前言

今天领导找到我提了这么一个需求：想实现权限的精细化控制，然后将这些角色授予给其他业务部门的DBA，不过仅能实现某些操作，这样就不需要我们全权负责了，对方也能做某类运维操作。权限体系共分为三个层级：

- 第一层，允许查询系统表和业务表
- 第二层：包含部分数据库的管理权限，比如杀会话、在SQL跑慢的时候允许创建索引，扩展表空间，收集统计信息，执行 vacuum 等管理操作
- 第三层：拥有业务表的修改权限

实现

让我们一起看看这个需求如何实现。首先第一层很好实现，只需创建一个 `readonly` 角色，相应表的查询权限授予给该角色，后续通过该角色来划分权限。

第三层也很好实现，直接 `grant delete,insert,update on xxx to xxx` 即可。

麻烦一点的是第二层，比如创建索引，需要是表的 owner 或者 superuser 才可以

```
postgres=# create table t1(id int);
CREATE TABLE
postgres=# \c postgres u1
psql (15beta1, server 14.2)
You are now connected to database "postgres" as user "u1".
postgres=> create index on t1(id);
ERROR:  must be owner of table t1
```

👆 报错很明显，同理，vacuum/analyze 等也是类似

```
postgres=> vacuum t1;
WARNING:  skipping "t1" --- only table or database owner can vacuum it
VACUUM
postgres=> analyze t1;
WARNING:  skipping "t1" --- only table or database owner can analyze it
ANALYZE
postgres=> analyze verbose t1;
WARNING:  skipping "t1" --- only table or database owner can analyze it
ANALYZE
```

而杀会话的权限，从 9.6 以后提供了 pg_signal_backend 角色，Signal another backend to cancel a query or terminate its session. 按需授予即可。

那么该如何实现上面这几个功能？这里就必须得提到PostgreSQL中的另一个骚操作了：函数安全性。

```
postgres=# \h create function
Command:     CREATE FUNCTION
Description: define a new function
Syntax:
```

```
CREATE [ OR REPLACE ] FUNCTION
    name ( [ [ argmode ] [ argname ] argtype [ { DEFAULT | = } default_expr ] [, ...] ]
)
    [ RETURNS rettype
      | RETURNS TABLE ( column_name column_type [, ...] ) ]
  { LANGUAGE lang_name
    | TRANSFORM { FOR TYPE type_name } [, ... ]
    | WINDOW
    | { IMMUTABLE | STABLE | VOLATILE }
    | [ NOT ] LEAKPROOF
    | { CALLED ON NULL INPUT | RETURNS NULL ON NULL INPUT | STRICT }
    | { [ EXTERNAL ] SECURITY INVOKER | [ EXTERNAL ] SECURITY DEFINER }      ---👈此处
    | PARALLEL { UNSAFE | RESTRICTED | SAFE }
    | COST execution_cost
    | ROWS result_rows
    | SUPPORT support_function
    | SET configuration_parameter { TO value | = value | FROM CURRENT }
    | AS 'definition'
    | AS 'obj_file', 'link_symbol'
    | sql_body
  } ...

URL: https://www.postgresql.org/docs/15/sql-createfunction.html
```

在创建函数的时候可以同时指定权限，分为 SECURITY INVOKER 和 SECURITY DEFINER，前者以调用者权限执行函数，这是默认的安全环境。后者以创建者权限执行函数，在函数执行期间以创建该函数的权限执行。

> `SECURITY INVOKER` indicates that the function is to be executed with the privileges of the user that calls it. That is the default. `SECURITY DEFINER` specifies that the function is to be executed with the privileges of the user that owns it.

那么我们完全可以使用这个特性来包装一层函数，如下

```
create or replace function dba_operation(operation_type anyelement,operation_details
text)
returns void
as $$
declare
  index_ops text :='create index';      ---创建索引
  kill_ops text  :='kill session';      ---查杀会话
  analyze_ops text :='analyze table';   ---统计信息
begin
  IF lower(operation_type) = index_ops or lower(operation_type) = kill_ops or
lower(operation_type) = analyze_ops THEN
    execute operation_details;
  ELSE
    raise exception 'error! undefined operations!';
  END IF;
END;
```

```
$$ language plpgsql SECURITY DEFINER;
```

将 dba_operation 函数定义为`SECURITY DEFINER`，这样以超级用户创建该函数并将函数的使用权限授予给用户，那么即使对应的用户没有对象的权限，也能以"superuser"的身份去执行，看个例子

```
postgres=> create index on t1(id);
ERROR:  must be owner of table t1
postgres=> select dba_operation('create index'::text,'create index on t1(id);');
 dba_operation
---------------

(1 row)

postgres=> \d t1
                Table "public.t1"
 Column |  Type   | Collation | Nullable | Default
--------+---------+-----------+----------+---------
 id     | integer |           |          |
Indexes:
    "t1_id_idx" btree (id)
```

可以看到索引创建成功了，成功绕过了权限的限制。但是这个函数有漏洞，我仅仅做了操作类型的判断，并没有判断具体执行的SQL，因此完全可以搞一些破坏，比如

```
postgres=> select dba_operation('create index'::text,'drop table t1');
 dba_operation
---------------

(1 row)

postgres=> \d t1
Did not find any relation named "t1".
```

好家伙，直接把表给干没了！因此，我们还需要去判断一下具体SQL，预防一些可能的风险，这里就简单写一下，各位可以按需扩充

```
create or replace function dba_operation(operation_type anyelement,operation_details
text)
returns void
as $$
declare
  ret boolean;
  index_ops text :='create index';
  kill_ops text  :='kill session';
  analyze_ops text :='analyze table';
begin
  IF lower(operation_type) = index_ops THEN
    select operation_details ~ 'index' into ret;
```

```
    raise notice 'ret is %',ret;
    IF ret is true THEN
      execute operation_details;
    ELSE
      raise exception 'error! dangerous ';
    END IF;
  ELSIF lower(operation_type) = kill_ops THEN
    select operation_details ~ 'cancel' into ret;
    IF ret is true THEN
      execute operation_details;
    ELSE
      raise exception 'error! dangerous ';
    END IF;
  ELSIF lower(operation_type) = analyze_ops THEN
    select operation_details ~ 'analyze' into ret;
    IF ret is true THEN
      execute operation_details;
    ELSE
      raise exception 'error! dangerous ';
    END IF;
  ELSE
    raise exception 'error! undefined operations!';
  END IF;
END;
$$ language plpgsql SECURITY DEFINER;
```

我使用正则表达式去判断具体执行的SQL是否包含对应的操作

```
ostgres=> create index on t1(id);
ERROR:  must be owner of table t1
postgres=> select dba_operation('create index','drop table t1');
ERROR:  could not determine polymorphic type because input has type unknown
postgres=> select dba_operation('create index'::text,'drop table t1');
NOTICE:  ret is f
ERROR:  error! dangerous
CONTEXT:  PL/pgSQL function dba_operation(anyelement,text) line 14 at RAISE
postgres=> select dba_operation('create index'::text,'alter table t1 add column t_time
timestamp');
NOTICE:  ret is f
ERROR:  error! dangerous
CONTEXT:  PL/pgSQL function dba_operation(anyelement,text) line 14 at RAISE
postgres=> select dba_operation('create index'::text,'create index on t1(id)');
NOTICE:  ret is t
 dba_operation
---------------

(1 row)

postgres=> \d t1
```

```
                Table "public.t1"
 Column |  Type   | Collation | Nullable | Default
--------+---------+-----------+----------+---------
 id     | integer |           |          |
Indexes:
    "t1_id_idx" btree (id)
```

这样就严谨多了，对于这类 SQL 注入会直接退出，perfect ~

小结

通过这个例子，想必各位能得到一些启发了，我可以利用函数安全性去实现一些精细化的权限控制，当然也可以"越权"去做一些事。

举个典型的例子，现在有个监控用户想去审计执行的SQL，常规方式是不行的，会提示权限不足，如下

```
postgres=# grant SELECT(query) on pg_stat_activity to u1;
GRANT
postgres=# \c postgres u1
You are now connected to database "postgres" as user "u1".
postgres=> select query from pg_stat_activity ;
                query
--------------------------------------
 <insufficient privilege>
 <insufficient privilege>
 <insufficient privilege>
 select query from pg_stat_activity ;
 <insufficient privilege>
 <insufficient privilege>
 <insufficient privilege>
(7 rows)
```

query 字段你是看不到的，因此我们也可以利用函数的这个特性来实现：用超级用户定义一个函数，并且函数的权限是`SECURITY DEFINER`，然后赋予函数的`execute`权限给相应用户即可。怎么样？是不是学到了一个非常棒的技巧？不过要切勿滥用啊，搞不好就提桶跑路了。

视图也有类似的权限控制，可以戳这篇 PostgreSQL 15新特性预览：视图的security invoker选项。

!