Programación de funciones en PL/pgSQL para PostgreSQL

Roberto Andrade Fonseca. ABL Consultores, S.A. de C.V.

8 de febrero de 2002

1. Objetivos

- Presentar a los asistentes las ventajas de contar con funciones que manejen las 'reglas del negocio' dentro de una base de datos, PostgreSQL en nuestro caso, para independizarlas del código en que se programen los clientes.
- Describir el lenguaje y mostrar ejemplos reales de funciones desarrolladas con PL/pgSQL.

2. Introducción

Es común que los desarrolladores de aplicaciones subutilicen las prestaciones de las bases de datos relacionales modernas, en ocasiones aimplemente por deconocer las ventajas que le ofrecen o por desconocer su manejo.

Dentro de PostgreSQL, la base de datos de código abierto más poderosa, se pueden desarrollar funciones en varios lenguajes. El lenguaje PL/pgSQL es uno de los más utilizados dentro de PostgreSQL, debido a que guarda cierta similitud con PL/SQL de Oracle y a su facilidad de uso.

En este tutorial se mostrará la sintaxis, el control de flujo y otras características del lenguaje, además de presentarán algunos ejmeplos reales.

2.1. Ventajas de usar PL/pgSQL

SQL es el lenguaje estándar para realizar consultas a un servidor de base de datos. Cada sentencia SQL se ejecuta de manera individual por el servidor, lo cual implica que las aplicaciones cliente deben enviar cada consulta al servidor, esperar a que la procese, recibir los resultados, procesar los datos y después enviar la siguiente sentencia.

Al usar PL/pgSQL es posible relizar cálculos, manejo de cadenas y consultas dentro del servidor de la base de datos, combinando el poder de un lenguaje procedimental y la facilidad de uso de SQL, minimizando el tiempo de conexión entre el cliente y el servidor.

3. Nuestra base de datos del tutorial

Para ejemplificar el uso de las funciones en PL/pgSQL, vamos a utilizar una base de datos que tiene el siguiente esquema:

```
-- Esquema de la base de datos del tutorial
-- de PL/pgSQL

CREATE TABLE asistente (
    id_asistente SERIAL,
    id_titulo    int REFERENCES titulo,
    ap_paterno   varchar NOT NULL,
    ap_materno   varchar,
    nombre   varchar NOT NULL,
    sexo     char(1) CHECK (sexo in ('M', 'F')),
    id_puesto    int REFERENCES puesto,
    compania     varchar,
    direccion    varchar,
    colonia         varchar(40),
    ciudad   varchar,
    codigo_postal   varchar,
    id_estado    varchar REFERENCES estado,
    id_pais  char(2) NOT NULL REFERENCES pais,
    lada     varchar,
    telefono1    varchar(8),
    telefono2    varchar(8),
    fax  varchar(8),
    email    varchar,
    url  varchar,
    id_categoria    int NOT NULL REFERENCES categoria DEFAULT 1,
    id_giro_empresa int REFERENCES giro_empresa,
    id_lugar_compra int REFERENCES lugar_compra,
    id_sistema_operativo int REFERENCES sistema_operativo,
    PRIMARY KEY (id_asistente)
 );
```

4. Estructura de PL/pgSQL

El lenguaje PL/pgSQL es estructura en bloques. Todas las palabras clave y los identificadores pueden escribirse mezclando letras mayúsculas y minúsculas.

Un bloque se definde de la siguiente manera:

```
[<<label>>]
[DECLARE
    declaraciones]
```

```
BEGIN
    sentencias
END;
```

Pueden existir varios bloques o sub-bloques en la sección de sentencias de un bloque. Los sub-bloques pueden ser usados para ocultar las variables a los bloques más externos.

Normalmente una de las sentencias es el valor de retorno, usando la palabra clave `RETURN`.

Las variables declaradas en la sección que antecede a un bloque se inicializan a su valor por omisión cada vez que se entra al bloque, no solamente al ser llamada la función. Por ejemplo:

```
CREATE FUNCTION estafunc() RETURNS INTEGER AS '
DECLARE
   cantidad INTEGER := 30;
BEGIN
   RAISE NOTICE ''Cantidad contiene aquí %'',cantidad;
   -- Cantidad contiene aquí 30
   cantidad := 50;
   --
   -- Creamos un sub-bloque
   --
   DECLARE
      cantidad INTEGER := 80;
   BEGIN
      RAISE NOTICE ''Cantidad contiene aquí %'',cantidad;
      -- Cantidad contiene aquí 80
   END;
   RAISE NOTICE ''Cantidad contiene aquí %'',cantidad;
   -- Cantidad contiene aquí 50

   RETURN cantidad;
END;
' LANGUAGE 'plpgsql';
```

No se debe confundir el uso de las sentencias de agrupamiento BEGIN/END de PL/pgSQL con los comandos de la base de datos que sirven para el control de las transacciones. Las funciones y procedimientos disparadores no pueden iniciar o realizar transacciones y Postgres no soporta transacciones anidadas.

5. Comentarios, constantes y variables

5.1. Comentarios

Existen dos tipo de comentarios en PL/pgSQL. Un doble guión – da inicio a un comentario, el cual se extiende hasta el final de la línea. Un `/*` inicia un

bloque que se extiende hasta la primera ocurrencia de `*/`.

5.2. Variables y constantes

Todas las variables, filas y registros usados en un bloque o en sus sub-bloques deben declararse en la sección de declaraciones del bloque.

La excepción es la variable de un ciclo FOR que itera sobre un rango de valores enteros.

Las variables en PL/pgSQL pueden ser de cualquier tipo de datos de SQL, como INTEGER, VARCHAR y CHAR. El valor por omisión de todas la variables es el valor NULL de SQL.

A continuación se muestran algunos ejemplos de declaración de variables:

```
user_id INTEGER;
quantity NUMBER(5);
url VARCHAR;
```

5.2.1. Constantes y variables con valores por omisión

La declaraciones tienen las siguiente sintaxis:

```
nombre [ CONSTANT ] tipo [ NOT NULL ] [ { DEFAULT | := } valor ];
```

El valor de una variable declarado como `CONSTANT` no puede ser modificado. Si acaso se especifica `NOT NULL`, la asignación de un valor `NULL` causa en un error en tiempo de ejecución. Puesto que el valor por omisión de todas las vriables es el valor `NULL` de SQL, todas las variables declaradas como `NOT NULL` deben contar con un valor por omisión específico.

EL valor se evalúa cada vez que se llama la función, así que asignar `now` a una variable de tipo `timestamp` causa que la variables almacene el tiempo real de la llamada a la función, no el de la hora en que se compiló en *bytecode*.

Ejemplos:

```
cantidad INTEGER := 32;
url varchar := ''http://misitio.com'';
user_id CONSTANT INTEGER := 10;
```

5.2.2. Variables pasadas a las funciones

Las variables que se pasan a las funciones son denominadas con los identificadores $1, $2, etc. (el máximo es 16).

Algunos ejemplos:

```
CREATE FUNCTION iva_venta(REAL) RETURNS REAL AS '
DECLARE
    subtotal ALIAS FOR $1;
BEGIN
```

```
        return subtotal * 1.15;
    END;
    ' LANGUAGE 'plpgsql';


CREATE FUNCTION instr(VARCHAR,INTEGER) RETURNS INTEGER AS '
    DECLARE
        v_string ALIAS FOR $1;
        index ALIAS FOR $2;
    BEGIN
        -- Algunos cáculos irían aquí.
    END;
    ' LANGUAGE 'plpgsql';
```

6. Atributos

Usando los atributos %TYPE and %ROWTYPE, es posible declarar variables con el mismo tipo de dato o estructura de otro item de la base de datos (por ejemplo, un campo de una tabla).

%TYPE Proporciona el tipo de dato de una variable o una columna. Se puede utilizar para declarar variables que almacenen valores de bases de datos. Por ejemplo, supongamos que usted tiene una columna llamada `user_id` en la tabla `users`. Para declarar una variable con el mismo tipo de dato que el usado en nuestra tabla de usuarios, lo que haría es:

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

Al usar %TYPE puede despreocuparse de los cambios futuros en la definición de la tabla.

nombre tabla %ROWTYPE Declara una renglón con la estructura de la tabla especificada. `tabla` puede ser una tabla o una vista que exista en la base de datos. Los campos del renglón se accesan con la notación punto. Los parámetros de una función pueden ser de tipo compuesto (renglones completos de una tabla). Es este caso, el identificador correspondiente $n será del tipo rowtype, pero debe usarse un seudónimo o alias usando el comando `ALIAS` que se describe más adelante.

Solamente los atributos del usuario de la tabla pueden ser accesibles en el renglón, ni los OID ni otros atributos del sistema (debido a que el renglón puede ser de una vista). Los campos de un rowtype heredan los tamaños de los campos o la precisión de los tipos de dato para char(), etc.

```
DECLARE
    users_rec users%ROWTYPE;
```

```
  user_id users%TYPE;
BEGIN
    user_id := users_rec.user_id;
    ...
create function cs_refresh_one_mv(integer) returns integer as '
   DECLARE
        key ALIAS FOR $1;
        table_data cs_materialized_views\%ROWTYPE;
   BEGIN
        SELECT INTO table_data * FROM cs_materialized_views
               WHERE sort_key=key;
        IF NOT FOUND THEN
           RAISE EXCEPTION ''View '' || key || '' not found'';
           RETURN 0;
        END IF;
        -- La columna mv_name de cs_materialized_views almacena
        -- los nombres de las vistas.
        TRUNCATE TABLE table_data.mv_name;
        INSERT INTO table_data.mv_name || '' '' || table_data.mv\query;
        return 1;
end;
' LANGUAGE 'plpgsql';
```

7. Expresiones

Todas las expresiones usadas en las sentencias de PL/pgSQL son procesadas usando el ejecutor del *backend*. Las expresiones que parecen contener constantes pueden requerir de una evaluación en tiempo de ejecución (por ejemplo, `now` para el tipo de dato timestamp) así que es imposible para el analizador sintáctico (*parser*) de PL/pgSQL identificar los valores de las constantes reales diferentes de `NULL`. Todas las expresiones son evaluadas internamente ejecutando una sentencia

```
SELECT expresión
```

usando el gestor de SPI. En la expresión, las ocurrencias de los identificadores de las variables se sustituyen por parámetros y los valores reales de las variables se pasan al ejecutor en el arreglo de parámetros. Todas las expresiones usadas en una función de PL/pgSQL son preparadas y almacenadas solamente una vez. La única excepción a esta regla es una sentencia `EXECUTE` si se requiere analizar una consulta cada vez que es encontrada.

La revisión del tipo realizada por el analizador sintáctico principal de Postgres tiene algunos efectos secundarios a la interpretación de valores constantes. En detalle, existe una diferencia entre lo que hacen estas dos funciones:

```
CREATE FUNCTION logfunc1 (text) RETURNS timestamp AS '
```

```
    DECLARE
        logtxt ALIAS FOR $1;
    BEGIN
        INSERT INTO logtable VALUES (logtxt, ''now'');
        RETURN ''now'';
    END;
' LANGUAGE 'plpgsql';
```

y

```
CREATE FUNCTION logfunc2 (text) RETURNS timestamp AS '
    DECLARE
        logtxt ALIAS FOR $1;
        curtime timestamp;
    BEGIN
        curtime := ''now'';
        INSERT INTO logtable VALUES (logtxt, curtime);
        RETURN curtime;
    END;
' LANGUAGE 'plpgsql';
```

En el caso de `logfunc1()`, el analizador principal de Postgres conoce, al preparar el plan para el `INSERT`, que la cadena 'now' debe ser interpretada como `timestamp` debido a que el campo destino de `logtable` es de ese tipo. Así, creará una constante a partir de ella en ese momento, y la usará en cada una de las invocaciones de logfunc1(), mientras esté vivo el *backend*. Es claro que esto no es lo que esperaría el programador.

En el caso de logfunc2(), el analizador principal de Postgres no sabe que tipo debe tener 'now', y por eso regresa un tipo de datos de texto, el cual contiene la cadena 'now'. Durante la asignación a la variable local curtime, el intérprete de PL/pgSQL cambia esta cadena a un tipo de timestamp por medio de una llamada a las funciones text_out() and timestamp_in() para efectuar la conversión.

8. Sentencias

Cualquier cosa no comprendida por el analizador PL/pgSQL tal como se especifica adelante será enviado al gestor de la base de datos, para su ejecución. La consulta resultante no devolverá ningún dato.

8.1. Asignación

Una asignación de un valor a una variable o campo de fila o de registro se escribe:

```
    identifier := expression;
```

Si el tipo de dato resultante de la expresión no coincide con el tipo de dato de las variables, o la variable tienen un tamaño o precisión conocido (como char(29)), el resultado será amoldado implícitamente por el interprete de bytecode de PL/pgSQL, usando los tipos de las variables para las funciones de entrada y los tipos resultantes en las funciones de salida. Nótese que esto puede potencialmente producir errores de ejecución generados por los tipos de las funciones de entrada.

Una asignación de una selección completa en un registro o fila puede hacerse del siguiente modo:

```
SELECT expressions INTO target FROM ...;
```

`target` puede ser un registro, una variable de fila o una lista separada por comas de variables y campo de de registros o filas.

Si una fila o una lista de variables se usa como objetivo, los valores seleccionados han de coincidir exactamente con la estructura de los objetivos o se producirá un error de ejecución. La palabra clave `FROM` puede preceder a cualquier calificador válido, agrupación, ordenación, etc. que pueda pasarse a una sentencia `SELECT`.

Existe una variable especial llamada `FOUND` de tipo booleano, que puede usarse inmediatamente después de `SELECT INTO` para comprobar si una asignación ha tenido éxito.

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

Si la selección devuelve múltiples filas, solo la primera se mueve a los campos objetivo; todas las demás se descartan.

8.2. Volviendo de la función

```
RETURN expresión
```

La función termina y el valor de `expresión` se devolverá al ejecutor superior. El valor devuelto por una función no puede quedar son definir. Si el control alcanza el fin del bloque de mayor nivel de la función sin encontrar una sentencia `RETURN`, ocurrirá un error de ejecución.

Las expresiones resultantes serán amoldadas automáticamente en los tipos devueltos por la función, tal como se ha descrito en el caso de las asignaciones.

8.3. Abortando la ejecución y mensajes

Use la sentencia RAISE para enviar mensajes al mecanismo elog de PostgreSQL.

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

Dentro del formato, " %"se usa como comodín para los subsecuentes identificadores, separados por comas. Los posibles niveles son DEBUG (suprimido en las bases de datos de producción), NOTICE (escribe en la bitácora de la base de datos y lo envía a la aplicación del cliente) y EXCEPTION (escribe en la bitácora de la base de datos y aborta la transacción).

```
RAISE NOTICE ''Id number '' || key || '' not found!'';
RAISE NOTICE ''Calling cs_create_job(%)'',v_job_id;
```

En este último ejemplo, v_job_id reeemplazará al % en la cadena

```
RAISE EXCEPTION ''Inexistent ID --> %'',user_id;
```

Esto abortará la transacción y escribirá en la bitácora de la base de datos.

9. Estructuras de control de flujo

9.1. Condiciones

```
IF expression THEN
    statements
[ELSE
    statements]
END IF;
```

expression debe devolver un valor que al menos pueda ser adaptado en un tipo booleano.

9.2. Bucles

Hay varios tipos de bucles.

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

Se trata de un bucle no condicional que ha de ser terminado de forma explicita, mediante una sentencia EXIT. La etiqueta opcional puede ser usado por las sentencias EXIT de otros bucles anidados, para especificar el nivel del bucle que ha de terminarse.

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

Se trata de un lazo condicional que se ejecuta mientras la evaluación de expression sea cierta.

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

Se trata de un bucle que se itera sobre un rango de valores enteros. La variable name se crea automáticamente con el tipo entero, y existe solo dentro del bucle. Las dos expresiones dan el limite inferior y superior del rango y son evaluados sólo cuando se entra en el bucle. El paso de la iteración es siempre 1.

```
    [<<label>>]
    FOR record | row IN select_clause LOOP
        statements
    END LOOP;
```

El registro o fila se asigna a todas las filas resultantes de la clausula de selección, y la sentencia se ejecuta para cada una de ellas. Si el bucle se termina con una sentencia EXIT, la ultima fila asignada es aún accesible después del bucle.

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

Si no se incluye `label`, se termina el lazo más interno, y se ejecuta la sentencia que sigue a `END LOOP`. Si se incluye `label` ha de ser la etiqueta del bucle actual u de otro de mayor nivel. EL bucle indicado se termina, y el control se pasa a la sentencia de después del `END` del bucle o bloque correspondiente.

9.3. Excepciones

Postgres no dispone de un modelo de manejo de excepciones muy elaborado. Cuando el analizador, el optimizador o el ejecutor deciden que una sentencia no puede ser procesada, la transacción completa es abortada y el sistema vuelve al lazo principal para procesar la siguiente consulta de la aplicación cliente.

Es posible introducirse en el mecanismo de errores para detectar cuando sucede esto. Pero lo que no es posible es saber qué ha causado en realidad el aborto (un error de conversión de entrada/salida, un error de punto flotante, un error de análisis). Y es posible que la base de datos haya quedado en un estado inconsistente, por lo que volver a un nivel de ejecución superior o continuar ejecutando comandos puede corromper toda la base de datos. E incluso aunque se pudiera enviar la información a la aplicación cliente, la transacción ya se abría abortado, por lo que carecería de sentido el intentar reanudar la operación.

Por todo esto, lo único que hace PL/pgSQL cuando se produce un aborto de ejecución durante la ejecución de una función o procedimiento disparador es enviar mensajes de depuración al nivel DEBUG, indicando en qué función y donde (numero de línea y tipo de sentencia) ha sucedido el error.

10. Ejemplos

Se incluyen unas pocas funciones para demostrar lo fácil que es escribir funciones en PL/pgSQL. Para ejemplos más complejos, el programador debería consultar el test de regresión de PL/pgSQL.

Un detalle doloroso a la hora de escribir funciones en PL/pgSQL es el manejo de la comilla simple. El texto de las funciones en CREATE FUNCTION ha de ser una cadena de texto. Las comillas simples en el interior de una cadena literal deben de duplicarse o anteponerse de una barra invertida. Aún estamos trabajando en una alternativa más elegante. Mientras tanto, duplique las comillas sencillas como en los ejemplos siguientes. Cualquier solución a este problema en futuras versiones de Postgres mantendrán la compatibilidad con esto.

10.1. Algunas funciones sencillas en PL/pgSQL

Las dos funciones siguientes son idénticas a sus contrapartidas que se verán cuando estudiemos el lenguaje C.

```
CREATE FUNCTION add_one (int4) RETURNS int4 AS '
    BEGIN
        RETURN $1 + 1;
    END;
' LANGUAGE 'plpgsql';

CREATE FUNCTION concat_text (text, text) RETURNS text AS '
    BEGIN
        RETURN $1 || $2;
    END;
' LANGUAGE 'plpgsql';
```

Funciones PL/pgSQL para tipos compuestos De nuevo, estas funciones PL/pgSQL tendrán su equivalente en lenguaje C.

```
CREATE FUNCTION c_overpaid (EMP, int4) RETURNS bool AS '
    DECLARE
        emprec ALIAS FOR $1;
        sallim ALIAS FOR $2;
    BEGIN
        IF emprec.salary ISNULL THEN
            RETURN ''f'';
        END IF;
        RETURN emprec.salary > sallim;
    END;
' LANGUAGE 'plpgsql';
```

10.2. Procedimientos desencadenados en PL/pgSQL

Estos procedimientos desencadenados aseguran que, cada vez que se inserte o actualice un fila en la tabla, se incluya el nombre del usuario y la fecha y hora. Y asegura que se proporciona un nombre de empleado y que el salario tiene un valor positivo.

```
    CREATE TABLE emp (
        empname text,
        salary int4,
        last_date datetime,
        last_user name);     CREATE FUNCTION emp_stamp () RETURNS OPAQUE AS
        BEGIN
            -- Check that empname and salary are given
            IF NEW.empname ISNULL THEN
                RAISE EXCEPTION ''empname cannot be NULL value'';
            END IF;
            IF NEW.salary ISNULL THEN
                RAISE EXCEPTION ''% cannot have NULL salary'', NEW.empname;
            END IF;             -- Who works for us when she must pay for?
            IF NEW.salary < 0 THEN
                RAISE EXCEPTION ''% cannot have a negative salary'', NEW.empname;
            END IF;             -- Remember who changed the payroll when
            NEW.last_date := ''now'';
            NEW.last_user := getpgusername();
            RETURN NEW;
        END;
    ' LANGUAGE 'plpgsql';

CREATE TRIGGER emp_stamp BEFORE INSERT OR UPDATE ON emp
        FOR EACH ROW EXECUTE PROCEDURE emp_stamp();
```

11. Referencias

PL/pgSQL:
http://lucas.hispalinux.es/Postgresql-es/web/navegable/programmer/x1503.html
Procedural Languages:
http://www.postgresql.org/idocs/index.php?programmer-pl.html
Oracle 8. Guía de aprendizaje, Michael Abbey y Michael J. Corey. Osborne.
Capítulo 7: PL/SQL.