BD 2 - Funzioni e Procedure

Luca Cosmo

Università Ca' Foscari Venezia

Università
Ca'Foscari
Venezia

Introduzione

Abbiamo visto come usare una funzione per definire il corpo di un trigger. In realtà il concetto di funzione è più generale ed ha altri utilizzi.

Perché definire una funzione in SQL?

- Incapsulare funzionalità di uso comune per favorirne il riutilizzo
- Offrire interfacce di accesso semplificate ad utenti inesperti di SQL
- Centralizzare lato server una sequenza di operazioni di cui non ci interessano i risultati intermedi (benefici di performance)

In questa lezione ci concentreremo su PL/pgSQL di Postgres, ma altri DBMS forniscono tecnologie simili. Il materiale di questa lezione è adattato dal manuale di Postgres (link per approfondimenti).

Dichiarazione di Funzioni

Una funzione si può dichiarare con la sintassi:

```
CREATE FUNCTION my_fun ( args ) RETURNS type
AS function_body
LANGUAGE plpgsql;
```

dove `function_body` è un blocco con la seguente struttura:

```
[ DECLARE declarations ]
BEGIN
    statements
END;
```

Postgres richiede che il *function_body* sia una stringa. Lo possiamo definire in più righe facendolo precedere e terminare da $$.

Dichiarazione di Variabili

Tutte le variabili utilizzate in un blocco vanno dichiarate nella sezione opportuna, associandole al loro tipo:

```
name [ CONSTANT ] type [ NOT NULL ] [ = expr ];
```

Il tipo di una variabile può essere un qualsiasi tipo SQL. Ci sono inoltre alcuni tipi e sintassi particolari che è importante menzionare:

- `var%TYPE`: il tipo della variabile o colonna chiamata `var`
- `tab%ROWTYPE`: il tipo record delle righe della tabella `tab`
- `RECORD`: un qualsiasi tipo record (da usare con attenzione!)
- `SETOF t`: un insieme di elementi di tipo `t` (solo per valori di ritorno!)

Esempio

```
CREATE FUNCTION my_concat(a text, b text,
                          uppercase boolean = false)
RETURNS text AS $$
BEGIN
    IF uppercase THEN RETURN UPPER(a || ' ' || b);
    END IF;
    RETURN LOWER(a || ' ' || b);
END; $$ LANGUAGE plpgsql;
```

Tutte le seguenti invocazioni sono lecite:

- `SELECT my_concat('Hello', 'World')`
- `SELECT my_concat('Hello', 'World', true)`
- `SELECT my_concat(b := 'World', a := 'Hello')`

Assegnamento

Un assegnamento ad una variabile si effettua con la sintassi:

```
var = expr;
```

Il risultato di un comando SQL che ritorna una singola riga, per esempio certe SELECT, può essere salvato in una variabile con la sintassi:

```
SELECT expr INTO [STRICT] var FROM ...
```

L'opzione `STRICT` richiede alla query di ritornare esattamente una riga, in caso contrario viene dato un errore a runtime. Se viene omessa, solo la prima riga del risultato viene assegnata (NULL in caso di risultato vuoto).

Ritornare un Valore

Una funzione che ritorna un singolo valore può usare la sintassi:

```
RETURN expr;
```

Se è necessario ritornare un record, è possibile utilizzare i cosiddetti parametri di output per definirne implicitamente il tipo:

```
CREATE FUNCTION sum_n_product(x int, y int,
                              OUT sum int, OUT prod int)
AS $$
BEGIN
    sum = x + y;
    prod = x * y;
END; $$ LANGUAGE plpgsql;
```

Ritornare un Insieme di Valori

Una funzione che ritorna un insieme di valori (`SETOF`) deve costruirlo in maniera incrementale tramite le sintassi:

```
RETURN NEXT expr;     -- aggiunge un record al risultato
RETURN QUERY query;   -- aggiunge un insieme al risultato
```

L'insieme di valori da ritornare può poi essere restituito con `RETURN` senza passare alcun argomento (o lasciando terminare la funzione).

Fate attenzione ai tipi di ritorno delle vostre funzioni! In particolare come possiamo ritornare un insieme di record in maniera corretta?

Esempio (1/3)

Example

Data la tabella `PC(model, speed, ram, hd, price)`, definire una funzione che ritorna un insieme di modelli associati ai rispettivi prezzi, cioè solo la prima e la quinta colonna della tabella.

Abbiamo almeno due diverse possibili opzioni:

1. abusare del polimorfismo, ritornando `SETOF RECORD`
2. usare la sintassi `RETURNS TABLE` al posto di `RETURNS`

Esempio (2/3)

Opzione 1: polimorfismo

```
CREATE FUNCTION f() RETURNS SETOF RECORD AS $$
BEGIN
RETURN QUERY SELECT model, price FROM pc;
END;
$$ LANGUAGE plpgsql;
```

Utilizzo:

```
SELECT m,p FROM f() AS (m character(20), p real);
```

Esempio (3/3)

Opzione 2: `RETURNS TABLE`

```
CREATE FUNCTION f()
RETURNS TABLE(m integer, p real) AS $$
BEGIN
RETURN QUERY SELECT model, price FROM pc;
END;
$$ LANGUAGE plpgsql;
```

Utilizzo:

```
SELECT * FROM f();
```

Nota di Implementazione

Si noti che la sintassi:

```
CREATE FUNCTION f()
RETURNS TABLE(m character(20), p real) ...
```

è solo una versione più fedele allo standard della sintassi:

```
CREATE FUNCTION f(OUT m character(20), OUT p real)
RETURNS SETOF RECORD ...
```

Nel caso di funzioni con parametri di output che ritornano un insieme di valori si può usare `RETURN NEXT` senza argomenti per aggiungere gli attuali valori dei parametri di output come nuova riga del risultato.

Condizionali

Sintassi del condizionale:

```
IF boolean-expr THEN
    statements
[ ELSIF boolean-expr THEN
    statements
    ... ]
[ ELSE
    statements ]
END IF;
```

C'è inoltre un costrutto CASE in due forme (vedi manuale).

Cicli

Tradizionale ciclo WHILE:

```
WHILE boolean-expr LOOP
    statements
END LOOP;
```

Il ciclo FOR ha invece una sintassi piuttosto complessa:

```
FOR name IN [ REVERSE ] int-expr .. int-expr
            [ BY int-expr ] LOOP
    statements
END LOOP;
```

La variabile di iterazione non deve essere dichiarata ed è locale al ciclo.

Cicli

Il ciclo FOR può essere utilizzato anche per iterare sui risultati prodotti da una certa query:

```
FOR target IN query LOOP
    statements
END LOOP;
```

L'iterazione su un array si effettua invece con FOREACH:

```
FOREACH target IN ARRAY expr LOOP
    statements
END LOOP
```

Variabile FOUND

Ciascuna funzione contiene una variabile booleana `FOUND`:

1. SELECT INTO imposta `FOUND` a true se viene assegnata una riga alla variabile corrispondente, a false altrimenti
2. UPDATE, INSERT e DELETE impostano `FOUND` a true se almeno una riga è stata toccata dall'operazione, a false altrimenti
3. un ciclo FOR imposta `FOUND` a true se ha iterato almeno una volta, a false altrimenti
4. RETURN QUERY imposta `FOUND` a true se la query ha ritornato almeno una riga, a false altrimenti

Esempio

Un altro modo per risolvere l'esempio precedente:

```
CREATE FUNCTION f3(OUT m character(20), OUT p real)
RETURNS SETOF RECORD AS $$
declare r RECORD;
BEGIN
    FOR r IN SELECT model, price FROM lab.pc LOOP
    SELECT r.model,r.price INTO m, p;
    RETURN NEXT;
    END LOOP;
END;
$$ LANGUAGE plpgsql;
```

Messaggi ed Eccezioni

Una funzione può riportare messaggi o errori con la sintassi:

```
RAISE [ level ] 'format' [, expr [, ... ]]
                [USING option = expr];
```

dove:

- `level` indica il livello di severità dell'errore (`DEBUG`, `LOG`, ...). Il livello di default `EXCEPTION` solleva anche un'eccezione
- `'format'` è una format string che specifica il messaggio da riportare
- la clausola `USING` permette di popolare informazioni aggiuntive sull'errore, per esempio il suo codice di errore `ERRCODE`

Si può usare `RAISE` senza parametri per rilanciare un'eccezione catturata

Messaggi ed Eccezioni

Per catturare un'eccezione si può usare la sintassi:

```
BEGIN
    statements
EXCEPTION
    WHEN cond [ OR cond ... ] THEN handler
    [ WHEN cond [ OR cond ... ] THEN handler ... ]
END;
```

Le condizioni `cond` seguono una sintassi particolare, si consulti il manuale per i dettagli. Per esempio è possibile fare azioni diverse in base al codice di errore, usando la condizione speciale `others` come fallback.

Messaggi ed Eccezioni

Quando un'eccezione viene catturata, il contenuto delle variabili locali persiste, ma tutti i cambiamenti al database effettuati nel blocco che ha sollevato l'eccezione vengono annullati!

```
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;
```

Qui `x` viene incrementata di 1, ma nel database troveremo `Tom Jones`.

Procedure

Una procedura è una funzione che non ritorna alcun risultato:

```
CREATE PROCEDURE my_proc ( args )
AS proc_body
LANGUAGE plpgsql;
```

Una procedura può essere invocata tramite il comando `CALL`. Questo è il metodo più moderno per gestire funzioni senza risultato:

- prima di Postgres 11: comando `PERFORM`
- una procedura differisce da una funzione void solo nella gestione delle transazioni (ne parleremo prossimamente)

Lab Time!

Definire delle funzioni PL/pgSQL sullo schema della scorsa lezione:

1. Dato un prezzo come argomento, ritornare il PC il cui prezzo è più vicino ad esso. Se ci sono più PC a pari merito, ritornarne uno.
2. Dato un prezzo come argomento, ritornare il numero di PC, laptop e stampanti che costano più di esso.
3. Dato un PC come argomento, inserirlo nella tabella corrispondente facendo sì che, se esiste già un PC più veloce che costa meno, il prezzo del nuovo prodotto sia abbassato fino a tale costo.
4. Dato un prezzo come argomento, usare un iteratore per ritornare una tabella che contenga tutti i PC che costano al più quel prezzo. Se non ce ne sono, lanciare un'eccezione col rispettivo messaggio di errore. Infine usare tale funzione in un'altra funzione che ne calcola il prezzo medio, gestendo la possibile eccezione.