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，http://qun.81nanchang.cn 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 $$
DECLARE
  quantity integer := 30;
BEGIN
  RAISE NOTICE 'Quantity here is %', quantity;      --在这里的数量是 30
  quantity := 50;
  --
  -- 创建一个子块
  -- http://www.qinglvfenzu.com
  DECLARE
    quantity integer := 80;
  BEGIN
    RAISE NOTICE 'Quantity here is %', quantity;   --在这里的数量是 80
  END;
  RAISE NOTICE 'Quantity here is %', quantity;      --在这里的数量是 50
  RETURN quantity;
END;
$$ 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;
```