**PL/pgSQL简介**

PostgreSQL允许使用各种过程语言来扩展数据库功能。

PostgreSQL将过程语言分为两大类：

任何用户都可以使用安全语言。SQL和PL/pgSQL是安全的语言。

沙盒语言仅由超级用户使用，因为沙盒语言提供了绕过安全性并允许访问外部源的功能。C语言是沙盒语言。

缺省情况下，PostgreSQL支持三种过程语言：SQL，PL/pgSQL和C。还可以使用扩展将其他过程语言（例如Perl，Python和TCL）加载到PostgreSQL中。

PL/pgSQL是一个功能强大的SQL脚本语言，该过程添加了许多过程元素，例如控制结构，循环和复杂的计算，以扩展SQL标准。它可以在PostgreSQL中开发复杂的功能，而这是使用普通SQL可能无法实现的。

**选择PL/pgSQL的原因：**

1.PL/pgSQL简单易学。

2.PostgreSQL默认附带PL/pgSQL。在PL/pgSQL中开发的用户定义函数可以像任何内置函数一样使用。

3.PL/pgSQL具有许多功能，可让您开发复杂的用户定义函数。

**PL/pgSQL的优点：**

减少应用程序和数据库服务器之间的往返次数。所有SQL语句都存储在PostgreSQL数据库服务器中的函数内，因此应用程序仅需发出函数调用即可返回结果，而不必发送多个SQL语句并等待每次调用之间的结果。

由于用户定义的函数和存储过程已预编译并存储在PostgreSQL数据库服务器中，因此提高了应用程序性能。

可在许多应用程序中重用。一旦开发了功能，就可以在任何应用程序中重用它。

**PL/pgSQL 的缺点**

软件开发速度较慢，因为 PL/pgSQL 需要许多开发人员不具备的专业技能。

难以管理版本且难以调试。

可能无法移植到其他数据库管理系统，例如MySQL或Microsoft SQL Server。

**$$符号的使用**

1.使用单引号作为字符串常量

select 'hello world';

2.当字符串常量包含单引号 ( ') 时，需要通过加倍单引号来进行转义。

select 'I''m postgresql';

PostgreSQL 引入了美元符号引用功能，使字符串常量更具可读性。

**语法：**

$tag$<string\_constant>$tag$

在此语法中，tag是可选的。它可能包含零个或多个字符。

在两个$tag$之间，你可以放置任何带有单引号 ( ') 和反斜杠 ( ) 的字符串。

select $$I'm postgresql $$;

使用带标记的美元符语法

SELECT $msg$I'm postgresql $msg$;

**在匿名块中使用$$**

do

'begin

raise notice ''this is a block'';

end';

块中的代码必须用单引号括起来。如果它内部有任何单引号，则需要将其转义：

raise notice ''this is a block'';

DO语句不属于该块。它用于执行匿名块。

do $$

begin

raise notice 'this is a block';

end $$;

**在函数中使用$$**

create function f\_add(x integer, y integer)

returns integer

language plpgsql

as

'begin

return x + y;

end';

SELECT f\_add(1, 2);

在这种语法中，function\_body是一个字符串常量，用单引号括起来。

如果函数有很多语句，就会变得难阅读。在这种情况下，可以使用美元符

create function f\_add(x integer, y integer)

returns integer

language plpgsql

as $$

begin

return x + y ;

end $$;

**在存储过程中使用$$**

create procedure sp\_delete(n int)

language plpgsql

as $$

begin

delete from t1 where id=n;

end $$;

**PL/pgSQL 块结构**

PL/pgSQL 是一种块结构语言，因此，PL/pgSQL函数或存储过程被组织成块。

**PL/pgSQL 中完整块的语法：**

[ <<**label**>> ]

[ DECLARE

**declarations** ]

BEGIN

statements;

END [ **label** ];

块说明：

1.每个块有两个部分：声明和正文。声明部分是可选的，而正文部分是必需的。块以关键字;后的分号结束。

2.块可能有一个位于开头和结尾的可选标签label。当想在块体的语句中指定它或者当你想限定在块中声明的变量的名称时，你可以使用块标签

3.声明部分是声明在正文中所使用的变量的地方。声明部分中的每条语句都以分号 ( ;) 结束。

4.正文部分是放置代码的地方。正文部分中的每个语句也以分号 ; 结束。

**PL/pgSQL 块结构示例**

DO $$

DECLARE

counter integer := 0;

BEGIN

counter := counter + 1;

RAISE NOTICE 'The current value of counter is %', counter;

END $$;

**PL/pgSQL 子块**

PL/pgSQL 允许将一个块放置在另一个块的主体内。嵌套在另一个块中的块称为子块。包含子块的块称为外块。

子块用于对语句进行分组，以便可以将大块划分为较小和更多的逻辑子块。子块中的变量可以具有与外部块中的变量相同的名称。

当在子块中声明与外部块中名称相同的变量时，外部块中的变量将隐藏在子块中。如果要访问外部块中的变量，请使用块标签来限定其名称：

DO $$

<<outer\_block>>

DECLARE

counter integer := 0;

BEGIN

counter = counter + 1;

RAISE NOTICE 'The current value of counter is %', counter;

DECLARE

counter integer := 0;

BEGIN

counter = counter + 10;

RAISE NOTICE 'The current value of counter in the subblock is %', counter;

RAISE NOTICE 'The current value of counter in the outer block is %', outer\_block.counter;

END;

RAISE NOTICE 'The current value of counter in the outer block is %', counter;

END outer\_block $$;

**小结**

1.PL/pgSQL 是一种块结构语言。它将程序组织成块。

2.块包含两部分：声明和主体。声明部分是可选的，而正文部分是强制性的。

3.块可以嵌套。嵌套块是放置在另一个块的主体内的块

**变量和常量**

**变量**

在使用变量之前，必须在PL/pgSQL 块的声明部分中声明它

声明语法:

variable\_name data\_type [:= expression];

1.使用特定数据类型与变量相关联。数据类型可以是任何有效的数据类型，例如integer、numeric、varchar和 char。

2.可选地为变量分配默认值。如果不这样做，变量的初始值为NULL

3.可以使用:=或=赋值运算符来初始化变量并将值分配给变量。

示例1：

do $$

declare

id integer := 1;

name text := 'zhangsan';

age varchar(50) := '18';

begin

raise notice 'the user row is %,%,%',id,name,age;

end $$;

示例2：

do $$

declare

create\_time time := now();

begin

raise notice '%', create\_time;

perform pg\_sleep(2);

raise notice '%', create\_time;

end $$;

**复制数据类型**

使用%type来复制表中某个字段（或某个已经存在的变量）的数据类型

variable\_name table\_name.column\_name%type;

示例

do $$

declare

v\_name users.name%type;

begin

select name from users into v\_name where id=20;

raise notice 'username is %s',v\_name;

end $$;

**行类型变量**

要存储语句返回的结果集的整行，需要先声明行类型变量或行变量

声明语法：

row\_variable table\_name%ROWTYPE;

要访问行变量的单个字段，使用点表示法 ( .) ：

row\_variable.field\_name

示例：（显示 id 10 的名字和年龄）

do $$

declare

v\_user users%rowtype;

begin

select \* from users into v\_user where id=10;

raise notice 'The user name is %,age is %', v\_user.name, v\_user.age;

end $$;

**记录类型变量**

记录类型允许定义可以保存结果集中单行的变量

record变量类似于行类型变量，不同的是，它只能保存结果集的一行

语法：variable\_name record;

示例1

do $$

declare

rec record;

begin

select name,age from users into rec where id=10;

raise notice 'name is %,age is %', rec.name, rec.age;

end;

$$;

示例2：（在for循环语句中使用记录变量）

1.首先，声明一个名为 rec 的变量，其类型为record。

2.其次，使用for loop语句从t1表中获取行。该语句在每次迭代中由和for loop组成的行分配给变量

3.使用点表示法显示记录变量的字段内容

do $$

declare

rec record;

begin

for rec in select name,age from users where id<=10

loop

raise notice '% (%)', rec.name, rec.age;

end loop;

end $$;

记录是一个占位符，可以保存结果集的单行。

记录没有像行变量那样的预定义结构。它的结构是在您为其分配行时确定的。

**常量**

与变量不同，常量的值一旦初始化就不能更改，常量使代码更具可读性和可维护性

定义常量：

constant\_name constant data\_type := expression;。

示例：

do $$

declare

age constant int := 10;

age\_new int := 5;

begin

raise notice 'The new age is %', age+age\_new;

end $$;

更改常量值会报错：

do $$

declare

age constant int := 10;

age\_new int := 5;

begin

raise notice 'The new age is %', age+age\_new;

age = 15;

end $$;

注意，在声明块中的函数，PostgreSQL会计算常量的值。

DO $$

DECLARE

start\_at CONSTANT time := now();

BEGIN

RAISE NOTICE 'Start executing block at %', start\_at;

perform pg\_sleep(2);

RAISE NOTICE 'Start executing block at %', start\_at;

END $$;

**消息和错误**

使用raise语句报告消息和引发错误，使用assert语句将调试检查信息插入到 PL/pgSQL 块中

**报告消息**

语法格式：

raise level format;

**等级**

level指定错误严重性级别，PostgreSQL 提供以下级别：

1.debug

2.log

3.notice

4.info

5.warning

6.exception

如果不指定level，默认为exception，它会引发错误并停止当前事务。

示例

do $$

begin

raise debug 'debug message %', now();

raise log 'log message %', now();

raise notice 'notice message %', now();

raise info 'information message %', now() ;

raise warning 'warning message %', now();

end $$;

注意：并非所有消息都报告给客户端。PostgreSQL仅将notice级别的消息返回给客户端。这由client\_min\_messages 配置参数控制的。

**引发错误**

要引发错误，请使用语句exception后的级别。raise语句默认使用级别为exception。

除了引发错误之外，还可以添加更多信息：

using option = expression

option可以是：

1.message: 设置错误信息

2.hint：提供提示信息，以便更容易发现错误的根本原因。

3.detail: 提供有关错误的详细信息。

4.errcode：识别错误代码，可以是条件名称，也可以直接是五字符SQLSTATE代码。

PostgreSQL错误代码：http://www.postgres.cn/docs/14/errcodes-appendix.html

示例：

do $$

declare

name varchar(50) := 'zhangsan';

begin

raise exception 'username: %', name

using hint = 'check the name again';

end $$;

示例：引发SQLSTATE及其对应的条件：

do $$

begin

raise sqlstate 'P0002';

end $$;

do $$

begin

raise no\_data\_found;

end $$;

**assert**

PostgreSQL 断言语句以及如何将其用于调试

assert语法：

assert condition [, message];

1.条件

是condition一个布尔表达式，预期总是返回true。

如果condition计算结果为true，则该assert语句不执行任何操作。

如果condition计算结果为false or null，PostgreSQL 会引发assert\_failure异常。

2.留言

该消息是可选的。

如果你不通过message，PostgreSQ Lassertion failed默认使用“ ”消息。如果您将 传递message给assert语句，它将使用它来替换默认消息。

请注意，您应该将该assert语句仅用于检测错误，而不是用于报告。要报告消息或错误，请改用raise语句。

**启用/禁用断言**

PostgreSQL 提供plpgsql.check\_asserts配置参数来启用或禁用断言测试。如果将此参数设置为off，则 assert 语句将不执行任何操作。

示例：使用该assert语句检查变量count是否大于5：

do $$

declare

count int:=10;

begin

raise notice 'this is assert test';

assert count > 5, 'count < 0,do something';

end $$;

因为表有数据，所以块没有发出任何消息。

以下示例发出错误，因为count<5。

do $$

declare

count int:=3;

begin

raise notice 'this is assert test';

assert count > 5, 'count < 0,do something';

end $$;

**小结：**

1.使用assert语句将调试检查添加到 PL/pgSQL 代码。

2.assert语句评估condition预期为ture，并在条件为false的情况下发出错误。

3.assert语句仅用于检测错误。要报告普通消息和错误，请改用该raise语句。

**控制语句**

**if语句**

PL/pgSQL 提供了三种形式的if语句。

if then

if then else

if then elsif

**if语句的简单形式：**

if condition then

statements;

end if;

示例：

do $$

declare

v\_user users%rowtype;

v\_id int=1;

begin

select \* from users into v\_user where id=v\_id;

if not found then

raise notice 'The user % could not be found',v\_id;

end if;

end $$;

这found是一个在 PL/pgSQL 过程语言中可用的全局变量

**if-then-else 语句**

语法

if condition then

statements;

else

alternative-statements;

END if;

示例：

do $$

declare

v\_user users%rowtype;

v\_id int=1;

begin

select \* from users into v\_user where id=v\_id;

if not found then

raise notice'The user % could not be found', v\_id;

else

raise notice 'The user is %', v\_id;

end if;

end $$;

**if-then-elsif 语句**

语法：

if condition\_1 then

statement\_1;

elsif condition\_2 then

statement\_2

...

elsif condition\_n then

statement\_n;

else

else-statement;

end if;

示例

do $$

declare

v\_user users%rowtype;

result text;

begin

select \* from users into v\_user where id=1;

if not found then

result:='The user could not be found';

elseif v\_user.age>=0 and v\_user.age<=10 then

result:='age bewtten 0 and 10';

elseif v\_user.age>=11 and v\_user.age<=20 then

result:='age bewtten 11 and 20';

elseif v\_user.age>=21 and v\_user.age<=30 then

result:='age bewtten 21 and 30';

else

result:='N/A';

end if;

raise notice'The user % ', result;

end $$;

**case语句**

case声明有两种形式：

1.简单的case语句

2.搜索case语句

**简单的case语句**

do $$

declare

score int=60;

result text;

begin

case score

when 90 then result:='优';

when 80 then result:='良';

when 70 then result:='中';

when 60 then result:='差';

when 0 then result:='不及格';

else result:='Unspecified';

end case;

raise notice 'The user score is %',result;

end $$;

**搜索case语句**

do $$

declare

score int=90;

result text;

begin

case

when score >= 90 then result:='优';

when score >= 80 then result:='良';

when score >= 70 then result:='中';

when score >= 60 then result:='差';

when score < 60 then result:='不及格';

end case;

raise notice 'The user score is %',result;

end $$;

**循环语句**

**loop循环**

loop重复执行一段代码，直到被exit or return语句终止

**语法：**

<<label>>

loop

statements;

end loop;

通常使用if循环内的语句来根据如下条件终止它：

<<label>>

loop

statements;

if condition then

exit;

end if;

end loop;

也可以嵌套循环

<<outer>>

loop

statements;

<<inner>>

loop

/\* ... \*/

exit <<inner>>

end loop;

end loop;

当有嵌套循环时，需要使用循环标签，以便可以在exit and continue语句中指定它以指示这些语句引用哪个循环。

**示例1：**（使用loop循环语句来显示计数器的值）

do $$

declare

counter integer := 0;

begin

loop

exit when counter = 5 ;

raise notice 'Counter %', counter;

counter := counter + 1;

end loop;

end $$;

**while循环**

while循环语句执行代码块，直到条件计算为false。

[ <<label>> ]

while condition loop

statements;

end loop;

**示例：**（使用while循环语句来显示计数器的值）

do $$

declare

counter integer := 0;

begin

while counter < 5 loop

raise notice 'Counter %', counter;

counter := counter + 1;

end loop;

end$$;

**for循环**

**在整数范围内循环的语法：**

[ <<label>> ]

for loop\_counter in [ reverse ] from.. to [ by step ] loop

statements

end loop [ label ];

示例1：

do $$

begin

for counter in 1..5 loop

raise notice 'counter: %', counter;

end loop;

end; $$;

示例2：

do $$

begin

for counter in reverse 5..1 loop

raise notice 'counter: %', counter;

end loop;

end; $$;

示例3：

do $$

begin

for v\_counter in 1..6 by 2 loop

raise notice 'counter: %', v\_counter;

end loop;

end; $$;

**使用 for 循环查询结果**

语法：

[ <<label>> ]

for target in query loop

statements

end loop [ label ];

示例：

do $$

declare

rec record;

begin

for rec in select name,age from t1 limit 10

loop

raise notice '%(% 岁)', rec.name, rec.age;

end loop;

end;$$;

**使用 for 循环动态查询结果**

语法：

[ <<label>> ]

for row in execute query\_expression [ using query\_param [, ... ] ]

loop

statements

end loop [ label ];

示例：

do $$

declare

sort\_type smallint := 1;

rec\_count int := 10;

rec record;

query text;

begin

query := 'select name,age from users ';

if sort\_type = 1 then

query := query || 'order by name';

elsif sort\_type = 2 then

query := query || 'order by age';

else

raise 'invalid sort type %', sort\_type;

end if;

query := query || ' limit $1';

for rec in execute query using rec\_count

loop

raise notice '% - %', rec.name, rec.age;

end loop;

end;$$;

**exit**

exit语句终止循环，包括loop循环、while 循环和for 循环

语法：

exit [label] [when boolean\_expression]

如果不使用标签，exit语句将终止当前循环

以下语句是等效的：

exit when counter > 10;

if counter > 10 then

exit;

end if;

exit when绝对更干净，更短

**示例1：**使用 Exit 语句终止loop

do $$

declare

i int = 0;

j int = 0;

begin

<<outer\_loop>>

loop

i = i + 1;

exit when i > 3;

-- inner loop

j = 0;

<<inner\_loop>>

loop

j = j + 1;

exit when j > 3;

raise notice '(i,j): (%,%)', i, j;

end loop inner\_loop;

end loop outer\_loop;

end $$;

**示例2**.使用 Exit 语句退出一个块

do $$

begin

<<simple\_block>>

begin

exit simple\_block;

raise notice '%', 'unreachable!';

end;

raise notice '%', 'End of block';

end;$$;

**continue**

continue语句跳过某次循环的当前迭代并跳转到下一个迭代，该语句可用于各种循环，包括无条件循环、while循环和for 循环

continue [loop\_label] [when condition]

loop\_label和when condition是可选的

**示例：**

以下示例使用continue无条件循环中的语句打印出 1 到 10 的奇数：

do $$

declare

counter int = 0;

begin

loop

counter = counter + 1;

exit when counter > 10;

continue when mod(counter,2) = 0;

raise notice '%', counter;

end loop;

end;$$;

**创建函数**

**创建函数**

用户可以使用create function定义一个新的函数

**语法：**

create [or replace] function function\_name(param\_list)

returns return\_type

language plpgsql

as $$

declare

-- variable declaration

begin

-- logic

end;$$;

使用language plpgsql指定函数的过程语言。请注意，PostgreSQL 支持许多过程语言，而不仅仅是plpgsql

最后使用$$在函数后面定义一个块

**示例1**：无参数名

create function f\_add(integer, integer)

returns integer

language plpgsql

as $$

begin

return $1 + $2;

end

$$;

select f\_add(1, 2);

**示例2**：

除了通过参数顺序，函数参数也可以通过名字进行传递和访问。如果通过名字访问参数，这样会增加函数代码的可读性。

create function f\_add(x integer, y integer)

returns integer

language plpgsql

as $$

begin

return x + y;

end

$$;

**示例3**：没有参数的函数

create function f\_one()

returns integer

language plpgsql

as $$

begin

return 1;

end

$$;

**函数参数**

PL/pgSQL 支持三种参数模式：in、out和inout. 如果您未明确指定参数，则默认情况下采用in模式。

|  |  |  |
| --- | --- | --- |
| in | out | inout |
| 默认 | 明确指定 | 明确指定 |
| 将值传递给函数 | 从函数返回一个值 | 将值传递给函数并返回更新的值。 |

**IN参数**

create function f\_add(in x integer,in y integer)

returns integer

language plpgsql

as $$

begin

return x + y;

end

$$;

**OUT参数**

示例1.返回一个参数

create function f\_add (IN x int, IN y int, OUT sum int)

language plpgsql

as $$

begin

sum = x + y;

end

$$;

select f\_add(3,7);

示例2.返回多个参数

create function f\_calc (x int, y int, OUT sum int, OUT product int)

language plpgsql

as $$

begin

sum = x + y;

product = x \* y;

end

$$;

select f\_calc(2,3);

select \* from f\_calc(11,42);

**inout模式**

inout模式是组合in和out模式。

这意味着调用者可以将参数传递给函数。该函数更改参数并返回更新后的值。

以下swap函数接受两个整数及其值：

create or replace function swap(inout x int,inout y int)

language plpgsql as $$

begin

select x,y into y,x;

end; $$;

调用函数：

select \* from swap(1,2);

**返回集合类型**

示例1.返回一个复合类型

create type sum\_prod as (sum int, product int);

create function f\_calc (x int,y int)

returns sum\_prod

language plpgsql

as $$

begin

return (x + y, x \* y);

end $$;

select f\_calc(2,5);

示例2.返回一个集合

create function get\_users(p\_age int)

returns setof users

language plpgsql

as $$

begin

return query select \* from users where age = p\_age;

end

$$;

select \* from get\_users(10);

create function get\_name\_name(p\_age int)

returns setof text

language plpgsql

as $$

begin

return query select name from age where age = p\_age;

end

$$;

select \* from get\_users(10);

示例3.返回一张表

create function get\_users(p\_age int)

returns table(v\_id int,v\_name text,v\_age int)

language plpgsql

as $$

begin

return query select \* from users where age = p\_age;

end

$$;

SELECT \* FROM get\_users(22);

**带有默认参数值的函数**

示例1：3个数相加

create function f\_sum(a int, b int default 2, c int default 3)

returns int

language plpgsql

as $$

begin

return a+b+c;

end

$$;

select f\_sum(10, 20, 30);--60

select f\_sum(10, 20);--33

select f\_sum(10);--15

select f\_sum(); -- 因为第一个参数没有默认值，所以会失败

示例2：统计年在age1到age2之间的学生

create function get\_student\_count(age1 int default 10, age2 int default 20)

returns int

language plpgsql

as $$

declare

count integer;

begin

select count(\*) into count from t1 where age between age1 and age2;

return count;

end

$$;

select get\_student\_count();

**如何调用函数：**

PostgreSQL 提供了三种调用函数的方法：

1.使用位置符号

2.使用命名符号

3.使用混合符号。

1.使用位置符号

要使用位置符号调用函数，需要以与参数相同的顺序指定参数。例如：

select get\_student\_count(10,15);

2.使用命名符号

下面显示了如何f\_get\_student\_count使用位置表示法调用函数：

select get\_student\_count(age1 => 10, age2 => 15);

在命名符号中，您使用=>分隔参数的名称和它的值。

为了向后兼容，PostgreSQL 支持基于:=以下的旧语法：

select get\_student\_count(age1 := 10, age2 := 15);

3.使用混合复合

select get\_student\_count(10,age2:=15);

**函数重载**

PostgreSQL 允许多个函数共享相同的名称，只要它们具有不同的参数。

如果两个或多个函数共享相同的名称，则函数名称将被重载

当您可以调用重载函数时，PostgreSQL 会根据函数参数列表选择最佳候选函数来执行。

示例1：返回年龄

create or replace function get\_age(p\_id integer)

returns integer

language plpgsql

as $$

declare

v\_age integer;

begin

select age into v\_age from users where id=p\_id;

return v\_age;

end; $$;

示例2：返回多个年龄的平均值

create or replace function get\_age(p\_id1 int,p\_id2 int)

returns numeric

language plpgsql

as $$

declare

v\_age1 integer;

v\_age2 integer;

begin

select score into v\_age1 from users where id=p\_id1;

select score into v\_age2 from users where id=p\_id2;

return (v\_age1+v\_age2)/2;

end; $$;

此函数与第一个函数同名，只是它有两个参数。

换句话说， get\_age(p\_id integer)函数被函数get\_sc(p\_id1 integer,p\_id2 int)重载了。

**函数重载和默认值**

在function get\_age(p\_id1 integer,p\_id2 int)函数中，如果要为第二个参数设置默认值，如下所示：

create or replace function get\_age(p\_id1 integer,p\_id2 int default 6)

returns numeric

language plpgsql

as $$

declare

v\_age1 integer;

v\_age2 integer;

begin

select score into v\_age1 from t1 where id=p\_id1;

select score into v\_age2 from t1 where id=p\_id2;

return (v\_age1+v\_age2)/2;

end $$;

调用get\_age()函数并传递id=2：

SELECT get\_age(2);--报错

在这种情况下，PostgreSQL 无法选择最佳候选函数来执行

在这种情况下，具有三个功能：

get\_age(p\_id integer)

get\_age(p\_id1 integer,p\_id2 int)

get\_age(p\_id1 integer,p\_id2 int default 6)

PostgreSQL 不知道应该执行第一个函数还是第三个函数

所以：当重载一个函数时，应该总是让它们的参数列表是唯一的

**函数管理**

重命名函数：

ALTER FUNCTION f\_sum(int,int) RENAME TO f\_sum2(int,int);

删除函数：

要删除函数及其依赖对象，您需要指定cascade选项。

DROP FUNCTION f\_sum(int,int);

删除多个函数：

DROP FUNCTION f\_sum(int,int),f\_calc(int,int);

如果函数名称在其模式中是唯一的，则可以在不带参数列表的情况下删除它：

DROP FUNCTION f\_sum;

**异常处理**

当块中发生错误时，PostgreSQL 将中止块的执行以及周围的事务。

要从错误中恢复，可以使用exception。

下面说明了exception子句的语法：

<<label>>

declare

begin

statements;

exception

when condition [or condition...] then

handle\_exception;

[when condition [or condition...] then

handle\_exception;]

[when others then

handle\_other\_exceptions;]

end;

**示例1：**处理 no\_data\_found 异常示例

do $$

declare

rec record;

v\_id int = 0;

begin

select name,age into strict rec from users where id=v\_id;

end $$;

使用exception子句捕获no\_data\_found异常并报告更有意义的消息：

do $$

declare

rec record;

v\_id int = 0;

begin

select name,age into strict rec from users where id=v\_id;

exception

when no\_data\_found then

raise exception 'user % not found', v\_id;

end $$ ;

**示例2：**处理 too\_many\_rows 异常示例

do $$

declare

rec record;

begin

select name,age into strict rec from users where name like '%a%';

exception

when too\_many\_rows then

raise exception 'Search query returns too many rows';

end $$;

在此示例中，too\_many\_rows发生异常是因为该select into语句返回多行，而它应该返回一行。

**示例3：**处理多个异常

do $$

declare

rec record;

v\_age int = 10;

begin

select name,age into strict rec from users where age=v\_age;

exception

when no\_data\_found then

raise exception 'The user with age % not found', v\_age;

when too\_many\_rows then

raise exception 'The user with age % is not unique', v\_age;

end $$;

**示例4**：将异常处理为 SQLSTATE 代码，而不是条件名称

do $$

declare

rec record;

v\_age int = 10;

begin

select name,age into strict rec from users where age=v\_age;

exception

when sqlstate 'P0002' then

raise exception 'The user with age % not found', v\_age;

when sqlstate 'P0003' then

raise exception 'The user with age % is not unique', v\_age;

end $$;

**PL/pgSQL调试**

安装调试器pldebugger

安装pgadmin使用调试器

PostgreSQL提供pldebugger插件，用于调试存储过程。

安装插件

git clone https://github.com/EnterpriseDB/pldebugger.git

export PATH=/usr/local/pgsql/bin:$PATH

cd /root/postgresql-14.5/contrib/pldebugger

export USE\_PGXS=1

make

make install

CREATE EXTENSION pldbgapi;

说明 仅superuser可以执行此命令。

删除插件

DROP EXTENSION pldbgapi;

说明 仅superuser可以执行此命令。

**调试示例**

1.使用pgAdmin客户端连接PostgreSQL数据库。

2.创建测试数据和函数

create function f\_test()

returns void

language plpgsql

as $$

declare

i int = 0;

j int = 0;

begin

<<outer\_loop>>

loop

i = i + 1;

exit when i > 3;

-- inner loop

j = 0;

<<inner\_loop>>

loop

j = j + 1;

exit when j > 3;

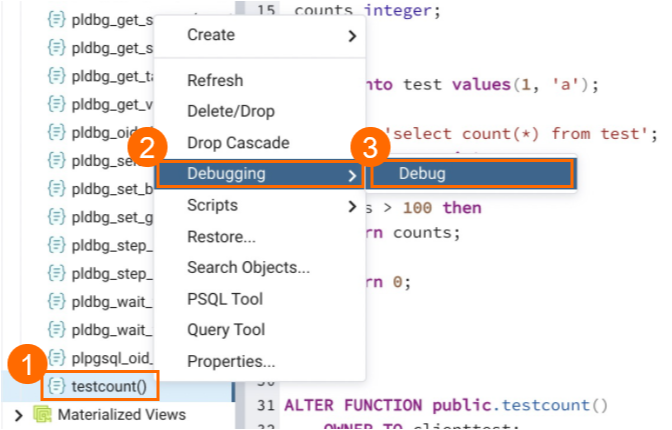
raise notice '(i,j): (%,%)', i, j;

end loop inner\_loop;

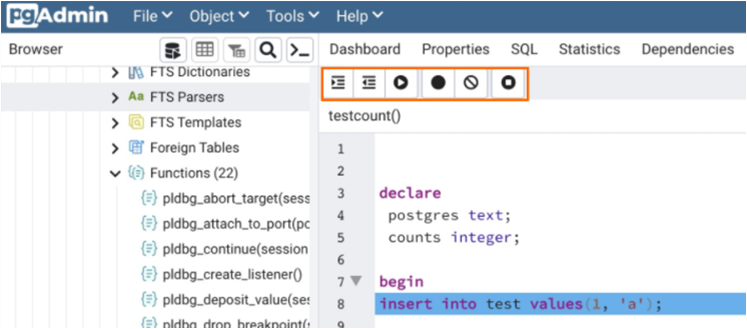
end loop outer\_loop;

end $$;

3.右键选择待调试的函数。



4.在pgAdmin界面右边的函数调试框中，您可以对目标函数进行单步调试，包含step into/over，continue，加设断点，停止等操作。底部有调试过程中显示的局部变量信息，调试结果与函数堆栈。



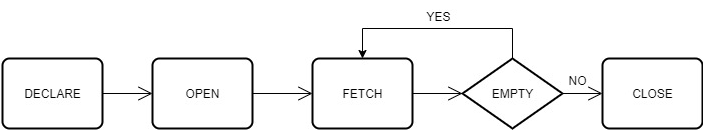
**游标的使用**

游标是SQL 的一种数据访问机制 ，是一种处理数据的方法。众所周知，使用SQL的select查询操作返回的结果是一个包含一行或者是多行的数据集，如果我们要对查询的结果再进行查询，比如（查看结果的第一行、下一行、最后一行、前十行等等操作）简单的通过select语句是无法完成的，因为这时候索要查询的结果不是数据表，而是已经查询出来的结果集。游标就是针对这种情况而出现的。

PL/pgSQL 游标允许开发者封装查询并依次处理每一行。

将大型结果集划分为多个部分并单独处理每个部分时，推荐使用游标。如果一次处理，可能会出现内存溢出错误，而且，可以开发一个返回对游标的引用的函数。这是从函数返回大型结果集的有效方法。函数的调用者可以根据游标引用处理结果集。

在PostgreSQL中使用游标的具体步骤：



1.声明一个游标—要访问游标，需要在块的声明部分中声明游标变量，游标变量的类型为cursor

2.打开游标

3.将结果集中的行提取到目标中。

4.检查是否还有更多行要获取。如果是，转步骤3，否则转步骤5。

5.关闭游标

**1.声明游标**

-- 声明游标

declare

cur refcursor;

-- 或者在查询语句中声明

cursor\_name [ [no] scroll ] cursor [( name datatype, name data type, ...)] for query

-- NO SCROLL 游标不能向后滚动

示例

declare

cur cursor for select \* from users; -- cur是封装表中所有行的游标

cur cursor (v\_age integer) for select \* from users where age=v\_age; -- cur封装了具有特定年龄的人。

**2.打开游标**

打开游标分为打开绑定游标和未绑定游标

打开未绑定游标：

-- 打开未绑定游标

OPEN unbound\_cursor\_variable [ [ NO ] SCROLL ] FOR query;

-- 因为未绑定游标变量在声明它时没有绑定到任何查询，所以必须在打开它时指定查询

open cur for select \* from users where age=17;

-- PostgreSQL 允许打开游标并将其绑定到动态查询

open unbound\_cursor\_variable[ [ no ] scroll ]

for execute query\_string [using expression [, ... ] ];

-- 示例：使用参数sort\_field对行进行排序的动态查询

query := 'select \* from users order by $1';

open cur for execute query using sort\_field;

打开绑定游标：

-- 打开绑定游标

open cursor\_variable[ (name:=value,name:=value,...)];

-- 示例

open cur;

open cur(age:=18);

**3.使用游标**

打开游标后，可以使用FETCH、MOVE、UPDATE或DELETE语句对其进行操作

-- 获取下一行

fetch [ direction { from | in } ] cursor\_variable into target\_variable;

-- 默认情况下，如果未明确指定方向，则光标将获取下一行next。以下对游标有效：

-- next

-- last

-- prior

-- first

-- absolute count

-- relative count

-- forward

-- backward

-- 示例

fetch cur into rec;

fetch last from cur into rec;

**4.移动光标**

如果只想移动光标而不检索任何行，则使用该MOVE语句。FETCH方向接受与语句相同的值

-- 移动语法

move [ direction { from | in } ] cursor\_variable;

-- 示例

move cur;

move last from cur;

move relative -1 from cur;

move forward 3 from cur;

**6.删除或更新行**

定位游标后，可以使用DELETE WHERE CURRENT OF or UPDATE WHERE CURRENT OF语句删除或更新游标标识的行，如下所示：

-- 更新、删除行语法

update table\_name set column = value, ... where current of cursor\_variable;

delete from table\_name where current of cursor\_variable;

-- 示例

update users set age = age+5 where current of cur;

**6.关闭游标**

close cursor\_variable;

**7.综合示例**

创建一个函数，在这个函数内使用游标实现查找年龄在23岁的人的信息。

select name,age from users where age=23;--有3条结果

create or replace function get\_users(p\_age integer)

returns text

language plpgsql

as $$

declare

v\_name text default '';

rec record;

cur cursor(p\_age integer) for select name,age from users where age=p\_age;

begin

open cur(p\_age);

loop

fetch cur into rec;

exit when not found;

v\_name := v\_name || ',' || rec.name || ':' || rec.age;

end loop;

close cur;

return v\_name;

end $$;

**函数的三态**

每个函数都有一个易变性的分类，有VOLATILE，STABLE或者 IMMUTABLE。如果创建函数命令没有明确声明范畴的话， 那么VOLATILE就是缺省的。

函数的易变性级别可在视图查知

select distinct provolatile from pg\_proc;

VOLATILE可以做任何事情，包括修改数据库。它可以在使用同样参数调用时返回不同的结果。优化器对这样的函数不做任何假设。一个使用易变函数的查询在需要数据值的时候每次都重新计算函数的值。

STABLE函数不会修改数据库， 并且保证在同一个查询的环境里给出相同参数的情况下，会给出相同的结果。这个范畴允许优化器在一个查询里把多个函数调用优化成一个。特别是在索引扫描的条件表达式里面包含这样的函数是安全的。因为索引扫描只计算一次比较值，而不是每行一次。在索引扫描条件里使用一个VOLATILE函数是非法的。

IMMUTABLE函数不会修改数据库， 并且保证在任何情况下同样的参数永远返回同样的结果。这个范畴允许优化器在查询调用函数的时候预先把函数计算成一个常量参数。比如，类似SELECT … WHERE x = 2 + 2的查询可以简化成 SELECT … WHERE x = 4，因为在加法操作符下层的函数是标记为 IMMUTABLE的。

**测试**

创建表并插入数据

create table test (id int);

insert into test select n from generate\_series(1,5) n;

创建volatile级别函数

create or replace function test(i int)

returns int

language plpgsql volatile

as $$

begin

raise notice '%',i;

return i;

end $$;

1、volatite级别函数在同一个查询中, 同样参数的情况下可能被多次执行

select \* from test where test(1)=1;

查询的结果有5行，test函数共执行了5次

2、优化器可根据实际场景优化stable级别函数的调用次数, 同样的参数多次调用可能减少成单次调用

alter function test(int) stable;

select \* from test where test(1)=1;

查询的结果有5行，test函数只调用一次，immutable级别也是一样。

3、immutable与stable的区别

优化器在处理immutable函数时, 先评估函数结果, 将结果替换为常量

**函数级别为stable时**

alter function test(int) stable;

explain analyze select \* from test where id>test(1);

顺序扫描表时，test函数执行了5次

create index on test(id);

set enable\_seqscan=off;

explain analyze select \* from test where id>test(1);

索引扫描表时,索引扫描仅评估被比较的表达式一次, 后多次与索引值进行比较,所以test函数调用了2次

**函数级别为immutable时**

alter function test(int) immutable;

explain analyze select \* from test where id>test(1);

优化器先评估函数结果, 将结果替换为常量

4、volatile级别函数没有限制, 可以修改数据, stable和immutable不允许

create or replace function test2()

returns int

language plpgsql volatile

as $$

begin

update test set id = id;

return 0;

end $$;

select test2();--0

函数级别为volatile修改成功

alter function test2() stable;

alter function test2() immutable;

select test2();--ERROR

stable或immutable级别时均报错

5、stable和immutable级别函数执行时，始终保持当前snapshot

create or replace function test3()

returns int

language plpgsql stable

as $$

declare

count int;

begin

for i in 1..5 loop

perform pg\_sleep(2);

select count(\*) into count from test;

raise notice 'counter:%',count;

end loop;

return 0;

end $$ ;

select test3();

insert into test values(6);

函数级别为stable或immutable时，对外部已提交的数据不可见

alter function test3() volatile;

select test3();

insert into test values(7);

函数级别为volatile时，对外部已提交的数据可见