

PosgreSQL PL/PgSQL



Federico Campoli

PostgreSQL PL/PgSQL

Federico Campoli

Prima Edizione

Pubblicato 2007

Copyright © 2007 PGHost di Federico Campoli

Opera rilasciata sotto licenza Creative Commons - **Attribuzione - Non commerciale - Non opere derivate 2.5**

Tu sei libero:

- di riprodurre, distribuire, comunicare al pubblico, esporre in pubblico, rappresentare, eseguire e recitare quest'opera

Alle seguenti condizioni:

- **Attribuzione.** Devi attribuire la paternità dell'opera nei modi indicati dall'autore o da chi ti ha dato l'opera in licenza.
- **Non commerciale.** Non puoi usare quest'opera per fini commerciali.
- **Non opere derivate.** Non puoi alterare o trasformare quest'opera, né usarla per crearne un'altra.

Nota: Ogni volta che usi o distribuisi quest'opera, devi farlo secondo i termini di questa licenza, che va comunicata con chiarezza. In ogni caso, puoi concordare col titolare dei diritti d'autore utilizzi di quest'opera non consentiti da questa licenza. Questa licenza non riduce o elimina in alcun modo i diritti morali dell'autore.

Sommario

1. Il linguaggio procedurale pl/pgsql.....	1
1.1. Perché usare un linguaggio procedurale	1
1.2. Installazione nel database.....	1
2. Funzioni	3
2.1. Struttura di una function pl/pgsql.....	3
2.2. Override delle funzioni	6
2.3. Cancellazione	6
2.4. I parametri	7
2.5. Esecuzione di una funzione.....	7
3. I costrutti	8
3.1. Decisioni con il costrutto IF.....	8
3.2. Iterazioni con il costrutto FOR.....	8
3.3. Iterazioni su risultati di una select.....	9
4. Debug	11
4.1. EXCEPTION	11
Webografia.....	12

Lista degli Esempi

1-1. Tabella pl_pgttemplate	2
2-1. Struttura blocco pl/pgsql	3
2-2. Esempio di funzione	3
2-3. Esecuzione di una funzione	7
2-4. Errore di cast	7
3-1. Esempio di costruito IF	8
3-2. Esempi di costrutti FOR	8
3-3. Esempio di costruito IF	10
4-1. Esempio di costruito IF	11

Capitolo 1. Il linguaggio procedurale pl/pgsql

1.1. Perché usare un linguaggio procedurale

Il call handler del PL/pgSQL esegue il parse del codice della funzione e crea una struttura binaria della funzione la prima volta, per sessione, che la funzione è richiamata. Le eventuali istruzioni SQL presenti nella funzione non sono però tradotte e vengono elaborate dall'ottimizzatore solo alla prima esecuzione, per sessione. Conseguentemente il piano di esecuzione per tali istruzioni SQL viene generato una sola volta e riutilizzato per tutta la sessione.

Così facendo viene ridotto il tempo di esecuzione specialmente se ci sono comandi sql ripetuti in loop. Lo svantaggio di questo approccio è che se vengono operate modifiche sul catalogo di sistema in maniera dinamica la funzione non è in grado di riconoscerle a meno di disconnettersi e riconnettersi.

PL/pgSQL permette di incapsulare istruzioni SQL all'interno del database senza dover mandare ogni volta la query dal backend al server. Usando PL/pgSQL si ottengono i seguenti vantaggi

- Eliminazione di scambio dati inutile tra client e server
- I dati prodotti durante le fasi intermedie dell'esecuzione non devono essere trasferiti tra client e server per l'elaborazione
- I piani di esecuzione vengono generati solo all'inizio della sessione

Queste "semplici" migliorie danno un incremento molto importante alle performance dell'applicazione se basata su PL/pgSQL. Con PL/pgSQL naturalmente è possibile usare tutti i tipi dato, operatori e funzioni a disposizione con SQL.

1.2. Installazione nel database

PostgreSQL grazie alla sua struttura modulare permette di aggiungere "on the fly" i linguaggi procedurali e di creare funzioni staticamente linkate a librerie esterne scritte in C. Pertanto, alla creazione del cluster con initdb il database template1, da cui normalmente vengono derivati tutti gli altri database creati con il comando CREATE DATABASE, non contiene nessun tipo di linguaggio procedurale.

L'installazione di base di PostgreSQL, che sia da binario o che sia da sorgente, installa di default nella directory lib di \$PGHOME una serie di librerie. Riferendoci all'installazione su sistema Linux, via ricompilazione con il solo comando di configurazione ./configure la directory lib contiene circa una quarantina tra file e link simbolici. Il file che corrisponde all'HANDLER del linguaggio procedurale pl/pgsql è plpgsql.so

Per installare nel database il linguaggio procedurale¹ è sufficiente il seguente comando:

CREATE LANGUAGE plpgsql;

PostgreSQL cercherà nella tabella di sistema pg_pltemplate la presenza del nome template che si è fornito con il comando CREATE LANGUAGE e, in caso di successo genererà il linguaggio procedurale con i parametri estratti da questa tabella.

Esempio 1-1. Tabella pl_pgtemplate

```
postgres=# select * from pg_pltemplate ;
```

tplname	tpltrusted	tplhandler	tplvalidator	tpllibrary	tplacl
plpgsql	t	plpgsql_call_handler	plpgsql_validator	\$libdir/plpgsql	
pltcl	t	pltcl_call_handler		\$libdir/pltcl	
pltclu	f	pltclu_call_handler		\$libdir/pltcl	
plperl	t	plperl_call_handler	plperl_validator	\$libdir/plperl	
plperlu	f	plperl_call_handler	plperl_validator	\$libdir/plperl	
plpythonu	f	plpython_call_handler		\$libdir/plpython	

Il campo tpltrusted se settato a true determina se il linguaggio è TRUSTED di default. Se il linguaggio procedurale, come nel caso di plpgsql, è trusted allora, poiché questo handler è affidabile, è possibile creare nuove function con i privilegi di utente ordinario. Una volta installato il linguaggio procedurale sarà possibile creare funzioni.

Note

1. l'installazione richiede i privilegi di superuser

Capitolo 2. Funzioni

2.1. Struttura di una function pl/pgsql

pl/pgsql e' un linguaggio strutturato a blocchi. Un blocco e' composto dalle seguenti sezioni:

Esempio 2-1. Struttura blocco pl/pgsql

```
DECLARE
    dichiarazioni di variabili
BEGIN
    istruzioni
END ;
```

Analizziamo la struttura di una funzione pl/pgsql

Esempio 2-2. Esempio di funzione

```
CREATE OR REPLACE FUNCTION auth_admins(character varying, character varying) RETURNS smallint AS
$BODY$
DECLARE
    v_username alias FOR $1;
    v_password alias FOR $2;
    b_status int2;

BEGIN
    SELECT INTO b_status count(*)
    FROM admins
    WHERE user_id = v_username
    AND passw = md5(v_password);

RETURN b_status;

END;
$BODY$
LANGUAGE plpgsql;
```

La tabella admins e' cosi' strutturata:

```
CREATE TABLE admins
(
    id_admin serial NOT NULL,
    user_id character varying(30) NOT NULL,
```

```

passw  character varying(100) NOT NULL,
flag_act  boolean NOT NULL,
name  character varying(50) NOT NULL,
surname character varying(50) NOT NULL,
email  character varying(50) NOT NULL,
flag_su  boolean DEFAULT false NOT NULL
);

```

- CREATE OR REPLACE FUNCTION auth_admins(character varying, character varying) RETURNS smallint AS

Questa sezione serve a creare o ricreare la funzione. La possibilita' di replace torna utile durante la fase di sviluppo per velocizzare le modifiche. Successivamente segue il nome della funzione con gli eventuali parametri che accetta e il tipo dati ritornato dalla funzione che nel nostro esempio e' uno smallint (int 2 byte). Alla parola AS segue poi l'intera definizione di funzione.

Importante: Le funzioni pl/pgsql non supportano i parametri con valore di default. E' quindi sempre necessario passare tutti i parametri quando la funzione viene richiamata.

- \$BODY\$

Fino alla versione 7.4 il corpo della funzione veniva incluso tra apici. Cio' provocava non pochi problemi nella scrittura di select con i delimitatori delle stringhe, costringendo a veri e propri numeri da circo per eseguire l'escape degli apici. Dalla versione 8.0 in poi e' possibile includere il corpo della funzione tra i segni \$\$ eventualmente inframmezzati da una label. Nel nostro esempio, per rendere piu' evidente che stiamo lavorando sul corpo funzione e' stata introdotta la label BODY. La chiusura del body avviene subito dopo l'ultimo END e prima della specifica LANGUAGE.

- DECLARE

Sezione dedicata alla dichiarazione delle variabili. Il pl/pgsql e' fortemente tipizzato e le variabili vanno dichiarate prima di essere utilizzate. Lo scope delle variabili e' relativo al blocco in cui vengono dichiarate e sottoblocchi eventuali.

- BEGIN

Inizio del blocco istruzioni. Puo' contenere i vari costrutti che vedremo in seguito, comandi SQL e ulteriori sottoblocchi.

Importante: Le variabili dichiarate in un blocco vengono inizializzate ai loro valori di default ogni volta che si entra in quel blocco.

- RETURN

Return di funzione. Il tipo dati ritornato deve essere lo stesso della clausola RETURNS in testa alla funzione. Se il tipo dato specificato da RETURNS e VOID l'istruzione RETURN deve essere omessa.

Importante: Fino alla versione 8.1 le funzioni che ritornavano VOID non effettuavano nessun controllo che RETURN fosse effettivamente omesso. Dalla 8.2 invece, se una funzione ritornante VOID contiene RETURN viene generato un errore.

- END

Termine del blocco o del sottoblocco istruzioni.

- LANGUAGE plpgsql;

Essendo PostgreSQL dotato di piu' linguaggi procedurali bisogna specificare al termine della funzione in che linguaggio questa e' scritta. Il nome del linguaggio viene ricercato nella tabella pg_pltemplate nel campo tplname.

Tutte le keyword e gli identificatori sono case insensitive. I commenti su singola linea, come per il linguaggio SQL iniziano con la sequenza --. I commenti multiriga vengono fatti alla stessa maniera del linguaggio C /* COMMENTO MULTIRIGA */.

Ogni istruzione di un blocco puo' essere un sottoblocco essa stessa.

Importante: Attenzione a non confondere BEGIN/END di plpgsql con i comandi di controllo transazioni tipici del database. BEGIN/END di plpgsql servono solo a raggruppare le istruzioni. Le funzioni e le procedure attivate da trigger sono sempre eseguite attraverso una transazione e

pertanto non e' possibile avviare o terminare un blocco transazionale al loro interno. Ad ogni modo una clausola EXCEPTION genera una subtransazione che puo' essere sottoposta a rollback senza produrre effetti sulla transazione originale.

2.2. Override delle funzioni

Abbiamo visto come con il comando CREATE OR REPLACE e' possibile creare una funzione pl/pgsql. I parametri passati richiedono la specifica del tipo dato. PostgreSQL permette di avere piu' funzioni aventi lo stesso nome ma con numero parametri o tipo dato di questi differenti. Ad esempio queste quattro funzioni aventi lo stesso nome sono di fatto quattro elementi distinti che possono essere programmati in maniera assolutamente indipendente.

```
auth_admins(character varying, character varying)
```

```
auth_admins(character varying, numeric)
```

```
auth_admins(character varying)
```

```
auth_admins()
```

2.3. Cancellazione

Per la cancellazione di una funzione si adopera il comando DROP FUNCTION. Poiche' ogni funzione e' strettamente legata ai parametri che accetta, numero e tipo dati, affinche' il DROP sia accettato bisogna specificare esattamente la funzione che si vuol droppare.

Rifacendoci al caso precedente per droppare la prima funzione

```
DROP FUNCTION auth_admins(character varying, character varying); CORRETTO
```

```
DROP FUNCTION auth_admins(,); SBAGLIATO
```

```
DROP FUNCTION auth_admins(); SBAGLIATO
```

```
DROP FUNCTION auth_admins; SBAGLIATO
```

2.4. I parametri

Ad ogni parametro passato corrisponde una variabile identificata con un progressivo. I parametri sono oggetti immutabili, non e' quindi possibile riassegnarvi dei valori. Nel blocco DECLARE e' possibile definire degli alias per i parametri in questo modo **v_username alias FOR \$1;**

Dalla versione 8.0 e successive e' possibile dichiarare gli alias direttamente nell'intestazione della funzione **CREATE OR REPLACE FUNCTION auth_admins(v_username character varying, v_password character varying)**

2.5. Esecuzione di una funzione

Una volta creata la funzione e' a nostra disposizione per l'esecuzione. Avviare una funzione e' banalmente semplice

Esempio 2-3. Esecuzione di una funzione

```
SELECT auth_admins('foo', 'bar');
```

Un errore molto frequente si verifica quando il cast implicito dell'espressione passata alla funzione come parametro fa perdere la corrispondenza tra nome funzione numero e tipo parametri.

Esempio 2-4. Errore di cast

```
SELECT auth_admins('0'::numeric, '0') ;
```

```
ERROR: function auth_admins(numeric, "unknown") does not exist
```

```
SQL state: 42883
```

```
Hint: No function matches the given name and argument types. You may need to add explicit type casts.
```

```
Character: 8
```

In questo caso il nome funzione corrisponde ma, un cast numeric del primo parametro, ha impedito a PostgreSQL di matchare la funzione memorizzata nel database con quella da noi indicata.

Capitolo 3. I costrutti

3.1. Decisioni con il costrutto IF

Similmente ad altri linguaggi di programmazione pl/pgsql permette di prendere decisioni con il costrutto IF. Nella forma piu' estesa possibile IF si presenta cosi'.

Esempio 3-1. Esempio di costrutto IF

```
IF number = 0 THEN
    result := 'zero';
ELSIF number > 0 THEN
    result := 'positive';
ELSIF number < 0 THEN
    result := 'negative';
ELSE
    result := 'NULL';
END IF;
```

3.2. Iterazioni con il costrutto FOR

Il costrutto for si compone della parola chiave FOR seguita dalla variabile sulla quale verra' eseguito il ciclo. Successivamente ci sono i limiti minore e maggiore che la variabile dovra' assumere seguito da LOOP---END LOOP; che conterra' le istruzioni da eseguire nel ciclo FOR. La parola opzionale REVERSE specifica che la variabile dovra' decrescere nell'esecuzione del FOR. La parola opzionale BY determina invece l'ammontare dell'incremento assunto dalla variabile ad ogni iterazione. Facciamo alcuni esempi.

Esempio 3-2. Esempi di costrutti FOR

```
--iterazione da 1 a 10
FOR i IN 1..10 LOOP
    RAISE NOTICE 'i è %', i;
END LOOP;

NOTICE: i è 1
NOTICE: i è 2
NOTICE: i è 3
NOTICE: i è 4
NOTICE: i è 5
NOTICE: i è 6
```

```
NOTICE: i è 7
NOTICE: i è 8
NOTICE: i è 9
NOTICE: i è 10

Total query runtime: 37 ms.
1 rows retrieved.
```

```
--iterazione da 1 a 10
FOR i IN REVERSE 10..1 LOOP
END LOOP;
NOTICE: i è 10
NOTICE: i è 9
NOTICE: i è 8
NOTICE: i è 7
NOTICE: i è 6
NOTICE: i è 5
NOTICE: i è 4
NOTICE: i è 3
NOTICE: i è 2
NOTICE: i è 1

Total query runtime: 44 ms.
1 rows retrieved.
```

```
--iterazione da 1 a 10
FOR i IN REVERSE 10..1 BY 2 LOOP
-- some computations here
RAISE NOTICE 'i is %', i;
END LOOP;

NOTICE: i è 10
NOTICE: i è 8
NOTICE: i è 6
NOTICE: i è 4
NOTICE: i è 2

Total query runtime: 45 ms.
1 rows retrieved.
```

3.3. Iterazioni su risultati di una select

L'iterazione su di una select avviene in maniera simile al FOR classico ma con alcune piccole varianti. La prima variante è che la variabile su cui iterare deve essere dichiarata nel blocco DECLARE come

RECORD. La seconda variante e' che non viene specificato un range di numeri ma la select su cui si vuole iterare.

Esempio 3-3. Esempio di costrutto IF

```
CREATE OR REPLACE FUNCTION test_select() RETURNS void AS
$BODY$
DECLARE
    r_results RECORD;

BEGIN
    FOR r_results IN SELECT * FROM admins LOOP
        RAISE NOTICE 'Il nome utente è %', r_results.user_id;

    END LOOP;
END;
$BODY$
LANGUAGE plpgsql;
```

La variabile di tipo RECORD `r_results` diviene depositaria dei dati estratti dalla select e permette l'accesso ai singoli campi, durante il FOR, semplicemente indicando `r_results.NOME_CAMPO`. E' quindi possibile, in questo modo, automatizzare elaborazioni senza scambio dati tra backend e server come indicato in precedenza.

Capitolo 4. Debug

Come ultimo argomento in questo breve escursus sul pl/pgsql vedremo come gestire gli errori. Fino alla versione 7.4 l'unico modo di gestire gli errori era inserire dei comandi di RAISE NOTICE o RAISE EXCEPTION che inviando messaggi al logger di PostgreSQL permettevano di capire l'evoluzione dell'esecuzione della funzione. Dalla versione 8.0 un nuovo costrutto, similmente a quanto accade per il pl/psql di oracle, è stato introdotto per la gestione delle eccezioni.

4.1. EXCEPTION

Il costrutto exception viene posizionato in fondo ad un blocco in modo da catturare le eccezioni che si possono verificare durante l'esecuzione del blocco. La sua struttura è la seguente.

Esempio 4-1. Esempio di costrutto IF

```
EXCEPTION
WHEN condition [ OR condition ... ] THEN
    handler_statements
[ WHEN condition [ OR condition ... ] THEN
    handler_statements
```

La condition può assumere il valore OTHERS come confronto jolly per catturare tutti gli errori oppure uno qualsiasi dei messaggi di errore elencati nell'appendice A della documentazione di PostgreSQL <http://www.postgresql.org/docs/8.2/static/errcodes-appendix.html>.

Il codice d'errore da inserire nella condition si ricava dalla suddetta tabella frammentando le parole con degli underscore. Ad esempio l'errore di CONNECTION FAILURE inserito nel contesto EXCEPTION diventa connection_failure.

Suggerimento: Un blocco con EXCEPTION è molto più dispendioso da eseguire rispetto ad uno senza. Pertanto è bene adoperare EXCEPTION solo dove è necessario.

Webografia

Riferimenti web

1996-2007, A cura di The PostgreSQL Global Development Group., *PostgreSQL 8.2 Documentation*.

<http://www.postgresql.org/docs/8.2/static/index.html>