据库都遵守，因此在进行数据库移植时可能会造成一定的麻烦。

5. 主键和外键：

从技术上来讲，主键约束只是唯一约束和非空约束的组合。

```
CREATE TABLE products (
    product_no integer PRIMARY KEY,  --字段 product_no 被定义为该表的唯一主键。
    name text,
    price numeric
```

```
);
```

和唯一性约束一样，主键可以同时作用于多个字段，形成联合主键：

```
CREATE TABLE example (
    a integer,
    b integer,
    c integer,
    PRIMARY KEY (b, c)
);
```

外键约束声明一个字段（或者一组字段）的数值必须匹配另外一个表中某些行出现的数值。 我们把这个行为称做两个相关表之间的参考完整性。

```
CREATE TABLE orders (
    order_id integer PRIMARY KEY, --该表也可以有自己的主键。
    --该表的product_no 字段为上面products 表主键(product_no)的外键。
    product_no integer REFERENCES products(product_no),
    quantity integer
);

CREATE TABLE t1 (
    a integer PRIMARY KEY,
    b integer,
    c integer,
    --该外键的字段数量和被引用表中主键的数量必须保持一致。
    FOREIGN KEY (b, c) REFERENCES example (b, c)
);
```

当多个表之间存在了主外键的参考性约束关系时，若想删除对应用表(主键表)中的某行记录，由于该行记录的主键字段值可能正在被其引用表(外键表)中某条记录所关联，此次删除操作将会失败。如果想完成此操作，一个显而易见的方法是先删除引用表中和该记录关联的行，之后再删除被引用表中的该行记录。然而需要说明的是，PostgreSQL 为我们提供了更为方便的方式完成此类操作。

```
CREATE TABLE products (
    product_no integer PRIMARY KEY,
    name text,
    price numeric
);

CREATE TABLE orders (
    order_id integer PRIMARY KEY,
    shipping_address text
);

CREATE TABLE order_items (
    product_no integer REFERENCES products ON DELETE RESTRICT, --限制选项
    order_id integer REFERENCES orders ON DELETE CASCADE, --级联删除选项
    quantity integer,
    PRIMARY KEY (product_no, order_id)
);
```

限制和级联删除是两种最常见的选项。RESTRICT 禁止删除被引用的行。 NO ACTION 的意思是如果在检查约束的时候，如果还存在任何引用行，则抛出错误； 如果你不声明任何东西，那么它就是缺省的行为。(这两个选择的实际区别是，NO ACTION 允许约束检查推迟到事务的晚些时候，而 RESTRICT 不行。) CASCADE 声明在删除一个被引用的行的时候，引用它的行也会被自动删除掉。 在外键字段上的动作还有两个选项： SET NULL 和 SET DEFAULT。 这样会导致在被引用行删除的时候，引用它们的字段分别设置为空或者缺省值。 请注意这些选项并不能让你逃脱被观察和约束的境地。比如，如果一个动作声明 SET DEFAULT，但是

缺省值并不能满足外键，那么动作就会失败。类似 ON DELETE，还有 ON UPDATE 选项，它是在被引用字段修改(更新)的时候调用的。可用的动作是一样的。

二、系统字段：

PostgreSQL 的每个数据表中都包含几个隐含定义的系统字段。因此，这些名字不能用于用户定义的字段名。这些系统字段的功能有些类似于 Oracle 中的 rownum 和 rowid 等。

oid: 行的对象标识符(对象 ID)。这个字段只有在创建表的时候使用了 WITH OIDS，或者是设置了配置参数 default_with_oids 时出现。这个字段的类型是 oid(和字段同名)。

tableoid: 包含本行的表的 OID。这个字段对那些从继承层次中选取的查询特别有用，因为如果没有它的话，我们就很难说明一行来自哪个独立的表。tableoid 可以和 pg_class 的 oid 字段连接起来获取表名字。

xmin: 插入该行版本的事务的标识(事务 ID)。

cmin: 在插入事务内部的命令标识(从零开始)。

xmax: 删除事务的标识(事务 ID)，如果不是被删除的行版本，那么是零。

cmax: 在删除事务内部的命令标识符，或者是零。

ctid: 一个行版本在它所处的表内的物理位置。请注意，尽管 ctid 可以用于非常快速地定位行版本，但每次 VACUUM FULL 之后，一个行的 ctid 都会被更新或者移动。因此 ctid 是不能作为长期的行标识符的。

OID 是 32 位的量，是在同一个集群内通用的计数器上赋值的。对于一个大型或者长时间使用的数据库，这个计数器是有可能重叠的。因此，假设 OID 是唯一的是非常错误的，除非你自己采取了措施来保证它们是唯一的。如果你需要标识表中的行，我们强烈建议使用序列号生成器。

三、表的修改：

1. 增加字段：

```
ALTER TABLE products ADD COLUMN description text;
```

新增的字段对于表中已经存在的行而言最初将先填充所给出的缺省值(如果你没有声明 DEFAULT 子句，那么缺省是空值)。

在新增字段时，可以同时给该字段指定约束。

```
ALTER TABLE products ADD COLUMN description text CHECK(description <> '');
```

2. 删除字段：

```
ALTER TABLE products DROP COLUMN description;
```

如果该表为被引用表，该字段为被引用字段，那么上面的删除操作将会失败。如果要想在删除被引用字段的同时级联的删除其所有引用字段，可以采用下面的语法形式。

```
ALTER TABLE products DROP COLUMN description CASCADE;
```

3. 增加约束：

```
ALTER TABLE products ADD CHECK(name <> '');  --增加一个表级约束
ALTER TABLE products ADD CONSTRAINT some_name UNIQUE(product_no);--增加命名的唯一性约束。
ALTER TABLE products ADD FOREIGN KEY(pdt_grp_id) REFERENCES pdt_grps; --增加外键约束。
ALTER TABLE products ALTER COLUMN product_no SET NOT NULL; --增加一个非空约束。
```

4. 删除约束：

```
ALTER TABLE products DROP CONSTRAINT some_name;
```

对于显示命名的约束，可以根据其名称直接删除，对于隐式自动命名的约束，可以通过 psql 的\d tablename 来获取该约束的名字。和删除字段一样，如果你想删除有着被依赖关系地约束，你需要用 CASCADE。一个例子是某个外键约束依赖被引用字段上的唯一约束或者主键约束。如：

```
MyTest=# \d products
```

```
      Table "public.products"
   Column     |  Type   | Modifiers
------------+---------+-----------
 product_no | integer |
 name       | text    |
 price      | numeric |
 Check constraints:
    "positive_price" CHECK (price > 0::numeric)
```

和其他约束不同的是，非空约束没有名字，因此只能通过下面的方式删除：

```
ALTER TABLE products ALTER COLUMN product_no DROP NOT NULL;
```

5. 改变字段的缺省值：

在为已有字段添加缺省值时，不会影响任何表中现有的数据行， 它只是为将来 INSERT 命令改变缺省值。

```
ALTER TABLE products ALTER COLUMN price SET DEFAULT 7.77;
```

下面为删除缺省值：

```
ALTER TABLE products ALTER COLUMN price DROP DEFAULT
```

6. 修改字段的数据类型：

只有在字段里现有的每个项都可以用一个隐含的类型转换转换成新的类型时才可能成功。比如当前的数据都是整型，而转换的目标类型为 numeric 或 varchar，这样的转换一般都可以成功。与此同时，PostgreSQL 还将试图把字段的缺省值（如果存在）转换成新的类型， 还有涉及该字段的任何约束。但是这些转换可能失败，或者可能生成奇怪的结果。 在修改某字段类型之前，你最好删除那些约束，然后再把自己手工修改过的添加上去。

```
ALTER TABLE products ALTER COLUMN price TYPE numeric(10,2);
```

7. 修改字段名：

```
ALTER TABLE products RENAME COLUMN product_no TO product_number;
```

8. 修改表名：

```
ALTER TABLE products RENAME TO items;
```

四、权限：

只有表的所有者才能修改或者删除表的权限。要赋予一个权限，我们使用 GRANT 命令，要撤销一个权限，使用 REVOKE 命令。

需要指出的是，PUBLIC 是特殊"用户"可以用于将权限赋予系统中的每一个用户。在声明权限的位置写 ALL 则将所有的与该对象类型相关的权限都赋予出去。

```
GRANT UPDATE ON table_name TO user;  --将表的更新权限赋予指定的 user。
GRANT SELECT ON table_name TO GROUP group; --将表的 select 权限赋予指定的组。
REVOKE ALL ON table_name FROM PUBLIC; --将表的所有权限从 Public 撤销。
```

最初，只有对象所有者(或者超级用户)可以赋予或者撤销对象的权限。但是，我们可以赋予一个**"with grant option"**权限，这样就给接受权限的人以授予该权限给其它人的权限。如果授予选项后来被撤销，那么所有那些从这个接受者接受了权限的用户(直接或者通过级连的授权)都将失去该权限。

PostgreSQL 学习手册(模式 Schema)

一个数据库包含一个或多个命名的模式，模式又包含表。模式还包含其它命名的对象，包括数据类型、函数，以及操作符。同一个对象名可以在不同的模式里使用而不会导致冲突； 比如，schema1 和 myschema 都可以包含叫做 mytable 的表。和数据库不同，模式不是严格分离的：一个用户可以访问他所连接的数据库中的任意模式中的对象，只要他有权限。

我们需要模式有以下几个主要原因：

1). 允许多个用户使用一个数据库而不会干扰其它用户。

2). 把数据库对象组织成逻辑组，让它们更便于管理。

3). 第三方的应用可以放在不同的模式中，这样它们就不会和其它对象的名字冲突。

1. 创建模式：

```
CREATE SCHEMA myschema;
```

通过以上命令可以创建名字为 myschema 的模式，在该模式被创建后，其便可拥有自己的一组逻辑对象，如表、视图和函数等。

2. public 模式：

在介绍后面的内容之前，这里我们需要先解释一下 public 模式。每当我们创建一个新的数据库时，PostgreSQL 都会为我们自动创建该模式。当登录到该数据库时，如果没有特殊的指定，我们将以该模式(public)的形式操作各种数据对象，如：

```
CREATE TABLE products ( ... ) 等同于 CREATE TABLE public.products ( ... )
```

3. 权限：

缺省时，用户看不到模式中不属于他们所有的对象。为了让他们看得见，模式的所有者需要在模式上赋予 USAGE 权限。为了让用户使用模式中的对象，我们可能需要赋予额外的权限，只要是适合该对象的。PostgreSQL 根据不同的对象提供了不同的权限类型，如：

```
GRANT ALL ON SCHEMA myschema TO public;
```

上面的 ALL 关键字将包含 **CREATE** 和 **USAGE** 两种权限。如果 public 模式拥有了 myschema 模式的 CREATE 权限，那么登录到该模式的用户将可以在 myschema 模式中创建任意对象，如：

```
CREATE TABLE myschema.products (
    product_no integer,
    name text,
    price numeric CHECK (price > 0),
);
```

在为模式下的所有表赋予权限时，需要将权限拆分为各种不同的表操作，如：

```
ALTER DEFAULT PRIVILEGES IN SCHEMA myschema
GRANT INSERT, SELECT, UPDATE, DELETE, TRUNCATE, REFERENCES, TRIGGER ON TABLES TO public;
```

在为模式下的所有 Sequence 序列对象赋予权限时，需要将权限拆分为各种不同的 Sequence 操作，如：

```
ALTER DEFAULT PRIVILEGES IN SCHEMA myschema
GRANT SELECT, UPDATE, USAGE ON SEQUENCES TO public;
```

在为模式下的所有函数赋予权限时，仅考虑执行权限，如：

```
ALTER DEFAULT PRIVILEGES IN SCHEMA myschema
GRANT EXECUTE ON FUNCTIONS TO public;
```

可以看出，通过以上方式在 public 模式下为 myschema 模式创建各种对象是极为不方便的。下面我们将要介绍另外一种方式，即通过 role 对象，直接登录并关联到 myschema 对象，之后便可以在 myschema 模式下直接创建各种所需的对象了。

```
CREATE ROLE myschema LOGIN PASSWORD '123456'; --创建了和该模式关联的角色对象。
CREATE SCHEMA myschema AUTHORIZATION myschema; --将该模式关联到指定的角色，模式名和角色名可以不相等。
```

在 Linux Shell 下，以 myschema 的角色登录到数据库 MyTest，在密码输入正确后将成功登录到该数据库。

```
/> psql -d MyTest -U myschema
Password:
```

```
MyTest=> CREATE TABLE test(i integer);
CREATE TABLE
MyTest=> \d   --查看该模式下，以及该模式有权限看到的tables 信息列表。
          List of relations
 Schema    |   Name   | Type  |  Owner
-----------+---------+------+----------
 myschema |   test     | table  | myschema
(1 rows)
```

4. 删除模式：

```
DROP SCHEMA myschema;
```

如果要删除模式及其所有对象，请使用级联删除：

```
DROP SCHEMA myschema CASCADE;
```

5. 模式搜索路径：

我们在使用一个数据库对象时可以使用它的全称来定位对象，然而这样做往往也是非常繁琐的，每次都不得不键入owner_name.object_name。PostgreSQL 中提供了模式搜索路径，这有些类似于 Linux 中的$PATH 环境变量，当我们执行一个 Shell 命令时，只有该命令位于$PATH 的目录列表中，我们才可以通过命令名直接执行，否则就需要输入它的全路径名。PostgreSQL 同样也通过查找一个搜索路径来判断一个表究竟是哪个表，这个路径是一个需要查找的模式列表。在搜索路径里找到的第一个表将被当作选定的表。如果在搜索路径中 没有匹配表，那么就报告一个错误，即使匹配表的名字在数据库其它的模式中存在也如此。

在搜索路径中的第一个模式叫做当前模式。除了是搜索的第一个模式之外，它还是在 CREATE TABLE 没有声明模式名的时候，新建表所属于的模式。要显示当前搜索路径，使用下面的命令：

```
MyTest=> SHOW search_path;
  search_path
----------------
 "$user",public
(1 row)
```

可以将新模式加入到搜索路径中，如：

```
SET search_path TO myschema,public;
```

为搜索路径设置指定的模式，如：

```
SET search_path TO myschema; --当前搜索路径中将只是包含myschema 一种模式。
```


PostgreSQL 学习手册(表的继承和分区)

一、表的继承：

这个概念对于很多已经熟悉其他数据库编程的开发人员而言会多少有些陌生，然而它的实现方式和设计原理却是简单易懂，现在就让我们从一个简单的例子开始吧。

1. 第一个继承表：

```
CREATE TABLE cities (   --父表
    name        text,
    population float,
    altitude     int
);
CREATE TABLE capitals ( --子表
    state      char(2)
```

) **INHERITS** (cities);

capitals 表继承自 cities 表的所有属性。在 PostgreSQL 里，一个表可以从零个或多个其它表中继承属性，而且一个查询既可以引用父表中的所有行，也可以引用父表的所有行加上其所有子表的行，其中后者是缺省行为。

```
MyTest=# INSERT INTO cities values('Las Vegas', 1.53, 2174);  --插入父表
INSERT 0 1
MyTest=# INSERT INTO cities values('Mariposa',3.30,1953);     --插入父表
INSERT 0 1
MyTest=# INSERT INTO capitals values('Madison',4.34,845,'WI');--插入子表
INSERT 0 1
MyTest=# SELECT name, altitude FROM cities WHERE altitude > 500; --父表和子表的数据均被取出。
   name     | altitude
-----------+----------
 Las Vegas |     2174
 Mariposa  |     1953
 Madison   |      845
(3 rows)

MyTest=# SELECT name, altitude FROM capitals WHERE altitude > 500; --只有子表的数据被取出。
  name    | altitude
---------+----------
 Madison |      845
(1 row)
```


如果希望只从父表中提取数据，则需要在 SQL 中加入 ONLY 关键字，如：

```
MyTest=# SELECT name,altitude FROM ONLY cities WHERE altitude > 500;
   name     | altitude
-----------+----------
 Las Vegas |     2174
 Mariposa  |     1953
(2 rows)
```

上例中 cities 前面的"ONLY"关键字表示该查询应该只对 cities 进行查找而不包括继承级别低于 cities 的表。许多我们已经讨论过的命令--SELECT，UPDATE 和 DELETE--支持这个"ONLY"符号。

在执行整表数据删除时，如果直接 truncate 父表，此时父表和其所有子表的数据均被删除，如果只是 truncate 子表，那么其父表的数据将不会变化，只是子表中的数据被清空。

```
MyTest=# TRUNCATE TABLE cities;  --父表和子表的数据均被删除。
TRUNCATE TABLE
MyTest=# SELECT * FROM capitals;
 name | population | altitude | state
------+------------+----------+-------
(0 rows)
```

2. 确定数据来源：

有时候你可能想知道某条记录来自哪个表。在每个表里我们都有一个系统隐含字段 **tableoid**，它可以告诉你表的来源：

```
MyTest=# SELECT tableoid, name, altitude FROM cities WHERE altitude > 500;
 tableoid |   name    | altitude
----------+-----------+----------
    16532 | Las Vegas |     2174
    16532 | Mariposa  |     1953
    16538 | Madison   |      845
```

(3 rows)

以上的结果只是给出了 **tableoid**，仅仅通过该值，我们还是无法看出实际的表名。要完成此操作，我们就需要和系统表 **pg_class** 进行关联，以通过 tableoid 字段从该表中提取实际的表名，见以下查询：

```
MyTest=# SELECT p.relname, c.name, c.altitude FROM cities c,pg_class p WHERE c.altitude > 500 and c.tableoid = p.oid;
 relname  |    name     | altitude
----------+-------------+----------
 cities   | Las Vegas   |     2174
 cities   | Mariposa    |     1953
 capitals | Madison     |      845
(3 rows)
```

3. 数据插入的注意事项：

继承并不自动从 INSERT 或者 COPY 中向继承级别中的其它表填充数据。在我们的例子里，下面的 INSERT 语句不会成功：

```
INSERT INTO cities (name, population, altitude, state) VALUES ('New York', NULL, NULL, 'NY');
```

我们可能希望数据被传递到 capitals 表里面去，但是这是不会发生的：INSERT 总是插入明确声明的那个表。

4. 多表继承：

一个表可以从多个父表继承，这种情况下它拥有父表们的字段的总和。子表中任意定义的字段也会加入其中。如果同一个字段名出现在多个父表中，或者同时出现在父表和子表的定义里，那么这些字段就会被"融合"，这样在子表里面就只有一个这样的字段。要想融合，字段必须是相同的数据类型，否则就会抛出一个错误。融合的字段将会拥有它所继承的字段的所有约束。

```
CREATE TABLE parent1 (FirstCol integer);
CREATE TABLE parent2 (FirstCol integer, SecondCol varchar(20));
CREATE TABLE parent3 (FirstCol varchar(200));
--子表 child1 将同时继承自 parent1 和 parent2 表，而这两个父表中均包含 integer 类型的 FirstCol 字段，因此 child1 可以创建成功。
CREATE TABLE child1 (MyCol timestamp) INHERITS (parent1,parent2);
--子表 child2 将不会创建成功，因为其两个父表中均包含 FirstCol 字段，但是它们的类型不相同。
CREATE TABLE child2 (MyCol timestamp) INHERITS (parent1,parent3);
--子表 child3 同样不会创建成功，因为它和其父表均包含 FirstCol 字段，但是它们的类型不相同。
CREATE TABLE child3 (FirstCol varchar(20)) INHERITS(parent1);
```

5. 继承和权限：

表访问权限并不会自动继承。因此，一个试图访问父表的用户还必须具有访问它的所有子表的权限，或者使用 ONLY 关键字只从父表中提取数据。在向现有的继承层次添加新的子表的时候，请注意给它赋予所有权限。

继承特性的一个严重的局限性是索引(包括唯一约束)和外键约束只施用于单个表，而不包括它们的继承的子表。这一点不管对引用表还是被引用表都是事实，因此在上面的例子里，如果我们声明 cities.name 为 UNIQUE 或者是一个 PRIMARY KEY，那么也不会阻止 capitals 表拥有重复了名字的 cities 数据行。 并且这些重复的行缺省时在查询 cities 表的时候会显示出来。实际上，缺省时 capitals 将完全没有唯一约束，因此可能包含带有同名的多个行。你应该给 capitals 增加唯一约束，但是这样做也不会避免与 cities 的重复。类似，如果我们声明 cities.name REFERENCES 某些其它的表，这个约束不会自动广播到 capitals。在这种条件下，你可以通过手工给 capitals 增加同样的 REFERENCES 约束来做到这点。

二、分区表：

1. 概述分区表：

分区的意思是把逻辑上的一个大表分割成物理上的几块儿，分区可以提供若干好处：

1). 某些类型的查询性能可以得到极大提升。

2). 更新的性能也可以得到提升，因为表的每块的索引要比在整个数据集上的索引要小。如果索引不能全部放在内存里，那么在索

引上的读和写都会产生更多的磁盘访问。

3). 批量删除可以用简单地删除某个分区来实现。

4). 将很少用的数据可以移动到便宜的、慢一些地存储介质上。

假设当前的数据库并不支持分区表，而我们的应用所需处理的数据量也非常大，对于这种应用场景，我们不得不人为的将该大表按照一定的规则，手工拆分成多个小表，让每个小表包含不同区间的数据。这样一来，我们就必须在数据插入、更新、删除和查询之前，先计算本次的指令需要操作的小表。对于有些查询而言，由于查询区间可能会跨越多个小表，这样我们又不得不将多个小表的查询结果进行 union 操作，以合并来自多个表的数据，并最终形成一个结果集返回给客户端。可见，如果我们正在使用的数据库不支持分区表，那么在适合其应用的场景下，我们就需要做很多额外的编程工作以弥补这一缺失。然而需要说明的是，尽管功能可以勉强应付，但是性能却和分区表无法相提并论。

目前 PostgreSQL 支持的分区形式主要为以下两种：

1). 范围分区：表被一个或者多个键字字段分区成"范围"，在这些范围之间没有重叠的数值分布到不同的分区里。比如，我们可以为特定的商业对象根据数据范围分区，或者根据标识符范围分区。

2). 列表分区：表是通过明确地列出每个分区里应该出现那些键字值实现的。

2. 实现分区：

1). 创建"主表"，所有分区都从它继承。

```
CREATE TABLE measurement (            --主表
    city_id       int     NOT NULL,
    logdate       date    NOT NULL,
    peaktemp int,
);
```

2). 创建几个"子"表，每个都从主表上继承。通常，这些"子"表将不会再增加任何字段。我们将把子表称作分区，尽管它们就是普通的 PostgreSQL 表。

```
CREATE TABLE measurement_yy04mm02 ( ) INHERITS (measurement);
CREATE TABLE measurement_yy04mm03 ( ) INHERITS (measurement);
...
CREATE TABLE measurement_yy05mm11 ( ) INHERITS (measurement);
CREATE TABLE measurement_yy05mm12 ( ) INHERITS (measurement);
CREATE TABLE measurement_yy06mm01 ( ) INHERITS (measurement);
```

上面创建的子表，均已年、月的形式进行范围划分，不同年月的数据将归属到不同的子表内。这样的实现方式对于清空分区数据而言将极为方便和高效，即直接执行 DROP TABLE 语句删除相应的子表，之后在根据实际的应用考虑是否重建该子表(分区)。相比于直接 DROP 子表，PostgreSQL 还提供了另外一种更为方便的方式来管理子表：

```
ALTER TABLE measurement_yy06mm01 NO INHERIT measurement;
```

和直接 DROP 相比，该方式仅仅是使子表脱离了原有的主表，而存储在子表中的数据仍然可以得到访问，因为此时该表已经被还原成一个普通的数据表了。这样对于数据库的 DBA 来说，就可以在此时对该表进行必要的维护操作，如数据清理、归档等，在完成诸多例行性的操作之后，就可以考虑是直接删除该表(DROP TABLE)，还是先清空该表的数据(TRUNCATE TABLE)，之后再让该表重新继承主表，如：

```
ALTER TABLE measurement_yy06mm01 INHERIT measurement;
```

3). 给分区表增加约束，定义每个分区允许的健值。同时需要注意的是，定义的约束要确保在不同的分区里不会有相同的键值。因此，我们需要将上面"子"表的定义修改为以下形式：

```
CREATE TABLE measurement_yy04mm02 (
    CHECK ( logdate >= DATE '2004-02-01' AND logdate < DATE '2004-03-01')
) INHERITS (measurement);
CREATE TABLE measurement_yy04mm03 (
    CHECK (logdate >= DATE '2004-03-01' AND logdate < DATE '2004-04-01')
) INHERITS (measurement);
...
CREATE TABLE measurement_yy05mm11 (
    CHECK (logdate >= DATE '2005-11-01' AND logdate < DATE '2005-12-01')
```

```
) INHERITS (measurement);
CREATE TABLE measurement_yy05mm12 (
    CHECK (logdate >= DATE '2005-12-01' AND logdate < DATE '2006-01-01')
) INHERITS (measurement);
CREATE TABLE measurement_yy06mm01 (
    CHECK (logdate >= DATE '2006-01-01' AND logdate < DATE '2006-02-01')
) INHERITS (measurement);
```

4). 尽可能基于键值创建索引。如果需要，我们也同样可以为子表中的其它字段创建索引。

```
CREATE INDEX measurement_yy04mm02_logdate ON measurement_yy04mm02 (logdate);
CREATE INDEX measurement_yy04mm03_logdate ON measurement_yy04mm03 (logdate);
...
CREATE INDEX measurement_yy05mm11_logdate ON measurement_yy05mm11 (logdate);
CREATE INDEX measurement_yy05mm12_logdate ON measurement_yy05mm12 (logdate);
CREATE INDEX measurement_yy06mm01_logdate ON measurement_yy06mm01 (logdate);
```

5). 定义一个规则或者触发器，把对主表的修改重定向到适当的分区表。

如果数据只进入最新的分区，我们可以设置一个非常简单的规则来插入数据。我们必须每个月都重新定义这个规则，即修改重定向插入的子表名，这样它总是指向当前分区。

```
CREATE OR REPLACE RULE measurement_current_partition AS
ON INSERT TO measurement
DO INSTEAD
INSERT INTO measurement_yy06mm01 VALUES (NEW.city_id, NEW.logdate, NEW.peaktemp);
```

其中 NEW 是关键字，表示新数据字段的集合。这里可以通过点(.)操作符来获取集合中的每一个字段。

我们可能想插入数据并且想让服务器自动定位应该向哪个分区插入数据。我们可以用像下面这样的更复杂的规则集来实现这个目标。

```
CREATE RULE measurement_insert_yy04mm02 AS
ON INSERT TO measurement WHERE (logdate >= DATE '2004-02-01' AND logdate < DATE '2004-03-01')
DO INSTEAD
INSERT INTO measurement_yy04mm02 VALUES (NEW.city_id, NEW.logdate, NEW.peaktemp);
...
CREATE RULE measurement_insert_yy05mm12 AS
ON INSERT TO measurement WHERE (logdate >= DATE '2005-12-01' AND logdate < DATE '2006-01-01')
DO INSTEAD
INSERT INTO measurement_yy05mm12 VALUES (NEW.city_id, NEW.logdate, NEW.peaktemp);
CREATE RULE measurement_insert_yy06mm01 AS
ON INSERT TO measurement WHERE (logdate >= DATE '2006-01-01' AND logdate < DATE '2006-02-01')
DO INSTEAD
INSERT INTO measurement_yy06mm01 VALUES (NEW.city_id, NEW.logdate, NEW.peaktemp);
```

请注意每个规则里面的 WHERE 子句正好匹配其分区的 CHECK 约束。

可以看出，一个复杂的分区方案可能要求相当多的 DDL。在上面的例子里我们需要每个月创建一次新分区，因此写一个脚本自动生成需要的 DDL 是明智的。除此之外，我们还不难推断出，分区表对于新数据的批量插入操作有一定的抑制，这一点在 Oracle 中也同样如此。

除了上面介绍的通过 Rule 的方式重定向主表的数据到各个子表，我们还可以通过触发器的方式来完成此操作，相比于基于 Rule 的重定向方法，基于触发器的方式可能会带来更好的插入效率，特别是针对非批量插入的情况。然而对于批量插入而言，由于 Rule 的额外开销是基于表的，而不是基于行的，因此效果会好于触发器方式。另一个需要注意的是，copy 操作将会忽略 Rules，如果我们想要通过 COPY 方法来插入数据，你只能将数据直接 copy 到正确的子表，而不是主表。这种限制对于触发器来说是不会造成任何问题的。基于 Rule 的重定向方式还存在另外一个问题，就是当插入的数据不在任何子表的约束中时，PostgreSQL 也不会报错，而是将数据直接保留在主表中。

6). 添加新分区：

这里将介绍两种添加新分区的方式，第一种方法简单且直观，我们只是创建新的子表，同时为其定义新的检查约束，如：

```
CREATE TABLE measurement_y2008m02 (
```

```
    CHECK ( logdate >= DATE '2008-02-01' AND logdate < DATE '2008-03-01' )
) INHERITS (measurement);
```

第二种方法的创建步骤相对繁琐，但更为灵活和实用。见以下四步：

```
/* 创建一个独立的数据表(measurement_y2008m02)，该表在创建时以将来的主表(measurement)为模板，包含模板表的缺省值(DEFAULTS)和一致性约束(CONSTRAINTS)。*/
CREATE TABLE measurement_y2008m02
    (LIKE measurement INCLUDING DEFAULTS INCLUDING CONSTRAINTS);
/* 为该表创建未来作为子表时需要使用的检查约束。*/
ALTER TABLE measurement_y2008m02 ADD CONSTRAINT y2008m02
    CHECK (logdate >= DATE '2008-02-01' AND logdate < DATE '2008-03-01');
/* 导入数据到该表。下面只是给出一种导入数据的方式作为例子。在导入数据之后，如有可能，还可以做进一步的数据处理，如数据转换、过滤等。*/
\copy measurement_y2008m02 from 'measurement_y2008m02'
/* 在适当的时候，或者说在需要的时候，让该表继承主表。*/
ALTER TABLE measurement_y2008m02 INHERIT measurement;
```

7). 确保 postgresql.conf 里的配置参数 constraint_exclusion 是打开的。没有这个参数，查询不会按照需要进行优化。这里我们需要做的是确保该选项在配置文件中没有被注释掉。

```
/> pwd
/opt/PostgreSQL/9.1/data
/> cat postgresql.conf | grep "constraint_exclusion"
constraint_exclusion = partition        # on, off, or partition
```

3. 分区和约束排除：

约束排除(Constraint exclusion)是一种查询优化技巧，它改进了用上面方法定义的表分区的性能。比如：

```
SET constraint_exclusion = on;
SELECT count(*) FROM measurement WHERE logdate >= DATE '2006-01-01';
```

如果没有约束排除，上面的查询会扫描 measurement 表中的每一个分区。打开了约束排除之后，规划器将检查每个分区的约束然后再视图证明该分区不需要被扫描，因为它不能包含任何符合 WHERE 子句条件的数据行。如果规划器可以证明这个，它就把该分区从查询规划里排除出去。

你可以使用 EXPLAIN 命令显示一个规划在 constraint_exclusion 打开和关闭情况下的不同。用上面方法设置的表的典型的缺省规划是：

```
SET constraint_exclusion = off;
EXPLAIN SELECT count(*) FROM measurement WHERE logdate >= DATE '2006-01-01';
                                QUERY PLAN
-------------------------------------------------------------------------------------------
 Aggregate  (cost=158.66..158.68 rows=1 width=0)
   ->  Append  (cost=0.00..151.88 rows=2715 width=0)
         ->  Seq Scan on measurement  (cost=0.00..30.38 rows=543 width=0)
               Filter: (logdate >= '2006-01-01'::date)
         ->  Seq Scan on measurement_yy04mm02 measurement  (cost=0.00..30.38 rows=543 width=0)
               Filter: (logdate >= '2006-01-01'::date)
         ->  Seq Scan on measurement_yy04mm03 measurement  (cost=0.00..30.38 rows=543 width=0)
               Filter: (logdate >= '2006-01-01'::date)
...
         ->  Seq Scan on measurement_yy05mm12 measurement  (cost=0.00..30.38 rows=543 width=0)
               Filter: (logdate >= '2006-01-01'::date)
         ->  Seq Scan on measurement_yy06mm01 measurement  (cost=0.00..30.38 rows=543 width=0)
               Filter: (logdate >= '2006-01-01'::date)
```

从上面的查询计划中可以看出，PostgreSQL 扫描了所有分区。下面我们再看一下打开约束排除之后的查询计划：

```
SET constraint_exclusion = on;
EXPLAIN SELECT count(*) FROM measurement WHERE logdate >= DATE '2006-01-01';
                                  QUERY PLAN
-------------------------------------------------------------------------------------------
 Aggregate  (cost=63.47..63.48 rows=1 width=0)
   ->  Append  (cost=0.00..60.75 rows=1086 width=0)
         ->  Seq Scan on measurement  (cost=0.00..30.38 rows=543 width=0)
               Filter: (logdate >= '2006-01-01'::date)
         ->  Seq Scan on measurement_yy06mm01 measurement  (cost=0.00..30.38 rows=543 width=0)
               Filter: (logdate >= '2006-01-01'::date)
```

请注意，约束排除只由 CHECK 约束驱动，而不会由索引驱动。

目前版本的 PostgreSQL 中该配置的缺省值是 **partition**，该值是介于 on 和 off 之间的一种行为方式，即规划器只会将约束排除应用于基于分区表的查询，而 on 设置则会为所有查询都进行约束排除，那么对于普通数据表而言，也将不得不承担由该机制而产生的额外开销。

约束排除在使用时有以下几点注意事项：

1). 约束排除只是在查询的 WHERE 子句包含约束的时候才生效。一个参数化的查询不会被优化，因为在运行时规划器不知道该参数会选择哪个分区。因此像 CURRENT_DATE 这样的函数必须避免。把分区键值和另外一个表的字段连接起来也不会得到优化。

2). 在 CHECK 约束里面要避免跨数据类型的比较，因为目前规划器会无法证明这样的条件为假。比如，下面的约束会在 x 是整数字段的时候可用，但是在 x 是一个 bigint 的时候不能用：

CHECK (x = 1)

对于 bigint 字段，我们必须使用类似下面这样的约束：

CHECK (x = 1::bigint)

这个问题并不仅仅局限于 bigint 数据类型，它可能会发生在任何约束的缺省数据类型与其比较的字段的数据类型不匹配的场合。在提交的查询里的跨数据类型的比较通常是 OK 的，只是不能在 CHECK 条件里。

3). 在主表上的 UPDATE 和 DELETE 命令并不执行约束排除。

4). 在规划器进行约束排除时，主表上的所有分区的所有约束都将会被检查，因此，大量的分区会显著增加查询规划的时间。

5). 在执行 ANALYZE 语句时，要为每一个分区都执行该命令，而不是仅仅对主表执行该命令。

PostgreSQL 学习手册(常用数据类型)

一、数值类型：

下面是 PostgreSQL 所支持的数值类型的列表和简单说明：

名字	存储空间	描述	范围
smallint	2 字节	小范围整数	-32768 到 +32767
integer	4 字节	常用的整数	-2147483648 到 +2147483647
bigint	8 字节	大范围的整数	-9223372036854775808 到 9223372036854775807
decimal	变长	用户声明精度，精确	无限制
numeric	变长	用户声明精度，精确	无限制
real	4 字节	变精度，不精确	6 位十进制数字精度

double	8 字节	变精度，不精确	15 位十进制数字精度
serial	4 字节	自增整数	1 到 +2147483647
bigserial	8 字节	大范围的自增整数	1 到 9223372036854775807

1. 整数类型：

类型 smallint、integer 和 bigint 存储各种范围的全部是数字的数，也就是没有小数部分的数字。试图存储超出范围以外的数值将导致一个错误。常用的类型是 integer，因为它提供了在范围、存储空间和性能之间的最佳平衡。一般只有在磁盘空间紧张的时候才使用 smallint。而只有在 integer 的范围不够的时候才使用 bigint，因为前者(integer)绝对快得多。

2. 任意精度数值：

类型 numeric 可以存储最多 1000 位精度的数字并且准确地进行计算。因此非常适合用于货币金额和其它要求计算准确的数量。不过，numeric 类型上的算术运算比整数类型或者浮点数类型要慢很多。

numeric 字段的最大精度和最大比例都是可以配置的。要声明一个类型为 numeric 的字段，你可以用下面的语法：

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

比如数字 23.5141 的精度为 6，而刻度为 4。

在目前的 PostgreSQL 版本中，**decimal** 和 **numeric** 是等效的。

3. 浮点数类型：

数据类型 real 和 double 是不准确的、牺牲精度的数字类型。不准确意味着一些数值不能准确地转换成内部格式并且是以近似的形式存储的，因此存储后再把数据打印出来可能显示一些缺失。

4. Serial(序号)类型：

serial 和 bigserial 类型不是真正的类型，只是为在表中设置唯一标识做的概念上的便利。

```
CREATE TABLE tablename (
    colname SERIAL
);
```

等价于

```
CREATE SEQUENCE tablename_colname_seq;
CREATE TABLE tablename(
    colname integer DEFAULT nextval('tablename_colname_seq') NOT NULL
);
```

这样，我们就创建了一个整数字段并且把它的缺省数值安排为从一个序列发生器取值。应用了一个 NOT NULL 约束以确保空值不会被插入。在大多数情况下你可能还希望附加一个 UNIQUE 或者 PRIMARY KEY 约束避免意外地插入重复的数值，但这个不是自动发生的。因此，如果你希望一个序列字段有一个唯一约束或者一个主键，那么你现在必须声明，就像其它数据类型一样。

还需要另外说明的是，一个 serial 类型创建的序列在其所属字段被删除时，该序列也将被自动删除，但是其它情况下是不会被删除的。因此，如果你想用同一个序列发生器同时给几个字段提供数据，那么就应该以独立对象的方式创建该序列发生器。

二、字符类型：

下面是 PostgreSQL 所支持的字符类型的列表和简单说明：

名字	描述
varchar(n)	变长，有长度限制
char(n)	定长,不足补空白
text	变长，无长度限制

SQL 定义了两种基本的字符类型，varchar(n)和 char(n)，这里的 n 是一个正整数。两种类型都可以存储最多 n 个字符长的字串，试图存储更长的字串到这些类型的字段里会产生一个错误，除非超出长度的字符都是空白，这种情况下该字串将被截断为最大长度。如果没有长度声明，char 等于 char(1)，而 varchar 则可以接受任何长度的字串。

```
MyTest=> CREATE TABLE testtable(first_col varchar(2));
CREATE TABLE
MyTest=> INSERT INTO testtable VALUES('333');  --插入字符串的长度，超过其字段定义的长度，因此报错。
ERROR:  value too long for type character varying(2)
--插入字符串中，超出字段定义长度的部分是空格，因此可以插入，但是空白符被截断。
MyTest=> INSERT INTO testtable VALUES('33 ');
INSERT 0 1
MyTest=> SELECT * FROM testtable;
 first_col
-----------
 33
(1 row)
```

这里需要注意的是，如果是将数值转换成 char(n)或者 varchar(n)，那么超长的数值将被截断成 n 个字符，而不会抛出错误。

```
MyTest=> select 1234::varchar(2);
 varchar
---------
 12
(1 row)
```

最后需要提示的是，这三种类型之间没有性能差别，只不过是在使用 char 类型时增加了存储尺寸。虽然在某些其它的数据库系统里，char(n)有一定的性能优势，但在 PostgreSQL 里没有。在大多数情况下，应该使用 text 或者 varchar。

三、日期/时间类型：

下面是 PostgreSQL 所支持的日期/时间类型的列表和简单说明：

名字	存储空间	描述	最低值	最高值	分辨率
timestamp[无时区]	8 字节	包括日期和时间	4713 BC	5874897AD	1 毫秒/14 位
timestamp[含时区]	8 字节	日期和时间，带时区	4713 BC	5874897AD	1 毫秒/14 位
interval	12 字节	时间间隔	-178000000 年	178000000 年	1 毫秒/14 位
date	4 字节	只用于日期	4713 BC	32767AD	1 天
time[无时区]	8 字节	只用于一日内时间	00:00:00	24:00:00	1 毫秒/14 位

1．日期/时间输入：

任何日期或者时间的文本输入均需要由单引号包围，就象一个文本字符串一样。

1)．日期:

以下为合法的日期格式列表：

例子	描述
January 8, 1999	在任何 datestyle 输入模式下都无歧义
1999-01-08	ISO-8601 格式，任何方式下都是 1999 年 1 月 8 号，(建议格式)
1/8/1999	歧义，在 MDY 下是 1 月 8 号；在 DMY 模式下读做 8 月 1 日

1/18/1999	在 MDY 模式下读做 1 月 18 日，其它模式下被拒绝
01/02/03	MDY 模式下的 2003 年 1 月 2 日；DMY 模式下的 2003 年 2 月 1 日；YMD 模式下的 2001 年 2 月 3 日
1999-Jan-08	任何模式下都是 1 月 8 日
Jan-08-1999	任何模式下都是 1 月 8 日
08-Jan-1999	任何模式下都是 1 月 8 日
99-Jan-08	在 YMD 模式下是 1 月 8 日，否则错误
08-Jan-99	1 月 8 日，除了在 YMD 模式下是错误的之外
Jan-08-99	1 月 8 日，除了在 YMD 模式下是错误的之外
19990108	ISO-8601；任何模式下都是 1999 年 1 月 8 日
990108	ISO-8601；任何模式下都是 1999 年 1 月 8 日

2). 时间：

以下为合法的时间格式列表：

例子	描述
04:05:06.789	ISO 8601
04:05:06	ISO 8601
04:05	ISO 8601
040506	ISO 8601
04:05 AM	与 04:05 一样；AM 不影响数值
04:05 PM	与 16:05 一样；输入小时数必须 <= 12
04:05:06.789-8	ISO 8601
04:05:06-08:00	ISO 8601
04:05-08:00	ISO 8601
040506-08	ISO 8601

3). 时间戳：

时间戳类型的有效输入由一个日期和时间的联接组成，后面跟着一个可选的时区。因此，1999-01-08 04:05:06 和 1999-01-08 04:05:06 -8:00 都是有效的数值。

2. 示例：

1). 在插入数据之前先查看 datestyle 系统变量的值：

```
MyTest=> show datestyle;
 DateStyle
-----------
 ISO, YMD
(1 row)
```

2). 创建包含日期、时间和时间戳类型的示例表：

```
MyTest=> CREATE TABLE testtable (id integer, date_col date, time_col time, timestamp_col timestamp);
CREATE TABLE
```

3). 插入数据：

```
MyTest=> INSERT INTO testtable(id,date_col) VALUES(1, DATE'01/02/03');  --datestyle 为 YMD
```

```
INSERT 0 1
MyTest=> SELECT id, date_col FROM testtable;
 id   |  date_col
----+------------
  1   | 2001-02-03
(1 row)

MyTest=> set datestyle = MDY;
SET
MyTest=> INSERT INTO testtable(id,date_col) VALUES(2, DATE'01/02/03');  --datestyle 为MDY
INSERT 0 1
MyTest=> SELECT id,date_col FROM testtable;
 id   |  date_col
----+------------
  1   | 2001-02-03
  2   | 2003-01-02

MyTest=> INSERT INTO testtable(id,time_col) VALUES(3, TIME'10:20:00');  --插入时间。
INSERT 0 1
MyTest=> SELECT id,time_col FROM testtable WHERE time_col IS NOT NULL;
 id   | time_col
----+----------
  3   | 10:20:00
(1 row)

MyTest=> INSERT INTO testtable(id,timestamp_col) VALUES(4, DATE'01/02/03');
INSERT 0 1
MyTest=> INSERT INTO testtable(id,timestamp_col) VALUES(5, TIMESTAMP'01/02/03 10:20:00');
INSERT 0 1
MyTest=> SELECT id,timestamp_col FROM testtable WHERE timestamp_col IS NOT NULL;
 id   |    timestamp_col
----+---------------------
  4   | 2003-01-02 00:00:00
  5   | 2003-01-02 10:20:00
(2 rows)
```


四、布尔类型：

PostgreSQL 支持标准的 SQL boolean 数据类型。boolean 只能有两个状态之一：真(True)或 假(False)。该类型占用 1 个字节。

"真"值的有效文本值是：

TRUE
't'
'true'
'y'
'yes'
'1'

而对于"假"而言，你可以使用下面这些：

FALSE
'f'
'false'
'n'
'no'
'0'
见如下使用方式：

```
MyTest=> CREATE TABLE testtable (a boolean, b text);
CREATE TABLE
MyTest=> INSERT INTO testtable VALUES(TRUE, 'sic est');
INSERT 0 1
MyTest=> INSERT INTO testtable VALUES(FALSE, 'non est');
INSERT 0 1
MyTest=> SELECT * FROM testtable;
 a  |    b
----+---------
 t  | sic est
 f  | non est
(2 rows)
MyTest=> SELECT * FROM testtable WHERE a;
 a  |    b
----+---------
 t  | sic est
(1 row)
MyTest=> SELECT * FROM testtable WHERE a = true;
 a  |    b
----+---------
 t  | sic est
(1 row)
```

五、位串类型：

位串就是一串 1 和 0 的字串。它们可以用于存储和视觉化位掩码。我们有两种类型的 SQL 位类型：bit(n)和 bit varying(n)；这里的 n 是一个正整数。bit 类型的数据必须准确匹配长度 n；试图存储短些或者长一些的数据都是错误的。类型 bit varying 数据是最长 n 的变长类型；更长的串会被拒绝。写一个没有长度的 bit 等效于 bit(1)，没有长度的 bit varying 相当于没有长度限制。

针对该类型，最后需要提醒的是，如果我们明确地把一个位串值转换成 bit(n)，那么它的右边将被截断或者在右边补齐零，直到刚好 n 位，而不会抛出任何错误。类似地，如果我们明确地把一个位串数值转换成 bit varying(n)，如果它超过 n 位，那么它的右边将被截断。 见如下具体使用方式：

```
MyTest=> CREATE TABLE testtable (a bit(3), b bit varying(5));
CREATE TABLE
MyTest=> INSERT INTO testtable VALUES (B'101', B'00');
INSERT 0 1
MyTest=> INSERT INTO testtable VALUES (B'10', B'101');
ERROR:  bit string length 2 does not match type bit(3)
MyTest=> INSERT INTO testtable VALUES (B'10'::bit(3), B'101');
INSERT 0 1
MyTest=> SELECT * FROM testtable;
  a  |  b
```

```
-----+-----
 101 | 00
 100 | 101
(2 rows)
MyTest=> SELECT B'11'::bit(3);
 bit
-----
 110
(1 row)
```

六、数组：

1. 数组类型声明：

1). 创建字段含有数组类型的表。

```
CREATE TABLE sal_emp (
    name            text,
    pay_by_quarter  integer[] --还可以定义为integer[4]或integer ARRAY[4]
);
```

2). 插入数组数据：

```
MyTest=# INSERT INTO sal_emp VALUES ('Bill', '{11000, 12000, 13000, 14000}');
INSERT 0 1
MyTest=# INSERT INTO sal_emp VALUES ('Carol', ARRAY[21000, 22000, 23000, 24000]);
INSERT 0 1
MyTest=# SELECT * FROM sal_emp;
 name  |      pay_by_quarter
--------+---------------------------
 Bill   | {11000,12000,13000,14000}
 Carol  | {21000,22000,23000,24000}
(2 rows)
```

2. 访问数组：

和其他语言一样，PostgreSQL 中数组也是通过下标数字(写在方括弧内)的方式进行访问，只是 PostgreSQL 中数组元素的下标是从 1 开始 n 结束。

```
MyTest=# SELECT pay_by_quarter[3] FROM sal_emp;
 pay_by_quarter
----------------
          13000
          23000
(2 rows)
MyTest=# SELECT name FROM sal_emp WHERE pay_by_quarter[1] <> pay_by_quarter[2];
 name
------
 Bill
 Carol
(2 rows)
```

PostgreSQL 中还提供了访问数组范围的功能，即 ARRAY[脚标下界:脚标上界]。

```
MyTest=# SELECT name,pay_by_quarter[1:3] FROM sal_emp;
 name  |    pay_by_quarter
```

```
--------+--------------------
 Bill   | {11000,12000,13000}
 Carol  | {21000,22000,23000}
(2 rows)
```

3. 修改数组：

1). 代替全部数组值：

```
--UPDATE sal_emp SET pay_by_quarter = ARRAY[25000,25000,27000,27000] WHERE name = 'Carol';  也可以。
MyTest=# UPDATE sal_emp SET pay_by_quarter = '{31000,32000,33000,34000}' WHERE name = 'Carol';
UPDATE 1
MyTest=# SELECT * FROM sal_emp;
 name  |      pay_by_quarter
--------+---------------------------
 Bill   | {11000,12000,13000,14000}
 Carol  | {31000,32000,33000,34000}
(2 rows)
```

2). 更新数组中某一元素：

```
MyTest=# UPDATE sal_emp SET pay_by_quarter[4] = 15000 WHERE name = 'Bill';
UPDATE 1
MyTest=# SELECT * FROM sal_emp;
 name  |      pay_by_quarter
--------+---------------------------
 Carol  | {31000,32000,33000,34000}
 Bill   | {11000,12000,13000,15000}
(2 rows)
```

3). 更新数组某一范围的元素：

```
MyTest=# UPDATE sal_emp SET pay_by_quarter[1:2] = '{37000,37000}' WHERE name = 'Carol';
UPDATE 1
MyTest=# SELECT * FROM sal_emp;
 name  |      pay_by_quarter
--------+---------------------------
 Bill   | {11000,12000,13000,15000}
 Carol  | {37000,37000,33000,34000}
(2 rows)
```

4). 直接赋值扩大数组：

```
MyTest=# UPDATE sal_emp SET pay_by_quarter[5] = 45000 WHERE name = 'Bill';
UPDATE 1
MyTest=# SELECT * FROM sal_emp;
 name  |        pay_by_quarter
--------+-------------------------------
 Carol  | {37000,37000,33000,34000}
 Bill   | {11000,12000,13000,15000,45000}
(2 rows)
```

4. 在数组中检索：

1). 最简单直接的方法：

```
SELECT * FROM sal_emp WHERE pay_by_quarter[1] = 10000 OR
                            pay_by_quarter[2] = 10000 OR
                            pay_by_quarter[3] = 10000 OR
```

```
                pay_by_quarter[4] = 10000;
```

2). 更加有效的方法：

```
SELECT * FROM sal_emp WHERE 10000 = ANY (pay_by_quarter); --数组元素中有任何一个等于10000，where 条件将成立。
SELECT * FROM sal_emp WHERE 10000 = ALL (pay_by_quarter); --只有当数组中所有的元素都等于10000 时，where 条件才成立。
```

七、复合类型：

PostgreSQL 中复合类型有些类似于 C 语言中的结构体，也可以被视为 Oracle 中的记录类型，但是还是感觉复合类型这个命名比较贴切。它实际上只是一个字段名和它们的数据类型的列表。PostgreSQL 允许像简单数据类型那样使用复合类型。比如，表字段可以声明为一个复合类型。

1. 声明复合类型：

下面是两个简单的声明示例：

```
CREATE TYPE complex AS (
    r double,
    i double
);
CREATE TYPE inventory_item AS (
    name            text,
    supplier_id     integer,
    price           numeric
);
```

和声明一个数据表相比，声明类型时需要加 AS 关键字，同时在声明 TYPE 时不能定义任何约束。下面我们看一下如何在表中指定复合类型的字段，如：

```
CREATE TABLE on_hand (
    item      inventory_item,
    count     integer
);
```

最后需要指出的是，在创建表的时候，PostgreSQL 也会自动创建一个与该表对应的复合类型，名字与表字相同，即表示该表的复合类型。

2. 复合类型值输入：

我们可以使用文本常量的方式表示复合类型值，即在圆括号里包围字段值并且用逗号分隔它们。你也可以将任何字段值用双引号括起，如果值本身包含逗号或者圆括号，那么就用双引号括起，对于上面的 inventory_item 复合类型的输入如下：

```
'("fuzzy dice",42,1.99)'
```

如果希望类型中的某个字段为 NULL，只需在其对应的位置不予输入即可，如下面的输入中 price 字段的值为 NULL，

```
'("fuzzy dice",42,)'
```

如果只是需要一个空字串，而非 NULL，写一对双引号，如：

```
'("",42,)'
```

在更多的场合中 PostgreSQL 推荐使用 ROW 表达式来构建复合类型值，使用该种方式相对简单，无需考虑更多标识字符问题，如：

```
ROW('fuzzy dice', 42, 1.99)
ROW('', 42, NULL)
```

注：对于 ROW 表达式，如果里面的字段数量超过 1 个，那么关键字 ROW 就可以省略，因此以上形式可以简化为：

```
('fuzzy dice', 42, 1.99)
('', 42, NULL)
```

3. 访问复合类型：

访问复合类型中的字段和访问数据表中的字段在形式上极为相似，只是为了对二者加以区分，PostgreSQL 设定在访问复合类型中的字段时，类型部分需要用圆括号括起，以避免混淆，如：

```
SELECT (item).name FROM on_hand WHERE (item).price > 9.99;
```

如果在查询中也需要用到表名，那么表名和类型名都需要被圆括号括起，如：

```
SELECT (on_hand.item).name FROM on_hand WHERE (on_hand.item).price > 9.99;
```

4. 修改复合类型：

见如下几个示例：

```
--直接插入复合类型的数据，这里是通过 ROW 表达式来完成的。
INSERT INTO on_hand(item) VALUES(ROW("fuzzy dice",42,1.99));
--在更新操作中，也是可以通过 ROW 表达式来完成。
UPDATE on_hand SET item = ROW("fuzzy dice",42,1.99) WHERE count = 0;
--在更新复合类型中的一个字段时，我们不能在 SET 后面出现的字段名周围加圆括号，
--但是在等号右边的表达式里引用同一个字段时却需要圆括号。
UPDATE on_hand SET item.price = (item).price + 1 WHERE count = 0;
--可以在插入中，直接插入复合类型中字段。
INSERT INTO on_hand (item.supplier_id, item.price) VALUES(100, 2.2);
```

PostgreSQL 学习手册(函数和操作符<一>)

一、逻辑操作符：

常用的逻辑操作符有：**AND**、**OR** 和 **NOT**。其语义与其它编程语言中的逻辑操作符完全相同。

二、比较操作符：

下面是 PostgreSQL 中提供的比较操作符列表：

操作符	描述
<	小于
>	大于
<=	小于或等于
>=	大于或等于
=	等于
!=	不等于

比较操作符可以用于所有可以比较的数据类型。所有比较操作符都是双目操作符，且返回 boolean 类型。除了比较操作符以外，我们还可以使用 BETWEEN 语句，如：

a **BETWEEN** x **AND** y 等效于 a >= x **AND** a <= y

a **NOT BETWEEN** x **AND** y 等效于 a < x **OR** a > y

三、 数学函数和操作符：

下面是 PostgreSQL 中提供的数学操作符列表：

操作符	描述	例子	结果
+	加	2 + 3	5
-	减	2 - 3	-1
*	乘	2 * 3	6
/	除	4 / 2	2
%	模	5 % 4	1
^	幂	2.0 ^ 3.0	8
|/	平方根	|/ 25.0	5
||/	立方根	||/ 27.0	3
!	阶乘	5 !	120
!!	阶乘	!! 5	120
@	绝对值	@ -5.0	5
&	按位 AND	91 & 15	11
|	按位 OR	32 | 3	35
#	按位 XOR	17 # 5	20
~	按位 NOT	~1	-2
<<	按位左移	1 << 4	16
>>	按位右移	8 >> 2	2

按位操作符只能用于整数类型，而其它的操作符可以用于全部数值数据类型。按位操作符还可以用于位串类型 bit 和 bit varying，

下面是 PostgreSQL 中提供的数学函数列表，需要说明的是，这些函数中有许多都存在多种形式，区别只是参数类型不同。除非特别指明，任何特定形式的函数都返回和它的参数相同的数据类型。

函数	返回类型	描述	例子	结果
abs(x)		绝对值	abs(-17.4)	17.4
cbrt(double)		立方根	cbrt(27.0)	3
ceil(double/numeric)		不小于参数的最小的整数	ceil(-42.8)	-42
degrees(double)		把弧度转为角度	degrees(0.5)	28.6478897565412
exp(double/numeric)		自然指数	exp(1.0)	2.71828182845905
floor(double/numeric)		不大于参数的最大整数	floor(-42.8)	-43
ln(double/numeric)		自然对数	ln(2.0)	0.693147180559945
log(double/numeric)		10 为底的对数	log(100.0)	2
log(b numeric,x numeric)		numeric 指定底数的对数	log(2.0, 64.0)	6.0000000000
mod(y, x)		取余数	mod(9,4)	1
pi()	double	"π"常量	pi()	3.14159265358979

power(a double, b double)	double	求 a 的 b 次幂	power(9.0, 3.0)	729
power(a numeric, b numeric)	numeric	求 a 的 b 次幂	power(9.0, 3.0)	729
radians(double)	double	把角度转为弧度	radians(45.0)	0.785398163397448
random()	double	0.0 到 1.0 之间的随机数值	random()	
round(double/numeric)		圆整为最接近的整数	round(42.4)	42
round(v numeric, s int)	numeric	圆整为 s 位小数数字	round(42.438,2)	42.44
sign(double/numeric)		参数的符号(-1,0,+1)	sign(-8.4)	-1
sqrt(double/numeric)		平方根	sqrt(2.0)	1.4142135623731
trunc(double/numeric)		截断(向零靠近)	trunc(42.8)	42
trunc(v numeric, s int)	numeric	截断为 s 小数位置的数字	trunc(42.438,2)	42.43

三角函数列表：

函数	描述
acos(x)	反余弦
asin(x)	反正弦
atan(x)	反正切
atan2(x, y)	正切 y/x 的反函数
cos(x)	余弦
cot(x)	余切
sin(x)	正弦
tan(x)	正切

四、字符串函数和操作符：

下面是 PostgreSQL 中提供的字符串操作符列表：

函数	返回类型	描述	例子	结果
string || string	text	字串连接	'Post' || 'greSQL'	PostgreSQL
bit_length(string)	int	字串里二进制位的个数	bit_length('jose')	32
char_length(string)	int	字串中的字符个数	char_length('jose')	4
convert(string using conversion_name)	text	使用指定的转换名字改变编码。	convert('PostgreSQL' using iso_8859_1_to_utf8)	'PostgreSQL'
lower(string)	text	把字串转化为小写	lower('TOM')	tom
octet_length(string)	int	字串中的字节数	octet_length('jose')	4
overlay(string placing string from int [for int])	text	替换子字串	overlay('Txxxxas' placing 'hom' from 2 for 4)	Thomas
position(substring in string)	int	指定的子字串的位置	position('om' in 'Thomas')	3

substring(string [from int] [for int])	text	抽取子字串	substring('Thomas' from 2 for 3)	hom
substring(string from pattern)	text	抽取匹配 POSIX 正则表达式的子字串	substring('Thomas' from '...$')	mas
substring(string from pattern for escape)	text	抽取匹配 SQL 正则表达式的子字串	substring('Thomas' from '%#"o_a#"_' for '#')	oma
trim([leading | trailing | both] [characters] from string)	text	从字串 string 的开头/结尾/两边/ 删除只包含 characters(缺省是一个空白)的最长的字串	trim(both 'x' from 'xTomxx')	Tom
upper(string)	text	把字串转化为大写。	upper('tom')	TOM
ascii(text)	int	参数第一个字符的 ASCII 码	ascii('x')	120
btrim(string text [, characters text])	text	从 string 开头和结尾删除只包含在 characters 里(缺省是空白)的字符的最长字串	btrim('xyxtrimyyx','xy')	trim
chr(int)	text	给出 ASCII 码的字符	chr(65)	A
convert(string text, [src_encoding name,] dest_encoding name)	text	把字串转换为 dest_encoding	convert('text_in_utf8', 'UTF8', 'LATIN1')	以 ISO 8859-1 编码表示的 text_in_utf8
initcap(text)	text	把每个单词的第一个子母转为大写，其它的保留小写。单词是一系列字母数字组成的字符，用非字母数字分隔。	initcap('hi thomas')	Hi Thomas
length(string text)	int	string 中字符的数目	length('jose')	4
lpad(string text, length int [, fill text])	text	通过填充字符 fill(缺省时为空白)，把 string 填充为长度 length。 如果 string 已经比 length 长则将其截断(在右边)。	lpad('hi', 5, 'xy')	xyxhi
ltrim(string text [, characters text])	text	从字串 string 的开头删除只包含 characters(缺省是一个空白)的最长的字串。	ltrim('zzzytrim','xyz')	trim
md5(string text)	text	计算给出 string 的 MD5 散列，以十六进制返回结果。	md5('abc')	
repeat(string text, number int)	text	重复 string number 次。	repeat('Pg', 4)	PgPgPgPg
replace(string text, from text, to text)	text	把字串 string 里出现地所有子字串 from 替换成子字串 to。	replace('abcdefabcdef', 'cd', 'XX')	abXXefabXXef
rpad(string text, length int [, fill text])	text	通过填充字符 fill(缺省时为空白)，把 string 填充为长度 length。如果 string 已经比 length 长则将其截断。	rpad('hi', 5, 'xy')	hixyx
rtrim(string text [, character text])	text	从字串 string 的结尾删除只包含 character(缺省是个空白)的最长的字	rtrim('trimxxxx','x')	trim

split_part(string text, delimiter text, field int)	text	根据 delimiter 分隔 string 返回生成的第 field 个子字串(1 Base)。	split_part('abc~@~def~@~ghi', '~@~', 2)	def
strpos(string, substring)	text	声明的子字串的位置。	strpos('high','ig')	2
substr(string, from [, count])	text	抽取子字串。	substr('alphabet', 3, 2)	ph
to_ascii(text [, encoding])	text	把 text 从其它编码转换为 ASCII。	to_ascii('Karel')	Karel
to_hex(number int/bigint)	text	把 number 转换成其对应地十六进制表现形式。	to_hex(9223372036854775807)	7fffffffffffffff
translate(string text, from text, to text)	text	把在 string 中包含的任何匹配 from 中的字符的字符转化为对应的在 to 中的字符。	translate('12345', '14', 'ax')	a23x5

五、位串函数和操作符：

对于类型 bit 和 bit varying，除了常用的比较操作符之外，还可以使用以下列表中由 PostgreSQL 提供的位串函数和操作符，其中&、|和#的位串操作数必须等长。在移位的时候，保留原始的位串的的长度。

操作符	描述	例子	结果
||	连接	B'10001' || B'011'	10001011
&	按位 AND	B'10001' & B'01101'	00001
|	按位 OR	B'10001' | B'01101'	11101
#	按位 XOR	B'10001' # B'01101'	11100
~	按位 NOT	~ B'10001'	01110
<<	按位左移	B'10001' << 3	01000
>>	按位右移	B'10001' >> 2	00100

除了以上列表中提及的操作符之外，位串还可以使用字符串函数：length，bit_length，octet_length，position，substring。此外，我们还可以在整数和 bit 之间来回转换，如：

```
MyTest=# SELECT 44::bit(10);
     bit
------------
 0000101100
(1 row)
MyTest=# SELECT 44::bit(3);
 bit
-----
 100
(1 row)
MyTest=# SELECT cast(-44 as bit(12));
      bit
--------------
 111111010100
(1 row)
MyTest=# SELECT '1110'::bit(4)::integer;
 int4
```

14

(1 row)

注意：如果只是转换为"bit"，意思是转换成 bit(1)，因此只会转换成整数的最低位。

PostgreSQL 学习手册(函数和操作符<二>)

六、模式匹配：

PostgreSQL 中提供了三种实现模式匹配的方法：SQL LIKE 操作符，更近一些的 SIMILAR TO 操作符，和 POSIX-风格正则表达式。

1. LIKE：

string **LIKE** pattern [**ESCAPE** escape-character]

string **NOT LIKE** pattern [**ESCAPE** escape-character]

每个 pattern 定义一个字串的集合。如果该 string 包含在 pattern 代表的字串集合里，那么 LIKE 表达式返回真。和我们想象的一样，如果 LIKE 返回真，那么 NOT LIKE 表达式返回假，反之亦然。在 pattern 里的下划线(_)代表匹配任何单个字符，而一个百分号(%)匹配任何零或更多字符，如：

'abc' LIKE 'abc' true

'abc' LIKE 'a%' true

'abc' LIKE '_b_' true

'abc' LIKE 'c' false

要匹配文本的下划线或者百分号，而不是匹配其它字符，在 pattern 里相应的字符必须前导转义字符。缺省的转义字符是反斜杠，但是你可以用 ESCAPE 子句指定一个。要匹配转义字符本身，写两个转义字符。我们也可以通过写成 **ESCAPE ''**的方式有效地关闭转义机制，此时，我们就不能关闭下划线和百分号的特殊含义了。

关键字 **ILIKE** 可以用于替换 LIKE，令该匹配就当前的区域设置是大小写无关的。这个特性不是 SQL 标准，是 PostgreSQL 的扩展。操作符**~~**等效于 **LIKE**， 而**~~***对应 **ILIKE**。还有**!~~**和**!~~***操作符分别代表 **NOT LIKE** 和 **NOT ILIKE**。所有这些操作符都是 PostgreSQL 特有的。

2. SIMILAR TO 正则表达式：

SIMILAR TO 根据模式是否匹配给定的字符串而返回真或者假。

string **SIMILAR TO** pattern [**ESCAPE** escape-character]

string **NOT SIMILAR TO** pattern [**ESCAPE** escape-character]

它和 LIKE 非常类似，支持 LIKE 的通配符(**'_'**和**'%'**)且保持其原意。除此之外，SIMILAR TO 还支持一些自己独有的元字符，如：

1). **|** 标识选择(两个候选之一)。

2). ***** 表示重复前面的项零次或更多次。

3). **+** 表示重复前面的项一次或更多次。

4). 可以使用圆括弧**()**把项组合成一个逻辑项。

5). 一个方括弧表达式**[...]**声明一个字符表，就像 POSIX 正则表达式一样。

见如下示例：

'abc' SIMILAR TO 'abc' true

'abc' SIMILAR TO 'a' false

'abc' SIMILAR TO '%(b|d)%' true

'abc' SIMILAR TO '(b|c)%' false

带三个参数的 **substring**，substring(string from pattern for escape-character)，提供了一个从字串中抽取一个匹配 SQL 正则表达式模式的子字串的函数。和 SIMILAR TO 一样，声明的模式必须匹配整个数据串，否则函数失效并返回 NULL。为了标识在

成功的时候应该返回的模式部分，模式必须出现后跟双引号(")的两个转义字符。匹配这两个标记之间的模式的字串将被返回，如：

```
MyTest=# SELECT substring('foobar' from '%#"o_b#"%' FOR '#'); --这里#是转义符，双引号内的模式是返回部分。
 substring
-----------
 oob
(1 row)
MyTest=# SELECT substring('foobar' from '#"o_b#"%' FOR '#');  --foobar 不能完全匹配后面的模式，因此返回 NULL。
 substring
-----------

(1 row)
```

七、数据类型格式化函数：

PostgreSQL 格式化函数提供一套有效的工具用于把各种数据类型(日期/时间、integer、floating point 和 numeric)转换成格式化的字符串以及反过来从格式化的字符串转换成指定的数据类型。下面列出了这些函数，它们都遵循一个公共的调用习惯：第一个参数是待格式化的值，而第二个是定义输出或输出格式的模板。

函数	返回类型	描述	例子
to_char(timestamp, text)	text	把时间戳转换成字串	to_char(current_timestamp, 'HH12:MI:SS')
to_char(interval, text)	text	把时间间隔转为字串	to_char(interval '15h 2m 12s', 'HH24:MI:SS')
to_char(int, text)	text	把整数转换成字串	to_char(125, '999')
to_char(double precision, text)	text	把实数/双精度数转换成字串	to_char(125.8::real, '999D9')
to_char(numeric, text)	text	把 numeric 转换成字串	to_char(-125.8, '999D99S')
to_date(text, text)	date	把字串转换成日期	to_date('05 Dec 2000', 'DD Mon YYYY')
to_timestamp(text, text)	timestamp	把字串转换成时间戳	to_timestamp('05 Dec 2000', 'DD Mon YYYY')
to_timestamp(double)	timestamp	把 UNIX 纪元转换成时间戳	to_timestamp(200120400)
to_number(text, text)	numeric	把字串转换成 numeric	to_number('12,454.8-', '99G999D9S')

1．用于日期/时间格式化的模式：

模式	描述
HH	一天的小时数(01-12)
HH12	一天的小时数(01-12)
HH24	一天的小时数(00-23)
MI	分钟(00-59)
SS	秒(00-59)
MS	毫秒(000-999)

US	微秒(000000-999999)
AM	正午标识(大写)
Y,YYY	带逗号的年(4 和更多位)
YYYY	年(4 和更多位)
YYY	年的后三位
YY	年的后两位
Y	年的最后一位
MONTH	全长大写月份名(空白填充为 9 字符)
Month	全长混合大小写月份名(空白填充为 9 字符)
month	全长小写月份名(空白填充为 9 字符)
MON	大写缩写月份名(3 字符)
Mon	缩写混合大小写月份名(3 字符)
mon	小写缩写月份名(3 字符)
MM	月份号(01-12)
DAY	全长大写日期名(空白填充为 9 字符)
Day	全长混合大小写日期名(空白填充为 9 字符)
day	全长小写日期名(空白填充为 9 字符)
DY	缩写大写日期名(3 字符)
Dy	缩写混合大小写日期名(3 字符)
dy	缩写小写日期名(3 字符)
DDD	一年里的日子(001-366)
DD	一个月里的日子(01-31)
D	一周里的日子(1-7；周日是 1)
W	一个月里的周数(1-5)(第一周从该月第一天开始)
WW	一年里的周数(1-53)(第一周从该年的第一天开始)

2．用于数值格式化的模板模式：

模式	描述
9	带有指定数值位数的值
0	带前导零的值
.(句点)	小数点
,(逗号)	分组(千)分隔符
PR	尖括号内负值
S	带符号的数值
L	货币符号
D	小数点
G	分组分隔符

MI	在指明的位置的负号(如果数字 < 0)
PL	在指明的位置的正号(如果数字 > 0)
SG	在指明的位置的正/负号

八、时间/日期函数和操作符：

1. 下面是 PostgreSQL 中支持的时间/日期操作符的列表：

操作符	例子	结果
+	date '2001-09-28' + integer '7'	date '2001-10-05'
+	date '2001-09-28' + interval '1 hour'	timestamp '2001-09-28 01:00'
+	date '2001-09-28' + time '03:00'	timestamp '2001-09-28 03:00'
+	interval '1 day' + interval '1 hour'	interval '1 day 01:00'
+	timestamp '2001-09-28 01:00' + interval '23 hours'	timestamp '2001-09-29 00:00'
+	time '01:00' + interval '3 hours'	time '04:00'
-	- interval '23 hours'	interval '-23:00'
-	date '2001-10-01' - date '2001-09-28'	integer '3'
-	date '2001-10-01' - integer '7'	date '2001-09-24'
-	date '2001-09-28' - interval '1 hour'	timestamp '2001-09-27 23:00'
-	time '05:00' - time '03:00'	interval '02:00'
-	time '05:00' - interval '2 hours'	time '03:00'
-	timestamp '2001-09-28 23:00' - interval '23 hours'	timestamp '2001-09-28 00:00'
-	interval '1 day' - interval '1 hour'	interval '23:00'
-	timestamp '2001-09-29 03:00' - timestamp '2001-09-27 12:00'	interval '1 day 15:00'
*	interval '1 hour' * double precision '3.5'	interval '03:30'
/	interval '1 hour' / double precision '1.5'	interval '00:40'

2. 日期/时间函数：

函数	返回类型	描述	例子	结果
age(timestamp, timestamp)	interval	减去参数，生成一个使用年、月的"符号化"的结果	age('2001-04-10', timestamp '1957-06-13')	43 years 9 mons 27 days
age(timestamp)	interval	从 current_date 减去得到的数值	age(timestamp '1957-06-13')	43 years 8 mons 3 days
current_date	date	今天的日期		
current_time	time	现在的时间		
current_timestamp	timestamp	日期和时间		
date_part(text, timestamp)	double	获取子域(等效于 extract)	date_part('hour', timestamp '2001-02-16 20:38:40')	20

date_part(text, interval)	double	获取子域(等效于 extract)	date_part('month', interval '2 years 3 months')	3
date_trunc(text, timestamp)	timestamp	截断成指定的精度	date_trunc('hour', timestamp '2001-02-16 20:38:40')	2001-02-16 20:00:00+00
extract(field from timestamp)	double	获取子域	extract(hour from timestamp '2001-02-16 20:38:40')	20
extract(field from interval)	double	获取子域	extract(month from interval '2 years 3 months')	3
localtime	time	今日的时间		
localtimestamp	timestamp	日期和时间		
now()	timestamp	当前的日期和时间(等效于 current_timestamp)		
timeofday()	text	当前日期和时间		

3. EXTRACT，date_part 函数支持的 field：

域	描述	例子	结果
CENTURY	世纪	EXTRACT(CENTURY FROM TIMESTAMP '2000-12-16 12:21:13');	20
DAY	(月分)里的日期域(1-31)	EXTRACT(DAY from TIMESTAMP '2001-02-16 20:38:40');	16
DECADE	年份域除以 10	EXTRACT(DECADE from TIMESTAMP '2001-02-16 20:38:40');	200
DOW	每周的星期号(0-6；星期天是 0) (仅用于 timestamp)	EXTRACT(DOW FROM TIMESTAMP '2001-02-16 20:38:40');	5
DOY	一年的第几天(1 -365/366) (仅用于 timestamp)	EXTRACT(DOY from TIMESTAMP '2001-02-16 20:38:40');	47
HOUR	小时域(0-23)	EXTRACT(HOUR from TIMESTAMP '2001-02-16 20:38:40');	20
MICROSECONDS	秒域，包括小数部分，乘以 1,000,000。	EXTRACT(MICROSECONDS from TIME '17:12:28.5');	28500000
MILLENNIUM	千年	EXTRACT(MILLENNIUM from TIMESTAMP '2001-02-16 20:38:40');	3
MILLISECONDS	秒域，包括小数部分，乘以 1000。	EXTRACT(MILLISECONDS from TIME '17:12:28.5');	28500
MINUTE	分钟域(0-59)	EXTRACT(MINUTE from TIMESTAMP '2001-02-16 20:38:40');	38
MONTH	对于 timestamp 数值，它是一年里的月份数(1-12)；对于 interval 数值，它是月的数目，然后对 12 取模(0-11)	EXTRACT(MONTH from TIMESTAMP '2001-02-16 20:38:40');	2
QUARTER	该天所在的该年的季度(1-4)(仅用于 timestamp)	EXTRACT(QUARTER from TIMESTAMP '2001-02-16 20:38:40');	1

SECOND	秒域，包括小数部分(0-59[1])	EXTRACT(SECOND from TIMESTAMP '2001-02-16 20:38:40');	40
WEEK	该天在所在的年份里是第几周。	EXTRACT(WEEK from TIMESTAMP '2001-02-16 20:38:40');	7
YEAR	年份域	EXTRACT(YEAR from TIMESTAMP '2001-02-16 20:38:40');	2001

4. 当前日期/时间：

我们可以使用下面的函数获取当前的日期和/或时间：

CURRENT_DATE
CURRENT_TIME
CURRENT_TIMESTAMP
CURRENT_TIME (precision)
CURRENT_TIMESTAMP (precision)
LOCALTIME
LOCALTIMESTAMP
LOCALTIME (precision)
LOCALTIMESTAMP (precision)

PostgreSQL 学习手册(函数和操作符<三>).

九、序列操作函数：

序列对象(也叫序列生成器)都是用 CREATE SEQUENCE 创建的特殊的单行表。一个序列对象通常用于为行或者表生成唯一的标识符。下面序列函数，为我们从序列对象中获取最新的序列值提供了简单和并发读取安全的方法。

函数	返回类型	描述
nextval(regclass)	bigint	递增序列对象到它的下一个数值并且返回该值。这个动作是自动完成的。即使多个会话并发运行 nextval，每个进程也会安全地收到一个唯一的序列值。
currval(regclass)	bigint	在当前会话中返回最近一次 nextval 抓到的该序列的数值。(如果在本会话中从未在该序列上调用过 nextval，那么会报告一个错误。)请注意因为此函数返回一个会话范围的数值，而且也能给出一个可预计的结果，因此可以用于判断其它会话是否执行过 nextval。
lastval()	bigint	返回当前会话里最近一次 nextval 返回的数值。这个函数等效于 currval，只是它不用序列名为参数，它抓取当前会话里面最近一次 nextval 使用的序列。如果当前会话还没有调用过 nextval，那么调用 lastval 将会报错。
setval(regclass, bigint)	bigint	重置序列对象的计数器数值。设置序列的 last_value 字段为指定数值并且将其 is_called 字段设置为 true，表示下一次 nextval 将在返回数值之前递增该序列。
setval(regclass, bigint, boolean)	bigint	重置序列对象的计数器数值。功能等同于上面的 setval 函数，只是 is_called 可以设置为 true 或 false。如果将其设置为 false，那么下一次 nextval 将返回该数值，随后的 nextval 才开始递增该序列。

对于 regclass 参数，仅需用单引号括住序列名即可，因此它看上去就像文本常量。为了达到和处理普通 SQL 对象一样的兼容性，这个字串将被转换成小写，除非该序列名是用双引号括起，如：

```
nextval('foo')      --操作序列号 foo
nextval('FOO')      --操作序列号 foo
nextval('"Foo"')    --操作序列号 Foo
SELECT setval('foo', 42);     -- 下次 nextval 将返回 43
SELECT setval('foo', 42, true);
SELECT setval('foo', 42, false);    -- 下次 nextval 将返回 42
```

十、条件表达式：

1. CASE：

SQL CASE 表达式是一种通用的条件表达式，类似于其它语言中的 if/else 语句。

```
CASE WHEN condition THEN result
    [WHEN ...]
    [ELSE result]
END
```

condition 是一个返回 boolean 的表达式。如果为真，那么 CASE 表达式的结果就是符合条件的 result。如果结果为假，那么以相同方式搜寻随后的 WHEN 子句。如果没有 WHEN condition 为真，那么 case 表达式的结果就是在 ELSE 子句里的值。如果省略了 ELSE 子句而且没有匹配的条件，结果为 NULL，如：

```
MyTest=> SELECT * FROM testtable;
 i
---
 1
 2
 3
(3 rows)
MyTest=> SELECT i, CASE WHEN i=1 THEN 'one'
MyTest->                WHEN i=2 THEN 'two'
MyTest->                ELSE 'other'
MyTest->           END
MyTest-> FROM testtable;
 i | case
---+-------
 1 | one
 2 | two
 3 | other
(3 rows)
```

注：CASE 表达式并不计算任何对于判断结果并不需要的子表达式。

2. COALESCE：

COALESCE 返回它的第一个非 NULL 的参数的值。它常用于在为显示目的检索数据时用缺省值替换 NULL 值。

```
COALESCE(value[, ...])
```

和 CASE 表达式一样，COALESCE 将不会计算不需要用来判断结果的参数。也就是说，在第一个非空参数右边的参数不会被计算。

3. NULLIF：

当且仅当 value1 和 value2 相等时，NULLIF 才返回 NULL。否则它返回 value1。

```
NULLIF(value1, value2)
```

```
MyTest=> SELECT NULLIF('abc','abc');
 nullif
--------

(1 row)
MyTest=> SELECT NULLIF('abcd','abc');
 nullif
--------
 abcd
(1 row)
```

4. GREATEST 和 LEAST：

GREATEST 和 LEAST 函数从一个任意的数字表达式列表里选取最大或者最小的数值。列表中的 NULL 数值将被忽略。只有所有表达式的结果都是 NULL 的时候，结果才会是 NULL。

GREATEST(value [, ...])

LEAST(value [, ...])

```
MyTest=> SELECT GREATEST(1,3,5);
 greatest
----------
        5
(1 row)
MyTest=> SELECT LEAST(1,3,5,NULL);
 least
-------
     1
(1 row)
```

十一、数组函数和操作符：

1. PostgreSQL 中提供的用于数组的操作符列表：

操作符	描述	例子	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
<=	小于或等于	ARRAY[1,2,3] <= ARRAY[1,2,3]	t
>=	大于或等于	ARRAY[1,4,3] >= ARRAY[1,4,3]	t
||	数组与数组连接	ARRAY[1,2,3] || ARRAY[4,5,6]	{1,2,3,4,5,6}
||	数组与数组连接	ARRAY[1,2,3] || ARRAY[[4,5,6],[7,8,9]]	{{1,2,3},{4,5,6},{7,8,9}}
||	元素与数组连接	3 || ARRAY[4,5,6]	{3,4,5,6}
||	元素与数组连接	ARRAY[4,5,6] || 7	{4,5,6,7}

2. PostgreSQL 中提供的用于数组的函数列表：

函数	返回类型	描述	例子	结果
array_cat(anyarray, anyarray)	anyarray	连接两个数组	array_cat(ARRAY[1,2,3], ARRAY[4,5])	{1,2,3,4,5}
array_append(anyarray, anyelement)	anyarray	向一个数组末尾附加一个元素	array_append(ARRAY[1,2], 3)	{1,2,3}
array_prepend(anyelement, anyarray)	anyarray	向一个数组开头附加一个元素	array_prepend(1, ARRAY[2,3])	{1,2,3}
array_dims(anyarray)	text	返回一个数组维数的文本表示	array_dims(ARRAY[[1,2,3], [4,5,6]])	[1:2][1:3]
array_lower(anyarray, int)	int	返回指定的数组维数的下界	array_lower(array_prepend(0, ARRAY[1,2,3]), 1)	0
array_upper(anyarray, int)	int	返回指定数组维数的上界	array_upper(ARRAY[1,2,3,4], 1)	4
array_to_string(anyarray, text)	text	使用提供的分隔符连接数组元素	array_to_string(ARRAY[1, 2, 3], '~^~')	1~^~2~^~3
string_to_array(text, text)	text[]	使用指定的分隔符把字串拆分成数组元素	string_to_array('xx~^~yy~^~zz', '~^~')	{xx,yy,zz}

十二、系统信息函数：

1. PostgreSQL 中提供的和数据库相关的函数列表：

名字	返回类型	描述
current_database()	name	当前数据库的名字
current_schema()	name	当前模式的名字
current_schemas(boolean)	name[]	在搜索路径中的模式名字
current_user	name	目前执行环境下的用户名
inet_client_addr()	inet	连接的远端地址
inet_client_port()	int	连接的远端端口
inet_server_addr()	inet	连接的本地地址
inet_server_port()	int	连接的本地端口
session_user	name	会话用户名
pg_postmaster_start_time()	timestamp	postmaster 启动的时间
user	name	current_user
version()	text	PostgreSQL 版本信息

2. 允许用户在程序里查询对象访问权限的函数：

名字	描述	可用权限
has_table_privilege(user,table,privilege)	用户是否有访问表的权限	SELECT/INSERT/UPDATE/DELETE/RULE/REFERENCES/TRIGGER
has_table_privilege(table,privilege)	当前用户是否有访问表的权限	SELECT/INSERT/UPDATE/DELETE/

		RULE/REFERENCES/TRIGGER
has_database_privilege(user,database,privilege)	用户是否有访问数据库的权限	CREATE/TEMPORARY
has_database_privilege(database,privilege)	当前用户是否有访问数据库的权限	CREATE/TEMPORARY
has_function_privilege(user,function,privilege)	用户是否有访问函数的权限	EXECUTE
has_function_privilege(function,privilege)	当前用户是否有访问函数的权限	EXECUTE
has_language_privilege(user,language,privilege)	用户是否有访问语言的权限	USAGE
has_language_privilege(language,privilege)	当前用户是否有访问语言的权限	USAGE
has_schema_privilege(user,schema,privilege)	用户是否有访问模式的权限	CREAT/USAGE
has_schema_privilege(schema,privilege)	当前用户是否有访问模式的权限	CREAT/USAGE
has_tablespace_privilege(user,tablespace,privilege)	用户是否有访问表空间的权限	CREATE
has_tablespace_privilege(tablespace,privilege)	当前用户是否有访问表空间的权限	CREATE

注：以上函数均返回 boolean 类型。要评估一个用户是否在权限上持有赋权选项，给权限键字附加 WITH GRANT OPTION；比如 'UPDATE WITH GRANT OPTION'。

3. 模式可视性查询函数：

那些判断一个对象是否在当前模式搜索路径中可见的函数。 如果一个表所在的模式在搜索路径中，并且没有同名的表出现在搜索路径的更早的地方，那么就说这个表视可见的。 它等效于表可以不带明确模式修饰进行引用。

名字	描述	应用类型
pg_table_is_visible(table_oid)	该表/视图是否在搜索路径中可见	regclass
pg_type_is_visible(type_oid)	该类/视图型是否在搜索路径中可见	regtype
pg_function_is_visible(function_oid)	该函数是否在搜索路径中可见	regprocedure
pg_operator_is_visible(operator_oid)	该操作符是否在搜索路径中可见	regoperator
pg_opclass_is_visible(opclass_oid)	该操作符表是否在搜索路径中可见	regclass
pg_conversion_is_visible(conversion_oid)	转换是否在搜索路径中可见	regoperator

注：以上函数均返回 boolean 类型，所有这些函数都需要对象 OID 标识作为检查的对象。

```
postgres=# SELECT pg_table_is_visible('testtable'::regclass);
 pg_table_is_visible
---------------------
 t
(1 row)
```

4. 系统表信息函数：

名字	返回类型	描述

format_type(type_oid,typemod)	text	获取一个数据类型的 SQL 名称
pg_get_viewdef(view_oid)	text	为视图获取 CREATE VIEW 命令
pg_get_viewdef(view_oid,pretty_bool)	text	为视图获取 CREATE VIEW 命令
pg_get_ruledef(rule_oid)	text	为规则获取 CREATE RULE 命令
pg_get_ruledef(rule_oid,pretty_bool)	text	为规则获取 CREATE RULE 命令
pg_get_indexdef(index_oid)	text	为索引获取 CREATE INDEX 命令
pg_get_indexdef(index_oid,column_no,pretty_bool)	text	为索引获取 CREATE INDEX 命令， 如果 column_no 不为零，则是只获取一个索引字段的定义
pg_get_triggerdef(trigger_oid)	text	为触发器获取 CREATE [CONSTRAINT] TRIGGER
pg_get_constraintdef(constraint_oid)	text	获取一个约束的定义
pg_get_constraintdef(constraint_oid,pretty_bool)	text	获取一个约束的定义
pg_get_expr(expr_text,relation_oid)	text	反编译一个表达式的内部形式，假设其中的任何 Vars 都引用第二个参数指出的关系
pg_get_expr(expr_text,relation_oid, pretty_bool)	text	反编译一个表达式的内部形式，假设其中的任何 Vars 都引用第二个参数指出的关系
pg_get_userbyid(roleid)	name	获取给出的 ID 的角色名
pg_get_serial_sequence(table_name,column_name)	text	获取一个 serial 或者 bigserial 字段使用的序列名字
pg_tablespace_databases(tablespace_oid)	setof oid	获取在指定表空间(OID 表示)中拥有对象的一套数据库的 OID 的集合

这些函数大多数都有两个变种，其中一个可以选择对结果的"漂亮的打印"。 漂亮打印的格式更容易读，但是缺省的格式更有可能被将来的 PostgreSQL 版本用同样的方法解释；如果是用于转储，那么尽可能避免使用漂亮打印。 给漂亮打印参数传递 false 生成的结果和那个没有这个参数的变种生成的结果完全一样。

十三、系统管理函数：

1. 查询以及修改运行时配置参数的函数：

名字	返回类型	描述
current_setting(setting_name)	text	当前设置的值
set_config(setting_name,new_value,is_local)	text	设置参数并返回新值

current_setting 用于以查询形式获取 setting_name 设置的当前数值。它和 SQL 命令 SHOW 是等效的。 比如：

```
MyTest=# SELECT current_setting('datestyle');
 current_setting
-----------------
 ISO, YMD
(1 row)
```

set_config 将参数 setting_name 设置为 new_value。如果 is_local 设置为 true，那么新数值将只应用于当前事务。如果你希望新的数值应用于当前会话，那么应该使用 false。它等效于 SQL 命令 SET。比如：

```
MyTest=# SELECT set_config('log_statement_stats','off', false);
 set_config
------------
```

off
(1 row)

2. 数据库对象尺寸函数：

名字	返回类型	描述
pg_tablespace_size(oid)	bigint	指定 OID 代表的表空间使用的磁盘空间
pg_tablespace_size(name)	bigint	指定名字的表空间使用的磁盘空间
pg_database_size(oid)	bigint	指定 OID 代表的数据库使用的磁盘空间
pg_database_size(name)	bigint	指定名称的数据库使用的磁盘空间
pg_relation_size(oid)	bigint	指定 OID 代表的表或者索引所使用的磁盘空间
pg_relation_size(text)	bigint	指定名称的表或者索引使用的磁盘空间。这个名字可以用模式名修饰
pg_total_relation_size(oid)	bigint	指定 OID 代表的表使用的磁盘空间，包括索引和压缩数据
pg_total_relation_size(text)	bigint	指定名字的表所使用的全部磁盘空间，包括索引和压缩数据。表名字可以用模式名修饰。
pg_size_pretty(bigint)	text	把字节计算的尺寸转换成一个人类易读的尺寸单位

3. 数据库对象位置函数：

名字	返回类型	描述
pg_relation_filenode(relationregclass)	oid	获取指定对象的文件节点编号(通常为对象的 oid 值)。
pg_relation_filepath(relationregclass)	text	获取指定对象的完整路径名。

```
mydatabase=# select pg_relation_filenode('testtable');
 pg_relation_filenode
----------------------
                17877
(1 row)
mydatabase=# select pg_relation_filepath('testtable');
            pg_relation_filepath
-----------------------------------------------
 pg_tblspc/17633/PG_9.1_201105231/17636/17877
(1 row)
```

PostgreSQL 学习手册(索引)

一、索引的类型：

PostgreSQL 提供了多　种索引类型：B-Tree、Hash、GiST 和 GIN，由于它们使用了不同的算法，因此每种索引类型都有其适合的查询类型，缺省时，CREATE INDEX 命令将创建 B-Tree 索引。

1. B-Tree:

```
CREATE TABLE test1 (
    id integer,
    content varchar
);
CREATE INDEX test1_id_index ON test1 (id);
```

B-Tree 索引主要用于等于和范围查询，特别是当索引列包含操作符" <、<=、=、>=和>"作为查询条件时，PostgreSQL 的查询规划器都会考虑使用 B-Tree 索引。在使用 BETWEEN、IN、IS NULL 和 IS NOT NULL 的查询中，PostgreSQL 也可以使用 B-Tree 索引。然而对于基于模式匹配操作符的查询，如 LIKE、ILIKE、~和 ~*，仅当模式存在一个常量，且该常量位于模式字符串的开头时，如 col LIKE 'foo%'或 col ~ '^foo'，索引才会生效，否则将会执行全表扫描，如：col LIKE '%bar'。

2. Hash：

```
CREATE INDEX name ON table USING hash (column);
```

散列(Hash)索引只能处理简单的等于比较。当索引列使用等于操作符进行比较时，查询规划器会考虑使用散列索引。

这里需要额外说明的是，PostgreSQL 散列索引的性能不比 B-Tree 索引强，但是散列索引的尺寸和构造时间则更差。另外，由于散列索引操作目前没有记录 WAL 日志，因此一旦发生了数据库崩溃，我们将不得不用 REINDEX 重建散列索引。

3. GiST：

GiST 索引不是一种单独的索引类型，而是一种架构，可以在该架构上实现很多不同的索引策略。从而可以使 GiST 索引根据不同的索引策略，而使用特定的操作符类型。

4. GIN：

GIN 索引是反转索引，它可以处理包含多个键的值(比如数组)。与 GiST 类似，GIN 同样支持用户定义的索引策略，从而可以使 GIN 索引根据不同的索引策略，而使用特定的操作符类型。作为示例，PostgreSQL 的标准发布中包含了用于一维数组的 GIN 操作符类型，如：<@、@>、=、&&等。

二、复合索引：

PostgreSQL 中的索引可以定义在数据表的多个字段上，如：

```
CREATE TABLE test2 (
    major int,
    minor int,
    name varchar
}
CREATE INDEX test2_mm_idx ON test2 (major, minor);
```

在当前的版本中，只有 B-tree、GiST 和 GIN 支持复合索引，其中最多可以声明 32 个字段。

1. B-Tree 类型的复合索引：

在 B-Tree 类型的复合索引中，该索引字段的任意子集均可用于查询条件，不过，只有当复合索引中的第一个索引字段(最左边)被包含其中时，才可以获得最高效率。

2. GiST 类型的复合索引：

在 GiST 类型的复合索引中，只有当第一个索引字段被包含在查询条件中时，才能决定该查询会扫描多少索引数据，而其他索引字段上的条件只是会限制索引返回的条目。假如第一个索引字段上的大多数数据都有相同的键值，那么此时应用 GiST 索引就会比较低效。

3. GIN 类型的复合索引：

与 B-Tree 和 GiST 索引不同的是，GIN 复合索引不会受到查询条件中使用了哪些索引字段子集的影响，无论是哪种组合，都会得到相同的效率。

使用复合索引应该谨慎。在大多数情况下，单一字段上的索引就已经足够了，并且还节约时间和空间。除非表的使用模式非常固定，否则超过三个字段的索引几乎没什么用处。

三、组合多个索引：

PostgreSQL 可以在查询时组合多个索引(包括同一索引的多次使用)，来处理单个索引扫描不能实现的场合。与此同时，系统还可以在多个索引扫描之间组成 AND 和 OR 的条件。比如，一个类似 WHERE x = 42 OR x = 47 OR x = 53 OR x = 99 的查询，可以被分解成四个独立的基于 x 字段索引的扫描，每个扫描使用一个查询子句，之后再将这些扫描结果 OR 在一起并生成最终的结果。另外一个例子是，如果我们在 x 和 y 上分别存在独立的索引，那么一个类似 WHERE x = 5 AND y = 6 的查询，就会分别基于这两个字段的索引进行扫描，之后再将各自扫描的结果进行 AND 操作并生成最终的结果行。

为了组合多个索引，系统扫描每个需要的索引，然后在内存里组织一个 BITMAP，它将给出索引扫描出的数据在数据表中的物理位置。然后，再根据查询的需要，把这些位图进行 AND 或者 OR 的操作并得出最终的 BITMAP。最后，检索数据表并返回数据行。表的数据行是按照物理顺序进行访问的，因为这是位图的布局，这就意味着任何原来的索引的排序都将消失。如果查询中有 ORDER BY 子句，那么还将会有一个额外的排序步骤。因为这个原因，以及每个额外的索引扫描都会增加额外的时间，这样规划器有时候就会选择使用简单的索引扫描，即使有多个索引可用也会如此。

四、唯一索引：

目前，只有 B-Tree 索引可以被声明为唯一索引。

CREATE UNIQUE INDEX name **ON** table (column [, ...]);

如果索引声明为唯一索引，那么就不允许出现多个索引值相同的行。我们认为 NULL 值相互间不相等。

五、表达式索引：

表达式索引主要用于在查询条件中存在基于某个字段的函数或表达式的结果与其他值进行比较的情况，如：

*SELECT * FROM test1 WHERE lower(col1) = 'value';*

此时，如果我们仅仅是在 col1 字段上建立索引，那么该查询在执行时一定不会使用该索引，而是直接进行全表扫描。如果该表的数据量较大，那么执行该查询也将会需要很长时间。解决该问题的办法非常简单，在 test1 表上建立基于 col1 字段的表达式索引，如：

CREATE INDEX test1_lower_col1_idx ON test1 ***(lower(col1))****;*

如果我们把该索引声明为 UNIQUE，那么它会禁止创建那种 col1 数值只是大小写有区别的数据行，以及 col1 数值完全相同的数据行。因此，在表达式上的索引可以用于强制那些无法定义为简单唯一约束的约束。现在让我们再看一个应用表达式索引的例子。

*SELECT * FROM people WHERE (first_name || ' ' || last_name) = 'John Smith';*

和上面的例子一样，尽管我们可能会为 first_name 和 last_name 分别创建独立索引，或者是基于这两个字段的复合索引，在执行该查询语句时，这些索引均不会被使用，该查询能够使用的索引只有我们下面创建的表达式索引。

CREATE INDEX people_names ON people ***((first_name || ' ' || last_name))****;*

CREATE INDEX 命令的语法通常要求在索引表达式周围书写圆括弧，就像我们在第二个例子里显示的那样。如果表达式只是一个

函数调用，那么可以省略，就像我们在第一个例子里显示的那样。

从索引维护的角度来看，索引表达式要相对低效一些，因为在插入数据或者更新数据的时候，都必须为该行计算表达式的结果，并将该结果直接存储到索引里。然而在查询时，PostgreSQL 就会把它们看做 WHERE idxcol = 'constant'，因此搜索的速度等效于基于简单索引的查询。通常而言，我们只是应该在检索速度比插入和更新速度更重要的场景下使用表达式索引。

六、部分索引：

部分索引**(partial index)**是建立在一个表的子集上的索引，而该子集是由一个条件表达式定义的(叫做部分索引的谓词)。该索引只包含表中那些满足这个谓词的行。

由于不是在所有的情况下都需要更新索引，因此部分索引会提高数据插入和数据更新的效率。然而又因为部分索引比普通索引要小，因此可以更好的提高确实需要索引部分的查询效率。见以下三个示例：

1. 索引字段和谓词条件字段一致：

```
CREATE INDEX access_log_client_ip_ix ON access_log(client_ip)
    WHERE NOT (client_ip > inet '192.168.100.0' AND client_ip < inet '192.168.100.255');
```

下面的查询将会用到该部分索引：

```
SELECT * FROM access_log WHERE url = '/index.html' AND client_ip = inet '212.78.10.32';
```

下面的查询将不会用该部分索引：

一个不能使用这个索引的查询可以是：

```
SELECT * FROM access_log WHERE client_ip = inet '192.168.100.23';
```

2. 索引字段和谓词条件字段不一致：

PostgreSQL 支持带任意谓词的部分索引，唯一的约束是谓词的字段也要来自于同样的数据表。注意，如果你希望你的查询语句能够用到部分索引，那么就要求该查询语句的条件部分必须和部分索引的谓词完全匹配。 准确说，只有在 PostgreSQL 能够识别出该查询的 WHERE 条件在数学上涵盖了该索引的谓词时，这个部分索引才能被用于该查询。

```
CREATE INDEX orders_unbilled_index ON orders(order_nr) WHERE billed is not true;
```

下面的查询一定会用到该部分索引：

```
SELECT * FROM orders WHERE billed is not true AND order_nr < 10000;
```

那么对于如下查询呢？

```
SELECT * FROM orders WHERE billed is not true AND amount > 5000.00;
```

这个查询将不像上面那个查询这么高效，毕竟查询的条件语句中没有用到索引字段，然而查询条件"billed is not true"却和部分索引的谓词完全匹配，因此 PostgreSQL 将扫描整个索引。这样只有在索引数据相对较少的情况下，该查询才能更有效一些。

下面的查询将不会用到部分索引。

```
SELECT * FROM orders WHERE order_nr = 3501;
```

3. 数据表子集的唯一性约束：

```
CREATE TABLE tests (
    subject text,
    target text,
    success boolean,
    ...
);
CREATE UNIQUE INDEX tests_success_constraint ON tests(subject, target) WHERE success;
```

该部分索引将只会对 success 字段值为 true 的数据进行唯一性约束。在实际的应用中，如果成功的数据较少，而不成功的数据较多时，该实现方法将会非常高效。

七、检查索引的使用：

见以下四条建议：

1．总是先运行 ANALYZE。

该命令将会收集表中数值分布状况的统计。在估算一个查询返回的行数时需要这个信息，而规划器则需要这个行数以便给每个可能的查询规划赋予真实的开销值。如果缺乏任何真实的统计信息，那么就会使用一些缺省数值，这样肯定是不准确的。因此，如果还没有运行 ANALYZE 就检查一个索引的使用状况，那将会是一次失败的检查。

2．使用真实的数据做实验。

用测试数据填充数据表，那么该表的索引将只会基于测试数据来评估该如何使用索引，而不是对所有的数据都如此使用。比如从 100000 行中选 1000 行，规划器可能会考虑使用索引，那么如果从 100 行中选 1 行就很难说也会使用索引了。因为 100 行的数据很可能是存储在一个磁盘页面中，然而没有任何查询规划能比通过顺序访问一个磁盘页面更加高效了。与此同时，在模拟测试数据时也要注意，如果这些数据是非常相似的数据、完全随机的数据，或按照排序顺序插入的数据，都会令统计信息偏离实际数据应该具有的特征。

3．如果索引没有得到使用，那么在测试中强制它的使用也许会有些价值。有一些运行时参数可以关闭各种各样的查询规划。

4．强制使用索引用法将会导致两种可能：一是系统选择是正确的，使用索引实际上并不合适，二是查询计划的开销计算并不能反映现实情况。这样你就应该对使用和不使用索引的查询进行计时，这个时候 EXPLAIN ANALYZE 命令就很有用了。

PostgreSQL 学习手册(事物隔离)

在 SQL 的标准中事物隔离级别分为以下四种：

1．读未提交(Read uncommitted)

2．读已提交(Read committed)

3．可重复读(Repeatable read)

4．可串行化(Serializable)

然而 PostgreSQL 在 9.1 之前的版本中只是实现了其中两种，即读已提交和可串行化，如果在实际应用中选择了另外两种，那么 PostgreSQL 将会自动向更严格的隔离级别调整。在 PostgreSQL v9.1 的版本中提供了三种实现方式，即在原有的基础上增加了可重复读。在这篇博客中我们将只是针对 2)和 4)进行说明和比较，因为在 9.1 中，3)和 4)的差别也是非常小的。

	读已提交	可串行化
PostgreSQL 缺省隔离级别	是	否
其它事物未提交数据是否可见	不可见	不可见
执行效率	高	低
适用场景	简单 SQL 逻辑，如果 SQL 语句中含有嵌套查询，那么在多次 SQL 查询中将极有可能获得不同版本的数据。	复杂 SQL 逻辑，特别是带有嵌套的查询比较适用。
SELECT 查询一致性时间点	从该 SELECT 查询开始执行时，在此查询执行期间，任何其它并发事物针对该查询结果集的数据操作都将不会被本次查询读到，即本次查询获取的数据版本是与查询开始执行时的数据版本相一致。	从该 SELECT 查询所在事物开始时，在此查询执行期间，任何其它并发事物针对该查询结果集的数据操作都将不会被本次查询读到，即本次查询获取的数据版本是与查询所在事物开始时的数据版本相一致。
同事物内的数据操作是否可见	比如在同一个事物内存在 update 和 select 操作，即使当前事物尚未提交，update 所作的修改，在当前事物后面的 select 中依然可见。	和读已提交相同。
同事物内多次	不同，由于该级别 select 的一致性时间点是该查询开	需要分两步来说，对于同一事物内的修改如果发生在两

相同的 select 所见的数据是否相同	始执行时，而多次查询的时间点将肯定不相同，如果在第一次查询开始到第二次查询开始之间，其它的并发事物修改并提交或当前事物仅修改了查询将要获取的数据，那么这些数据操作的结果将会在第二个查询中有所体现。	次查询语句之间，那么第二个查询将会看到这些修改的结果。然而对于其它并发事物的修改，将不会造成任何影响，即两次 select 的结果是相同的。原因显而易见，该隔离级别的 select 一致性时间点是与事物开始时相一致的。
相同行数据的修改	如果此时两个并发事物在修改同一行数据，先修改的事物将会给该行加行级锁，另外一个事物将进入等待状态，直到第一个事物操作该行结束。那么倘若第一个针对该行的修改操作最终被其事物回滚，第二个修改操作在结束等待后，将直接修改该数据。然而如果第一个操作是被正常提交的话，那么就需要进一步判断该操作的类型，如果是删除(delete)该行，第二个修改操作将直接被忽略。如果是 update 该行的记录，第二个修改操作则需要重新评估该行是否依然符合之前定义的修改条件。	和读已提交隔离级别的机制基本相同，只是在第一个修改操作提交后，第二个操作将不再区分之前的修改是 delete 还是 update，而是直接并返回下面信息：Error: Can't serialize access due to concurrent update. 这是因为一个可串行化的事务在可串行化事务开始之后不能更改或者锁住被其他事务更改过的行。因此，当应用收到这样的错误信息时，它应该退出当前的事务然后从头开始重新进行整个事务。在应用程序中，也应该有必要的代码来专门处理该类错误。

最后需要说明的是，在绝大多数的情况下，读已提交级别均可适用，而且该级别的并发效率更高。只有在比较特殊的情况下，才手工将当前的事物隔离级别调整为可串行化或可重复读。

PostgreSQL 学习手册(性能提升技巧)

一、使用 EXPLAIN：

PostgreSQL 为每个查询都生成一个查询规划，因为选择正确的查询路径对性能的影响是极为关键的。PostgreSQL 本身已经包含了一个规划器用于寻找最优规划，我们可以通过使用 EXPLAIN 命令来查看规划器为每个查询生成的查询规划。

PostgreSQL 中生成的查询规划是由 1 到 n 个规划节点构成的规划树，其中最底层的节点为表扫描节点，用于从数据表中返回检索出的数据行。然而，不同的扫描节点类型代表着不同的表访问模式，如：顺序扫描、索引扫描，以及位图索引扫描等。如果查询仍然需要连接、聚集、排序，或者是对原始行的其它操作，那么就会在扫描节点"之上"有其它额外的节点。并且这些操作通常都有多种方法，因此在这些位置也有可能出现不同的节点类型。EXPLAIN 将为规划树中的每个节点都输出一行信息，显示基本的节点类型和规划器为执行这个规划节点计算出的预计开销值。第一行(最上层的节点)是对该规划的总执行开销的预计，这个数值就是规划器试图最小化的数值。

这里有一个简单的例子，如下：

```
EXPLAIN SELECT * FROM tenk1;
                        QUERY PLAN
-------------------------------------------------------------
 Seq Scan on tenk1  (cost=0.00..458.00 rows=10000 width=244)
```

EXPLAIN 引用的数据是：

1). 预计的启动开销(在输出扫描开始之前消耗的时间，比如在一个排序节点里做排续的时间)。

2). 预计的总开销。

3). 预计的该规划节点输出的行数。

4). 预计的该规划节点的行平均宽度(单位：字节)。

这里开销(cost)的计算单位是磁盘页面的存取数量，如 1.0 将表示一次顺序的磁盘页面读取。其中上层节点的开销将包括其所有子节点的开销。这里的输出行数(rows)并不是规划节点处理/扫描的行数，通常会更少一些。一般而言，顶层的行预计数量会更接近于查询实际返回的行数。

现在我们执行下面基于系统表的查询：

SELECT relpages, reltuples FROM pg_class WHERE relname = 'tenk1';

从查询结果中可以看出 tenk1 表占有 358 个磁盘页面和 10000 条记录，然而为了计算 cost 的值，我们仍然需要知道另外一个系统参数值。

```
postgres=# show cpu_tuple_cost;
 cpu_tuple_cost
----------------
 0.01
(1 row)
```

cost = 358(磁盘页面数) + 10000(行数) * 0.01(cpu_tuple_cost 系统参数值)

下面我们再来看一个带有 WHERE 条件的查询规划。

*EXPLAIN SELECT * FROM tenk1 WHERE unique1 < 7000;*

```
                      QUERY PLAN
------------------------------------------------------------
 Seq Scan on tenk1  (cost=0.00..483.00 rows=7033 width=244)
   Filter: (unique1 < 7000)
```

EXPLAIN 的输出显示，WHERE 子句被当作一个"filter"应用，这表示该规划节点将扫描表中的每一行数据，之后再判定它们是否符合过滤的条件，最后仅输出通过过滤条件的行数。这里由于 WHERE 子句的存在，预计的输出行数减少了。即便如此，扫描仍将访问所有 10000 行数据，因此开销并没有真正降低，实际上它还增加了一些因数据过滤而产生的额外 CPU 开销。

上面的数据只是一个预计数字，即使是在每次执行 ANALYZE 命令之后也会随之改变，因为 ANALYZE 生成的统计数据是通过从该表中随机抽取的样本计算的。

如果我们将上面查询的条件设置的更为严格一些的话，将会得到不同的查询规划，如：

*EXPLAIN SELECT * FROM tenk1 WHERE unique1 < 100;*

```
                            QUERY PLAN
------------------------------------------------------------------------------
 Bitmap Heap Scan on tenk1  (cost=2.37..232.35 rows=106 width=244)
   Recheck Cond: (unique1 < 100)
   ->  Bitmap Index Scan on tenk1_unique1  (cost=0.00..2.37 rows=106 width=0)
         Index Cond: (unique1 < 100)
```

这里，规划器决定使用两步规划，最内层的规划节点访问一个索引，找出匹配索引条件的行的位置，然后上层规划节点再从表里读取这些行。单独地读取数据行比顺序地读取它们的开销要高很多，但是因为并非访问该表的所有磁盘页面，因此该方法的开销仍然比一次顺序扫描的开销要少。这里使用两层规划的原因是因为上层规划节点把通过索引检索出来的行的物理位置先进行排序，这样可以最小化单独读取磁盘页面的开销。节点名称里面提到的"位图(bitmap)"是进行排序的机制。

现在我们还可以将 WHERE 的条件设置的更加严格，如：

*EXPLAIN SELECT * FROM tenk1 WHERE unique1 < 3;*

```
                            QUERY PLAN
------------------------------------------------------------------------------
 Index Scan using tenk1_unique1 on tenk1  (cost=0.00..10.00 rows=2 width=244)
   Index Cond: (unique1 < 3)
```

在该 SQL 中，表的数据行是以索引的顺序来读取的，这样就会令读取它们的开销变得更大，然而事实上这里将要获取的行数却少得可怜，因此没有必要在基于行的物理位置进行排序了。

现在我们需要向 WHERE 子句增加另外一个条件，如：

*EXPLAIN SELECT * FROM tenk1 WHERE unique1 < 3 AND stringu1 = 'xxx';*

```
                                QUERY PLAN
-------------------------------------------------------------------------------
 Index Scan using tenk1_unique1 on tenk1  (cost=0.00..10.01 rows=1 width=244)
   Index Cond: (unique1 < 3)
   Filter: (stringu1 = 'xxx'::name)
```

新增的过滤条件 stringu1 = 'xxx'只是减少了预计输出的行数，但是并没有减少实际开销，因为我们仍然需要访问相同数量的数据行。而该条件并没有作为一个索引条件，而是被当成对索引结果的过滤条件来看待。

如果 WHERE 条件里有多个字段存在索引，那么规划器可能会使用索引的 AND 或 OR 的组合，如：

*EXPLAIN SELECT * FROM tenk1 WHERE unique1 < 100 AND unique2 > 9000;*

```
                                QUERY PLAN
-------------------------------------------------------------------------------------
 Bitmap Heap Scan on tenk1  (cost=11.27..49.11 rows=11 width=244)
   Recheck Cond: ((unique1 < 100) AND (unique2 > 9000))
   ->  BitmapAnd  (cost=11.27..11.27 rows=11 width=0)
         ->  Bitmap Index Scan on tenk1_unique1  (cost=0.00..2.37 rows=106 width=0)
               Index Cond: (unique1 < 100)
         ->  Bitmap Index Scan on tenk1_unique2  (cost=0.00..8.65 rows=1042 width=0)
               Index Cond: (unique2 > 9000)
```

这样的结果将会导致访问两个索引，与只使用一个索引，而把另外一个条件只当作过滤器相比，这个方法未必是更优。

现在让我们来看一下基于索引字段进行表连接的查询规划，如：

*EXPLAIN SELECT * FROM tenk1 t1, tenk2 t2 WHERE t1.unique1 < 100 AND t1.unique2 = t2.unique2;*

```
                                QUERY PLAN
-------------------------------------------------------------------------------------
 Nested Loop  (cost=2.37..553.11 rows=106 width=488)
   ->  Bitmap Heap Scan on tenk1 t1  (cost=2.37..232.35 rows=106 width=244)
         Recheck Cond: (unique1 < 100)
         ->  Bitmap Index Scan on tenk1_unique1  (cost=0.00..2.37 rows=106 width=0)
               Index Cond: (unique1 < 100)
   ->  Index Scan using tenk2_unique2 on tenk2 t2  (cost=0.00..3.01 rows=1 width=244)
         Index Cond: ("outer".unique2 = t2.unique2)
```

从查询规划中可以看出(Nested Loop)该查询语句使用了嵌套循环。外层的扫描是一个位图索引，因此其开销与行计数和之前查询的开销是相同的，这是因为条件 unique1 < 100 发挥了作用。 这个时候 t1.unique2 = t2.unique2 条件子句还没有产生什么作用，因此它不会影响外层扫描的行计数。然而对于内层扫描而言，当前外层扫描的数据行将被插入到内层索引扫描中，并生成类似的条件 t2.unique2 = constant。所以，内层扫描将得到和 EXPLAIN SELECT * FROM tenk2 WHERE unique2 = 42 一样的计划和开销。最后，以外层扫描的开销为基础设置循环节点的开销，再加上每个外层行的一个迭代(这里是 106 * 3.01)，以及连接处理需要的一点点 CPU 时间。

如果不想使用嵌套循环的方式来规划上面的查询，那么我们可以通过执行以下系统设置，以关闭嵌套循环，如：

SET enable_nestloop = off;

*EXPLAIN SELECT * FROM tenk1 t1, tenk2 t2 WHERE t1.unique1 < 100 AND t1.unique2 = t2.unique2;*

```
                                QUERY PLAN
-------------------------------------------------------------------------------------
 Hash Join  (cost=232.61..741.67 rows=106 width=488)
```

```
Hash Cond: ("outer".unique2 = "inner".unique2)
->  Seq Scan on tenk2 t2  (cost=0.00..458.00 rows=10000 width=244)
->  Hash  (cost=232.35..232.35 rows=106 width=244)
      ->  Bitmap Heap Scan on tenk1 t1  (cost=2.37..232.35 rows=106 width=244)
            Recheck Cond: (unique1 < 100)
            ->  Bitmap Index Scan on tenk1_unique1  (cost=0.00..2.37 rows=106 width=0)
                  Index Cond: (unique1 < 100)
```

这个规划仍然试图用同样的索引扫描从 tenk1 里面取出符合要求的 100 行，并把它们存储在内存中的散列(哈希)表里，然后对 tenk2 做一次全表顺序扫描，并为每一条 tenk2 中的记录查询散列(哈希)表，寻找可能匹配 t1.unique2 = t2.unique2 的行。读取 tenk1 和建立散列表是此散列联接的全部启动开销，因为我们在开始读取 tenk2 之前不可能获得任何输出行。

此外，我们还可以用 EXPLAIN ANALYZE 命令检查规划器预估值的准确性。这个命令将先执行该查询，然后显示每个规划节点内实际运行时间，以及单纯 EXPLAIN 命令显示的预计开销，如：

*EXPLAIN ANALYZE SELECT * FROM tenk1 t1, tenk2 t2 WHERE t1.unique1 < 100 AND t1.unique2 = t2.unique2;*

```
                                                                  QUERY PLAN
-----------------------------------------------------------------------------------------------------------------------------------------
 Nested Loop  (cost=2.37..553.11 rows=106 width=488) (actual time=1.392..12.700 rows=100 loops=1)
   ->  Bitmap Heap Scan on tenk1 t1  (cost=2.37..232.35 rows=106 width=244) (actual time=0.878..2.367 rows=100 loops=1)
         Recheck Cond: (unique1 < 100)
         ->  Bitmap Index Scan on tenk1_unique1  (cost=0.00..2.37 rows=106 width=0) (actual time=0.546..0.546 rows=100 loops=1)
               Index Cond: (unique1 < 100)
   ->  Index Scan using tenk2_unique2 on tenk2 t2  (cost=0.00..3.01 rows=1 width=244) (actual time=0.067..0.078 rows=1 loops=100)
         Index Cond: ("outer".unique2 = t2.unique2)
 Total runtime: 14.452 ms
```

注意"actual time"数值是以真实时间的毫秒来计算的，而"cost"预估值是以磁盘页面读取数量来计算的，所以它们很可能是不一致的。然而我们需要关注的只是两组数据的比值是否一致。

在一些查询规划里，一个子规划节点很可能会运行多次，如之前的嵌套循环规划，内层的索引扫描会为每个外层行执行一次。在这种情况下，"loops"将报告该节点执行的总次数，而显示的实际时间和行数目则是每次执行的平均值。这么做的原因是令这些真实数值与开销预计显示的数值更具可比性。如果想获得该节点所花费的时间总数，计算方式是用该值乘以"loops"值。

EXPLAIN ANALYZE 显示的"Total runtime"包括执行器启动和关闭的时间，以及结果行处理的时间，但是它并不包括分析、重写或者规划的时间。

如果 EXPLAIN 命令仅能用于测试环境，而不能用于真实环境，那它就什么用都没有。比如，在一个数据较少的表上执行 EXPLAIN，它不能适用于数量很多的大表，因为规划器的开销计算不是线性的，因此它很可能对大些或者小些的表选择不同的规划。一个极端的例子是一个只占据一个磁盘页面的表，在这样的表上，不管它有没有索引可以使用，你几乎都总是得到顺序扫描规划。规划器知道不管在任何情况下它都要进行一个磁盘页面的读取，所以再增加几个磁盘页面读取用以查找索引是毫无意义的。

二、批量数据插入：

有以下几种方法用于优化数据的批量插入。

1. 关闭自动提交：

在批量插入数据时，如果每条数据都被自动提交，当中途出现系统故障时，不仅不能保障本次批量插入的数据一致性，而且由于有

多次提交操作的发生，整个插入效率也会受到很大的打击。解决方法是，关闭系统的自动提交，并且在插入开始之前，显示的执行 begin transaction 命令，在全部插入操作完成之后再执行 commit 命令提交所有的插入操作。

2. 使用 COPY：

使用 COPY 在一条命令里装载所有记录，而不是一系列的 INSERT 命令。COPY 命令是为装载数量巨大的数据行优化过的，它不像 INSERT 命令那样灵活，但是在装载大量数据时，系统开销也要少很多。因为 COPY 是单条命令，因此在填充表的时就没有必要关闭自动提交了。

3. 删除索引：

如果你正在装载一个新创建的表，最快的方法是创建表，用 COPY 批量装载，然后创建表需要的任何索引。因为在已存在数据的表上创建索引比维护逐行增加要快。当然在缺少索引期间，其它有关该表的查询操作的性能将会受到一定的影响，唯一性约束也有可能遭到破坏。

4. 删除外键约束：

和索引一样，"批量地"检查外键约束比一行行检查更加高效。因此，我们可以先删除外键约束，装载数据，然后在重建约束。

5. 增大 maintenance_work_mem：

在装载大量数据时，临时增大 maintenance_work_mem 系统变量的值可以改进性能。这个系统参数可以提高 CREATE INDEX 命令和 ALTER TABLE ADD FOREIGN KEY 命令的执行效率，但是它不会对 COPY 操作本身产生多大的影响。

6. 增大 checkpoint_segments：

临时增大 checkpoint_segments 系统变量的值也可以提高大量数据装载的效率。这是因为在向 PostgreSQL 装载大量数据时，将会导致检查点操作(由系统变量 checkpoint_timeout 声明)比平时更加频繁的发生。在每次检查点发生时，所有的脏数据都必须 flush 到磁盘上。通过提高 checkpoint_segments 变量的值，可以有效的减少检查点的数目。

7. 事后运行 ANALYZE：

在增加或者更新了大量数据之后，应该立即运行 ANALYZE 命令，这样可以保证规划器得到基于该表的最新数据统计。换句话说，如果没有统计数据或者统计数据太过陈旧，那么规划器很可能会选择一个较差的查询规划，从而导致查询效率过于低下。

PostgreSQL 学习手册(服务器配置)

一、服务器进程的启动和关闭：

下面是 pg_ctl 命令的使用方法和常用选项，需要指出的是，该命令是 postgres 命令的封装体，因此在使用上比直接使用 postgres 更加方便。

```
pg_ctl init[db] [-D DATADIR] [-s] [-o "OPTIONS"]
pg_ctl start    [-w] [-t SECS] [-D DATADIR] [-s] [-l FILENAME] [-o "OPTIONS"]
pg_ctl stop     [-W] [-t SECS] [-D DATADIR] [-s] [-m SHUTDOWN-MODE]
pg_ctl restart  [-w] [-t SECS] [-D DATADIR] [-s] [-m SHUTDOWN-MODE]
pg_ctl reload   [-D DATADIR] [-s]
pg_ctl status   [-D DATADIR]
pg_ctl promote [-D DATADIR] [-s]
```

选项	描述
-D	指定数据库存储的路径
-l	指定服务器进程的日志文件

-s	仅打印错误信息，不打印普通信息
-t SECS	当使用-w 选项时等待的秒数
-w	等待直到数据库操作完成(对于 stop 而言，该选项时缺省选项)
-W	不等待任何操作的完成
--help	显示帮助信息
--version	显示版本信息
-m	对于 stop 和 restart 操作，可以指定关闭模式
系统关闭模式	
smart	不在接收新的连接，直到当前已有的连接都断开之后才退出系统
fast	不在接收新的连接请求，主动关闭已经建立的连接，之后退出系统
immediate	立即退出，但是在 restart 的时候需要有恢复的操作被执行

这里我们只是给出最为常用的使用方式，即数据库服务器的正常启动和关闭。

#start 表示启动 postgres 服务器进程。

#-D 指定数据库服务器的初始目录的存放路径。

#-l 指定数据库服务器进程的日志文件

/> pg_ctl -w start -D /opt/PostgreSQL/9.1/data -l /opt/PostgreSQL/9.1/data/pg_log/startup.log

#stop 表示停止 postgres 服务器进程

#-m fast 在关闭系统时，使用 fast 的关闭模式。

/> pg_ctl stop -m fast -w -D /opt/PostgreSQL/9.1/data

二、服务器配置：

1. 设置参数：

在 PostgreSQL 中，所有配置参数名都是大小写不敏感的。每个参数都可以接受四种类型的值，它们分别是布尔、整数、浮点数和字符串。其中布尔值可以是 ON、OFF、TRUE、FALSE、YES、NO、1 和 0。包含这些参数的配置文件是 **postgresql.conf**，该文件通常存放在 initdb 初始化的数据(data)目录下，见如下配置片段：

```
# 这是一个注释
log_connections = yes
log_destination = 'syslog'
search_path = '$user, public'
```

井号(#)开始的行为注释行，如果配置值中包含数字，则需要用单引号括起。如果参数值本身包含单引号，我们可以写两个单引号(推荐方法)或用反斜扛包围。

这里需要注意的是，并非所有配置参数都可以在服务器运行时执行动态修改，有些参数在修改后，只能等到服务器重新启动后才能生效。

PostgreSQL 还提供了另外一种修改配置参数的方法，即在命令行上直接执行修改命令，如：

/> postgres -c log_connections=yes -c log_destination='syslog'

如果此时命令行设置的参数和配置文件中的参数相互冲突，那么命令行中给出的参数将覆盖配置文件中已有的参数值。除此之外，我们还可以通过 ALTER DATABASE 和 ALTER USER 等 PostgreSQL 的数据定义命令来分别修改指定数据库或指定用户的配置信息。其中针对数据库的设置将覆盖任何从 postgres 命令行或者配置文件从给出的设置，然后又会被针对用户的设置覆盖，最后又都会被每会话的选项覆盖。下面是当服务器配置出现冲突时，PostgreSQL 服务器将会采用哪种方式的优先级，如：

1). 基于会话的配置；

2). 基于用户的配置；

3). 基于数据库的配置；

4). postgres 命令行指定的配置；

5). 配置文件 postgresql.conf 中给出的配置。

最后需要说明的是，有些设置可以通过 PostgreSQL 的 set 命令进行设置，如在 psql 中我们可以输入：

SET ENABLE_SEQSCAN TO OFF;

也可以通过 show 命令来显示指定配置的当前值，如：

SHOW ENABLE_SEQSCAN;

与此同时，我们也可以手工查询 pg_settings 系统表的方式来检索感兴趣的系统参数。

三、内存相关的参数配置：

1. shared_buffers(integer)：

设置数据库服务器可以使用的共享内存数量。缺省情况下可以设置为 32MB，但是不要少于 128KB。因为该值设置的越高对系统的性能越有好处。该配置参数只能在数据库启动时设置。

此时，如果你有一台专用的数据库服务器，其内存为 1G 或者更多，那么我们推荐将该值设置为系统内存的 25%。

2. work_mem(integer)：

PostgreSQL 在执行排序操作时，会根据 work_mem 的大小决定是否将一个大的结果集拆分为几个小的和 work_mem 差不多大小的临时文件。显然拆分的结果是降低了排序的速度。因此增加 work_mem 有助于提高排序的速度。然而需要指出的是，如果系统中同时存在多个排序操作，那么每个操作在排序时使用的内存数量均为 work_mem，因此在我们设置该值时需要注意这一问题。

3. maintence_work_mem(integer)：

指定在维护性操作中使用的最大内存数，如 VACUUM、CREATE INDEX 和 ALTER TABLE ADD FOREIGN KEY 等，该配置的缺省值为 16MB。因为每个会话在同一时刻只能执行一个该操作，所以使用的频率不高，但是这些指令往往消耗较多的系统资源，因此应该尽快让这些指令快速执行完毕。

PostgreSQL 学习手册(角色和权限)

PostgreSQL 是通过角色来管理数据库访问权限的，我们可以将一个角色看成是一个数据库用户，或者一组数据库用户。角色可以拥有数据库对象，如表、索引，也可以把这些对象上的权限赋予其它角色，以控制哪些用户对哪些对象拥有哪些权限。

一、数据库角色：

1. 创建角色：

CREATE ROLE role_name;

2. 删除角色：

DROP ROLE role_name;

3. 查询角色：

检查系统表 pg_role，如：

SELECT usename FROM **pg_role**;

也可以在 psql 中执行\du 命令列出所有角色。

二、角色属性：

一个数据库角色可以有一系列属性，这些属性定义他的权限，以及与客户认证系统的交互。

1. 登录权限：

只有具有 LOGIN 属性的角色才可以用于数据库连接，因此我们可以将具有该属性的角色视为登录用户，创建方法有如下两种：

CREATE ROLE name **LOGIN PASSWORD** '123456';

CREATE USER name **PASSWORD** '123456';

2. 超级用户：

数据库的超级用户拥有该数据库的所有权限，为了安全起见，我们最好使用非超级用户完成我们的正常工作。和创建普通用户不同，创建超级用户必须是以超级用户的身份执行以下命令：

CREATE ROLE name **SUPERUSER**;

3. 创建数据库：

角色要想创建数据库，必须明确赋予创建数据库的属性，见如下命令：

CREATE ROLE name **CREATEDB**;

4. 创建角色：

一个角色要想创建更多角色，必须明确给予创建角色的属性，见如下命令：

CREATE ROLE name **CREATEROLE**;

三、权限：

数据库对象在被创建时都会被赋予一个所有者，通常而言，所有者就是执行对象创建语句的角色。对于大多数类型的对象，其初始状态是只有所有者(或超级用户)可以对该对象做任何事情。如果要允许其它用户可以使用该对象，必须赋予适当的权限。PostgreSQL 中预定义了许多不同类型的内置权限，如：**SELECT、INSERT、UPDATE、DELETE、RULE、REFERENCES、TRIGGER、CREATE、TEMPORARY、EXECUTE 和 USAGE**。

我们可以使用 GRANT 命令来赋予权限，如：

GRANT UPDATE ON accounts **TO** joe;

对于上面的命令，其含义为将 accounts 表的 update 权限赋予 joe 角色。此外，我们也可以用特殊的名字 PUBLIC 把对象的权限赋予系统中的所有角色。在权限声明的位置上写 ALL，表示把适用于该对象的所有权限都赋予目标角色。

要撤销权限，使用合适的 REVOKE 命令：

REVOKE ALL ON accounts **FROM PUBLIC**;

其含义为：对所有角色(PUBLIC)撤销在 accounts 对象上的所有权限(ALL)。

四、角色成员：

在系统的用户管理中，通常会把多个用户赋予一个组，这样在设置权限时只需给该组设置即可，撤销权限时也是从该组撤消。在 PostgreSQL 中，首先需要创建一个代表组的角色，之后再将该角色的 membership 权限赋给独立的用户角色即可。

1. 创建一个组角色，通常而言，该角色不应该具有 LOGIN 属性，如：

CREATE ROLE name;

2. 使用 GRANT 和 REVOKE 命令添加和撤消权限：

GRANT group_role TO role1, ... ;

REVOKE group_role FROM role1, ... ;

一个角色成员可以通过两种方法使用组角色的权限，如：

1. 每个组成员都可以用 SET ROLE 命令将自己临时"变成"该组成员，此后再创建的任何对象的所有者将属于该组，而不是原有的登录用户。

2. 拥有 INHERIT 属性的角色成员自动继承它们所属角色的权限。

见如下示例：

```
CREATE ROLE joe LOGIN INHERIT;  --INHERIT 是缺省属性。
CREATE ROLE admin NOINHERIT;
CREATE ROLE wheel NOINHERIT;
GRANT admin TO joe;
GRANT wheel TO admin;
```

现在我们以角色 joe 的身份与数据库建立连接，那么该数据库会话将同时拥有角色 joe 和角色 admin 的权限，这是因为 joe"继承(INHERIT)"了 admin 的权限。然而与此不同的是，赋予 wheel 角色的权限在该会话中将不可用，因为 joe 角色只是 wheel 角色的一个间接成员，它是通过 admin 角色间接传递过来的，而 admin 角色却含有 NOINHERIT 属性，这样 wheel 角色的权限将无法被 joe 继承。

这样 wheel 角色的权限将无法被 joe 继承。此时，我们可以在该会话中执行下面的命令：

```
SET ROLE admin;
```

在执行之后，该会话将只拥有 admin 角色的权限，而不再包括赋予 joe 角色的权限。同样，在执行下面的命令之后，该会话只能使用赋予 wheel 的权限。

```
SET ROLE wheel;
```

在执行一段时间之后，如果仍然希望将该会话恢复为原有权限，可以使用下列恢复方式之一：

```
SET ROLE joe;
SET ROLE NONE;
RESET ROLE;
```

注意：SET ROLE 命令总是允许选取当前登录角色的直接或间接组角色。因此，在变为 wheel 之前没必要先变成 admin。

角色属性 **LOGIN**、**SUPERUSER** 和 **CREATEROLE** 被视为特殊权限，它们不会像其它数据库对象的普通权限那样被继承。如果需要，必须在调用 SET ROLE 时显示指定拥有该属性的角色。比如，我们也可以给 admin 角色赋予 CREATEDB 和 CREATEROLE 权限，然后再以 joe 的角色连接数据库，此时该会话不会立即拥有这些特殊权限，只有当执行 SET ROLE admin 命令之后当前会话才具有这些权限。

要删除一个组角色，执行 **DROP ROLE group_role** 命令即可。然而在删除该组角色之后，它与其成员角色之间的关系将被立即撤销(成员角色本身不会受影响)。不过需要注意的是，在删除之前，任何属于该组角色的对象都必须先被删除或者将对象的所有者赋予其它角色，与此同时，任何赋予该组角色的权限也都必须被撤消。

PostgreSQL 学习手册(数据库管理) .

一、概述：

数据库可以被看成是 SQL 对象(数据库对象)的命名集合，通常而言，每个数据库对象(表、函数等)只属于一个数据库。不过对于部分系统表而言，如 pg_database，是属于整个集群的。更准确地说，数据库是模式的集合，而模式包含表、函数等 SQL 对象。因此完整的对象层次应该是这样的：服务器、数据库、模式、表或其他类型的对象。

在与数据库服务器建立连接时，该连接只能与一个数据库形成关联，不允许在一个会话中进行多个数据库的访问。如以 postgres 用户登录，该用户可以访问的缺省数据库为 postgres，在登录后如果执行下面的 SQL 语句将会收到 PostgreSQL 给出的相关错误信息。

```
postgres=# SELECT * FROM MyTest."MyUser".testtables;
ERROR:  cross-database references are not implemented: "otherdb.otheruser.sometable"
LINE 1: select * from otherdb.otheruser.sometable
```

在 PostgreSQL 中，数据库在物理上是相互隔离的，对它们的访问控制也是在会话层次上进行的。然而模式只是逻辑上的对象管理结构，是否能访问某个模式的对象是由权限系统来控制的。

执行下面的基于系统表的查询语句可以列出现有的数据库集合。

```
SELECT datname FROM pg_database;
```

注：**psql** 应用程序的**\l** 元命令和**-l** 命令行选项也可以用来列出当前服务器中已有的数据库。

二、创建数据库：

在 PostgreSQL 服务器上执行下面的 SQL 语句可以创建数据库。

CREATE DATABASE db_name;

在数据库成功创建之后，当前登录角色将自动成为此新数据库的所有者。在删除该数据库时，也需要该用户的特权。如果你想让当前创建的数据库的所有者为其它角色，可以执行下面的 SQL 语句。

CREATE DATABASE db_name **OWNER** role_name;

三、修改数据库配置：

PostgreSQL 服务器提供了大量的运行时配置变量，我们可以根据自己的实际情况为某一数据库的某一配置变量指定特殊值，通过执行下面的 SQL 命令可以使该数据库的某一配置被设置为指定值，而不再使用缺省值。

ALTER DATABASE db_name **SET** varname **TO** new_value;

这样在之后基于该数据库的会话中，被修改的配置值已经生效。如果要撤消这样的设置并恢复为原有的缺省值，可以执行下面的 SQL 命令。

ALTER DATABASE dbname **RESET** varname;

四、删除数据库：

只有数据库的所有者和超级用户可以删除数据库。删除数据库将会删除数据库中包括的所有对象，该操作是不可恢复的。见如下删除 SQL 命令：

DROP DATABASE db_name;

五、表空间：

在 PostgreSQL 中，表空间表示一组文件存放的目录位置。在创建之后，就可以在该表空间上创建数据库对象。通过使用表空间，管理员可以控制一个 PostgreSQL 服务器的磁盘布局。这样管理员就可以根据数据库对象的数据量和数据使用频度等参照来规划这些对象的存储位置，以便减少 IO 等待，从而优化系统的整体运行性能。比如，将一个使用频繁的索引放在非常可靠、高效的磁盘设备上，如固态硬盘。而将很少使用的数据库对象存放在相对较慢的磁盘系统上。下面的 SQL 命令用于创建表空间。

CREATE TABLESPACE fastspace **LOCATION** '/mnt/sda1/postgresql/data';

需要说明的是，表空间指定的位置必须是一个现有的空目录，且属于 PostgreSQL 系统用户，如 postgres。在成功创建之后，所有在该表空间上创建的对象都将被存放在这个目录下的文件里。

在 PostgreSQL 中只有超级用户可以创建表空间，但是在成功创建之后，就可以允许普通数据库用户在其上创建数据库对象了。要完成此操作，必须在表空间上给这些用户授予 CREATE 权限。表、索引和整个数据库都可以放在特定的表空间里。见如下 SQL 命令：

CREATE TABLE foo(i int) **TABLESPACE** space1;

此外，我们还可以通过修改 **default_tablespace** 配置变量，以使指定的表空间成为缺省表空间，这样在创建任何数据库对象时，如果没有显示指定表空间，那么该对象将被创建在缺省表空间中，如：

```
SET default_tablespace = space1;
CREATE TABLE foo(i int);
```

与数据库相关联的表空间用于存储该数据库的系统表，以及任何使用该数据库的服务器进程创建的临时文件。

要删除一个空的表空间，可以直接使用 **DROP TABLESPACE** 命令，然而要删除一个包含数据库对象的表空间，则需要先将该表空间上的所有对象全部删除后，才可以再在删除该表空间。

要检索当前系统中有哪些表空间，可以执行以下查询，其中 pg_tablespace 为 PostgreSQL 中的系统表。

SELECT spcname FROM **pg_tablespace**;

我们还可以通过 **psql** 程序的**\db** 元命令列出现有的表空间。

PostgreSQL 学习手册(数据库维护)

一、恢复磁盘空间：

在 PostgreSQL 中，使用 delete 和 update 语句删除或更新的数据行并没有被实际删除，而只是在旧版本数据行的物理地址上将该行的状态置为已删除或已过期。因此当数据表中的数据变化极为频繁时，那么在一段时间之后该表所占用的空间将会变得很大，然而数据量却可能变化不大。要解决该问题，需要定期对数据变化频繁的数据表执行 VACUUM 操作。

VACUUM 命令存在两种形式，**VACUUM** 和 **VACUUM FULL**，它们之间的区别见如下表格：

	无 VACUUM	VACUUM	VACUUM FULL
删除大量数据之后	只是将删除数据的状态置为已删除，该空间不能记录被重新使用。	如果删除的记录位于表的末端，其所占用的空间将会被物理释放并归还操作系统。如果不是末端数据，该命令会将指定表或索引中被删除数据所占用空间重新置为可用状态，那么在今后有新数据插入时，将优先使用该空间，直到所有被重用的空间用完时，再考虑使用新增的磁盘页面。	不论被删除的数据是否处于数据表的末端，这些数据所占用的空间都将被物理的释放并归还于操作系统。之后再有新数据插入时，将分配新的磁盘页面以供使用。
执行效率		由于只是状态置为操作，因此效率较高。	在当前版本的 PostgreSQL(v9.1)中，该命令会为指定的表或索引重新生成一个数据文件，并将原有文件中可用的数据导入到新文件中，之后再删除原来的数据文件。因此在导入过程中，要求当前磁盘有更多的空间可用于此操作。由此可见，该命令的执行效率相对较低。
被删除的数据所占用的物理空间是否被重新规划给操作系统。	不会	不会	会
在执行 VACUUM 命令时，是否可以并发执行针对该表的其他操作。		由于该操作是共享锁，因此可以与其他操作并行进行。	由于该操作需要在指定的表上应用排它锁，因此在执行该操作期间，任何基于该表的操作都将被挂起，知道该操作完成。
推荐使用方式	在进行数据清空是，可以使用 truncate 操作，因为该操作将会物理的清空数据表，并将其所占用的	为了保证数据表的磁盘页面数量能够保持在一个相对稳定值，可以定期执行该操作，如每天或每周中数据操作相对较少的时段。	考虑到该操作的开销，以及对其他错误的排斥，推荐的方式是，定期监控数据量变化较大的表，只有确认其磁盘页面占有量接近临界值时，才考虑执行一次该操作。即便如此，也需要注意尽量选

	空间直接归还于操作系统。		择数据操作较少的时段来完成该操作。
执行后其它操作的效率	对于查询而言，由于存在大量的磁盘页面碎片，因此效率会逐步降低。	相比于不执行任何 VACUUM 操作，其效率更高，但是插入的效率会有所降低。	在执行完该操作后，所有基于该表的操作效率都会得到极大的提升。

二、更新规划器统计：

PostgreSQL 查询规划器在选择最优路径时，需要参照相关数据表的统计信息用以为查询生成最合理的规划。这些统计是通过 ANALYZE 命令获得的，你可以直接调用该命令，或者把它当做 VACUUM 命令里的一个可选步骤来调用，如 **VACUUM ANAYLYZE** table_name，该命令将会先执行 VACUUM 再执行 ANALYZE。与回收空间(VACUUM)一样，对数据更新频繁的表保持一定频度的 ANALYZE，从而使该表的统计信息始终处于相对较新的状态，这样对于基于该表的查询优化将是极为有利的。然而对于更新并不频繁的数据表，则不需要执行该操作。

我们可以为特定的表，甚至是表中特定的字段运行 ANALYZE 命令，这样我们就可以根据实际情况，只对更新比较频繁的部分信息执行 ANALYZE 操作，这样不仅可以节省统计信息所占用的空间，也可以提高本次 ANALYZE 操作的执行效率。这里需要额外说明的是，ANALYZE 是一项相当快的操作，即使是在数据量较大的表上也是如此，因为它使用了统计学上的随机采样的方法进行行采样，而不是把每一行数据都读取进来并进行分析。因此，可以考虑定期对整个数据库执行该命令。

事实上，我们甚至可以通过下面的命令来调整指定字段的抽样率，如：

ALTER TABLE testtable **ALTER COLUMN** test_col **SET STATISTICS** 200

注意：该值的取值范围是 **0--1000**，其中值越低采样比例就越低，分析结果的准确性也就越低，但是 ANALYZE 命令执行的速度却更快。如果将该值设置为**-1**，那么该字段的采样比率将恢复到系统当前默认的采样值，我们可以通过下面的命令获取当前系统的缺省采样值。

```
postgres=# show default_statistics_target;
 default_statistics_target
---------------------------
 100
(1 row)
```

从上面的结果可以看出，该数据库的缺省采样值为 100(10%)。

三、VACUUM 和 ANALYZE 的示例：

```
#1. 创建测试数据表。
postgres=# CREATE TABLE testtable (i integer);
CREATE TABLE
#2. 为测试表创建索引。
postgres=# CREATE INDEX testtable_idx ON testtable(i);
CREATE INDEX
#3. 创建批量插入测试数据的函数。
postgres=# CREATE OR REPLACE FUNCTION test_insert() returns integer AS $$
DECLARE
    min integer;
    max integer;
BEGIN
    SELECT COUNT(*) INTO min from testtable;
```

```
        max := min + 10000;
        FOR i IN min..max LOOP
            INSERT INTO testtable VALUES(i);
        END LOOP;
        RETURN 0;
    END;
    $$ LANGUAGE plpgsql;
    CREATE FUNCTION
    #4. 批量插入数据到测试表(执行四次)
    postgres=# SELECT test_insert();
     test_insert
    -------------
              0
    (1 row)
    #5. 确认四次批量插入都成功。
    postgres=# SELECT COUNT(*) FROM testtable;
     count
    -------
     40004
    (1 row)
    #6. 分析测试表，以便有关该表的统计信息被更新到PostgreSQL 的系统表。
    postgres=# ANALYZE testtable;
    ANALYZE
    #7. 查看测试表和索引当前占用的页面数量(通常每个页面为8k)。
    postgres=# SELECT relname,relfilenode, relpages FROM pg_class WHERE relname = 'testtable' or relname =
'testtable_idx';
         relname       | relfilenode    | relpages
    ---------------+-------------+----------
     testtable       |      17601   |      157
     testtable_idx  |      17604   |       90
    #8. 批量删除数据。
    postgres=# DELETE FROM testtable WHERE i < 30000;
    DELETE 30003
    #9. 执行vacuum 和analyze，以便更新系统表，同时为该表和索引记录高水标记。
    #10. 这里需要额外说明的是，上面删除的数据均位于数据表的前部，如果删除的是末尾部分，
    #       如where i > 10000，那么在执行VACUUM ANALYZE 的时候，数据表将会被物理的缩小。
    postgres=# VACUUM ANALYZE testtable;
    ANALYZE
    #11. 查看测试表和索引在删除后，再通过VACUUM ANALYZE 更新系统统计信息后的结果(保持不变)。
    postgres=# SELECT relname,relfilenode, relpages FROM pg_class WHERE relname = 'testtable' or relname =
'testtable_idx';
         relname       | relfilenode     | relpages
    ---------------+-------------+----------
     testtable       |      17601   |      157
     testtable_idx  |      17604   |       90
    (2 rows)
    #12. 再重新批量插入两次，之后在分析该表以更新其统计信息。
    postgres=# SELECT test_insert(); --执行两次。
     test_insert
```

```
------------
         0
(1 row)
postgres=# ANALYZE testtable;
ANALYZE
#13. 此时可以看到数据表中的页面数量仍然为之前的高水标记数量，索引页面数量的增加
#       是和其内部实现方式有关，但是在后面的插入中，索引所占的页面数量就不会继续增加。
postgres=# SELECT relname,relfilenode, relpages FROM pg_class WHERE relname = 'testtable' or relname =
'testtable_idx';
      relname       | relfilenode    | relpages
--------------+-------------+----------
 testtable        |      17601   |     157
 testtable_idx  |      17604   |     173
(2 rows)
postgres=# SELECT test_insert();
 test_insert
------------
         0
(1 row)
postgres=# ANALYZE testtable;
ANALYZE
#14. 可以看到索引的页面数量确实没有继续增加。
postgres=# SELECT relname,relfilenode, relpages FROM pg_class WHERE relname = 'testtable' or relname =
'testtable_idx';
      relname       | relfilenode    | relpages
--------------+-------------+----------
 testtable        |      17601   |     157
 testtable_idx  |      17604   |     173
(2 rows)
#15. 重新批量删除数据。
postgres=# DELETE FROM testtable WHERE i < 30000;
DELETE 19996
#16. 从后面的查询可以看出，在执行VACUUM FULL 命令之后，测试表和索引所占用的页面数量
#       确实降低了，说明它们占用的物理空间已经缩小了。
postgres=# VACUUM FULL testtable;
VACUUM
postgres=# SELECT relname,relfilenode, relpages FROM pg_class WHERE relname = 'testtable' or relname =
'testtable_idx';
      relname       | relfilenode    | relpages
--------------+-------------+----------
 testtable        |      17602   |     118
 testtable_idx  |      17605   |     68
(2 rows)
```

四、定期重建索引：

在 PostgreSQL 中，为数据更新频繁的数据表定期重建索引**(REINDEX INDEX)**是非常有必要的。对于 B-Tree 索引，只有那些已经完全清空的索引页才会得到重复使用，对于那些仅部分空间可用的索引页将不会得到重用，如果一个页面中大多数索引键值都

被删除，只留下很少的一部分，那么该页将不会被释放并重用。在这种极端的情况下，由于每个索引页面的利用率极低，一旦数据量显著增加，将会导致索引文件变得极为庞大，不仅降低了查询效率，而且还存在整个磁盘空间被完全填满的危险。

对于重建后的索引还存在另外一个性能上的优势，因为在新建立的索引上，逻辑上相互连接的页面在物理上往往也是连在一起的，这样可以提高磁盘页面被连续读取的几率，从而提高整个操作的 IO 效率。见如下示例：

```
#1. 此时已经在该表中插入了大约 6 万条数据，下面的 SQL 语句将查询该索引所占用的磁盘空间。
postgres=# SELECT relname, pg_relation_size(oid)/1024 || 'K' AS size FROM pg_class WHERE relkind='i' AND relname = 'testtable_idx';
     relname      | size
----------------+------
 testtable_idx | 1240K
(1 row)
#2. 删除数据表中大多数的数据。
postgres=# DELETE FROM testtable WHERE i > 20000;
DELETE 50006
#3. 分析一个该表，以便于后面的 SQL 语句继续查看该索引占用的空间。
postgres=# ANALYZE testtable;
ANALYZE
#4. 从该查询结果可以看出，该索引所占用的空间并未减少，而是和之前的完全一样。
postgres=# SELECT pg_relation_size('testtable_idx')/1024 || 'K' AS size;
 size
------
 1240K
(1 row)
#5. 重建索引。
postgres=# REINDEX INDEX testtable_idx;
REINDEX
#6. 查看重建后的索引实际占用的空间，从结果中可以看出索引的尺寸已经减少。
postgres=# SELECT pg_relation_size('testtable_idx')/1024 || 'K' AS size;
 size
------
 368K
(1 row)
#7. 最后一点需要记住的是，在索引重建后一定要分析数据表。
postgres=# ANALYZE testtable;
ANALYZE
```

五、观察磁盘使用情况：

1. 查看数据表所占用的磁盘页面数量。

```
#relpages 只能被 VACUUM、ANALYZE 和几个 DDL 命令更新，如 CREATE INDEX。通常一个页面的长度为 8K 字节。
postgres=# SELECT relfilenode, relpages FROM pg_class WHERE relname = 'testtable';
 relfilenode | relpages
-------------+----------
       16412 |       79
(1 row)
```

2. 查看指定数据表的索引名称和索引占用的磁盘页面数量。

```
postgres=# SELECT c2.relname, c2.relpages FROM pg_class c, pg_class c2, pg_index i
```

```
    WHERE c.relname = 'testtable' AND c.oid = i.indrelid AND c2.oid = i.indexrelid
    ORDER BY c2.relname;
     relname     | relpages
-----------------+----------
 testtable_idx |       46
(1 row)
```

PostgreSQL 学习手册(系统表)

一、pg_class:

该系统表记录了数据表、索引(仍然需要参阅 pg_index)、序列、视图、复合类型和一些特殊关系类型的元数据。注意：不是所有字段对所有对象类型都有意义。

名字	类型	引用	描述
relname	name		数据类型名字。
relnamespace	oid	pg_namespace.oid	包含这个对象的名字空间(模式)的 OI。
reltype	oid	pg_type.oid	对应这个表的行类型的数据类型。
relowner	oid	pg_authid.oid	对象的所有者。
relam	oid	pg_am.oid	对于索引对象，表示该索引的类型(B-tree，hash)。
relfilenode	oid		对象存储在磁盘上的文件名，如果没有则为 0。
reltablespace	oid	pg_tablespace.oid	对象所在的表空间。如果为零，则表示使用该数据库的缺省表空间。(如果对象在磁盘上没有文件，这个字段就没有什么意义)
relpages	int4		该数据表或索引所占用的磁盘页面数量，查询规划器会借助该值选择最优路径。
reltuples	float4		表中行的数量，该值只是被规划器使用的一个估计值。
reltoastrelid	oid	pg_class.oid	与此表关联的 TOAST 表的 OID，如果没有为 0。TOAST 表在一个从属表里"离线"存储大字段。
reltoastidxid	oid	pg_class.oid	如果是 TOAST 表，该字段为它索引的 OID，如果不是 TOAST 表则为 0。
relhasindex	bool		如果这是一个数据表而且至少有(或者最近有过)一个索引，则为真。它是由 CREATE INDEX 设置的，但 DROP INDEX 不会立即将它清除。如果 VACUUM 发现一个表没有索引，那么它清理 relhasindex。
relisshared	bool		如果该表在整个集群中由所有数据库共享，则为真。
relkind	char		r = 普通表，i = 索引，S = 序列，v = 视图， c = 复合类型，s = 特殊，t = TOAST 表
relnatts	int2		数据表中用户字段的数量(除了系统字段以外，如 oid)。在 pg_attribute 里肯定有相同数目的数据行。见 pg_attribute.attnum.
relchecks	int2		表中检查约束的数量，参阅 pg_constraint 表。
reltriggers	int2		表中触发器的数量；参阅 pg_trigger 表。
relhasoids	bool		如果我们为对象中的每行都生成一个 OID，则为真。
relhaspkey	bool		如果该表存在主键，则为真。
relhasrules	bool		如表有规则就为真；参阅 pg_rewrite 表。

relhassubclass	bool		如果该表有子表，则为真。
relacl	aclitem[]		访问权限。

见如下应用示例：

 # 查看指定表对象 testtable 的模式

postgres=# SELECT relname,relnamespace,nspname FROM pg_class c,pg_namespace n WHERE relname = 'testtable' AND relnamespace = n.oid;

```
 relname   | relnamespace | nspname
-----------+--------------+---------
 testtable |         2200 | public
(1 row)
```

 # 查看指定表对象 testtable 的 owner(即 role)。

postgres=# select relname,rolname from pg_class c,pg_authid au where relname = 'testtable' and relowner = au.oid;

```
 relname   | rolname
-----------+----------
 testtable | postgres
(1 row)
```

二、pg_attribute:

该系统表存储所有表(包括系统表，如 pg_class)的字段信息。数据库中的每个表的每个字段在 pg_attribute 表中都有一行记录。

名字	类型	引用	描述
attrelid	oid	pg_class.oid	此字段所属的表。
attname	name		字段名。
atttypid	oid	pg_type.oid	字段的数据类型。
attstattarget	int4		attstattarget 控制 ANALYZE 为这个字段设置的统计细节的级别。零值表示不收集统计信息，负数表示使用系统缺省的统计对象。正数值的确切信息是和数据类型相关的。
attlen	int2		该字段所属类型的长度。(pg_type.typlen 的拷贝)
attnum	int2		字段的编号，普通字段是从 1 开始计数的。系统字段，如 oid，是任意的负数。
attndims	int4		如果该字段是数组，该值表示数组的维数，否则是 0。
attcacheoff	int4		在磁盘上总是-1，但是如果装载入内存中的行描述器中， 它可能会被更新为缓冲在行中字段的偏移量。
atttypmod	int4		表示数据表在创建时提供的类型相关的数据(比如，varchar 字段的最大长度)。其值对那些不需要 atttypmod 的类型而言通常为-1。
attbyval	bool		pg_type. typbyval 字段值的拷贝。
attstorage	char		pg_type.typstorage 字段值的拷贝。
attalign	char		pg_type.typalign 字段值的拷贝。
attnotnull	bool		如果该字段带有非空约束，则为真，否则为假。
atthasdef	bool		该字段是否存在缺省值，此时它对应 pg_attrdef 表里实际定义此值的记录。
attisdropped	bool		该字段是否已经被删除。如果被删除，该字段在物理上仍然存在表中，但会被分析器忽略，因此不能再通过 SQL 访问。

attislocal	bool		该字段是否局部定义在对象中的。
attinhcount	int4		该字段所拥有的直接祖先的个数。如果一个字段的祖先个数非零，那么它就不能被删除或重命名。

见如下应用示例：

#查看指定表中包含的字段名和字段编号。

postgres=# SELECT relname, attname,attnum FROM pg_class c,pg_attribute attr WHERE relname = 'testtable' AND c.oid = attr.attrelid;

```
 relname   | attname   | attnum
-----------+----------+--------
 testtable   | tableoid   |     -7
 testtable   | cmax       |     -6
 testtable   | xmax       |     -5
 testtable   | cmin       |     -4
 testtable   | xmin       |     -3
 testtable   | ctid       |     -1
 testtable   | i          |      1
(7 rows)
```

#只查看用户自定义字段的类型

postgres=# SELECT relname,attname,typname FROM pg_class c,pg_attribute a,pg_type t WHERE c.relname = 'testtable' AND c.oid = attrelid AND atttypid = t.oid AND attnum > 0;

```
 relname   | attname   | typname
-----------+----------+---------
 testtable   | i          | int4
(7 rows)
```

三、pg_attrdef:

该系统表主要存储字段缺省值，字段中的主要信息存放在 pg_attribute 系统表中。注意：只有明确声明了缺省值的字段在该表中才会有记录。

名字	类型	引用	描述
adrelid	oid	pg_class.oid	这个字段所属的表
adnum	int2	pg_attribute.attnum	字段编号，其规则等同于 pg_attribute.attnum
adbin	text		字段缺省值的内部表现形式。
adsrc	text		缺省值的人可读的表现形式。

见如下应用示例：

#查看指定表有哪些字段存在缺省值，同时显示出字段名和缺省值的定义方式

postgres=# CREATE TABLE testtable2 (i integer DEFAULT 100);

```
CREATE TABLE
```

postgres=# SELECT c.relname, a.attname, ad.adnum, ad.adsrc FROM pg_class c, pg_attribute a, pg_attrdef ad WHERE relname = 'testtable2' AND ad.adrelid = c.oid AND adnum = a.attnum AND attrelid = c.oid;

```
 relname    | attname | adnum  | adsrc
------------+---------+--------+-------
 testtable2  | i          |      1  | 100
(1 row)
```

四、pg_authid:

该系统表存储有关数据库认证的角色信息，在 PostgreSQL 中角色可以表现为用户和组两种形式。对于用户而言只是设置了 rolcanlogin 标志的角色。由于该表包含口令数据，所以它不是公共可读的。PostgreSQL 中提供了另外一个建立在该表之上的系统视图 pg_roles，该视图将口令字段填成空白。

名字	类型	引用	描述
rolname	name		角色名称。
rolsuper	bool		角色是否拥有超级用户权限。
rolcreaterole	bool		角色是否可以创建其它角色。
rolcreatedb	bool		角色是否可以创建数据库。
rolcatupdate	bool		角色是否可以直接更新系统表(如果该设置为假，即使超级用户也不能更新系统表)。
rolcanlogin	bool		角色是否可以登录，换句话说，这个角色是否可以给予会话认证标识符。
rolpassword	text		口令(可能是加密的)；如果没有则为 NULL。
rolvaliduntil	timestamptz		口令失效时间(只用于口令认证)；如果没有失效期，则为 NULL。
rolconfig	text[]		运行时配置变量的会话缺省。

见如下应用示例：

```
#从输出结果可以看出口令字段已经被加密。
postgres=# SELECT rolname,rolpassword FROM pg_authid;
 rolname  |             rolpassword
----------+-------------------------------------
 postgres | md5a3556571e93b0d20722ba62be61e8c2d
```

五、pg_auth_members:

该系统表存储角色之间的成员关系。

名字	类型	引用	描述
roleid	oid	pg_authid.oid	组角色的 ID。
member	oid	pg_authid.oid	属于组角色 roleid 的成员角色的 ID。
grantor	oid	pg_authid.oid	赋予此成员关系的角色的 ID。
admin_option	bool		如果具有把其它成员角色加入组角色的权限，则为真。

见如下应用示例：

```
#1. 先查看角色成员表中有哪些角色之间的隶属关系，在当前结果集中只有一个成员角色隶属于一个组角色，
#    如果有多个成员角色隶属于同一个组角色，这样将会有多条记录。
postgres=# SELECT * FROM pg_auth_members ;
 roleid  | member | grantor | admin_option
--------+--------+---------+--------------
  16446 |  16445  |     10   | f
(1 row)
#2. 查看组角色的名字。
```

```
postgres=# SELECT rolname FROM pg_authid a,pg_auth_members am WHERE a.oid = am.roleid;
 rolname
---------
 mygroup
(1 row)
#3. 查看成员角色的名字。
#4. 如果需要用一个结果集获取角色之间的隶属关系，可以将这两个结果集作为子查询后再进行关联。
postgres=# SELECT rolname FROM pg_authid a,pg_auth_members am WHERE a.oid = am.member;
 rolname
---------
 myuser
(1 row)
```

六、pg_constraint:

该系统表存储 PostgreSQL 中表对象的检查约束、主键、唯一约束和外键约束。

名字	类型	引用	描述
conname	name		约束名字(不一定是唯一的)。
connamespace	oid	pg_namespace.oid	包含这个约束的名字空间(模式)的 OID。
contype	char		c = 检查约束， f = 外键约束， p = 主键约束， u = 唯一约束
condeferrable	bool		该约束是否可以推迟。
condeferred	bool		缺省时这个约束是否是推迟的？
conrelid	oid	pg_class.oid	该约束所在的表，如果不是表约束则为 0。
contypid	oid	pg_type.oid	该约束所在的域，如果不是域约束则为 0。
confrelid	oid	pg_class.oid	如果为外键，则指向参照的表，否则为 0。
confupdtype	char		外键更新动作代码。
confdeltype	char		外键删除动作代码。
confmatchtype	char		外键匹配类型。
conkey	int2[]	pg_attribute.attnum	如果是表约束，则是约束控制的字段列表。
confkey	int2[]	pg_attribute.attnum	如果是外键，则是参照字段的列表。
conbin	text		如果是检查约束，则表示表达式的内部形式。
consrc	text		如果是检查约束，则是表达式的人可读的形式。

七、pg_tablespace:

该系统表存储表空间的信息。注意：表可以放在特定的表空间里，以帮助管理磁盘布局和解决 IO 瓶颈。

名字	类型	引用	描述
spcname	name		表空间名称。
spcowner	oid	pg_authid.oid	表空间的所有者，通常是创建它的角色。
spclocation	text		表空间的位置(目录路径)。

spcacl	aclitem[]		访问权限。

见如下应用示例：

```
#1. 创建表空间。
postgres=# CREATE TABLESPACE my_tablespace LOCATION '/opt/PostgreSQL/9.1/mydata';
CREATE TABLESPACE
#2. 将新建表空间的 CREATE 权限赋予 public。
postgres=# GRANT CREATE ON TABLESPACE my_tablespace TO public;
GRANT
#3. 查看系统内用户自定义表空间的名字、文件位置和创建它的角色名称。
#4. 系统创建时自动创建的两个表空间(pg_default 和 pg_global)的文件位置为空(不是 NULL)。
postgres=# SELECT spcname,rolname,spclocation FROM pg_tablespace ts,pg_authid a WHERE ts.spcowner = a.oid AND spclocation <> '';
    spcname    | rolname  |        spclocation
---------------+----------+---------------------------
 my_tablespace | postgres | /opt/PostgreSQL/9.1/mydata
(1 row)
```

八、pg_namespace:

该系统表存储名字空间(模式)。

名字	类型	引用	描述
nspname	name		名字空间(模式)的名称。
nspowner	oid	pg_authid.oid	名字空间(模式)的所有者
nspacl	aclitem[]		访问权限。

见如下应用示例：

```
# 查看当前数据库 public 模式的创建者的名称。
postgres=# SELECT nspname,rolname FROM pg_namespace n, pg_authid a WHERE nspname = 'public' AND nspowner = a.oid;
 nspname | rolname
---------+----------
 public  | postgres
(1 row)
```

九、pg_database:

该系统表存储数据库的信息。和大多数系统表不同的是，在一个集群里该表是所有数据库共享的，即每个集群只有一份 pg_database 拷贝，而不是每个数据库一份。

名字	类型	引用	描述
datname	name		数据库名称。
datdba	oid	pg_authid.oid	数据库所有者，通常为创建该数据库的角色。
encoding	int4		数据库的字符编码方式。
datistemplate	bool		如果为真，此数据库可以用于 CREATE DATABASE TEMPLATE 子句，把新数据库创建为此数

			据库的克隆。
datallowconn	bool		如果为假，则没有人可以联接到这个数据库。
datlastsysoid	oid		数据库里最后一个系统 OID，此值对 pg_dump 特别有用。
datvacuumxid	xid		
datfrozenxid	xid		
dattablespace	text	pg_tablespace.oid	该数据库的缺省表空间。在这个数据库里，所有 pg_class.reltablespace 为零的表都将保存在这个表空间里，特别要指出的是，所有非共享的系统表也都存放在这里。
datconfig	text[]		运行时配置变量的会话缺省值。
datacl	aclitem[]		访问权限。

十、pg_index:

该系统表存储关于索引的一部分信息。其它的信息大多数存储在 pg_class。

名字	类型	引用	描述
indexrelid	oid	pg_class.oid	该索引在 pg_class 里的记录的 OID。
indrelid	oid	pg_class.oid	索引所在表在 pg_class 里的记录的 OID。
indnatts	int2		索引中的字段数量(拷贝的 pg_class.relnatts)。
indisunique	bool		如果为真，该索引是唯一索引。
indisprimary	bool		如果为真，该索引为该表的主键。
indisclustered	bool		如果为真，那么该表在这个索引上建了簇。
indkey	int2vector	pg_attribute.attnum	该数组的元素数量为 indnatts，数组元素值表示建立这个索引时所依赖的字段编号，如 1 3，表示第一个字段和第三个字段构成这个索引的键值。如果为 0，则表示是表达式索引，而不是基于简单字段的索引。
indclass	oidvector	pg_opclass.oid	对于构成索引键值的每个字段，这个字段都包含一个指向所使用的操作符表的 OID。
indexprs	text		表达式树用于那些非简单字段引用的索引属性。它是一个列表，在 indkey 里面的每个零条目一个元素。如果所有索引属性都是简单的引用，则为空。
indpred	text		部分索引断言的表达式树。如果不是部分索引， 则是空字串。

见如下应用示例：

 # 查看该索引所在表的名称，以及构成该索引的键值数量和具体键值的字段编号。

postgres=# SELECT indnatts,indkey,relname FROM pg_index i, pg_class c WHERE c.relname = 'testtable2' AND indrelid = c.oid;

```
 indnatts | indkey |  relname
----------+--------+------------
        2 | 1 3    | testtable2
(1 row)
```

 # 查看指定表包含的索引，同时列出索引的名称。

postgres=# SELECT t.relname AS table_name, c.relname AS index_name FROM (SELECT relname,indexrelid FROM pg_index i, pg_class c WHERE c.relname = 'testtable2' AND indrelid = c.oid) t, pg_index i,pg_class c WHERE t.indexrelid = i.indexrelid AND i.indexrelid = c.oid;

```
 table_name |  index_name
------------+----------------
```

testtable2 | testtable2_idx
(1 row)

PostgreSQL 学习手册(系统视图)

一、pg_tables:

该视图提供了对有关数据库中每个表的有用信息地访问。

名字	类型	引用	描述
schemaname	name	pg_namespace.nspname	包含表的模式名字。
tablename	name	pg_class.relname	表的名字。
tableowner	name	pg_authid.rolname	表的所有者的名字。
tablespace	name	pg_tablespace.spcname	包含表的表空间名字(如果是数据库缺省，则为 NULL)。
hasindexes	bool	pg_class.relhasindex	如果表拥有(或者最近拥有)任何索引，则为真。
hasrules	bool	pg_class.relhasrules	如果表存在规则，则为真。
hastriggers	bool	pg_class.reltriggers	如果表有触发器，则为真。

二、pg_indexes:

该视图提供对数据库中每个索引的有用信息的访问。

名字	类型	引用	描述
schemaname	name	pg_namespace.nspname	包含表和索引的模式的名字。
tablename	name	pg_class.relname	索引所在表的名字。
indexname	name	pg_class.relname	索引的名字。
tablespace	name	pg_tablespace.spcname	包含索引的表空间名字(如果是数据库缺省，则为 NULL)。
indexdef	text		索引定义(一个重建的创建命令)。

三、pg_views:

该视图提供了对数据库里每个视图的有用信息的访问途径。

名字	类型	引用	描述
schemaname	name	pg_namespace.nspname	包含此视图的模式名字。
viewname	name	pg_class.relname	视图的名字。
viewowner	name	pg_authid.rolname	视图的所有者的名字。
definition	text		视图定义(一个重建的 SELECT 查询)。

四、pg_user:

该视图提供了对数据库用户的相关信息的访问。这个视图只是 pg_shadow 表的公众可读的部分的视图化，但是不包含口令字段。

名字	类型	引用	描述
usename	name		用户名。
usesysid	int4		用户 ID(用于引用这个用户的任意数字)。
usecreatedb	bool		用户是否可以创建数据库。
usesuper	bool		用户是否是一个超级用户。
usecatupd	bool		用户是否可以更新系统表。(即使超级用户也不能这么干，除非这个字段为真。)
passwd	text		口令(可能加密了)。
valuntil	abstime		口令失效的时间(只用于口令认证)。
useconfig	text[]		运行时配置参数的会话缺省。

五、pg_roles:

该视图提供访问数据库角色有关信息的接口。这个视图只是 pg_authid 表的公开可读部分的视图化，同时把口令字段用空白填充。

名字	类型	引用	描述
rolname	name		角色名。
rolsuper	bool		是否有超级用户权限的角色。
rolcreaterole	bool		是否可以创建更多角色的角色。
rolcreatedb	bool		是否可以创建数据库的角色。
rolcatupdate	bool		是否可以直接更新系统表的角色。
rolcanlogin	bool		如果为真，表示是可以登录的角色。
rolpassword	text		不是口令(总是 ********)。
rolvaliduntil	timestamptz		口令失效日期(只用于口令认证)；如果没有失效期，为 NULL。
rolconfig	text[]		运行时配置变量的会话缺省。

六、pg_rules:

该视图提供对查询重写规则的有用信息访问的接口。

名字	类型	引用	描述
schemaname	name	pg_namespace.nspname	包含表的模式的名字。
tablename	name	pg_class.relname	规则施加影响的表的名字。
rulename	name	pg_rewrite.rulename	规则的名字。
definition	text		规则定义(一个重新构造的创建命令)。

七、pg_settings:

该视图提供了对服务器运行时参数的访问。它实际上是 SHOW 和 SET 命令的另外一种方式。它还提供一些用 SHOW 不能直接获取的参数的访问，比如最大和最小值。

名字	类型	引用	描述
name	text		运行时配置参数名。
setting	text		参数的当前值。
category	text		参数的逻辑组。
short_desc	text		参数的一个简短的描述。
extra_desc	text		有关参数的额外的、更详细的信息。
context	text		设置这个参数的值要求的环境。
vartype	text		参数类型(bool、integer、real 和 string)。
source	text		当前参数值的来源。
min_val	text		该参数允许的最小值(非数字值为 NULL)。
max_val	text		该参数允许的最大值(非数字值为 NULL)。

我们不能对 pg_settings 视图进行插入或者删除， 只能更新。对 pg_settings 中的一行进行 UPDATE 等效于在该命名参数上执行 *SET* 命令。这个修改值影响当前会话使用的数值。如果在一个最后退出的事务中发出了 UPDATE 命令，那么 UPDATE 命令的效果将在事务回滚之后消失。一旦包围它的事务提交，这个效果将固化，直到会话结束。

PostgreSQL 学习手册(客户端命令<一>)

零、口令文件:

在给出其它 PostgreSQL 客户端命令之前，我们需要先介绍一下 PostgreSQL 中的口令文件。之所以在这里提前说明该文件，是因为我们在后面的示例代码中会大量应用该文件，从而保证我们的脚本能够自动化完成。换句话说，如果在客户端命令执行时没有提供该文件，PostgreSQL 的所有客户端命令均会被口令输入提示中断。

在当前用户的 HOME 目录下,我们需要手工创建文件名为 .pgpass 的口令文件,这样就可以在我们连接 PostgreSQL 服务器时，客户端命令自动读取该文件已获得登录时所需要的口令信息。该文件的格式如下：

hostname:port:database:username:password

以上数据是用冒号作为分隔符，总共分为五个字段，分别表示服务器主机名(IP)、服务器监听的端口号、登录访问的数据库名、登录用户名和密码，其中前四个字段都可以使用星号(*)来表示匹配任意值。见如下示例：

/> cat > .pgpass

**:5432:postgres:postgres:123456*

CTRL+D

#.pgpass 文件的权限必须为0600，从而防止任何全局或者同组的用户访问，否则这个文件将被忽略。

/> chmod 0600 .pgpass

在学习后面的客户端命令之前，我们需要根据自己的应用环境手工创建该文件，以便后面所有的示例代码都会用到该口令文件，这样它们就都可以以批处理的方式自动完成。

一、createdb:

创建一个新的 PostgreSQL 数据库。该命令的使用方式如下：

createdb [option...] [dbname] [description]

1. 命令行选项列表：

选项	说明
-D(--tablespace=tablespace)	指定数据库的缺省表空间。
-e(--echo)	回显 createdb 生成的命令并且把它发送到服务器。
-E(--encoding=encoding)	指定用于此数据库的字符编码方式。
-l(--locale=locale)	指定用于此数据库的本地化设置。
-O(--owner=owner)	指定新建数据库的拥有者，如果未指定此选项，该值为当前登录的用户。
-T(--template=template)	指定创建此数据库的模板数据库。
-h(--host=host)	指定 PostgreSQL 服务器的主机名。
-p(--port=port)	指定服务器的侦听端口，如不指定，则为缺省的 5432。
-U(--username=username)	本次操作的登录用户名，如果-O 选项没有指定，此数据库的 Owner 将为该登录用户。
-w(--no-password)	如果当前登录用户没有密码，可以指定该选项直接登录。

2. 应用示例：

#1. 以 postgres 的身份登录。(详情参照上面口令文件的内容)

/> psql

#2. 创建表空间。

postgres=# CREATE TABLESPACE my_tablespace LOCATION '/opt/PostgreSQL/9.1/mydata';

CREATE TABLESPACE

#3. 创建新数据库的 owner。

postgres=# CREATE ROLE myuser LOGIN PASSWORD '123456';

CREATE ROLE

postgres=# \q

#4. 创建新数据库，其中本次连接的登录用户为 postgres，新数据库的 owner 为 myuser，表空间为 my_tablespace，新数据库名为 mydatabase。

*/> createdb **-U** postgres **-O** myuser **-D** my_tablespace **-e mydatabase***

CREATE DATABASE mydatabase OWNER myuser TABLESPACE my_tablespace;

#5. 重新登录，通过查询系统表查看该数据库是否创建成功，以及表空间和所有者是否一致。

/> psql

postgres=# SELECT datname,rolname,spcname FROM pg_database db, pg_authid au, pg_tablespace ts WHERE datname = 'mydatabase' AND datdba = au.oid AND dattablespace = ts.oid;

```
  datname   | rolname |    spcname
------------+---------+---------------
 mydatabase | myuser  | my_tablespace
(1 row)
```

二、dropdb：

删除一个现有 PostgreSQL 数据库。

dropdb [option...] dbname

1. 命令行选项列表：

选项	说明
-e(--echo)	回显 dropdb 生成的命令并且把它发送到服务器。
-i(--interactive)	在做任何破坏性动作前提示。
-q(--quiet)	不显示响应。
-h(--host=host)	指定 PostgreSQL 服务器的主机名。
-p(--port=port)	指定服务器的监听端口，如不指定，则为缺省的 5432。
-U(--username=username)	本次操作的登录用户名。
-w(--no-password)	如果当前登录用户没有密码，可以指定该选项直接登录。

2. 应用示例：

```
# 以 postgres 的身份连接服务器，删除 mydatabase 数据库。
/> dropdb -U postgres -e mydatabase
DROP DATABASE mydatabase;
# 通过查看系统表验证该数据库是否已经被删除。
/> psql
postgres=# SELECT count(*) FROM pg_database WHERE datname = 'mydatabase';
 count
-------
     0
(1 row)
```

三、reindexdb：

为一个指定的 PostgreSQL 数据库重建索引。

reindexdb [connection-option...] [--table | -t table] [--index | -i index] [dbname]

reindexdb [connection-option...] [--all | -a]

reindexdb [connection-option...] [--system | -s] [dbname]

1. 命令行选项列表：

选项	说明
-a(-all)	重建整个数据库的索引。
-e(--echo)	回显 reindexdb 生成的命令并且把它发送到服务器。
-i(--index=index)	仅重建指定的索引。
-q(--quiet)	不显示响应。
-s(--system)	重建数据库系统表的索引。
-t(--table=table)	仅重建指定数据表的索引。
-h(--host=host)	指定 PostgreSQL 服务器的主机名。
-p(--port=port)	指定服务器的监听端口，如不指定，则为缺省的 5432。
-U(--username=username)	本次操作的登录用户名。
-w(--no-password)	如果当前登录用户没有密码，可以指定该选项直接登录。

2. 应用示例：

```
# 仅重建数据表 testtable 上的全部索引。
```

```
/> reindexdb -t testtable -e -U postgres postgres
REINDEX TABLE testtable;
#仅重建指定索引 testtable_idx
/> reindexdb -i testtable_idx -e -U postgres postgres
REINDEX INDEX testtable_idx;
#重建指定数据库 mydatabase 的全部索引。
/> reindexdb mydatabase
```

四、vacuumdb：

收集垃圾并且分析一个 PostgreSQL 数据库。

vacuumdb [-options] [--full | -f] [--verbose | -v] [--analyze | -z] [-t table [(column [,...])]] [dbname]

vacuumdb [-options] [--all | -a] [--full | -f] [--verbose | -v] [--analyze | -z]

1. 命令行选项列表：

选项	说明
-a(--all)	清理所有数据库。
-e(--echo)	回显 vacuumdb 生成的命令并且把它发送到服务器。
-f(--full)	执行完全清理。
-q(--quiet)	不显示响应。
-t *table* [(*column*[,...])]	仅仅清理或分析指定的数据表，字段名只是在与--analyze 选项联合使用时才需要声明。
-v(--verbose)	在处理过程中打印详细信息。
-z(--analyze)	计算用于规划器的统计值。
-h(--host=host)	指定 PostgreSQL 服务器的主机名。
-p(--port=port)	指定服务器的监听端口，如不指定，则为缺省的 5432。
-U(--username=username)	本次操作的登录用户名。
-w(--no-password)	如果当前登录用户没有密码，可以指定该选项直接登录。

2. 应用示例：

```
#清理整个数据库 mydatabase。
/> vacuumdb -e mydatabase
VACUUM;
#清理并分析 postgres 数据库中的 testtable 表。
/> vacuumdb -e --analyze --table 'testtable' postgres
VACUUM ANALYZE testtable;
#清理并分析 postgres 数据库中的 testtable 表的 i 字段。
/> vacuumdb -e --analyze -t 'testtable(i)' postgres
VACUUM ANALYZE testtable(i);
```

五、createuser:

定义一个新的 PostgreSQL 用户帐户，需要说明的是只有超级用户或者是带有 CREATEROLE 权限的用户才可以执行该命令。如果希望创建的是超级用户，那么只能以超级用户的身份执行该命令，换句话说，带有 CREATEROLE 权限的普通用户无法创建超级用

户。该命令的使用方式如下：

createuser [option...] [username]

1. 命令行选项列表：

选项	说明
-c number	设置新创建用户的最大连接数，缺省为没有限制。
-d(--createdb)	允许该新建用户创建数据库。
-D(--no-createdb)	禁止该新建用户创建数据库。
-e(--echo)	回显 createuser 生成的命令并且把它发送到服务器。
-E(--encrypted)	对保存在数据库里的用户口令加密。如果没有声明， 则使用缺省值。
-i(--inherit)	新创建的角色将自动继承它的组角色的权限。
-I(--no-inherit)	新创建的角色不会自动继承它的组角色的权限。
-l(--login)	新角色将被授予登录权限，该选项为缺省选项。
-L(--no-login)	新角色没有被授予登录权限。
-N(--unencrypted)	不对保存在数据库里的用户口令加密。如果没有声明， 则使用缺省值。
-P(--pwprompt)	如果给出该选项，在创建用户时将提示设置口令。
-r(--createrole)	新角色被授予创建数据库的权限。
-R(--no-createrole)	新角色没有被授予创建数据库的权限。
-s(--superuser)	新角色为超级用户。
-S(--no-superuser)	新角色不是超级用户。
-h(--host=host)	指定 PostgreSQL 服务器的主机名。
-p(--port=port)	指定服务器的监听端口，如不指定，则为缺省的 5432。
-U(--username=username)	本次操作的登录用户名。
-w(--no-password)	如果当前登录用户没有密码，可以指定该选项直接登录。

2. 应用示例：

 # 对于有些没有缺省设置的选项，如-(d/D)、-(s/S)和-(r/R)，如果在命令行中没有直接指定，那么在执行该命令是将会给出提示信息。

 # 需要注意的是该提示将会挂起自动化脚本，直到输入后命令才会继续执行。

/> createuser -U postgres myuser

Shall the new role be a superuser? (y/n) ***n***

Shall the new role be allowed to create databases? (y/n) ***y***

Shall the new role be allowed to create more new roles? (y/n) ***n***

CREATE ROLE myuser NOSUPERUSER CREATEDB NOCREATEROLE INHERIT LOGIN;

 # 通过 psql 登录后查看系统视图，以验证该用户是否成功创建，以及新角色的权限是否正确。

/> psql

postgres=# SELECT rolname,rolsuper,rolinherit,rolcreaterole,rolcreatedb,rolcanlogin FROM pg_roles WHERE rolname = 'myuser';

```
 rolname | rolsuper | rolinherit | rolcreaterole | rolcreatedb | rolcanlogin
---------+----------+------------+---------------+-------------+-------------
 myuser  | f        | t          | f             | t           | t
(1 row)
```

 # 为了保证自动化脚本不会被该命令的提示挂起，我们需要在执行该命令时指定所有没有缺省值的选项。

/> createuser -U postgres -e -S -D -R myuser2

CREATE ROLE myuser2 NOSUPERUSER NOCREATEDB NOCREATEROLE INHERIT LOGIN;

 # 我们可以在创建用户时即刻指定该用户的密码，该操作由-P 选项完成，然而这样的用法一定会挂起自动化脚本，

 # 因此我们可以采用一种折中的办法，即在创建用户时不指定密码，在自动化脚本执行成功后再手工该用户的密码。

/> createuser -P -s -e myuser3

Enter password for new role:

Enter it again:

CREATE ROLE myuser3 PASSWORD 'md5fe54c4f3129f2a766f53e4f4c9d2a698' SUPERUSER CREATEDB CREATEROLE INHERIT LOGIN;

六、dropuser：

删除一个 PostgreSQL 用户帐户，需要说明的是只有超级用户或带有 CREATEROLE 权限的用户可以执行该命令，如果要删除超级用户，只能通过超级用户的身份执行该命令。该命令的使用方式如下：

dropuser [option...] [username]

1．命令行选项列表：

选项	说明
-e(--echo)	回显 dropuser 生成的命令并且把它发送到服务器。
-i(--interactive)	在做任何破坏性动作前提示。
-h(--host=host)	指定 PostgreSQL 服务器的主机名。
-p(--port=port)	指定服务器的监听端口，如不指定，则为缺省的 5432。
-U(--username=username)	本次操作的登录用户名。
-w(--no-password)	如果当前登录用户没有密码，可以指定该选项直接登录。

2．应用示例：

 # 直接删除指定用户。

/> dropuser -e myuser3

DROP ROLE myuser3;

 # 在删除指定用户时，该命令会给出提示信息，以免误操作。

/> dropuser -e -i myuser2

Role "myuser2" will be permanently removed.

Are you sure? (y/n) **y**

DROP ROLE myuser2;

PostgreSQL 学习手册(客户端命令<二>)

七、pg_dump：

pg_dump 是一个用于备份 PostgreSQL 数据库的工具。它甚至可以在数据库正在并发使用时进行完整一致的备份，而不会阻塞其它用户对数据库的访问。该工具生成的转储格式可以分为两种，脚本和归档文件。其中脚本格式是包含许多 SQL 命令的纯文本格式，这些 SQL 命令可以用于重建该数据库并将之恢复到生成此脚本时的状态，该操作需要使用 psql 来完成。至于归档格式，如果需要重建数据库就必须和 pg_restore 工具一起使用。在重建过程中，可以对恢复的对象进行选择，甚至可以在恢复之前对需要恢复的条目进行重新排序。该命令的使用方式如下：

pg_dump [option...] [dbname]

1. 命令行选项列表：

选项	说明
-a(--data-only)	只输出数据，不输出模式(数据对象的定义)。这个选项只是对纯文本格式有意义。对于归档格式，你可以在调用 pg_restore 时指定选项。
-b(--blobs)	在 dump 中包含大对象。
-c(--clean)	在输出创建数据库对象的 SQL 命令之前，先输出删除该数据库对象的 SQL 命令。这个选项只是对纯文本格式有意义。对于归档格式，你可以在调用 pg_restore 时指定选项。
-C(--create)	先输出创建数据库的命令，之后再重新连接新创建的数据库。对于此种格式的脚本，在运行之前是和哪个数据库进行连接就不这么重要了。这个选项只是对纯文本格式有意义。对于归档格式，你可以在调用 pg_restore 时指定选项。
-E encoding	以指定的字符集创建该 dump 文件。
-f file	输出到指定文件，如果没有该选项，则输出到标准输出。
-F format	**p(plain)**: 纯文本格式的 SQL 脚本文件(缺省)。**c(custom)**: 输出适合于 pg_restore 的自定义归档格式。 这是最灵活的格式，它允许对装载的数据和对象定义进行重新排列。这个格式缺省的时候是压缩的。**t(tar)**: 输出适合于 pg_restore 的 tar 归档文件。使用这个归档允许在恢复数据库时重新排序和/或把数据库对象排除在外。同 i 时也可能可以在恢复的时候限制对哪些数据进行恢复。
-n schema	只转储 schema 的内容。如果没有声明该选项，目标数据库中的所有非系统模式都会被转储。该选项也可以被多次指定，以指定不同 pattern 的模式。
-N schema	不转储匹配 schema 的内容，其他规则和-n 一致。
-o(--oids)	作为数据的一部分，为每个表都输出对象标识(OID)。
-O(--no-owner)	不输出设置对象所有权的 SQL 命令。
-s(--schema-only)	只输出对象定义(模式)，不输出数据。
-S username	指定关闭触发器时需要用到的超级用户名。它只有在使用--disable-triggers 的时候才有关系。
-t table	只输出表的数据。很可能在不同模式里面有多个同名表，如果这样，那么所有匹配的表都将被转储。通过多次指定该参数，可以一次转储多张表。这里还可以指定和 psql 一样的 pattern，以便匹配更多的表。*(关于 pattern，基本的使用方式是可以将它视为 unix 的通配符，即*表示任意字符，?表示任意单个字符，.(dot)表示 schema 和 object 之间的分隔符，如 a*.b*，表示以 a 开头的 schema 和以 b 开头的数据库对象。如果没有.(dot)，将只是表示数据库对象。这里也可以使用基本的正则表达式，如[0-9]表示数字。)*
-T table	排除指定的表，其他规则和-t 选项一致。
-x(--no-privileges)	不导出访问权限信息(grant/revoke 命令)。
-Z 0..9	声明在那些支持压缩的格式中使用的压缩级别。 (目前只有自定义格式支持压缩)
--column-inserts	导出数据用 insert into table_name(columns_list) values(values_list)命令表示，这样的操作相对其它操作而言是比较慢的，但是在特殊情况下，如数据表字段的位置有可能发生变化或有新的字段插入到原有字段列表的中间等。由于 columns_list 被明确指定，因此在导入时不会出现数据被导入到错误字段的问题。
--inserts	导出的数据用 insert 命令表示，而不是 copy 命令。即便使用 insert 要比 copy 慢一些，但是对于今后导入到其他非 PostgreSQL 的数据库是比较有意义的。

--no-tablespaces	不输出设置表空间的命令，如果带有这个选项，所有的对象都将恢复到执行 pg_restore 时的缺省表空间中。
--no-unlogged-table-data	对于不计入日志(unlogged)的数据表，不会导出它的数据，至于是否导出其 Schema 信息，需要依赖其他的选项而定。
-h(--host=host)	指定 PostgreSQL 服务器的主机名。
-p(--port=port)	指定服务器的侦听端口，如不指定，则为缺省的 5432。
-U(--username=username)	本次操作的登录用户名，如果-O 选项没有指定，此数据库的 Owner 将为该登录用户。
-w(--no-password)	如果当前登录用户没有密码，可以指定该选项直接登录。

2. 应用示例：

 # -h: PostgreSQL 服务器的主机为 192.168.149.137。

 # -U: 登录用户为 postgres。

 # -t: 导出表名以 test 开头的数据表，如 testtable。

 # -a: 仅仅导出数据，不导出对象的 schema 信息。

 # -f: 输出文件是当前目录下的 my_dump.sql

 # mydatabase 是此次操作的目标数据库。

/> pg_dump -h 192.168.149.137 -U postgres -t test -a -f ./my_dump.sql mydatabase*

#-c: 先输出删除数据库对象的 SQL 命令，在输出创建数据库对象的 SQL 命令，这对于部署干净的初始系统或是搭建测试环境都非常方便。

/> pg_dump -h 192.168.220.136 -U postgres -c -f ./my_dump.sql mydatabase

 # 导出 mydatabase 数据库的信息。在通过 psql 命令导入时可以重新指定数据库，如：/> psql -d newdb -f my_dump.sql

/> pg_dump -h 192.168.220.136 -U postgres -f ./my_dump.sql mydatabase

 # 导出模式为 my_schema 和以 test 开头的数据库对象名，但是不包括 my_schema.employee_log 对象。

/> pg_dump -t 'my_schema.test' -T my_schema.employee_log mydatabase > my_dump.sql*

 # 导出 east 和 west 模式下的所有数据库对象。下面两个命令是等同的，只是后者使用了正则。

/> pg_dump -n 'east' -n 'west' mydatabase -f my_dump.sql

/> pg_dump -n '(east|west)' mydatabase -f my_dump.sql

八、pg_restore:

pg_restore 用于恢复 pg_dump 导出的任何非纯文本格式的文件，它将数据库重建成保存它时的状态。对于归档格式的文件，pg_restore 可以进行有选择的恢复，甚至也可以在恢复前重新排列数据的顺序。

pg_restore 可以在两种模式下操作。如果指定数据库，归档将直接恢复到该数据库。否则，必须先手工创建数据库，之后再通过 pg_restore 恢复数据到该新建的数据库中。该命令的使用方式如下：

pg_restore [option...] [filename]

1. 命令行选项列表：

选项	说明
filename	指定要恢复的备份文件，如果没有声明，则使用标准输入。
-a(--data-only)	只恢复数据，而不恢复表模式(数据对象定义)。
-c(--clean)	创建数据库对象前先清理(删除)它们。
-C(--create)	在恢复数据库之前先创建它。(在使用该选项时，数据库名需要由-d 选项指定，该选项只是执行最基本的 CREATE DATABASE 命令。需要说明的是，归档文件中所有的数据都将恢复到归档文件里指定的数据库中)。

-d dbname	与数据库 dbname 建立连接并且直接恢复数据到该数据库中。
-e(--exit-on-error)	如果在向数据库发送 SQL 命令的时候遇到错误，则退出。缺省是继续执行并且在恢复结束时显示一个错误计数。
-F format	指定备份文件的格式。由于 pg_restore 会自动判断格式，因此指定格式并不是必须的。如果指定，它可以是以下格式之一：**t(tar)**: 使用该格式允许在恢复数据库时重新排序和/或把表模式信息排除出去，同时还可能在恢复时限制装载的数据。**c(custom)**:该格式是来自 pg_dump 的自定义格式。这是最灵活的格式，因为它允许重新对数据排序，也允许重载表模式信息，缺省情况下这个格式是压缩的。
-I index	只恢复指定的索引。
-l(--list)	列出备份中的内容，这个操作的输出可以作为-L 选项的输入。注意，如果过滤选项-n 或-t 连同-l 选项一起使用的话，他们也将限制列出的条目。
-L list-file	仅恢复在 list-file 中列出的条目，恢复的顺序为各个条目在该文件中出现的顺序，你也可以手工编辑该文件，并重新排列这些条目的位置，之后再进行恢复操作，其中以分号(;)开头的行为注释行，注释行不会被导入。
-n namespace	仅恢复指定模式(Schema)的数据库对象。该选项可以和-t 选项联合使用，以恢复指定的数据对象。
-O(--no-owner)	不输出设置对象所有权的 SQL 命令。
-P function-name(argtype [, ...])	只恢复指定的命名函数。该名称应该和转储的内容列表中的完全一致。
-s(--schema-only)	只恢复表结构(数据定义)。不恢复数据，序列值将重置。
-S username	指定关闭触发器时需要用到的超级用户名。它只有在使用--disable-triggers 的时候才有关系。
-t table	只恢复指定表的 Schema 和/或数据，该选项也可以连同-n 选项指定模式。
-x(--no-privileges)	不恢复访问权限信息(grant/revoke 命令)。
-1(--single-transaction)	在一个单一事物中执行恢复命令。这个选项隐含包括了--exit-on-error 选项。
--no-tablespaces	不输出设置表空间的命令，如果带有这个选项，所有的对象都将恢复到执行 pg_restore 时的缺省表空间中。
--no-data-for-failed-tables	缺省情况下，即使创建表失败了，如该表已经存在，数据加载的操作也不会停止，这样的结果就是很容易导致大量的重复数据被插入到该表中。如果带有该选项，那么一旦出现针对该表的任何错误，对该数据表的加载将被忽略。
--role=rolename	以指定的角色名执行 restore 的操作。通常而言，如果连接角色没有足够的权限用于本次恢复操作，那么就可以利用该选项在建立连接之后再切换到有足够权限的角色。
-h(--host=host)	指定 PostgreSQL 服务器的主机名。
-p(--port=port)	指定服务器的侦听端口，如不指定，则为缺省的 5432。
-U(--username=username)	本次操作的登录用户名，如果-O 选项没有指定，此数据库的 Owner 将为该登录用户。
-w(--no-password)	如果当前登录用户没有密码，可以指定该选项直接登录。

2. 应用示例：

 # 先通过 createdb 命令，以 myuser 用户的身份登录，创建带恢复的数据 newdb

/> createdb -U myuser newdb

#用pg_restore 命令的-l 选项导出my_dump.dat 备份文件中导出数据库对象的明细列表。

/> pg_restore -l my_dump.dat > db.list

/> cat db.list

2; 145344 TABLE species postgres

4; 145359 TABLE nt_header postgres

6; 145402 TABLE species_records postgres

8; 145416 TABLE ss_old postgres

10; 145433 TABLE map_resolutions postgres

#将以上列表文件中的内容修改为以下形式。

#主要的修改是注释掉编号为2、4 和8 的三个数据库对象，同时编号10 的对象放到该文件的头部，这样在基于该列表

#文件导入时，2、4 和8 等三个对象将不会被导入，在恢复的过程中将先导入编号为10 的对象的数据，再导入对象6 的数据。

/> cat new_db.list

10; 145433 TABLE map_resolutions postgres

;2; 145344 TABLE species postgres

;4; 145359 TABLE nt_header postgres

6; 145402 TABLE species_records postgres

;8; 145416 TABLE ss_old postgres

#恢复时指定的数据库是newdb，导入哪些数据库对象和导入顺序将会按照new_db.list 文件中提示的规则导入。

/> pg_restore -d newdb -L new_db.list my_dump.dat

九、psql:

PostgreSQL 的交互终端，等同于 Oracle 中的 sqlplus。

1. 常用命令行选项列表：

选项	说明
-c command	指定 psql 执行一条 SQL 命令 command(用双引号括起)，执行后退出。
-d dbname	待连接的数据库名称。
-E	回显由\d 和其他反斜杠命令生成的实际查询。
-f filename	使用 filename 文件中的数据作为命令输入源，而不是交互式读入查询。在处理完文件后，psql 结束并退出。
-h hostname	声明正在运行服务器的主机名
-l	列出所有可用的数据库，然后退出。
-L filename	除了正常的输出源之外，把所有查询记录输出到文件 filename。
-o filename	将所有查询重定向输出到文件 filename。
-p port	指定 PostgreSQL 服务器的监听端口。
-q --quiet	让 psql 安静地执行所处理的任务。缺省时 psql 将输出打印欢迎和许多其他信息。
-t --tuples-only	关闭打印列名称和结果行计数脚注等信息。
-U username	以用户 username 代替缺省用户与数据库建立连接。

2. 命令行选项应用示例：

#-d: 指定连接的数据库。

#-U: 指定连接的用户。

#-c: 后面的SQL 语句是本次操作需要执行的命令。

*/> psql -d posgres -U postgres -c "select * from testtable"*

```
 i
---
 1
 2
 3
 5
(4 rows)
#-t: 没有输出上面输出结果中的字段标题信息和行数统计信息。
#-q：该选项和-t 选项联合使用，非常有利于自动化脚本。如：
#    select 'copy ' || tablename || ' to ' || tablename || '.sql' from pg_tables
#由以上 sql 语句生成的结果集，在重定向到输出文件后，可以作为下一次 psql 的输入执行。
/> psql -t -q -c "select * from testtable"
 1
 2
 3
 5
#-l: 列出当前系统中可用的数据库。
/> psql -l
                                List of databases
   Name       |  Owner   | Encoding |  Collation  |    Ctype    |   Access privileges
-------------+----------+----------+-------------+-------------+-----------------------
 mydatabase | myuser   | UTF8     | zh_CN.UTF-8 | zh_CN.UTF-8 |
 postgres    | postgres | UTF8     | zh_CN.UTF-8 | zh_CN.UTF-8 |

 ... ...

(4 rows)
#-o: 将查询语句的数据结果输出到指定文件。
/> psql -c "select * from testtable" -o out
/> cat out
 i
---
 1
 2
 3
 5
(4 rows)
```

3. 内置命令列表：

psql 内置命令的格式为反斜杠后面紧跟一个命令动词，之后是任意参数。参数与命令动词以及其他参数之间可以用空白符隔开，如果参数里面包含空白符，该参数必须用单引号括起，如果参数内包含单引号，则需要用反斜杠进行转义，此外单引号内的参数还支持类似 C 语言 printf 函数所支持的转义关键字，如\t、\n 等。

命令	说明
\a	如果目前的表输出格式是不对齐的，切换成对齐的。如果是对齐的，则切换成不对齐。
\cd [directory]	把当前工作目录切换到 directory。没有参数则切换到当前用户的主目录。
\C [title]	为查询结果添加表头(title)，如果没有参数则取消当前的表头。
\c [dbname[username]]	连接新的数据库，同时断开当前连接。如果 dbname 参数为-，表示仍然连接当前数据库。如果忽略 username，则表示继续使用当前的用户名。

\copy	其参数类似于 SQL copy，功能则几乎等同于 SQL copy，一个重要的差别是该内置命令可以将表的内容导出到本地，或者是从本地导入到数据库指定的表，而 SQL copy 则是将表中的数据导出到服务器的某个文件，或者是从服务器的文件导入到数据表。由此可见，SQL copy 的效率要优于该内置命令。
\d [pattern]	显示和 pattern 匹配的数据库对象，如表、视图、索引或者序列。显示所有列，它们的类型，表空间(如果不是缺省的)和任何特殊属性。
\db [pattern]	列出所有可用的表空间。如果声明了 pattern， 那么只显示那些匹配模式的表空间。
\db+ [pattern]	和上一个命令相比，还会新增显示每个表空间的权限信息。
\df [pattern]	列出所有可用函数，以及它们的参数和返回的数据类型。如果声明了 pattern，那么只显示匹配(正则表达式)的函数。
\df+ [pattern]	和上一个命令相比，还会新增显示每个函数的附加信息，包括语言和描述。
\distvS [pattern]	这不是一个单独命令名称：字母 i、s、t、v、S 分别代表索引(index)、序列(sequence)、表(table)、视图(view)和系统表(system table)。你可以以任意顺序声明部分或者所有这些字母获得这些对象的一个列表。
\dn [pattern]	列出所有可用模式。如果声明了 pattern，那么只列出匹配模式的模式名。
\dn+ [pattern]	和上一个命令相比，还会新增显示每个对象的权限和注释。
\dp [pattern]	生成一列可用的表和它们相关的权限。如果声明了 pattern， 那么只列出名字可以匹配模式的表。
\dT [pattern]	列出所有数据类型或只显示那些匹配 pattern 的。
\du [pattern]	列出所有已配置用户或者只列出那些匹配 pattern 的用户。
\echo text[...]	向标准输出打印参数，用一个空格分隔并且最后跟着一个新行。如：\echo `date`
\g[{filename||command}]	把当前的查询结果缓冲区的内容发送给服务器并且把查询的输出存储到可选的 filename 或者把输出定向到一个独立的在执行 command 的 Unix shell。
\i filename	从文件 filename 中读取并把其内容当作从键盘输入的那样执行查询。
\l	列出服务器上所有数据库的名字和它们的所有者以及字符集编码。
\o[{filename||command}]	把后面的查询结果保存到文件 filename 里或者把后面的查询结果定向到一个独立的 shell command。
\p	打印当前查询缓冲区到标准输出。
\q	退出 psql 程序。
\r	重置(清空)查询缓冲区。
\s [filename]	将命令行历史打印出或是存放到 filename。如果省略 filename，历史将输出到标准输出。
\t	切换是否输出列/字段名的信息头和行记数脚注。
\w{filename ||command}	将当前查询缓冲区输出到文件 filename 或者定向到 Unix 命令 command。
\z [pattern]	生成一个带有访问权限列表的数据库中所有表，视图和序列的列表。如果给出任何 pattern，则被当成一个规则表达式，只显示匹配的表，视图和序列。
\! [command]	返回到一个独立的 Unix shell 或者执行 Unix 命令 command。参数不会被进一步解释，shell 将看到全部参数。

4. 内置命令应用示例：

在 psql 中，大部分的内置命令都比较易于理解，因此这里只是给出几个我个人认为相对容易混淆的命令。

 # \c: 其中横线(-)表示仍然连接当前数据库，myuser 是新的用户名。

```
postgres=# \c - myuser
Password for user myuser:
postgres=> SELECT user;
 current_user
--------------
 myuser
(1 row)
# 执行任意SQL 语句。
postgres=# SELECT * FROM testtable WHERE i = 2;
 i
---
 2
(1 row)
# \g 命令会将上一个SQL 命令的结果输出到指定文件。
postgres=# \g my_file_for_command_g
postgres=# \! cat my_file_for_command_g
 i
---
 2
(1 row)
# \g 命令会将上一个SQL 命令的结果从管道输出到指定的Shell 命令，如cat。
postgres=# \g | cat
 i
---
 2
(1 row)
# \p 打印上一个SQL 命令。
postgres=# \p
SELECT * FROM testtable WHERE i = 2;
# \w 将上一个SQL 命令输出到指定的文件。
postgres=# \w my_file_for_option_w
postgres=# \! cat my_file_for_option_w
SELECT * FROM testtable WHERE i = 2;
# \o 和\g 相反，该命令会将后面psql 命令的输出结果输出到指定的文件，直到遇到下一个独立的\o，
# 此后的命令结果将不再输出到该文件。
postgres=# \o my_file_for_option_o
postgres=# SELECT * FROM testtable WHERE i = 1;
# 终止后面的命令结果也输出到my_file_for_option_o 文件中。
postgres=# \o
postgres=# \! cat my_file_for_option_o
 i
---
 1
(1 row)
```

PostgreSQL 学习手册(SQL 语言函数)

一、基本概念：

SQL 函数可以包含任意数量的查询，但是函数只返回最后一个查询(必须是 SELECT)的结果。在简单情况下，返回最后一条查询结果的第一行。如果最后一个查询不返回任何行，那么该函数将返回 NULL 值。如果需要该函数返回最后一条 SELECT 语句的所有行，可以将函数的返回值定义为集合，即 SETOF sometype。

SQL 函数的函数体应该是用分号分隔的 SQL 语句列表，其中最后一条语句之后的分号是可选的。除非函数声明为返回 void，否则最后一条语句必须是 SELECT。事实上，在 SQL 函数中，不仅可以包含 SELECT 查询语句，也可以包含 INSERT、UPDATE 和 DELETE 等其他标准的 SQL 语句，但是和事物相关的语句不能包含其中，如 BEGIN、COMMIT、ROLLBACK 和 SAVEPOINT 等。

CREATE FUNCTION 命令的语法要求函数体写成一个字符串文本。通常来说，该文本字符串常量使用美元符($$)围住，如：

```
CREATE FUNCTION clean_emp() RETURNS void AS $$
    DELETE FROM emp WHERE salary < 0;
$$ LANGUAGE SQL;
```

最后需要说明的是 SQL 函数中的参数，PostgreSQL 定义$1 表示第一个参数，$2 为第二个参数并以此类推。如果参数是复合类型，则可以使用点表示法，即$1.name 访问复合类型参数中的 name 字段。需要注意的是函数参数只能用作数据值，而不能用于标识符，如：

```
INSERT INTO mytable VALUES ($1);   --合法
INSERT INTO $1 VALUES (42);        --不合法(表名属于标示符之一)
```

二、基本类型：

最简单的 SQL 函数可能就是没有参数且返回基本类型的函数了，如：

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

下面的例子声明了基本类型作为函数的参数。

```
CREATE FUNCTION add_em(integer, integer) RETURNS integer AS $$
    SELECT $1 + $2;
$$ LANGUAGE SQL;
# 通过 select 调用函数。
postgres=# SELECT add_em(1,2) AS answer;
 answer
--------
      3
(1 row)
```

在下面的例子中，函数体内包含多个 SQL 语句，它们之间是用分号进行分隔的。

```
CREATE FUNCTION tf1 (integer, numeric) RETURNS numeric AS $$
    UPDATE bank SET balance = balance - $2 WHERE accountno = $1;
    SELECT balance FROM bank WHERE accountno = $1;
$$ LANGUAGE SQL;
```

三、复合类型：

见如下示例：

1). 创建数据表，这样与之对应的复合类型也随之生成。

```
CREATE TABLE emp (
    name        text,
    salary      numeric,
    age         integer,
);
```

2). 创建函数，其参数为复合类型。在函数体内，可以像引用基本类型参数那样引用复合类型，如$1。访问复合类型的字段使用点表达式即可，如：$1.salary。

```
CREATE FUNCTION double_salary(emp) RETURNS integer AS $$
    SELECT ($1.salary * 2)::integer AS salary;
$$ LANGUAGE SQL;
```

3). 在 select 语句中，可以使用 emp.*表示 emp 表的一整行数据。

```
SELECT name, double_salary(emp.*) AS dream FROM emp WHERE age > 30;
```

4). 我们也可以使用 ROW 表达式构造自定义的复合类型，如：

```
SELECT name, double_salary(ROW(name, salary*1.1, age)) AS dream FROM emp;
```

5). 创建一个函数，其返回值为复合类型，如：

```
CREATE FUNCTION new_emp() RETURNS emp AS $$
    SELECT ROW('None', 1000.0, 25)::emp;
$$ LANGUAGE SQL;
```

6). 调用返回复合类型的函数。

```
SELECT new_emp();
```

7). 调用返回复合类型的函数，同时访问该返回值的某个字段。

```
SELECT (new_emp()).name;
```

四、带输出参数的函数：

还有一种方法可以用于返回函数执行的结果，即输出参数，如：

```
CREATE FUNCTION add_em2 (IN x int, IN y int, OUT sum int) AS $$
    SELECT $1 + $2
$$ LANGUAGE SQL;
```

调用方法和返回结果与 add_em(带有返回值的函数)完全一致，如：

```
SELECT add_em(3,7);
```

这个带有输出参数的函数和之前的 add_em 函数没有本质的区别。事实上，输出参数的真正价值在于它为函数提供了返回多个字段的途径。如，

```
CREATE FUNCTION sum_n_product (x int, y int, OUT sum int, OUT product int) AS $$
    SELECT $1 + $2, $1 * $2
$$ LANGUAGE SQL;
```

调用方式没有改变，只是返回结果多出一列。

```
SELECT * FROM sum_n_product(11,42);
 sum | product
-----+---------
  53 |     462
(1 row)
```

在上面的示例中，IN 用于表示该函数参数为输入参数(缺省值，可以忽略)，OUT 则表示该参数为输出参数。

五、返回结果作为表数据源：

所有 SQL 函数都可以在查询的 FROM 子句里使用。该方法对于返回复合类型的函数而言特别有用，如果该函数定义为返回一个基本类型，那么该函数生成一个单字段表，如果该函数定义为返回一个复合类型，那么该函数生成一个复合类型里每个属性组成的行。见如下示例：

1). 创建一个数据表。

```
CREATE TABLE foo (
    fooid    int,
    foosubid int,
    fooname  text
);
```

2). 创建 SQL 函数，其返回值为与 foo 表对应的复合类型。

```
CREATE FUNCTION getfoo(int) RETURNS foo AS $$
    SELECT * FROM foo WHERE fooid = $1;
$$ LANGUAGE SQL;
```

3). 在 FROM 子句中调用该函数。

```
SELECT *, upper(fooname) FROM getfoo(1) AS t1;
```

六、返回集合的 SQL 函数：

如果 SQL 函数的返回值为 SETOF sometype，那么在调用该函数时，将返回最后一个 SELECT 查询的全部数据。这个特性通常用于把函数放在 FROM 子句里调用，见如下示例：

```
CREATE FUNCTION getfoo(int) RETURNS setof foo AS $$
    SELECT * FROM foo WHERE fooid = $1;
$$ LANGUAGE SQL;
```

在 FROM 子句中调用了返回复合类型集合的函数，其结果等同于：*SELECT * FROM (SELECT * FROM foo WHERE fooid = 1) t1;*

```
SELECT * FROM getfoo(1) AS t1;
```

七、多态的 SQL 函数：

SQL 函数可以声明为接受多态类型(**anyelement** 和 **anyarray**)的参数或返回多态类型的返回值，见如下示例：

1). 函数参数和返回值均为多态类型。

```
CREATE FUNCTION make_array(anyelement, anyelement) RETURNS anyarray AS $$
    SELECT ARRAY[$1, $2];
$$ LANGUAGE SQL;
```

其调用方式和调用其它类型的 SQL 函数完全相同，只是在传递字符串类型的参数时，需要显式转换到目标类型，否则将会被视为 unknown 类型，如：

```
SELECT make_array(1, 2) AS intarray, make_array('a'::text, 'b') AS textarray;
```

2). 函数的参数为多态类型，而返回值则为基本类型。

```
CREATE FUNCTION is_greater(anyelement, anyelement) RETURNS boolean AS $$
    SELECT $1 > $2;
$$ LANGUAGE SQL;
```

3). 多态类型用于函数的输出参数。

```
CREATE FUNCTION dup (f1 anyelement, OUT f2 anyelement, OUT f3 anyarray) AS $$
    SELECT $1, ARRAY[$1,$1]
$$ LANGUAGE sql;
```

八、函数重载：

多个函数可以定义成相同的函数名，但是它们的参数一定要有所区分。换句话说，函数名可以重载，此规则有些类似于面向对象语言中的函数重载，见如下示例：

```
CREATE FUNCTION test(int, real) RETURNS ...
CREATE FUNCTION test(smallint, double) RETURNS ...
```

由于在 PostgreSQL 中函数支持重载，因此在删除函数时，也必须指定参数列表，如：

```
DROP FUNCTION test(int, real);
DROP FUNCTION test(smallint,double);
```

PostgreSQL 学习手册(PL/pgSQL 过程语言)

一、概述：

PL/pgSQL 函数在第一次被调用时，其函数内的源代码(文本)将被解析为二进制指令树，但是函数内的表达式和 SQL 命令只有在首次用到它们的时候，PL/pgSQL 解释器才会为其创建一个准备好的执行规划，随后对该表达式或 SQL 命令的访问都将使用该规划。如果在一个条件语句中，有部分 SQL 命令或表达式没有被用到，那么 PL/pgSQL 解释器在本次调用中将不会为其准备执行规划，这样的好处是可以有效地减少为 PL/pgSQL 函数里的语句生成分析和执行规划的总时间，然而缺点是某些表达式或 SQL 命令中的错误只有在其被执行到的时候才能发现。

由于 PL/pgSQL 在函数里为一个命令制定了执行计划，那么在本次会话中该计划将会被反复使用，这样做往往可以得到更好的性能，但是如果你动态修改了相关的数据库对象，那么就有可能产生问题，如：

```
CREATE FUNCTION populate() RETURNS integer AS $$
DECLARE
    -- 声明段
BEGIN
    PERFORM my_function();
END;
$$ LANGUAGE plpgsql;
```

在调用以上函数时，PERFORM 语句的执行计划将引用 my_function 对象的 OID。在此之后，如果你重建了 my_function 函数，那么 populate 函数将无法再找到原有 my_function 函数的 OID。要解决该问题，可以选择重建 populate 函数，或者重新登录建立新的会话，以使 PostgreSQL 重新编译该函数。要想规避此类问题的发生，在重建 my_function 时可以使用 CREATE OR REPLACE FUNCTION 命令。

鉴于以上规则，在 PL/pgSQL 里直接出现的 SQL 命令必须在每次执行时均引用相同的表和字段，换句话说，不能将函数的参数用作 SQL 命令的表名或字段名。如果想绕开该限制，可以考虑使用 PL/pgSQL 中的 EXECUTE 语句动态地构造命令，由此换来的代价是每次执行时都要构造一个新的命令计划。

使用 PL/pgSQL 函数的一个非常重要的优势是可以提高程序的执行效率，由于原有的 SQL 调用不得不在客户端与服务器之间反复传递数据，这样不仅增加了进程间通讯所产生的开销，而且也会大大增加网络 IO 的开销。

二、PL/pgSQL 的结构：

PL/pgSQL 是一种块结构语言，函数定义的所有文本都必须在一个块内，其中块中的每个声明和每条语句都是以分号结束，如果某一子块在另外一个块内，那么该子块的 END 关键字后面必须以分号结束，不过对于函数体的最后一个 END 关键字，分号可以省略，如：

```
[ <<label>> ]
[ DECLARE declarations ]
BEGIN
    statements
END [ label ];
```

在 PL/pgSQL 中有两种注释类型，双破折号**(--)**表示单行注释。**/* */**表示多行注释，该注释类型的规则等同于 C 语言中的多行注释。

在语句块前面的声明段中定义的变量在每次进入语句块(BEGIN)时都会将声明的变量初始化为它们的缺省值，而不是每次函数调用时初始化一次。如：

```
CREATE FUNCTION somefunc() RETURNS integer AS $$
DECLARE
  quantity integer := 30;
BEGIN
  RAISE NOTICE 'Quantity here is %', quantity;      --在这里的数量是 30
  quantity := 50;
  --
  -- 创建一个子块
  --
  DECLARE
    quantity integer := 80;
  BEGIN
    RAISE NOTICE 'Quantity here is %', quantity;   --在这里的数量是 80
  END;
  RAISE NOTICE 'Quantity here is %', quantity;      --在这里的数量是 50
  RETURN quantity;
END;
$$ LANGUAGE plpgsql;
#执行该函数以进一步观察其执行的结果。
postgres=# select somefunc();
NOTICE:  Quantity here is 30
NOTICE:  Quantity here is 80
NOTICE:  Quantity here is 50
 somefunc
----------
       50
(1 row)
```

最后需要说明的是，目前版本的 PostgreSQL 并不支持嵌套事务，函数中的事物总是由外层命令(函数的调用者)来控制的，它们本身无法开始或提交事务。

三、声明：

所有在块里使用的变量都必须在块的声明段里先进行声明，唯一的例外是 FOR 循环里的循环计数变量，该变量被自动声明为整型。变量声明的语法如下：

```
variable_name [ CONSTANT ] variable_type [ NOT NULL ] [ { DEFAULT | := } expression ];
```

1). SQL 中的数据类型均可作为 PL/pgSQL 变量的数据类型，如 integer、varchar 和 char 等。

2). 如果给出了 DEFAULT 子句，该变量在进入 BEGIN 块时将被初始化为该缺省值，否则被初始化为 SQL 空值。缺省值是在每次进入该块时进行计算的。因此，如果把 now()赋予一个类型为 timestamp 的变量，那么该变量的缺省值将为函数实际调用时的时间，而不是函数预编译时的时间。

3). CONSTANT 选项是为了避免该变量在进入 BEGIN 块后被重新赋值，以保证该变量为常量。

4). 如果声明了 NOT NULL，那么赋予 NULL 数值给该变量将导致一个运行时错误。因此所有声明为 NOT NULL 的变量也必须在声明时定义一个非空的缺省值。

1. 函数参数的别名：

传递给函数的参数都是用$1、$2 这样的标识符来表示的。为了增加可读性，我们可以为其声明别名。之后别名和数字标识符均可指向该参数值，见如下示例：

1). 在函数声明的同时给出参数变量名。

```
CREATE FUNCTION sales_tax(subtotal real) RETURNS real AS $$
BEGIN
    RETURN subtotal * 0.06;
END;
$$ LANGUAGE plpgsql;
```

2). 在声明段中为参数变量定义别名。

```
CREATE FUNCTION sales_tax(REAL) RETURNS real AS $$
DECLARE
    subtotal ALIAS FOR $1;
BEGIN
    RETURN subtotal * 0.06;
END;
$$ LANGUAGE plpgsql;
```

3). 对于输出参数而言，我们仍然可以遵守 1)和 2)中的规则。

```
CREATE FUNCTION sales_tax(subtotal real, OUT tax real) AS $$
BEGIN
    tax := subtotal * 0.06;
END;
$$ LANGUAGE plpgsql;
```

4). 如果 PL/pgSQL 函数的返回类型为多态类型(anyelement 或 anyarray)，那么函数就会创建一个特殊的参数：$0。我们仍然可以为该变量设置别名。

```
CREATE FUNCTION add_three_values(v1 anyelement, v2 anyelement, v3 anyelement)
RETURNS anyelement AS $$
DECLARE
    result ALIAS FOR $0;
BEGIN
    result := v1 + v2 + v3;
    RETURN result;
END;
$$ LANGUAGE plpgsql;
```


2. 拷贝类型：

见如下形式的变量声明：

```
variable%TYPE
```

%TYPE 表示一个变量或表字段的数据类型，PL/pgSQL 允许通过该方式声明一个变量，其类型等同于 variable 或表字段的数据类型，见如下示例：

```
user_id users.user_id%TYPE;
```

在上面的例子中，变量 user_id 的数据类型等同于 users 表中 user_id 字段的类型。

通过使用%TYPE，一旦引用的变量类型今后发生改变，我们也无需修改该变量的类型声明。最后需要说明的是，我们可以在函数的参数和返回值中使用该方式的类型声明。

3. 行类型：

见如下形式的变量声明：

```
name table_name%ROWTYPE;
name composite_type_name;
```

table_name%ROWTYPE 表示指定表的行类型，我们在创建一个表的时候，PostgreSQL 也会随之创建出一个与之相应的复合类型，该类型名等同于表名，因此，我们可以通过以上两种方式来声明行类型的变量。由此方式声明的变量，可以保存 SELECT 返回结果中的一行。如果要访问变量中的某个域字段，可以使用点表示法，如 rowvar.field，但是行类型的变量只能访问自定义字段，无法访问系统提供的隐含字段，如 OID 等。对于函数的参数，我们只能使用复合类型标识变量的数据类型。最后需要说明的是，推荐使用%ROWTYPE 的声明方式，这样可以具有更好的可移植性，因为在 Oracle 的 PL/SQL 中也存在相同的概念，其声明方式也为%ROWTYPE。见如下示例：

```
CREATE FUNCTION merge_fields(t_row table1) RETURNS text AS $$
DECLARE
    t2_row table2%ROWTYPE;
BEGIN
    SELECT * INTO t2_row FROM table2 WHERE id = 1 limit 1;
    RETURN t_row.f1 || t2_row.f3 || t_row.f5 || t2_row.f7;
END;
$$ LANGUAGE plpgsql;
```

4. 记录类型：

见如下形式的变量声明：

```
name RECORD;
```

记录变量类似于行类型变量，但是它们没有预定义的结构，只能通过 SELECT 或 FOR 命令来获取实际的行结构，因此记录变量在被初始化之前无法访问，否则将引发运行时错误。

注：RECORD 不是真正的数据类型，只是一个占位符。

四、基本语句：

1. 赋值：

PL/pgSQL 中赋值语句的形式为：**identIFier := expression**，等号两端的变量和表达式的类型或者一致，或者可以通过 PostgreSQL 的转换规则进行转换，否则将会导致运行时错误，见如下示例：

```
user_id := 20;
tax := subtotal * 0.06;
```

2. SELECT INTO：

通过该语句可以为记录变量或行类型变量进行赋值，其表现形式为：**SELECT INTO target select_expressions FROM ...**，该赋值方式一次只能赋值一个变量。表达式中的 target 可以表示为是一个记录变量、行变量，或者是一组用逗号分隔的简单变量和记录/行字段的列表。select_expressions 以及剩余部分和普通 SQL 一样。

如果将一行或者一个变量列表用做目标，那么选出的数值必需精确匹配目标的结构，否则就会产生运行时错误。如果目标是一个记录变量，那么它自动将自己构造成命令结果列的行类型。如果命令返回零行，目标被赋予空值。如果命令返回多行，那么将只有第一行被赋予目标，其它行将被忽略。在执行 SELECT INTO 语句之后，可以通过检查内置变量 FOUND 来判断本次赋值是否成功，如：

```
SELECT INTO myrec * FROM emp WHERE empname = myname;
IF NOT FOUND THEN
    RAISE EXCEPTION 'employee % not found', myname;
END IF;
```

要测试一个记录/行结果是否为空，可以使用 IS NULL 条件进行判断，但是对于返回多条记录的情况则无法判断，如：

```
DECLARE
    users_rec RECORD;
```

```
BEGIN
    SELECT INTO users_rec * FROM users WHERE user_id = 3;
    IF users_rec.homepage IS NULL THEN
        RETURN 'http://';
    END IF;
END;
```

3. 执行一个没有结果的表达式或者命令：

在调用一个表达式或执行一个命令时，如果对其返回的结果不感兴趣，可以考虑使用 PERFORM 语句：**PERFORM query**，该语句将执行 PERFORM 之后的命令并忽略其返回的结果。其中 query 的写法和普通的 SQL SELECT 命令是一样的，只是把开头的关键字 SELECT 替换成 PERFORM，如：

```
PERFORM create_mv('cs_session_page_requests_mv', my_query);
```

4. 执行动态命令：

如果在 PL/pgSQL 函数中操作的表或数据类型在每次调用该函数时都可能会发生变化，在这样的情况下，可以考虑使用 PL/pgSQL 提供的 EXECUTE 语句：**EXECUTE command-string [INTO target]**，其中 command-string 是用一段文本表示的表达式，它包含要执行的命令。而 target 是一个记录变量、行变量或者一组用逗号分隔的简单变量和记录/行域的列表。这里需要特别注意的是，该命令字符串将不会发生任何 PL/pgSQL 变量代换，变量的数值必需在构造命令字符串时插入到该字符串中。

和所有其它 PL/pgSQL 命令不同的是，一个由 EXECUTE 语句运行的命令在服务器内并不会只 prepare 和保存一次。相反，该语句在每次运行的时候，命令都会 prepare 一次。因此命令字符串可以在函数里动态的生成以便于对各种不同的表和字段进行操作，从而提高函数的灵活性。然而由此换来的却是性能上的折损。见如下示例：

```
EXECUTE 'UPDATE tbl SET ' || quote_ident(columnname) || ' = ' || quote_literal(newvalue);
```

五、控制结构：

1. 函数返回：

1). RETURN expression

该表达式用于终止当前的函数，然后再将 expression 的值返回给调用者。如果返回简单类型，那么可以使用任何表达式，同时表达式的类型也将被自动转换成函数的返回类型，就像我们在赋值中描述的那样。如果要返回一个复合类型的数值，则必须让表达式返回记录或者匹配的行变量。

2). RETURN NEXT expression

如果 PL/pgSQL 函数声明为返回 SETOF sometype，其行记录是通过 RETURN NEXT 命令进行填充的，直到执行到不带参数的 RETURN 时才表示该函数结束。因此对于 RETURN NEXT 而言，它实际上并不从函数中返回，只是简单地把表达式的值保存起来，然后继续执行 PL/pgSQL 函数里的下一条语句。随着 RETURN NEXT 命令的迭代执行，结果集最终被建立起来。该类函数的调用方式如下：

```
SELECT * FROM some_func();
```

它被放在 FROM 子句中作为数据源使用。最后需要指出的是，如果结果集数量很大，那么通过该种方式来构建结果集将会导致极大的性能损失。

2. 条件：

在 PL/pgSQL 中有以下三种形式的条件语句。

1). IF-THEN

```
IF boolean-expression THEN
    statements
END IF;
```

2). IF-THEN-ELSE

```
IF boolean-expression THEN
    statements
```

```
ELSE
    statements
END IF;
```

3). IF-THEN-ELSIF-ELSE

```
IF boolean-expression THEN
    statements
ELSIF boolean-expression THEN
    statements
ELSIF boolean-expression THEN
    statements
ELSE
    statements
END IF;
```

关于条件语句，这里就不在做过多的赘述了。

3. 循环：

1). LOOP

```
LOOP
    statements
END LOOP [ label ];
```

LOOP 定义一个无条件的循环，直到由 EXIT 或者 RETURN 语句终止。可选的 label 可以由 EXIT 和 CONTINUE 语句使用，用于在嵌套循环中声明应该应用于哪一层循环。

2). EXIT

```
EXIT [ label ] [ WHEN expression ];
```

如果没有给出 label，就退出最内层的循环，然后执行跟在 END LOOP 后面的语句。如果给出 label，它必须是当前或更高层的嵌套循环块或语句块的标签。之后该命名块或循环就会终止，而控制则直接转到对应循环/块的 END 语句后面的语句上。

如果声明了 WHEN，EXIT 命令只有在 expression 为真时才被执行，否则将直接执行 EXIT 后面的语句。见如下示例：

```
LOOP
    -- do something
    EXIT WHEN count > 0;
END LOOP;
```

3). CONTINUE

```
CONTINUE [ label ] [ WHEN expression ];
```

如果没有给出 label，CONTINUE 就会跳到最内层循环的开始处，重新进行判断，以决定是否继续执行循环内的语句。如果指定 label，则跳到该 label 所在的循环开始处。如果声明了 WHEN，CONTINUE 命令只有在 expression 为真时才被执行，否则将直接执行 CONTINUE 后面的语句。见如下示例：

```
LOOP
    -- do something
    EXIT WHEN count > 100;
    CONTINUE WHEN count < 50;
END LOOP;
```

4). WHILE

```
[ <<label>> ]
WHILE expression LOOP
    statements
END LOOP [ label ];
```

只要条件表达式为真，其块内的语句就会被循环执行。条件是在每次进入循环体时进行判断的。见如下示例：

```
WHILE amount_owed > 0 AND gift_certificate_balance > 0 LOOP
    --do something
```

```
END LOOP;
```

5). FOR

```
[ <<label>> ]
FOR name IN [ REVERSE ] expression .. expression LOOP
    statements
END LOOP [ label ];
```

变量 name 自动被定义为 integer 类型，其作用域仅为 FOR 循环的块内。表示范围上下界的两个表达式只在进入循环时计算一次。每次迭代 name 值自增 1，但如果声明了 REVERSE，name 变量在每次迭代中将自减 1，见如下示例：

```
FOR i IN 1..10 LOOP
    --do something
    RAISE NOTICE 'i IS %', i;
END LOOP;

FOR i IN REVERSE 10..1 LOOP
    --do something
END LOOP;
```

4. 遍历命令结果：

```
[ <<label>> ]
FOR record_or_row IN query LOOP
    statements
END LOOP [ label ];
```

这是另外一种形式的 FOR 循环，在该循环中可以遍历命令的结果并操作相应的数据，见如下示例：

```
FOR rec IN SELECT * FROM some_table LOOP
    PERFORM some_func(rec.one_col);
END LOOP;
```

PL/pgSQL 还提供了另外一种遍历命令结果的方式，和上面的方式相比，唯一的差别是该方式将 SELECT 语句存于字符串文本中，然后再交由 EXECUTE 命令动态的执行。和前一种方式相比，该方式的灵活性更高，但是效率较低。

```
[ <<label>> ]
FOR record_or_row IN EXECUTE text_expression LOOP
    statements
END LOOP [ label ];
```

5. 异常捕获：

在 PL/pgSQL 函数中，如果没有异常捕获，函数会在发生错误时直接退出，与其相关的事物也会随之回滚。我们可以通过使用带有 EXCEPTION 子句的 BEGIN 块来捕获异常并使其从中恢复。见如下声明形式：

```
[ <<label>> ]
[ DECLARE
    declarations ]
BEGIN
    statements
EXCEPTION
    WHEN condition [ OR condition ... ] THEN
        handler_statements
    WHEN condition [ OR condition ... ] THEN
        handler_statements
END;
```

如果没有错误发生，只有 BEGIN 块中的 statements 会被正常执行，然而一旦这些语句中有任意一条发生错误，其后的语句都将被跳过，直接跳转到 EXCEPTION 块的开始处。此时系统将搜索异常条件列表，寻找匹配该异常的第一个条件，如果找到匹配，则执

行相应的 handler_statements，之后再执行 END 的下一条语句。如果没有找到匹配，该错误就会被继续向外抛出，其结果与没有 EXCEPTION 子句完全等同。如果此时 handler_statements 中的语句发生新错误，它将不能被该 EXCEPTION 子句捕获，而是继续向外传播，交由其外层的 EXCEPTION 子句捕获并处理。见如下示例：

```
INSERT INTO mytab(firstname, lastname) VALUES('Tom', 'Jones');
BEGIN
    UPDATE mytab SET firstname = 'Joe' WHERE lastname = 'Jones';
    x := x + 1;
    y := x / 0;
EXCEPTION
    WHEN division_by_zero THEN
        RAISE NOTICE 'caught division_by_zero';
        RETURN x;
END;
```

当以上函数执行到 y := x / 0 语句时，将会引发一个异常错误，代码将跳转到 EXCEPTION 块的开始处，之后系统会寻找匹配的异常捕捉条件，此时 division_by_zero 完全匹配，这样该条件内的代码将会被继续执行。需要说明的是，RETURN 语句中返回的 x 值为 x := x + 1 执行后的新值，但是在除零之前的 update 语句将会被回滚，BEGIN 之前的 insert 语句将仍然生效。

六、游标：

1．声明游标变量：

在 PL/pgSQL 中对游标的访问都是通过游标变量实现的，其数据类型为 refcursor。 创建游标变量的方法有以下两种：

1). 和声明其他类型的变量一样，直接声明一个游标类型的变量即可。

2). 使用游标专有的声明语法，如：

```
name CURSOR [ ( arguments ) ] FOR query;
```

其中 arguments 为一组逗号分隔的 name datatype 列表，见如下示例：

```
curs1 refcursor;
curs2 CURSOR FOR SELECT * FROM tenk1;
curs3 CURSOR (key integer) IS SELECT * FROM tenk1 WHERE unique1 = key;
```

在上面三个例子中，只有第一个是未绑定游标，剩下两个游标均已被绑定。

2．打开游标：

游标在使用之前必须先被打开，在 PL/pgSQL 中有三种形式的 OPEN 语句，其中两种用于未绑定的游标变量，另外一种用于绑定的游标变量。

1). OPEN FOR:

其声明形式为：

```
OPEN unbound_cursor FOR query;
```

该形式只能用于未绑定的游标变量，其查询语句必须是 SELECT，或其他返回记录行的语句，如 EXPLAIN。在 PostgreSQL 中，该查询和普通的 SQL 命令平等对待，即先替换变量名，同时也将该查询的执行计划缓存起来，以供后用。见如下示例：

```
OPEN curs1 FOR SELECT * FROM foo WHERE key = mykey;
```

2). OPEN FOR EXECUTE

其声明形式为：

```
OPEN unbound_cursor FOR EXECUTE query-string;
```

和上面的形式一样，该形式也仅适用于未绑定的游标变量。EXECUTE 将动态执行其后以文本形式表示的查询字符串。

```
OPEN curs1 FOR EXECUTE 'SELECT * FROM ' || quote_ident($1);
```

3). 打开一个绑定的游标

其声明形式为：

```
OPEN bound_cursor [ ( argument_values ) ];
```

该形式仅适用于绑定的游标变量，只有当该变量在声明时包含接收参数，才能以传递参数的形式打开该游标，这些参数将被实际代

入到游标声明的查询语句中，见如下示例：

```
OPEN curs2;
OPEN curs3(42);
```

3. 使用游标：

游标一旦打开，就可以按照以下方式进行读取。然而需要说明的是，游标的打开和读取必须在同一个事物内，因为在 PostgreSQL 中，如果事物结束，事物内打开的游标将会被隐含的关闭。

1). FETCH

其声明形式为：

```
FETCH cursor INTO target;
```

FETCH 命令从游标中读取下一行记录的数据到目标中，其中目标可以是行变量、记录变量，或者是一组逗号分隔的普通变量的列表，读取成功与否，可通过 PL/pgSQL 内置变量 FOUND 来判断，其规则等同于 SELECT INTO。见如下示例：

```
FETCH curs1 INTO rowvar;  --rowvar 为行变量
FETCH curs2 INTO foo, bar, baz;
```

2). CLOSE

其声明形式为：

```
CLOSE cursor;
```

关闭当前已经打开的游标，以释放其占有的系统资源，见如下示例：

```
CLOSE curs1;
```

七、错误和消息：

在 PostgreSQL 中可以利用 RAISE 语句报告信息和抛出错误，其声明形式为：

```
RAISE level 'format' [, expression [, ...]];
```

这里包含的级别有 DEBUG(向服务器日志写信息)、LOG(向服务器日志写信息，优先级更高)、INFO、NOTICE 和 WARNING(把信息写到服务器日志以及转发到客户端应用，优先级逐步升高)和 EXCEPTION 抛出一个错误(通常退出当前事务)。某个优先级别的信息是报告给客户端还是写到服务器日志，还是两个均有，是由 log_min_messages 和 client_min_messages 这两个系统初始化参数控制的。

在 format 部分中，%表示为占位符，其实际值仅在 RAISE 命令执行时由后面的变量替换，如果要在 format 中表示%自身，可以使用%%的形式表示，见如下示例：

```
RAISE NOTICE 'Calling cs_create_job(%)',v_job_id;  --v_job_id 变量的值将替换 format 中的%。
RAISE EXCEPTION 'Inexistent ID --> %',user_id;
```