Info:结构体
入口

Create A Trigger

Basic form(Triggers on Data Changes)

A trigger is a specification that the database should automatically execute a particular function whenever a certain type of operation is performed.

```
CREATE [ CONSTRAINT ] TRIGGER name { BEFORE | AFTER | INSTEAD OF } { event [ OR
... ] }
    ON table_name
    [ FROM referenced_table_name ]
    [ NOT DEFERRABLE | [ DEFERRABLE ] [ INITIALLY IMMEDIATE | INITIALLY DEFERRED
] ]
    [ REFERENCING { { OLD | NEW } TABLE [ AS ] transition_relation_name } [ ...
] ]
    [ FOR [ EACH ] { ROW | STATEMENT } ]
    [ WHEN ( condition ) ]
    EXECUTE { FUNCTION | PROCEDURE } function_name ( arguments )
```

Tips: `event` here could be one of follow

- INSERT
- UPDATE [OF column_name [, ...]]
- DELETE
- TRUNCATE

The following conventions are used in the synopsis of a command:
brackets ([and]) indicate optional parts. (In the synopsis of a Tcl command, question marks (?) are used instead, as is usual in Tcl.)
Braces ({ and }) and vertical lines (|) indicate that you must choose one alternative.
Dots (...) mean that the preceding element can be repeated.

Classification

1. by location where triggers can be attached to

- tables (partitioned or not)
- views
- foreign tables

2. by which trigger affected on

- `row`: With a per-row trigger, the trigger function is invoked **once for each row** that is affected by the statement that fired the trigger
- `statement`: a per-statement trigger is invoked **only once** when an appropriate statement is executed, regardless of the number of rows affected by that statement. In particular, a statement that affects zero rows will still result in the execution of any applicable per-statement triggers

3. by whether they fire *before*, *after*, or *instead of* the operation

- `before`:
- `after`:
- `instead of`: may only be defined on views, and only at row level; they fire immediately as each row in the view is identified as needing to be operated on.

Suitable uses summarizes

when	Event	Row-level	Statement-level
Before	INSERT/UPDATE/DELETE	Tables and foreign tables	Tables, views, and foreign tables
Before	TRUNCATE	-	Tables
AFTER	INSERT/UPDATE/DELETE	Tables and foreign tables	Tables
AFTER	TRUNCATE	-	Tables
INSTEAD OF	INSERT/UPDATE/DELETE	Views	-
INSTEAD OF	TRUNCATE	-	-

Trigger and Trigger Function

Experiment 1:

```
CREATE FUNCTION emp_stamp() RETURNS trigger
AS
$emp_stamp$
    BEGIN
        -- Check that salary are given
        IF NEW.name IS NULL THEN
            RAISE EXCEPTION 'emp_name cannot be null';
        END IF;
        IF NEW.salary IS NULL THEN
            RAISE EXCEPTION '% cannot have null salary', NEW.name;
        END IF;

        -- Who works for us when they must pay for it?
        IF NEW.salary < 0 THEN
            RAISE EXCEPTION '% cannot have a negative salary', NEW.name;
        END IF;
    END;
$emp_stamp$ LANGUAGE plpgsql;
```

The trigger function must be defined before the trigger itself can be created. The trigger function must be declared as a function taking no arguments and returning type `trigger`.

The same trigger function can be used for multiple triggers.

The trigger function should be **writen in procedure language(PL) or C**, **could not in pure SQL**.

Tips:

- First comes trigger function, then triggers
- trigger function : triggers = 1 : n

Data passed in Trigger procedure

When a PL/pgSQL function is called as a trigger, several special variables are **created automatically in the top-level block**. They are:

You can use them without declare

Variable	Data type	Detail
NEW	RECORD	variable holding the new database row for `INSERT`/`UPDATE` operations in row-level triggers. This variable is null in statement-level triggers and for `DELETE` operations.
OLD	RECORD	variable holding the old database row for UPDATE/DELETE operations in row-level triggers. This variable is null in statement-level triggers and for INSERT operations.
TG_NAME	name	variable holding the old database row for `UPDATE`/`DELETE` operations in row-level triggers. This variable is null in statement-level triggers and for `INSERT` operations.
TG_WHEN	text	a string of `BEFORE`, `AFTER`, or `INSTEAD OF`, depending on the trigger's definition.
TG_LEVEL	text	a string of either `ROW` or `STATEMENT` depending on the trigger's definition.
TG_OP	text	a string of `INSERT`, `UPDATE`, `DELETE`, or `TRUNCATE` telling for which operation the trigger was fired.
TG_RELID	oid	the object ID of the table that caused the trigger invocation.
TG_RELNAME	name	the name of the table that caused the trigger invocation. This is now deprecated, and could disappear in a future release. Use `TG_TABLE_NAME` instead.
TG_TABLE_NAME	name	the name of the table that caused the trigger invocation.
TG_TABLE_SCHEMA	name	the name of the schema of the table that caused the trigger invocation.
TG_NARGS	integer	the number of arguments given to the trigger function in the `CREATE TRIGGER` statement.
TG_ARGV[]	array of text	the arguments from the `CREATE TRIGGER` statement. The index counts from 0. Invalid indexes (less than 0 or greater than or equal to `tg_nargs`) result in a null value.

trigger operation

Here we use following tables to be attached to

```
create table dept(
id int primary key,
name varchar(40),
office varchar(40) default 'Nanshan'
```

```
);

insert into dept values(1, '开发部','Bao''an');
insert into dept values(2, '测试部','Bao''an');
insert into dept values(3, '财务部');
insert into dept values(4, '市场部','Bao''an');
insert into dept values(5, '总经办');
insert into dept values(6, '生产部','Bao''an');

create table emp(
id int primary key,
name varchar(10),
salary numeric(7,2) default 0.0,
deptId int,
constraint fk_emp3_dept foreign key(deptId) references dept(id)
);

insert into emp values
                        (001,'张三',15000, 1),
                        (002,'李四',25000, 1),
                        (003,'王五',10000, 2),
                        (004,'赵六',6000, 6),
                        (005,'庄七',5000, 6),
                        (006,'郝八',12000, 3),
                        (007,'段九',7500, 3),
                        (008,'殷十',15000, 4),
                        (009,'周一',15000, 4),
                        (010,'范二',25000, 5),
                        (011,'何玲',15000, 5),
                        (012,'李雷',7000, 6),
                        (013,'刘小毛',4500, 6),
                        (014,'张大壮',6000, 6);
```

Experiment 2 : Create a trigger function

```
CREATE TABLE emp_log(
    operation         char(1)   NOT NULL,
    stamp             timestamp NOT NULL,
    userid            text      NOT NULL,
    level             text      NOT NULL,
    when_trigger      text      NOT NULL,
    id                integer   ,
    name              varchar(10) ,
    salary numeric
);

CREATE OR REPLACE FUNCTION process_emp_check() RETURNS TRIGGER AS $emp_check$
    BEGIN
        --
        -- Create a row in emp_audit to reflect the operation performed on emp,
        -- making use of the special variable TG_OP to work out the operation.
        --
        IF (TG_OP = 'DELETE') THEN
            INSERT INTO emp_log SELECT 'D', now(), user, TG_LEVEL, TG_WHEN,
OLD.id, OLD.name, OLD.salary;--
```

```
        ELSIF (TG_OP = 'UPDATE') THEN
            INSERT INTO emp_log SELECT 'U', now(), user, TG_LEVEL, TG_WHEN,
NEW.id, NEW.name, NEW.salary;
        ELSIF (TG_OP = 'INSERT') THEN
            INSERT INTO emp_log SELECT 'I', now(), user, TG_LEVEL, TG_WHEN,
NEW.id, NEW.name, NEW.salary;
        END IF;
        RETURN NULL; -- result is ignored since this is an AFTER trigger
    END;
$emp_check$ LANGUAGE plpgsql;
```

Be careful : the String is in SINGLE quotation

Experiment 3: after + each row + table

```
CREATE TRIGGER check_after
    AFTER INSERT OR UPDATE OR DELETE ON emp
    FOR EACH ROW
    EXECUTE FUNCTION process_emp_check();

--test, and see result in emp_log
insert into emp values (017,'贺小山', 7000, 1);
UPDATE emp set salary=8000 where id=017;
delete from emp where id=017;
```

Experiment 4: before + each row + table

```
CREATE TRIGGER check_before
    BEFORE INSERT OR UPDATE OR DELETE ON emp
    FOR EACH ROW
    EXECUTE FUNCTION process_emp_check();

--test, and see result in emp_log
insert into emp values (017,'贺小山', 7000, 1);
UPDATE emp set salary=8000 where id=017;
delete from emp where id=017;
```

```
CREATE TRIGGER check_before_insert
    BEFORE INSERT ON emp
    FOR EACH ROW
    EXECUTE FUNCTION process_emp_check();

CREATE TRIGGER check_before_update
    BEFORE UPDATE ON emp
    FOR EACH ROW
    EXECUTE FUNCTION process_emp_check();

CREATE TRIGGER check_before_delete
    BEFORE DELETE ON emp
    FOR EACH ROW
    EXECUTE FUNCTION process_emp_check();

--test
```

```
insert into emp values (017,'贺小山', 7000, 1);
UPDATE emp set salary=8000 where id=017;
delete from emp where id=017;

UPDATE emp set salary=salary*1.2;

delete from emp where id=013;
```

Experiment 5: after + each row + table.col

```
CREATE TRIGGER check_update_col
    AFTER UPDATE OF salary ON emp
    FOR EACH ROW
    --WHEN (OLD.salary IS DISTINCT FROM NEW.salary)
    EXECUTE FUNCTION process_emp_check();

--test
UPDATE emp set deptId = 1 where id=12;
UPDATE emp set salary=salary*1.2 where id=13;
```

Experiment 6: after + each row + table.col + when, the trigger function is executed when col has in fact changed value

```
CREATE TRIGGER check_update_col_when
    AFTER UPDATE OF salary ON emp
    FOR EACH ROW
    WHEN (OLD.salary IS DISTINCT FROM NEW.salary)
    EXECUTE FUNCTION process_emp_check();

--test
UPDATE emp set deptId = 6 where id=12;
UPDATE emp set salary=salary*0.9 where id=13;
```

Experiment 7: after + each row + table + when, the trigger function is executed when table has in fact changed value, you can test yourself

```
CREATE TRIGGER check_update_col_when
    AFTER UPDATE ON emp
    FOR EACH ROW
    WHEN (OLD.* IS DISTINCT FROM NEW.*)
    EXECUTE FUNCTION process_emp_check();
```

Experiment 8: after + each statement + table

```
CREATE TRIGGER check_after_statement
    AFTER INSERT OR UPDATE OR DELETE ON emp
    FOR EACH STATEMENT
    EXECUTE FUNCTION process_emp_check();

--test
insert into emp values (017,'贺小山', 7000, 1);
UPDATE emp set salary=8000 where id=017;
delete from emp where id=017;

UPDATE emp set salary=salary*1.15;
```

Experiment 9: before + each statement + table

```
CREATE TRIGGER check_before_statement
    BEFORE INSERT OR UPDATE OR DELETE ON emp
    FOR EACH STATEMENT
    EXECUTE FUNCTION process_emp_check();

--test
insert into emp values (017,'贺小山', 7000, 1);
UPDATE emp set salary=8000 where id=017;
delete from emp where id=017;

UPDATE emp set salary=salary*0.95;
```

Notice: triggers on view and other situations, please read example in following reference

ref:http://postgres.cn/docs/11/plpgsql-trigger.html

Event Trigger

Basic Form

```
CREATE EVENT TRIGGER name
    ON event
    [ WHEN filter_variable IN (filter_value [, ... ]) [ AND ... ] ]
    EXECUTE { FUNCTION | PROCEDURE } function_name()
```

Experiment 10

```
CREATE OR REPLACE FUNCTION event_tr() RETURNS event_trigger AS $$
BEGIN
    RAISE NOTICE 'Operation here: % %', tg_event, tg_tag;
END;
$$ LANGUAGE plpgsql;

CREATE EVENT TRIGGER tr_fun_event ON ddl_command_start EXECUTE FUNCTION
event_tr();

create table test(
    test1 int,
    test2 int
);
drop table test;
```

classification

- ddl_command_start
- ddl_command_end
- sql_drop
- table_rewrite

Tips: Show triggers information

```
select *
from pg_catalog.pg_trigger;
```