PostgreSQL설명과 예제

주효식nogadax@chollian.net 2000/03/23

이 문서는 포스트그레스의 Index,function,trigger,grant,revoke,Large Object를 예제와 함께 설명한 글이다.

목 차

제 1 절 INDEX, SEQUENCE, FUNCTION(1)

1.1 Create Index

INDEX 는 데이타베이스내의 relation(테이블)에 대한 검색의 성능을 높여준다.

```
CREATE  [UNIQUE]  INDEX  index_name
 ON  table_name  (name_of_attribute);


CREATE  [UNIQUE]  INDEX  index_name  ON  table_name
 [USING  acc_name] (column [ops_name] [,...]);


CREATE  [UNIQUE]  INDEX  index_name  ON  table_name
 [USING  acc_name] (func_name() ops_name );


 acc_name   :  ACCESS METHOD . 디폴트는 BTREE 이다.(BTREE,RTREE,HASH)
 func_name  :  사용자 정의 함수.
 ops_name   :  operator class (int4_ops,int24_ops,int42_ops)

 btree(sid int4_ops) 는 btree 를 이용한 INDEX 자료형이 4-BYTE 정수임.
 디폴트 operator class 는 Field Type 이다.
 현재 btree 는 7개까지의 Multi-Column INDEX를 지원한다.
```

1.1.1 example

```
 example 1) INDEX 생성 1

  CREATE  INDEX  indx1
   ON  supplier(sid);

  supplier  relation(테이블) 의 sname attribute(column) 을  INDEX 로 지정


 example 2) INDEX 생성 2

  CREATE INDEX indx2
   ON supplier USING btree(sid int4_pos);


 example 3) INDEX 생성 3

  CREATE INDEX indx3
   ON supplier USING btree(sid int8_ops);
```

```
example 4) INDEX 생성 4

 CREATE INDEX indx4
  ON supplier USING btree(sid, tid);


example 5)  INDEX  삭제

 DROP INDEX indx1;
 DROP INDEX indx2;
 DROP INDEX indx3;
 DROP INDEX indx4;
```

1.2 Create SEQUENCE

SEQUENCE 는 순차적인 숫자 발생기이다.

```
CREATE  SEQUENCE  seq_name [INCREMENT increment]
  [MINVALUE  minvalue]  [MANVALUE  maxvalue]
  [START  start]  [CACHE  cache]  [CYCLE]


INCREMENT : 이값이 -1 이면 -1 만큼 감소 , 3 이면 3씩 증가, 디폴트는 1 이다.
MAXVALUE  : optional clause , 증가할수 있는 최고값을 명시적으로 지정
START     : 시작값
CACHE     : sequence 값을 먼저 메모리에 할당하여 빠른 ACCESS 를 가능케 한다.
CYCLE     : 최고값으로 증가되면 다시 최소값으로 순환하게 한다.
```

1.2.1 example

```
CREATE  SEQUENCE  seq_name1 START 101;
SELECT  NEXTVAL('seq_name1);


결과

nextval
-------
    114
```

1.3 Create FUNCTION

FUNCTION 은 새로운 함수를 정의한다.

```
CREATE  FUNCTION  func_name([type[,...]])
   RETURNS  return_type [with (attribute [,...])]
   AS ' definition '
   LANGUAGE 'language_name';
```

```
LANGUAGE : sql, pgsql, c 등이 있다.
```

1.3.1 example

```
CREATE FUNCTION test() RETURNS int4
AS ' SELECT 1 '
LANGUAGE 'sql';


실행
SELECT test() AS answer;


결과
  answer
  ------
       1
```

```
AS ' 와 ' 사이에 함수의 본문을 기입하면 된다. 참고로 문자열일 경우,
'seq_test1' 와 같은 경우 다음처럼 한다.


CREATE FUNCTION test() RETURNS int4
AS ' SELECT NEXTVAL(''seq_test1'') '
LANGUAGE 'sql';


여기서 NEXTVAL 은 SEQUENCE 관련 내장함수이다.
```

1.4 예제

다음 예제의 이름은 test.sql 입니다. 다음 예제를 화일로 만들어 다음처럼 실행하시면 됩니다.

nogadax=> \i /usr/local/src/test.sql

```
-------------------------------------------------------cut here!!
--code By  nogadax@chollian.net  /2000/02/18
--drop all object for safe_test
DROP SEQUENCE seq_test1;
DROP SEQUENCE seq_test2;
DROP SEQUENCE seq_test3;
DROP INDEX    ind_test1;
DROP INDEX    ind_test2;
DROP INDEX    ind_test3;
DROP TABLE    tab_test1;
DROP TABLE    tab_test2;
DROP TABLE    tab_test3;
DROP FUNCTION func_test();
```

```
--create  sequence  seq_test1,seq_test2,seq_test3
CREATE  SEQUENCE  seq_test1  START  101;
CREATE  SEQUENCE  seq_test2  START    1;
CREATE  SEQUENCE  seq_test3  START    1;

--create table tab_test1,tab_test2,tab_test3
CREATE  TABLE  tab_test1(
    tab1_id    int4  NOT NULL,
    tab1_name  text,
    tab1_tel   text,
    teb1_memo  text
 );

CREATE  TABLE  tab_test2(
    tab2_id    int4  NOT NULL,
    tab2_name  text,
    tab2_tel   text,
    teb2_memo  text
 );

CREATE  TABLE  tab_test3(
    tab3_id    int4 DEFAULT  nextval('seq_test3') NOT NULL,
    tab3_name  text,
    tab3_tel   text,
    tab3_memo  text
 );

--craete  index
CREATE  UNIQUE  INDEX   ind_test1  ON  tab_test1(tab1_id);
CREATE  UNIQUE  INDEX   ind_test2  ON  tab_test2(tab2_id);
CREATE  UNIQUE  INDEX   ind_test3  ON  tab_test3  USING  btree(tab3_id  int4_ops);

--FUNCTION  func_test()
CREATE  FUNCTION  func_test()  RETURNS  INT4
   AS  ' SELECT NEXTVAL(''seq_test1'') '
   LANGUAGE 'sql';

--transaction 1
BEGIN;
   INSERT  INTO  tab_test1  VALUES (func_test(),'jini1','000-0000','No_Memo1');
   INSERT  INTO  tab_test2  VALUES (nextval('seq_test2'),'winob1','000-0001','No_Memo1');
   INSERT  INTO  tab_test3  (tab3_name,tab3_tel,tab3_memo)
                            VALUES ('nogadax1','000-0003','No_Memo1');

   INSERT  INTO  tab_test1  VALUES (func_test(),'jini2','100-0000','No_Memo2');
   INSERT  INTO  tab_test2  VALUES (nextval('seq_test2'),'winob2','100-0001','No_Memo2');
   INSERT  INTO  tab_test3  (tab3_name,tab3_tel,tab3_memo)
                            VALUES ('nogadax2','100-0003','No_Memo2');

   INSERT  INTO  tab_test1  VALUES (func_test(),'jini3','200-0000','No_Memo3');
   INSERT  INTO  tab_test2  VALUES (nextval('seq_test2'),'winob3','200-0001','No_Memo3');
   INSERT  INTO  tab_test3  (tab3_name,tab3_tel,tab3_memo)
                            VALUES ('nogadax3','200-0003','No_Memo3');

   INSERT  INTO  tab_test1  VALUES (func_test(),'jini4','300-0000','No_Memo4');
```

```
    INSERT  INTO  tab_test2  VALUES (nextval('seq_test2'),'winob4','300-0001','No_Memo4');
    INSERT  INTO  tab_test3  (tab3_name,tab3_tel,tab3_memo)
                             VALUES ('nogadax4','300-0003','No_Memo4');

    INSERT  INTO  tab_test1  VALUES (func_test(),'jini5','400-0000','No_Memo5');
    INSERT  INTO  tab_test2  VALUES (nextval('seq_test2'),'winob5','400-0001','No_Memo5');
    INSERT  INTO  tab_test3  (tab3_name,tab3_tel,tab3_memo)
                             VALUES ('nogadax5','400-0003','No_Memo5');

END;

--transaction 2
BEGIN;
   SELECT * FROM tab_test1;
   SELECT * FROM tab_test2;
   SELECT * FROM tab_test3;
   VACUUM VERBOSE ANALYZE tab_test1;
   VACUUM VERBOSE ANALYZE tab_test2;
   VACUUM VERBOSE ANALYZE tab_test3;
END;

------------------------------------------------------------------End !!
```

제 2 절 GRANT and REVOKE

2.1 GRANT

GRANT는 user,group 혹은 모든 user들에게 해당 객체에 대한 사용권한을 승인한다. REVOKE는 user,group 혹은 모든 user로부터 객체에 대한 사용권한을 무효화한다.

```
GRANT privilege [,...] ON object [,...]
    TO { PUBLIC | GROUP group | username}
```

privilege

```
SELECT : 특정 TABLE/VIEW 의 column에 대한 access 을 승인
INSERT : 특정 TABLE의 모든 column 에 데이타의 삽입에 대한 권한 승인
UPDTAE : 특정 TABLE의 모든 column 의 갱신에 대한 권한 승인
DELETE : 특정 TABLE 의 row 의 삭제에 대한 권한 승인
RULE : 특정 TABLE/VIEW에 대한 rule 을 정의하는 권한에 대한 승인
ALL : 모든 권한을 승인한다.
```

object

```
access 를 승인하는 객체의 이름으로서 다음과 같은 객체들이 있다.

Table
Sequence
View
Index
```

PUBLIC

모든 유저를 승인

GROUP group

사용 권한을 획득할 group을 지정, group 을 명시적으로 생성되어져 있어야 함.

username

사용권한을 획득할 사용자명. PUBLIC 은 모든 유저에 대해서 적용된다.

Description

GRANT 는 객체 (object) 를 생성한 유저가 모든 유저, 혹은 개인 유저, 혹은 그룹에 대해 해당 객체의 사용 권한을 허용하도록 한다. 객체를 생성한 유저가 아닌 다른 유저들은 그 객체에 대한 사용권한이 없어서 사용할 수가 없다. 단지 그 해당 객체를 생성한 유저만이 이를 허용할 수 가 있는데 이는 GRANT 를 사용함으로서 다른 유저들이 사용할 수 있도록 허용한다. 어떤 객체를 생성한 유저는 자동적으로 모든 권한을 가지며 이 권한들은 SELECT INSERT, UPDATE, DELETE, RULE 등이며 또한 그 객체 자체를 삭제할 수 있다.

Notes

psql 에서 "\z" 를 사용하여 존재하는 객체에 대한 permission 등을 참조할 수 있다.

permission 정보의 형식

```
username=arwR : 유저에게 승인된 사용권한
group gname=arwR : GROUP 에게 승인된 사용권한
=arwR : 모든 유저에게 승인된 사용권한

a : INSERT privilege
r : SELECT privilege
w : UPDATE/DELETE privilege
R : RULE privilege
arwR : ALL privilege
```

USAGE(사용예)

```
GRANT INSERT ON imsi_table TO PUBLIC

GRANT ALL ON imsi_table TO nogadax
```

2.2 REVOKE

유저,그룹, 혹은 모든 유저로부터 access privilege 를 취소

```
REVOKE privilege [,...]
 ON object [,...]
 FROM { PUBLIC | GROUP gname | username }
```

privilege

SELECT ,INSERT ,UPDATE, DELETE, RULE, ALL

object

적용될 수 있는 객체 : table, view, sequence, index

group

privilege 를 취소할 그룹명

username

PUBLIC

Description

REVOKE 는 객체의 생성자가 모든 유저, 유저, 그룹들로부터 전에 승인했던 퍼미션을 해제한다.

USAGE(사용예)

REVOKE INSERT ON imsi_table FROM PUBLIC

REVOKE ALL ON imsi_table FROM nogadax

제 3 절 TRIGGER

3.1 TRIGGER 1

```
CREATE  TRIGGER  name  { BEFORE | AFTER } { event [ OR ...]}
ON  table  FOR  EACH  { ROW  |  STATEMENT }
EXECUTE  PROCEDURE  func_name ( )

event     : INPUT , UPDATE , DELETE 등이 올 수 있다.
func_name : 사용자 정의 함수이다. plpgsql 을 권장하고 싶다.
            또한 이 함수에는 인자가 들어올 수 없다.
            또한 이 함수의 RETURN TYPE 로는 OPAQUE 이어야 한다.
            인수를 위해 TRIGGER 의 자체 변수를 사용하여야 한다.
            (new,old,...)

TRIGGER 를  촌스럽게 표현하자면 방아쇠를 당기면 총알이 나가는 것에 비유할 수 있다.
TRIGGER 은 TUPLE(ROW or RECORD) 에 대한  INSERT, UPDATE, DELETE 를 사건(event) 로
보고 이 사건에 정의된 어떤 행동으로 반응하는 것이다.TEL'
```

example

```
CREATE   TRIGGER    trg_test
BEFORE   DELETE     OR   UPDATE  ON  supplier  FOR  EACH  ROW
EXECUTE  PROCEDURE  check_sup();


supplier 테이블에 DELETE, UPDATE 가 발생하면 이 동작이 행해지기 전에
check_sup() 를 실행하라. 이 TRIGGER 의 이름은 trg_test 이다.
```

plpgsql 맛보기

```
CREATE  FUNCTION  func_name() RETURNS type
AS '
   [DECLARE  declarations]
   BEGIN
      statements
   END;
'LANGUAGE 'plpgsql';
```

example

```
CREATE  FUNCTION  pgsql_test() RETURNS  datetime
AS '
  DECLARE  curtime  datetime;
  BEGIN
    curtime:= ''now'';
    return  curtime;
  END;
' LANGUAGE 'plpgsql';
```

3.2 TRIGGER 실행을 위한 PL/pgSQL등록 방법

다음은 실행 예제이다. 우선 다음 예제를 적절히 복사를 해서 파일로 만든다.(예 : trg.sql) 다음 예제를 실행에 앞서 먼저 해야할 일이 있는데 그것은 Procedural Languages 를 등록하는 것이다. 다음예제에는 PL/pgSQL 을 이용한 함수를 사용하므로 이것을 등록하는 것은 필수이다.

방법은 두가지가 있다.

1. template1 데이타 베이스에 등록하는 것이다. 이 데이타베이스에 등록이 된후 template1 데이타베이스에서 create database 명령으로 데이타베이스를 생성하면 자동적으로 생성된 데이타베이스에 PL/pgSQL이 등록이 되므로 편리하다.

등록에 앞서 다음을 확인하라.

postgreSQL의 PATH : 여기서의 PATH 는 /usr/local/pgsql 이다. 또한 pgsql 디렉토리 밑의 lib 디렉토리에서 plpgsql.so 를 확인하라. 아마도 이 화일은 다 존재할 것이다.

```
등록 과정

[postgres@nogadax postgres]# psql  template1
template1=>
template1=> CREATE  FUNCTION  plpgsql_call_handler()  RETURNS  OPAQUE  AS
template1-> '/usr/local/pgsql/lib/plpgsql.so'  LANGUAGE  'c';
template1=>
template1=> CREATE  TRUSTED  PROCEDURAL  LANGUAGE  'plpgsql'
```

```
template1-> HANDLER  plpgsql_call_handler
template1-> LANCOMPILER  'PL/pgSQL';
template1=> CREATE  DATABASE  nogadax;
template1=> \q
[postgres@nogadax postgres]#



2. 다음 방법은 생성한 데이타베이스마다 하나하나 다 등록을 하는 것이다.

등록 과정

[postgres@nogadax postgres]# psql  nogadax
nogadax=>
nogadax=> CREATE  FUNCTION  plpgsql_call_handler()  RETURNS  OPAQUE  AS
nogadax-> '/usr/local/pgsql/lib/plpgsql.so'  LANGUAGE  'c';
nogadax=>
nogadax=> CREATE  TRUSTED  PROCEDURAL  LANGUAGE  'plpgsql'
nogadax-> HANDLER  plpgsql_call_handler
nogadax-> LANCOMPILER  'PL/pgSQL';
nogadax=>
nogadax=> \q
[postgres@nogadax postgres]#


이제는 위의 두가지 방법중 하나를 선택하여 등록을 하면 된다.
```

3.3 TRIGGER 예제 1

다음은 아래 예제를 실행하는 방법이다.

```
[podtgres@nogadax postgres]$ psql  nogadax
nogadax=> \i  /home/postgres/trg.sql
.....
.....


-------------------------------------------------------Cut here!!
--coded  BY  NoGaDa-X  2000/02/19

--DROP all Object for safe_test
 DROP FUNCTION ins_row();
 DROP TRIGGER trg_test ON test1;
 DROP TABLE  test1;
 DROP TABLE  test2;

--Create Table
 CREATE  TABLE  test1(
   tab1_id    int4,
   tab1_name  text
 );

 CREATE  TABLE  test2(
```

```
  tab2_id    int4,
  tab2_memo  text  DEFAULT 'None'
);


--Create Function
CREATE  FUNCTION ins_row() RETURNS OPAQUE
 AS '
   BEGIN
     INSERT INTO test2(tab2_id) VALUES(new.tab1_id);
     RETURN new;
   END;
 ' LANGUAGE 'plpgsql';


--Create Trigger
CREATE  TRIGGER  trg_test
AFTER   INSERT   ON test1 FOR  EACH  ROW
EXECUTE PROCEDURE   ins_row();


--INSERT  Transaction
BEGIN;
    INSERT  INTO  test1 values(1,'nogadax');
    INSERT  INTO  test1 values(2,'winob');
    INSERT  INTO  test1 values(3,'diver708');
    INSERT  INTO  test1 values(4,'jini');
    INSERT  INTO  test1 values(5,'opensys');
    INSERT  INTO  test1 values(6,'Linuz');
END;

--SELECT TRACTION
BEGIN;
    SELECT  *  FROM  test1;
    SELECT  *  FROM  test2;
END;

----------------------------------------------------------End !!
```

3.4 TRIGGER 2

CREATE TRIGGER

```
CREATE  TRIGGER  trigger_name  { BEFORE | AFTER } { event [OR,...] }
 ON  table  FOR  EACH  { ROW | STATEMENT }
 EXECUTE  PROCEDURE  funcname ();
```

```
trigger_name : TRIGGER 의 이름
table        : Table 의 이름
event        : INSERT , DELETE , UPDATE 들 중에서 하나 혹은 두세개를
               TRIGGER 를 기동시키기 위한 event 로 봄
               ( 예 : INSERT OR UPDATE )
func_name    : user 가 제공한 함수. 이 함수는 트리거가 생성되기 전에 만들어져야 한다.
```

또한, 이 함수의 return 형은 opaque이며 인수가 없어야 한다.
(이 부분은 PostgreSQL Programmer's Guide 에 나와 있는 부분인데 function 에서 왜 인수가 쓰이면 안되는지 그리고 opaque 형의 리턴 값만 되는지 를 정확히 설명한 부분이 없다.)

"CREATE TRIGGER"로 인해 TRIGGER 가 성공적으로 생성되면 CREATE 라는 메시지가 출력된다.

DESCRIPTION

CREATE TRIGGER은 현재의 데이타베이스에 새로운 TRIGGER을 등록할 것이다.Trigger 은 테이블(릴레이션)과 연계되어서 미리 규정된 함수 func_name을 실행한다.

트리거는 트리거의 생성시 BEFORE키를 사용하여 Tuple(row,record)에 어떤 event가 발생하기 전에 기동되어 질수 있도록 규정되어질수 있으며 반대로 AFTER키를 사용하여 event가 완료 후에 기동되게 할수도 있다.

(다음은 부분적으로 Postgres data changes visibility rule 이 참조되었다.)
트리거가 BEFORE에 의해 event전에 기동되어 진다면, 트리거는 현재의 Tuple에 대한 event를 건너뛰게 한다. 특히 INSERT나 UPDATE의 event에 대해서는 삽입되어질 튜플의 변화를 인지할 수 없다. 즉, BEFORE성격의 트리거는 변경되어질 튜플들에 대해서 "invisible"한 상태이다. 단지, 처리되어질 event 만 인식할 수 있다.

또한, 트리거가 AFTER 키에 의해 event후에 기동되어지면, 최근의 삽입,UPDATE,삭제 등이 트리거에 "visible" 이다. 즉, 변경된 부분을 트리거가 인지할 수 있다.

event는 다중의 event 를 OR 에 의해 규정할 수 있다. 또한 동일한 릴레이션에 동일한 event 를 지정하는 하나 이상의 트리거를 정의할 수 있으나, 이는 트리거의 기동 순서를 예측할 수 없게 된다.

트리거가 SQL 쿼리를 실행할때 다른 트리거들을 기동시킬 수 있으며 이를 CASCADE 트리거라 한다. 이러한 캐스케이드 트리거의 레벨에는 제한이 없으므로 한번의 트리거로 여러개의 다중의 트리거를 기동시킬 수 있다.

동일한 릴레이션에 대한 INSERT 트리거가 있다면 이 트리거는 다시 동일한 릴레이션에 대한 트리거가 기동되어질 수 있다. 하지만 아직 PostgreSQL은 이런 트리거에 대한 튜플의 동기화가 지원되지 않으므로 주의를 하여야 할 것이다.

NOTES

CREATE TRIGGER 은 PostgreSQL의 확장된 기능이다.
단지 릴레이션(Table) 의 소유자만이 그 릴레이션에 트리거를 생성할 수 있다.
버전 6.4에서 STATEMENT 는 구현되지 않았다.

3.5 TRIGGER 예제 2

CASCADING TRIGGER SAMPLE

```
----------------------------------------------------Cut here !!
--coded  by NoGaDa-X
--cascading tigger

DROP TRIGGER tab1_trg ON test1;
DROP TRIGGER tab2_trg ON test2;
DROP TRIGGER tab3_trg ON test3;

DROP FUNCTION tab1_func();
DROP FUNCTION tab2_func();
DROP FUNCTION tab3_func();

DROP TABLE test1;
DROP TABLE test2;
DROP TABLE test3;
DROP TABLE test4;


--Create Table
CREATE TABLE test1(
 tab1_id   INT4
);

CREATE TABLE test2(
  tab2_id  INT4
);

CREATE TABLE test3(
  tab3_id   INT4
);

CREATE TABLE test4(
  tab4_id   INT4
);


--Create Function
CREATE FUNCTION tab1_func() RETURNS opaque
AS '
   BEGIN
      INSERT  INTO  test2 values( new.tab1_id);
      RETURN new;
   END;
' LANGUAGE 'plpgsql';

CREATE FUNCTION tab2_func() RETURNS opaque
AS '
   BEGIN
      INSERT  INTO  test3 values( new.tab2_id);
      RETURN new;
   END;
' LANGUAGE 'plpgsql';

CREATE FUNCTION tab3_func() RETURNS opaque
```

```
AS '
   BEGIN
      INSERT  INTO  test4 values( new.tab3_id);
      RETURN new;
   END;
' LANGUAGE 'plpgsql';


--Create Trigger
CREATE TRIGGER tab1_trg AFTER
INSERT OR UPDATE ON test1 FOR EACH ROW
EXECUTE PROCEDURE tab1_func();

CREATE TRIGGER tab2_trg AFTER
INSERT OR UPDATE ON test2 FOR EACH ROW
EXECUTE PROCEDURE tab2_func();

CREATE TRIGGER tab3_trg AFTER
INSERT OR UPDATE ON test3 FOR EACH ROW
EXECUTE PROCEDURE tab3_func();


--transaction
BEGIN;
  INSERT INTO test1 VALUES (1);
  SELECT * from test1;

  INSERT INTO test1 VALUES (2);
  SELECT * from test2;

  INSERT INTO test1 VALUES (3);
  SELECT * from test3;

  INSERT INTO test1 VALUES (4);
  SELECT * from test4;
END;

-----------------------------------------------End !!
```

제 4 절 PL/pgSQL

4.1 PL/pgSQL 1 (PL/pgSQL처리기 등록)

1. Procedure Language

Postgres 6.3 버젼 부터 PostgreSQL은 procedural Language(PL)를 지원하기 시작했다. 이것은 PostgreSQL 만의 특별한 경우로서 oracle 의 PL/SQL과 비유될 수 있다. 하지만 특수한 언어인 PostgreSQL의 PL은 PostgreSQL에 내장된것 아니고 모듈화된 Handler를 다시 데이타베이스에 등록을 해주어야 한다. 그렇지 않으면 데이타베이스는 PL 로 쓰 여진 function의 내용을 이해할 수 없을 것이다. 결론적으로 처리기는 공유객체로서 컴파일되며 동적으로 Load 되는 특별한 언어 처리 기능이다.

여기서는 PL Handler의 등록의 예로서 PL의 한종류인 PL/pgSQL 언어를 등록하겠다.

Installing Procedural Languages

공유객체인 처리기는 컴파일된 후 인스톨되어야 하는데 디폴트로 PL/pgSQL 은 PostgreSQL 설치시 자동으로 컴파일된후 라이브러리 디렉토리에 놓여진다. 다 른 처리기인 PL/Tcl 은 PostgreSQL 컴파일시 명시적으로 설정되어야지만 컴파 일되며 라이브러리 디렉토리에 놓여진다. 라이브러리 디렉토리는 설치되어질 PostgreSQL의 바로 밑의 lib 이다. 예를 들어 PostgreSQL의 절대경로가 다음과 같다고 하자.

```
/usr/local/pgsql
```

그러면 라이브러리 디렉토리의 절대경로는 다음과 같다.

```
/usr/local/pgsql/lib
```

PL/pgSQL 언어 처리기를 설치하기 위해서는 먼저 위의 라이브러리 디렉토리에서 "plpgsql.so" 를 먼저 확인하여야 한다.

확인 후 CREATE FUNCTION 와 CREATE PROCEDURAL LANGUAGE 에 의해 각 데이타베이스 에서 등록을 하여야 한다. 각 데이타베이스에서 등록을 하지않고 일괄적으로 처리하고 싶다면 PostgreSQL 의 특별힌 데이타베이스인 "template1" 에서 등록을 하면된다. template1 에서 등록이 되었다면 차후 생성되는 데이타베이스에는 자동적으로 처리기가 등록된다.

PL/pgSQL 처리기 등록 예제

```
[postgres@nogadax postgresql]psql template1
 template1=>
 template1=> CREATE  FUNCTION  plpgsql_call_handler()  RETURNS  OPAQUE
 template1-> AS  ' /usr/local/pgsql/lib/plpgsql.so '
 template1-> LANGUAGE  'C' ;
 template1=>
 template1=> CREATE  TRUSTED  PROCEDURAL  LANGUAGE  'plpgsql'
 template1-> HANDLER  plpgsql_call_handler
 template1-> LANCOMPILER  'PL/pgSQL' ;
 template1=>
 template1=> CREATE  DATABASE  nogadax ;
 template1=>
```

혹은 위의 문을 화일로 저장한후 \i 를 사용할 수 있다.

```
template1=> \i   /usr/local/src/plsql_inst.sql
```

"CREATE TRUSTED PROCEDURAL LANGUAGE" 에서 TRUSTED 키워드는 PostgreSQL 의 슈퍼유저 권한이 없는 일반 유저가 "CREATE FUNCTION" 이나 "CREATE TRIGGER" 문을 사용할 때 등록 된 procedure language(PL) 를 사용할 수 있도록 해준다.

2. PL/pgSQL

PL/pgSQL 은 PostgreSQL 데이타베이스 시스템에서 "Loadable Procedural Language"이다. 이 패키지는 Jan Wieck 에의해 작성되었다.

OVERVIEW

```
1. PL/pgSQL 은 function 이나 trigger procedure 를 만드는데 사용되어 질 수 있다.
2. SQL 문에 제어 구조를 추가할 수 있다.
```

```
3. 복잡한 계산을 구현할 수 있다.
4. user가 정의한 Type, Function, Operation을 상속할 수 있다.
5. Server 에 의해 Trusted(Authentication 관련의 뜻)된것을 정의할 수 있다.
6. 사용하기 쉽다.
```

설명

```
PL/pgSQL 은 대소문자의 구분이 없으므로 키워드나 Identifier 들은 대소문자
구분없이 혼용되어 쓰일 수 있다.
```

PL/pgSQL 은 블럭 지향언어이다. 블럭은 다음처럼 정의되어진다.

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

블럭의 statements구역내에 여러개의 sub-block이 생성될 수 있으며 이는 서브블럭내의 변수 들을 서브블럭 외부로부터 숨기기 위해 사용되어질수 있다. 블럭 앞부분의 declarations구역 에서 선언된 변수는 function 이 호출될 때 단지 한번 초기화가 되는 것이 아니라 블럭으로 진입할 때마다 매번 디폴트 값으로 초기화된다.

PL/pgSQL 의 BEGIN/END 와 Transaction(BEGIN; END;)을 위한 데이타베이스의 명령문과는 다르다는 것을 이해해야 한다. 또한 Function과 Trigger Procedure 에서는 트랜잭션을 시작 하거나 commit 을 할 수 없고 Postgres는 중첩된 트랜잭션을 가질 수 없다.

```
--       : 한 라인의 주석처리
/*   */  : 블럭 단위 주석 처리
```

example

```
CREATE  FUNCTION  logfunc2(text,text,text)  RETURNS  datetime
AS '
    DECLARE  logtxt1  ALIAS  FOR  $1;
             logtxt2  ALIAS  FOR  $2;
             logtxt2  ALIAS  FOR  $3;
             curtime  datetime;
    BEGIN
       curtime :=''now'';
       INSERT  INTO  logtable  VALUES (logtxt1,logtxt2,logtxt3,curtime);
       RETURN  curtime;
    END;
' LANGUAGE 'plpgsql';
```

설명

$1,$2,$3 은 함수의 인자들로서 나열된 순서로서 참조되어진다. DECLARE 의 ALIAS FOR 변수 $1 에 대한 별명을 설정한다. 이로서 $1 에 대한 가독성이 높아질수 있다. curtime := "now"; 는 변수 curtime에 현재의 시각값을("now") 할당한다.":=" 은 변수에 값을 할당할때 쓰인다. 마지막으로 위의 함수의 리턴값이 datetime 이므로 datetime 타입의 변수 curtime 을 리턴하게 된다.

4.2 PL/pgSQL 2

example 1

다음은 예제입니다. 적당히 화일로 복사해서 실행을 하면 됩니다.

```
DROP FUNCTION test1();
DROP TABLE tab1;

CREATE TABLE tab1 (
   id    int4,
   name  text
);

CREATE FUNCTION test1() RETURNS int4
AS '
   DECLARE
      var1  tab1.id%TYPE:=1;
      var2  tab1.name%TYPE;
      var3  var2%TYPE:=''nogada'';
   BEGIN
      INSERT INTO tab1(id,name) VALUES(var1,var3);
      RETURN  var1;
   END;
'  LANGUAGE 'plpgsql';

SELECT test1();
SELECT * FROM tab1;
```

설명

위의 예제는 DROP명령문으로부터 시작한다. 별다른 이유는 없고 안전한 테스트를 위해 기존에 있을지 모를 function이나 table 을 먼저 삭제한다. DROP 명령어로 인한 에러는 무시해도 된다.Function의 DECLARE 부분은 변수의 선언 구역으로 보면 되겠다. var1 , var2,var3은 변수명이다. tab.id%TYPE 은 var1 의 변수형으로서 tab.id 의 속성을 참조하며 이속성의 바로 뒤의 %TYPE 에의해 지정된다. 또한 %TYPE 은 앞전에 선언된 변수의 자률형을 참조 할 수 있다. var3 는 바로 전에 선언된 var2 의 자료형을 참조한다.

Trigger Procedure

PL/pgSQL 은 트리거 프로시져를 정의하는데 사용되어질 수 있는데 CREATE FUNCTION문을 사용 하여 생성되어진다. 생성될 트리거 프로시져는 대체로 인자가없고 opaque형을 리턴하는 함수 로서 생성되어진다.

트리거 프로시져로서 생성된 함수에는 약간의 특수한 변수를 가지며 이는 자동으로 생성되어 지며 다음과 같다.

```
NEW       : ROW 레벨 트리거상에서 INSERT/UPDATE 로 인해 새로티 생성된
            ROW 를 유지하는 변수로서 데이타타입은 RECORD 이다. RECORD
            형은 미리 구조화되지 않은 ROWTYPE로서 selection이나 insert
            ,update 시 결과로 생성된 하나의 row 를 유지하는 형이다.
OLD       : new 와 대조되는 변수로서 UPDATE나 DELETE형 으로 인해 변경
            되기 전의 ROW를 유지하는 변수이다.
TG_NAME   : 데이타 타입은 NAME 이고 실제로 기동된 트리거의 이름에 대한
            변수이다.
```

```
TG_WHEN     :  text형이고 BEFORE나  AFTER를  가진다.
TG_LEVEL    :  text형이고 ROW나 STATEMENT를  가진다.
TG_OP       :  text형이고 INSERT나 UPDATE나 DELETE 를  가진다.
TG_RELID    :  oid형이고(Object ID) 트리거를  기동시키는  테이블의 Object ID
               이다.
TG_RELNAME :  name형이고  트리거를  기동시키는  테이블의  name  을  가지는  변수
               이다.
TG_NARGS    :  Integer형이고  트리거  프로시져에  주어지는  인자의  개수이다.
TG_ARGV[]   :  array of text  형이고  트리거  프로시져에  주어지는  인자들을
               값으로  가지는  텍스트  배열형의  변수이다.
```

4.3 예제

다음을 화일로 만들어 실행해보세요.

```
-------------------------------------------------------Cut here !!
    DROP  TRIGGER   emp_stamp  ON  emp;
    DROP  FUNCTION  emp_stamp() ;
    DROP  TABLE  emp;


    CREATE  TABLE  emp(
      empname      text,
      salary       int4,
      last_date    datetime,
      last_user    name
    );


    CREATE  FUNCTION  emp_stamp()  RETURNS  OPAQUE
    AS '
     BEGIN

       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;


       IF  NEW.salary  <  0  THEN
         RAISE  NOTICE ''% cannot have a negative salary'', NEW.empname ;
       END  IF;


       --NOTICE TEST
       RAISE NOTICE ''TRIGGER NAME : %'',TG_NAME ;
       RAISE NOTICE ''TRIGGER LEVEL : %   TRIGGER OPERATION : %'',TG_LEVEL , TG_OP;

       --EXCEPTION TEST
       RAISE EXCEPTION ''TRIGGER WHEN : %'',TG_WHEN;
       RAISE NOTICE ''TRIGGER LEVEL : %   TRIGGER OPERATION : %'',TG_LEVEL , TG_OP;
```

```
    NEW.last_date := ''now'';
    NEW.last_user := getpgusername();
    RETURN  NEW;
  END;
' LANGUAGE 'plpgsql';


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


INSERT  INTO  emp(empname,salary)  VALUES('nogadax',20);
INSERT  INTO  emp(empname) VALUES('winob');
INSERT  INTO  emp(salary)  VALUES(10);
INSERT  INTO  emp(empname,salary)  VALUES('diver',30);
INSERT  INTO  emp(salary)  VALUES(-20);


SELECT  *  FROM  emp;

--------------------------------------------------------------------------End !!
```

설명

RAISE는 메시지를 던지는 것입니다. EXCEEPTION은 포스트그레스의 DEBUG레벨로서 데이타베이스에 log 를 남기고 트랜잭션을 중지한다. 다른 키워드로 NOTICE가 있 는데 이것은 데이타베이스를 작성하고 이를 클라이언트측으로도 전송한다.

```
RAISE EXCEPTION ''% cannot have NULL salary'', NEW.empname ;
```

위의 문에서 " 와 " 사이의 % 는 NEW.empname 의 값을 출력한다.

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

NEW.empname의 NEW는 RECORD타입으로서 트리거의 이벤트의 원인이 되는 emp 의 구조를 가지며 NEW 의 값은 event의 결과로서 추가되거나 변경된 행을 가진다.

정리

Declaration : 선언부

name [CONSTANT] type [NOT NULL] [DEFAULT | :=value]

```
example )
   DECLARE
       var_a  int4  DEFAULT  5;

--var_a  변수의  DEFAULT의 값은  5이다.
```

name class%ROWTYPE

```
example )
   DECLARE
        var_a  test_table%ROWTYPE;
```

var_a 변수는 test_table 의 테이블 구조를 가진다.

name RECORD

```
example )
   DECLARE
      var_a  RECORD ;
```

특정 테이블의 구조를 가지지 않고 selection 의 결과에
대한 구조를 가질 수 있다.(NEW,OLD)cc'

name ALIAS FOR $n;

$n 에 대한 별칭

RENAME oldname TO newname

oldname를 newname로 바꿈

Data Type : 자료형

Postgres-BaseType : 포스트 그레스의 기본 자료형(int4,integer,text,char,..)

variable%TYPE

class.field%TYPE

Expression

Select expression

Statement : 처리문장

Assignment : 값의 할당

```
identifier :=expression;

SELECT  expressions  INTO  target  FROM ...

PERFORM  query  : Calling another function

RETURN  expression;

RAISE  [NOTICE | EXCEPTION]  '' message % '',variable
```

제어문

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


[label]
LOOP statements END LOOP;


[label]
WHILE expression LOOP statements END LOOP;


[label]
FOR name IN [REVERSE] expression LOOP statements END LOOP;


[label]
FOR record | row IN select_clause LOOP statement
END LOOP;


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

제 5 절 Large Object with Transaction

5.1 Large Object와 예제

포스트그레스에서는 한 튜플의 사이즈가 8192 Byte (8k Bytes) 로 제한되어 있다. 하나의 레코드에 들어갈 수 있는 데이타의 총 크기가 제한되어 있으므로 이미지나 사이즈가 8K 를 넘는 문서들은 다르게 저장되어야 한다.

포스트그레스는 Large Object 라는 개념으로 이를 극복하려한다. 과거에 포스트그 레스는 이런 큰 사이즈의 데이타를 위해 3가지의 지원이 있었으나 사용자들사이의 잦은 혼란으로 하나만을 지원하게 되었고 그것은 단순히 데이타베이스안의 데이타로 서의 Large Object 를 지원한다. 이것은 액세스를 할때 느릴수 있지만 엄격한 데이타 무결성을 제공한다.

포스트그레스는 Large Object 를 쪼개어 이를 데이타베이스의 튜플들에 저장한다. B-tree 인덱스는 랜덤한 resd-write 액세스에 대한 빠른 검색을 보증한다.

```
다음은 예제이다.

-------------------------------------------------------------------
   drop table image;

   BEGIN WORK;

   SET TRANSACTION ISOLATION LEVEL SERIALIZABLE;
```

```
create table image(
  name    text,
  raster  oid
);

insert  into    image (name , raster)
     values ('snapshot' , lo_import('/usr/local/src/snapshot01.gif') );

select  lo_export (image.raster , '/tmp/snap.gif')
     from  image  where name='snapshot';

COMMIT  WORK;
----------------------------------------------------------------------
```

Large Object Note

위의 예제에서 명시적으로 트랜잭션내에서 Large Object 의 처리가 이루어지고 있다. 이는 포스트그레스 6.5 버젼대에서부터의 Large Object 처리에 대한 요구사항으로서 6.5 이전 버젼의 암시적인 트랜잭션 요구사항과는 달리 명시적인 트랜잭션을 요구한 다. 이 요구사항이 무시된다면, 즉 명시적인 트랜잭션문이 작성되지 않는다면 비정 성적인 결과를 만든다.

설명

```
BEGIN  WORK;
  사용자 정의 트랜잭션 시작

SET  TRANSACTION  ISOLATION  LEVEL  SERIALIZABLE;
  트랜잭션 레벨 중 가장 강력한 SERIZABLE 레벨로 재 설정을 함.

포스트그레스의 트랜잭션의 디폴트 레벨은 "READ COMMITTED" 로서 트랜잭션 내의
query 는 이 query 가 실행되기전에 commit 된 데이타만 다룰 수 있다.

SERIALIZABLE 는 포스트그레스의 가장 강력한 트랜잭션 레벨로서 트랜잭션내의
query는 query시작전이 아닌 그 트랜잭션이 시작되기전에 commit된 데이타만을
다룰수 있다.

OID 는 객체에대한 포스트그레스의 시스템 관련 식별자이다.

lo_import(읽어올 데이타의 PATH); 는 데이타를 읽어들이는 Large Object 관련
내장 함수이다.

lo_export( OID , 데이타가 쓰여질 시스템의 PATH); 는 데이타를 읽어서 꺼내는
Large Object 관련 내장함수이다.

COMMIT WORK; 는 트랜잭션의 완료를 의미한다. 이로 인해 실질적인 갱신이나 삭제등이
이루어진다.
```

5.2 TRANSACTION

5.2.1 트랜잭션의 성격(ACID)

```
원자성       : 하나의 트랜잭션은 다수의 query를 실행하지만 이는 단지 하나의
(ATOMIC)       query 인양 실행되어야 한다.
```

일관성 (CONSISTENT)	:	트랜잭션의 수행에 대해 데이타베이스의 데이타들의 일관성은 유지되어야 한다.
분리 (ISOLATABLE)	:	각 트랜잭션은 분리되어 다른 트랜잭션중에 간섭해서는 안된다. 이는 병렬 (CONCURRENCY) 제어의 개념으로 데이타베이스는 멀티 유저 환경일 수 있으므로 각 유저의 트랜잭션은 안전하게 이루어져야 한다.
영구성 (DURABLE)	:	트랜잭션의 수행후 commit 된 데이타들은 영구적으로 유지되어야 한다.

5.2.2 트랜잭션 관련 SQL 명령어 정리

BEGIN [WORK | TRANSACTION]

```
BEGIN : 새로운 트랜잭션이 Chain Mode로 시작했음을 알린다.
WORK , TRANCTION : Optional Keyword. They have no effect.
```

COMMIT [WORK | TRANSACTION]

```
트랜잭션후 변경된 결과를 저장.
```

END [WORK | TRANCTION]

```
현재 트랜잭션을 COMMIT.
END는 포스트그레스 확장으로서 COMMIT 와 같은 의미이다.
```

LOCK [TABLE] name

LOCK [TABLE] name IN [ROW | ACCESS] {SHARE | EXCLUSIVE} MODE

LOCK [TABLE] name IN SHARE ROW EXCLUSIVE MODE

```
명시적으로 트랜잭션 내의 테이블을 잠금.
```

ROLLBACK [WORK | TRANSACTION]

```
현재 트랜잭션을 중지한다.
```

ABORT [WORK | TRANSACTION]

```
현재 트랜잭션을 중지한다. ABORT 는 포스트그레스 확장으로 ROLLBACK와
같은 의미로서 쓰인다.
```

SET TRANSACTION ISOLATION LEVEL {READ COMMITTED | SERIALIZABLE}

```
현재 트랜잭션에 대한 분리 레벨을 설정한다.
```

설명

```
INSERT INTO tab VALUES('qwe','www',123);
```

위의 INSERT문 이 성공적으로 수행되었다면 commit 될것이다.
아니면 RollBack 될것이다. 다시 말해, 위의 문이 성공하면
데이타베이스에 그에 따른 데이타가 저장되고 그렇지 않고
INSERT 의 실행결과가 ERROR 이면 데이타는 저장되지 않는다.

이를 autocommit 라 하는데 또한 다른말로 unchained mode
라고도 한다.

포스트그레스에서의 일반적인 명령들의 실행은 unchained mode 이다.
그리고 이를 좀 더 그술적으로 서술하면 다음과 같다.

"각각의 문장(statement)들은 암시적인 트랜잭션내에서 실행되어지고
그 문장의 끝부분에서 commit가 이루어지는데 실행이 성공적이면 commit
가 행해지고 반대로 실행이 성공적이지 않으면 rollback 되어진다."

결국은 개별적인 SQL 문들의 실행에 있어 사용자들은 자신도 모르게
트랜잭션내에서 수행하고 있고 또한 그 결과도 자신도 모르게 commit
이거나 rollback이 이루어진다.

BEGIN 은 명시적으로 트랜잭션을 시작함을 의미하며 autocommit 이 되지
않는다(chained mode). 명시적인 commit 문이 올때까지 작업들의 결과들이
데이타베이스에 저장되지 않는다.

BEGIN 문 바로 뒤에 SET 문을 사용하여 그 트랜잭션의 트랜잭션 분리 레벨
을 지정할 수 있다. SET 문의 예는 다음과 같다.

```
BEGIN WORK;
  SET  TRANSACTION  ISOLATION  LEVEL  SERIALIZABLE
  INSERT INTO tab VALUES(1,2,3);
  INSERT INTO tab VALUES(3,4,5);
COMMIT WORK;
```

트랜잭션 분리 정책은 여러 유저의 동시성 에 대한 보다 강력한 제한이라 할
수 있겠다. 포스트그레스에의 디폴트 트랜잭션 분리레벨은 "READ COMMITTED"
이다. READ COMMITTED 보다 더욱더 엄격한 레벨이 SERIALIZABLE 이다.

제 6 절 참고 문서와 기타

http://database.sarang.net

http://www.postgresql.org

postgresql 도큐먼트

```
programmer's guide
user's guide
```

오라클 관련 도서.

```
의외로 포스트그레스와 오라클이 비슷한 부분이 많더군요.
개념적으로 잘 이해가 되지 않을 때 참조를 하면서 포스트그레스를
```

공부하면 좋을 겁니다.