空间数据库原理及开发技术

第8章 空间数据库开发技术

中南大学地信系 李光强
QQ：41733233

教学目标

- 掌握PL/pgSQL变量常量定义和程序结构
- 掌握PL/pgSQL开发方法
- 掌握存储过程、自定义函数的定义和调用
- 熟悉触发器的定义和使用
- 熟悉PL/pgSQL实际应用方法
- 了解PostGIS桌面应用开发技术

教学重点

- 存储过程、自定义函数的定义和调用

教学内容

- 8.1 PostgreSQL开发
- 8.2 PL/pgSQL开发方法
- 8.3 PostGIS的PL/pgSQL开发实例
- 8.4 PostGIS桌面应用开发

8.1 PostgreSQL开发

- 8.1.1 PL程序概述
- 8.1.2 PL/pgSQL变量常量定义
- 8.1.3 PL/pgSQL程序结构

8.1.1 PL程序概述

- PostgreSQL拥有服务器端程序设计特性，允许用户开发自己的程序，包括**用户自定义函数**（user defined function,UDF）、**存储过程**（Storage Procedure）、**触发器**（Trigger）等。
- PostgreSQL为开发者提供了**数据库过程语言**（Procedural Language，**PL**），开发的程序称为**PL/pgSQL**。
- PL/pgSQL为开发者提供了多种程序结构形式，拓展了数据库本身的处理能力。
- 除了PL/pgSQL，PostgreSQL还支持多种语法开发PL程序，这些语法包括C/C++、PL/TCL、PL/Python、PL/Perl、PL/Java 等。
- PL程序的优点：
  - 减少应用程序和数据库之间的会话和网络传输
  - 提高应用的性能
  - 可重用性，易于维护，程序安全
  - 扩展了数据完整性检查功能

8.1.2 PL/pgSQL变量常量定义

- (一）变量
  - 变量的声明必须在声明部分进行定义，语法如下：

    variable_name data_type [NOT NULL] [{ DEFAULT | := | = } expression];
  - 例8-1. 定义一个整型变量i，并赋初始值“100”，语句如下：

    i integer = 100; 或 i integer :=100;
  - 例8-2. 定义一个记录型变量，语句如下：

    arow RECORD;
- （二）常量
  - 常量的值一旦被初始化就不能被改变，包括**常数型常量**和**变量型常量**两类。
  - （1）常数型常量是固定值，直接参与表达式计算。
    - 例如，100是一个整型常数型常量；’ hello world’是一个字符型常数型常量。
  - （2）变量型常量又称为常变量，在定义常变量时要使用 **CONSTANT** 关键字。常变量在声明时初始化，在程序中不允许被修改。
  - 例8-3. 定义π常变量PI，用于计算圆的面积，语句如下：

    PI CONSTANT NUMERIC := 3.14159265;

8.1.2 PL/pgSQL变量常量定义

（三）运算符

类别	运算符	描述	实例
算术运算符	+	加	a + b 结果为 5
	-	减	a - b 结果为 -1
	*	乘	a * b 结果为 6
	/	除	b / a 结果为 1
	%	模（取余）	b % a 结果为 1
	^	指数	a ^ b 结果为 8
	|/	平方根	|/ 25.0 结果为 5
	||/	立方根	||/ 27.0 结果为 3
	!	阶乘	5 ! 结果为 120
	!!	阶乘	!! 5 结果为 120
字符运算符	||	字符串连接	'Hello '||' CSU'结果为"Hell CSU"

类别	运算符	描述	实例
日期时间运算符	+	日期时间加	date '2023-06-28' + integer '7'结果为2023-07-05
	-	日期时间减	date '2023-06-28' - integer '7'结果为2023-06-21
比较运算符	=	等于	(1 = 2) 为 false。
	!=	不等于	(1 != 2) 为 true。
	<>	不等于	(1 < > 2) 为 true。
	>	大于	(1 > 2) 为 false。
	<	小于	(1 < 2) 为 true。
	>=	大于等于	(1 >= 2) 为 false。
	<=	小于等于	(1 <= 2) 为 true。
逻辑运算符	AND	逻辑与运算符	(1>2) AND(3>4) 为false
	NOT	逻辑非运算符	NOT (1>2) 为true
	OR	逻辑或运算符	(1>2) OR (3<4) 为true

8.1.3 PL/pgSQL程序结构

（一）顺序结构

- 例8-4. 顺序结构示例，语句如下：

```
DO $$
BEGIN
        RAISE NOTICE 'Hello world!';
        RAISE NOTICE 'Hello China!';
        RAISE NOTICE 'Hello CSU!';
END $$;
```

- 运行结果：
  - > NOTICE: Hello world!
  - > NOTICE: Hello China!
  - > NOTICE: Hello CSU!

8.1.3 PL/pgSQL程序结构

- （二）**分支结构**
  - 分支结构包括IF...ELSE...END IF和CASE WHEN...ELSE...END CASE两类。
  - （1）IF分支结构

```
IF condition1 THEN
  Statements1;
ELSE
  Statements2;
END IF;
```

  - 嵌套分支：

```
IF condition1 THEN
  Statements1;
ELSIF condition2 THEN
    Statements2;
ELSIF condition3 THEN
 Statements3；
ELSE
  Statements4;
END IF;
```

- 例8-5. IF..ELSE判断示例，语句如下：

```
① DO $$
② DECLARE
③   a integer := 10;
④   b integer := 20;
⑤ BEGIN
⑥   IF a > b THEN
⑦     RAISE NOTICE 'a大于 b';
⑧   ELSE
⑨     RAISE NOTICE 'a 不大于 b';
⑩   END IF;
⑪ END $$;
```

运行结果：

> NOTICE: a 不大于 b

8.1.3 PL/pgSQL程序结构

- （二）**分支结构**
  - （2）CASE分支结构

```
CASE search-expression
  WHEN expression_1 [, expression_2, ...] THEN
    when-statements
 [ ... ]
 [ELSE
    else-statements ]
END CASE;
```

  - 例8-7. CASE分支结构示例，语句如下：

```
DO $$
DECLARE
 Rate                    NUMERIC=3;
 price_segment           VARCHAR(50);
BEGIN
 CASE rate
   WHEN 0.99 THEN
     price_segment = '低';
   WHEN 2.99 THEN
      price_segment = '中';
   WHEN 4.99 THEN
      price_segment = '高';
   ELSE
      price_segment = '未指定';
END CASE;

  raise notice '价格等次: %', price_segment;
END; $$
```

运行结果：

> NOTICE: 价格等次: 未指定

8.1.3 PL/pgSQL程序结构

（三）循环结构

- PL/pgSQL循环结构分为LOOP、WHILE...LOOP、FOR...LOOP三类。
- （1）LOOP循环

```
LOOP
  Statements;
  EXIT [<label>] WHEN condition;
END LOOP;
```

- 例8-8. LOOP循环示例如下：

```
DO $$
DECLARE
  i INTEGER := 0 ;
BEGIN
 LOOP
    EXIT WHEN i = 10 ;
    RAISE NOTICE 'i = %',i;
    i := i+1;
 END LOOP ;
END ; $$
```

运行结果：

```
> NOTICE:  i = 0
> NOTICE:  i = 1
> NOTICE:  i = 2
> NOTICE:  i = 3
> NOTICE:  i = 4
> NOTICE:  i = 5
> NOTICE:  i = 6
> NOTICE:  i = 7
> NOTICE:  i = 8
> NOTICE:  i = 9
```

8.1.3 PL/pgSQL程序结构

- （三）**循环结构**
  - （2）WHILE...LOOP循环

```
WHILE condition LOOP
  statements;
END LOOP;
```

  - 例8-9. WHILE...LOOP循环示例如下：

```
DO $$
DECLARE
  i INTEGER := 0 ;
BEGIN
   WHILE i < 10 LOOP
      RAISE NOTICE 'i = %',i;
      i := i+1;
   END LOOP ;
END ; $$
```

运行结果：

```
> NOTICE:  i = 0
> NOTICE:  i = 1
> NOTICE:  i = 2
> NOTICE:  i = 3
> NOTICE:  i = 4
> NOTICE:  i = 5
> NOTICE:  i = 6
> NOTICE:  i = 7
> NOTICE:  i = 8
> NOTICE:  i = 9
```

8.1.3 PL/pgSQL程序结构

（三）循环结构

（3）FOR...LOOP循环结构

```
FOR loop_counter IN [ REVERSE ] from.. to [ BY expression ]
LOOP
   statements
END LOOP ;
```

例8-10. FOR...LOOP循环示例如下：

```
DO $$
DECLARE
  i INTEGER := 0 ;
BEGIN
   FOR i IN 0..9 LOOP
      RAISE NOTICE 'i = %',i;
   END LOOP ;
END ; $$
```

执行结果如下：

```
> NOTICE:  i = 0
> NOTICE:  i = 1
> NOTICE:  i = 2
> NOTICE:  i = 3
> NOTICE:  i = 4
> NOTICE:  i = 5
> NOTICE:  i = 6
> NOTICE:  i = 7
> NOTICE:  i = 8
> NOTICE:  i = 9
```

8.2 PL/pgSQL开发方法

- 8.2.1 PL/pgSQL代码块
- 8.2.2 PL/pgSQL存储过程
- 8.2.3 PL/pgSQL自定义函数
- 8.2.4 PL/pgSQL触发器

8.2.1 PL/pgSQL代码块

- PL/pgSQL代码块是由多行代码组成的程序块体，代码块定义语法如下：

```
[ <label> ]
[ DECLARE
 declarations ]
BEGIN
 statements;
 ...
END [ label ];
```

  - label 是可选的代码块标签，是PostgreSQL控制代码块运行结束的标志，如“$$”
  - DECLARE 是可选的声明部分，用于定义变量、常变量
  - BEGIN 和 END 之间是代码主体，是程序的主要功能代码
  - 所有的语句都使用分号（;）结束，END 之后的分号表示代码块结束。

8.2.2 PL/pgSQL存储过程

- **存储过程**是由多条SQL语句和控制语句组成的特殊程序，经数据库管理系统编译后，驻留在数据库中，可以被PL/pgSQL程序调用，也可以从另一个存储过程调用。
- 存储过程也有**输入参数和输出**参数。根据返回值类型的不同，存储过程可以分为三类：
  - （1）返回记录集的存储过程：存储过程执行并返回记录集结果，例如一个存储过程从数据库中检索出符合条件的记录集；
  - （2）返回数值的存储过程：存储过程执行并返回一个数值，也称为标量存储过程；
  - （3）行为存储过程：存储过程仅用来实现数据库的某个功能，没有返回值，例如向数据表插入一行记录的存储过程。
- 存储过程定义语法如下：

```
CREATE [ OR REPLACE ] PROCEDURE name (
[ [ argmode ] [ argname ] argtype [ { DEFAULT | = } default_expr ] [, ...] ]
)
AS <label>
[DECLARE
 declarations;]
BEGIN
 statements;
 ...
END; <label>
LANGUAGE plpgsql;
```

8.2.2 PL/pgSQL存储过程

- 例8-11. 存储过程示例。

```
CREATE OR REPLACE PROCEDURE PROC_1(n int)
AS $$
DECLARE
  i INTEGER := 0 ;
BEGIN
        FOR i IN 0..n LOOP
         RAISE NOTICE 'i = %',i;
 END LOOP ;
END ; $$
LANGUAGE plpgsql;
```

- 执行结果：

```
> NOTICE:  i = 0
> NOTICE:  i = 1
> NOTICE:  i = 2
> NOTICE:  i = 3
> NOTICE:  i = 4
> NOTICE:  i = 5
> NOTICE:  i = 6
> NOTICE:  i = 7
> NOTICE:  i = 8
> NOTICE:  i = 9
```

- 存储过程调用语句如下：
  - CALL PROC_1(9);

8.2.3 PL/pgSQL自定义函数

- PL/pgSQL**自定义函数**是使用SQL语句和控制语句开发具有传入参数和返回值的程序体。自定义函数经数据库管理系统编译后，驻留在数据库中，可以被SQL语句和PL/pgSQL程序调用。
- 自定义函数语法如下：

```
CREATE [ OR REPLACE ] FUNCTION name (
[ [ argmode ] [ argname ] argtype [ { DEFAULT | = } default_expr ] [, ...] ] )
 RETURNS rettype
AS $$
DECLARE
 declarations
BEGIN
 statements;
 ...
END; $$
LANGUAGE plpgsql;
```

8.2.3 PL/pgSQL自定义函数

- 例8-12. 自定义函数示例。

```
CREATE OR REPLACE FUNCTION FUN_1(n int)
RETURNS integer
AS $$
DECLARE
  i INTEGER := 0 ;
        sum integer:=0;
BEGIN
       FOR i IN 0..n LOOP
        sum := sum + i;
 END LOOP ;
 return sum;
END ; $$
LANGUAGE plpgsql;
```

- 函数调用语句如下：

```
SELECT FUN_1(100);
```

- 返回结果如下：

```
fun_1
-----------
5050
```

8.2.4 PL/pgSQL触发器

- PostgreSQL **触发器**（trigger）是一种特殊的函数，侦测数据表发生变更事件（如INSERT、UPDATE、DELETE 或者 TRUNCATE 语句）或者数据库事件（DDL 语句），自动触发和执行这些函数，而不是由用户或者应用程序调用。
- 对于数据变更触发器，PostgreSQL 支持两种级别的触发方式：行级（row-level）触发器和语句级（statement-level）触发器。
- 触发器可以在事件发生之前（BEFORE）触发，也可以事件之后（AFTER）触发。

触发时机	触发事件	行级触发器	语句级触发器
BEFORE	INSERT、UPDATE、DELETE	表和外部表	表、视图和外部表
BEFORE	TRUNCATE		表
AFTER	INSERT、UPDATE、DELETE	表和外部表	表、视图和外部表
AFTER	TRUNCATE		表
INSTEAD OF	INSERT、UPDATE、DELETE	视图	
INSTEAD OF	TRUNCATE		

8.2.4 PL/pgSQL触发器

- PostgreSQL 触发器的创建分为两步：
  - （1）使用 CREATE FUNCTION 语句创建触发器函数，创建触发器函数语法和创建自定义函数语法相同。但是值得注意的是：触发器函数与普通函数的区别在于它没有输入参数，并且返回类型为 trigger。
  - （2）使用 CREATE TRIGGER 语句将该函数与表进行关联，语法如下：

```
CREATE TRIGGER trigger_name
<BEFORE | AFTER | INSTEAD OF> <event [OR ...]>
 ON <table_name>
 [FOR [EACH] <ROW | STATEMENT>]
 [WHEN ( condition ) ]
 EXECUTE FUNCTION <trigger_function>;
```

8.2.4 PL/pgSQL触发器

- PostgreSQL 触发器示例
  - 例8-13. 若存在多边形数据表mypolygon(id serial8 primary, geom geometry)，如果要为该表添加触发器mytrigger和触发器函数fn_beforeinsert( )，实现：当向该表插入一条记录时，判断新记录多边形是否与表中已有多边形相交，若存在相交记录，则拒绝插入；否则完成插入。代码如下：

```
-- 创建数据表mypolygon
DROP TABLE IF EXISTS mypolygon;
CREATE TABLE mypolygon
(
id                    SERIAL8 PRIMARY KEY,
geom      GEOMETRY
);

-- 触发器函数
CREATE OR REPLACE FUNCTION "CH08"."fn_beforeinsert"()
  RETURNS trigger
            AS $$
BEGIN
            If EXISTS (
       SELECT
          1
       FROM
          mypolygon a
          WHERE
             st_intersects(a.geom, NEW.geom)
       ) THEN
          RETURN NULL;
       END IF;
                RETURN new;
END; $$
  LANGUAGE plpgsql;

-- 创建触发器
CREATE TRIGGER mytrigger
    BEFORE INSERT ON mypolygon
    FOR EACH ROW
    EXECUTE PROCEDURE CH08.fn_beforeinsert();
```

8.2.4 PL/pgSQL触发器

■ 例8-13. 测试如下：

执行以下语句验证触发器：

① INSERT INTO mypolygon(geom)
VALUES('POLYGON((0 0,10 0, 10 10, 0 10, 0 0))'::geometry);

执行结果：

- Affected rows: 1

② INSERT INTO mypolygon(geom)
VALUES('POLYGON((20 0,30 0, 30 10, 30 10, 20 0))'::geometry);

执行结果：

- Affected rows: 1

③ INSERT INTO mypolygon(geom)
VALUES('POLYGON((0 0,10 0, 10 10, 0 10, 0 0))'::geometry);

执行结果：

> Affected rows: 0

结果分析：执行①~②插入语句都向数据表插入了新记录，当执行第③条插入语句时，没有通过触发器函数fn_beforeinsert的检查，拒绝插入数据，即“Affeected rows 0”

8.3 PostGIS的PL/pgSQL开发实例

- 8.3.1 PostGIS的PL/pgSQL代码块程序
- 8.3.2 PostGIS的PL/pgSQL自定义函数
- 8.3.3 PostGIS的PL/pgSQL存储过程

8.3.1 PostGIS的PL/pgSQL代码块程序

- 例8-14. 使用代码块程序创建空间数据表points，随机生成（0，0）到（100，100）范围内的100个点，并存入points数据表中。程序如下

```
DO $$
DECLARE
  point          geometry;       -- 定义点几何图形变量
  x              float;          -- 定义点x坐标变量
  y              float;          -- 定义点y坐标变量
  i              int =0;         -- 循环变量
BEGIN
   -- 生成点的外边界
   DROP TABLE IF EXISTS CH08.box;
   CREATE TABLE CH08.box
    AS
      SELECT
        'POLYGON((0 0,100 0,100 100,0 100,0 0))'::geometry as geom;

   -- 创建点空间数据表
   DROP TABLE IF EXISTS CH08.points;
   CREATE TABLE CH08.points(
      id          serial primary key,
      geom        geometry
);
   -- 循环生成100个点
   FOR i IN 1..100 LOOP
      x        = random()*100;
      y        = random()*100;
      point    = st_makepoint(x,y);

      INSERT INTO CH08.points(geom)  VALUES(point);
   END LOOP;
END; $$
```

扩展：上述随机生成100个点的语句也可以改为:

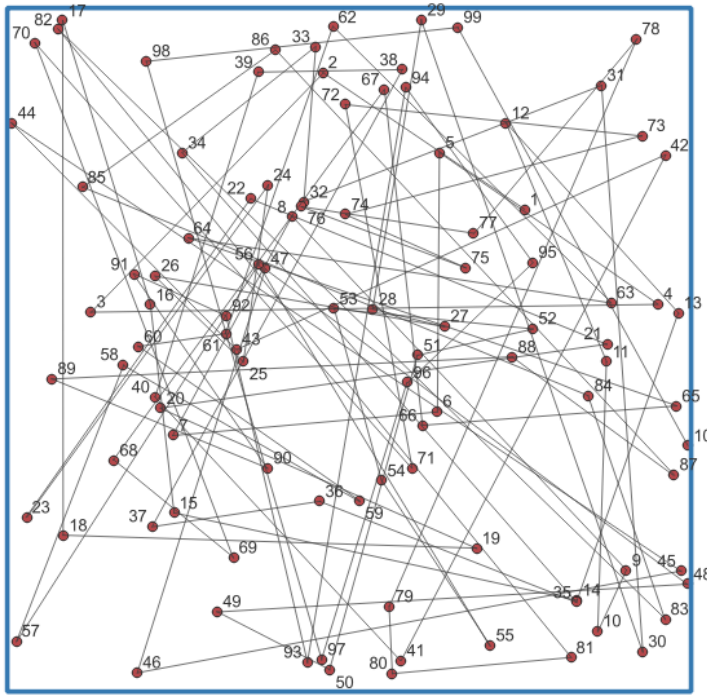
```
INSERT INTO CH08.points(geom)
 SELECT St_MakePoint(random()*100,random()*100)
 FROM generate_series(1,100);
```

其中，**generate_series(m,n)**函数生成从m～n的序列整数。

8.3.1 PostGIS的PL/pgSQL代码块程序

- 例8-15. 利用上例生成的点集，按序将相邻的两个点生成线段并存入lines空间数据表。例如点1和点2生成一个线段，点2和点3生成一个线段等等。代码如下

```
DO $$
DECLARE
  i             int;          -- 循环变量
  rec1          record;       -- 记录1
  rec2          record;       -- 记录2
  line          geometry;     -- 定义线几何图形变量
BEGIN
   -- 创建lines数据表
  DROP TABLE IF EXISTS CH08.lines;
  CREATE TABLE CH08.lines
  (
  id            serial primary key,
  geom          geometry
  );

FOR i IN 1..99 LOOP
  -- 取连接两个点的记录
  SELECT geom  FROM CH08.points WHERE id=i      INTO rec1 ;
  SELECT geom  FROM CH08.points  WHERE id=i+1 INTO rec2;
    -- 生成线段，并插入到lines数据表
    line = ST_MakeLine(rec1.geom,rec2.geom);
    INSERT INTO CH08.lines(geom) VALUES(line);
  END LOOP;
END; $$
```



8.3.2 PostGIS的PL/pgSQL自定义函数

- 例8-17. 定义函数fn_getCenterClosestPoint，查询例8-14所有点里，距离点集几何中心最近的点，并返回。函数定义的语句如下

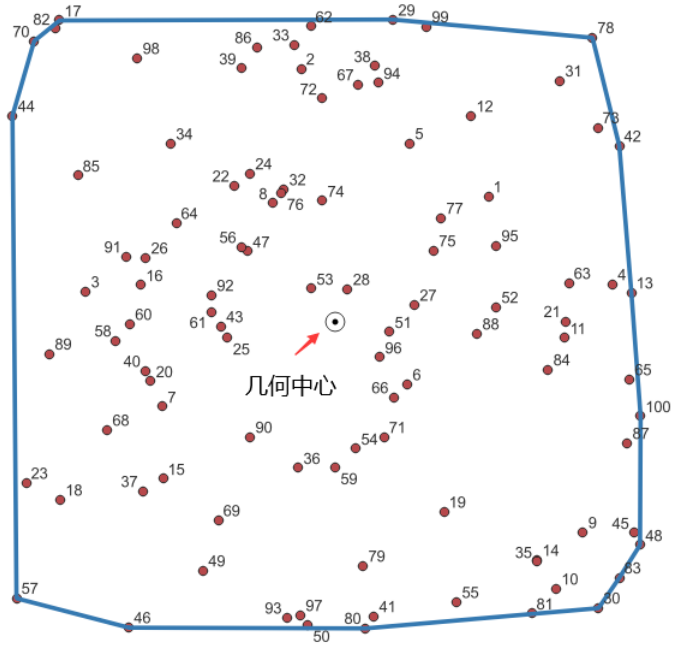
```
CREATE OR REPLACE FUNCTION CH08.fn_getCenterClosestPoint()
RETURNS geometry
AS $$
DECLARE
  center        geometry;      -- 点集的几何中心
  point         geometry;      -- 距离几何中心的最近点
BEGIN
   -- 计算点集的几何中心并赋给center变量
   SELECT ST_centroid(st_union(geom))
     FROM CH08.points
     INTO center;

   -- 寻找距离中心点最近的点
   SELECT ST_ClosestPoint(st_union(geom), center)
      FROM CH08.points
      INTO point;

   RETURN point;
END; $$
LANGUAGE plpgsql;
```

函数调用语句如下：

```
SELECT st_astext(CH08.fn_getCenterClosestPoint());
```

调用结果如下：

```
st_astext
-----------------------------------------------------------
POINT(53.42901654476151 55.96479227149267)
```



8.3.2 PostGIS的PL/pgSQL自定义函数

- 例8-19. 定义函数fn_getNearPoints，输入点id、距离d值，从例8-14点集中查询指定id点的d近邻点集（即距离点d范围内的点），返回MultiPoint结果。定义语句如下：

```
CREATE OR REPLACE FUNCTION CH08.fn_getNearPoints(_id int,d float)
RETURNS geometry
AS $$
DECLARE
   nn          geometry;      -- 近邻点集
   point       geometry;      -- 目标点
   buffer      geometry;      -- 目标点缓冲区
BEGIN
   -- 查询指定id的点
   SELECT geom  FROM CH08.points WHERE id=_id  INTO point;

   -- 判断是否存在id=_id的点,如果不存在，则返回null
   IF point is null THEN
      RETURN null;
   END IF;
    -- 生成点缓冲区
   buffer = ST_Buffer(point,d);
   -- 查询被缓冲区覆盖的点集
   SELECT ST_Union(geom)  FROM CH08.points
     WHERE ST_Contains(buffer, geom)   INTO nn;
    RETURN nn;
END; $$
LANGUAGE plpgsql;
```

函数调用语句如下：

```
SELECT st_astext(CH08.fngetNearPoints(28,10.0));
```

返回结果如下：

```
st_astext
---------------------------------
MULTIPOINT(47.79196 56.008191,53.429016 55.964792,60.002324 49.2133313)
```

8.3.3 PostGIS的PL/pgSQL存储过程

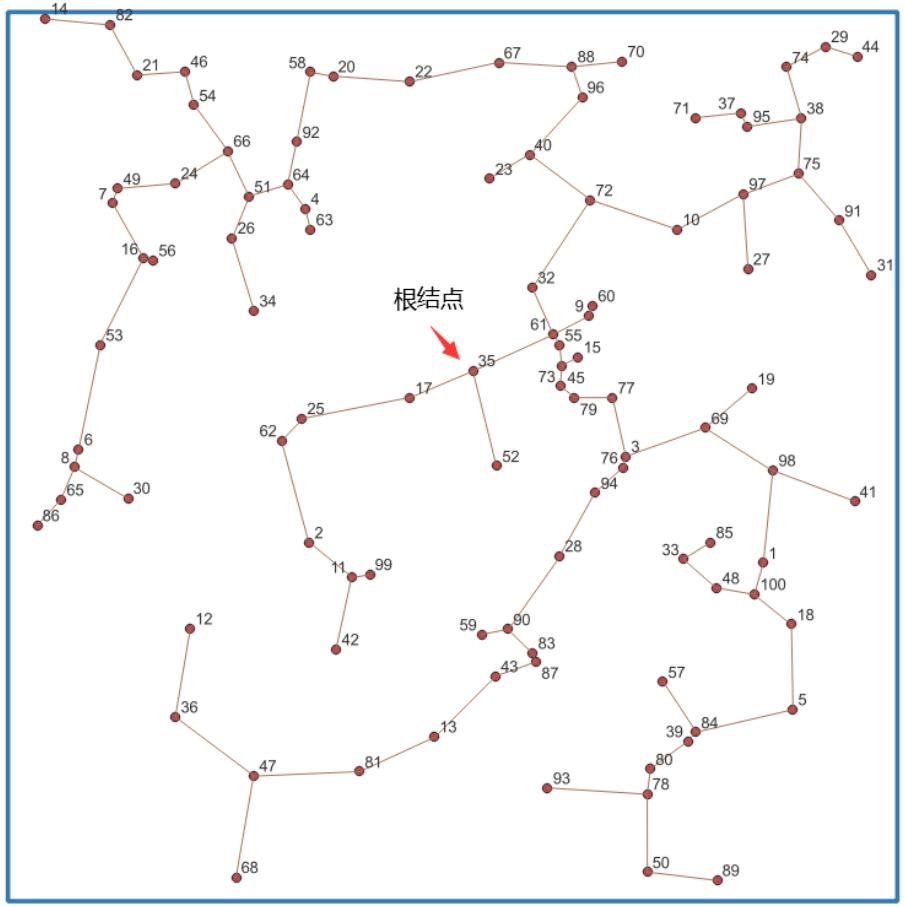
- 例8-21. 定义存储过程sp_CreateMST，生成例8-14点集的最小生成树（minimum-cost spanning tree，MST），树的根结点为点集几何中心最近的点。定义语句如下：

```
CREATE OR REPLACE PROCEDURE sp_CreateMST()
AS $$
DECLARE
  ids          int[];          -- 最小生成树点集id数组
  geoms        geometry[] ;    -- MST结点集
  line         geometry;       -- 最小生成树连线
  point        geometry;       -- MST结点集中与新加入点最近的点
  rec          record;         -- 新加入点记录
  i            int =0;
  _id          int;

BEGIN
  -- 创建MST连接线数据表
  DROP TABLE IF EXISTS CH08.mst;
  CREATE TABLE CH08.mst (
     ID          serial PRIMARY KEY,
     start_id    int,            -- 连线起点id
     end_id      int,            -- 连线终点id
     geom        geometry
  );
  -- 调用例8-14函数，获取距离点集几何中心最近点的id
  _id =CH08.fn_getCenterClosestPoint();
  SELECT id,geom FROM CH08.points where id=_id into rec;
  -- 将id加入到MST树结点集
  ids      = array[rec.id];
  geoms  = array[rec.geom];
  point    = rec.geom;

FOR i IN 1..99 LOOP
    -- 寻找与MST结点集最近的点
    SELECT a.* FROM CH08.points a,
                  (SELECT st_closestpoint(st_union(geom), st_collect(geoms)) as geom
                    FROM CH08.points
                      WHERE id<>all(ids)
                   ) foo
       WHERE st_equals(a.geom, foo.geom)
        INTO rec;

    -- 找出与新结点最近的MST结点
    point = st_closestpoint(st_collect(geoms), rec.geom);
    -- 获取point对应的id
    _id = ids[array_position(geoms, point)];

    -- 将id加入到MST树结点集
    ids       = array_append(ids,rec.id);
    geoms  = array_append(geoms,rec.geom);
    line = st_makeline(point,rec.geom);
    INSERT INTO CH08.mst(start_id,end_id,geom) VALUES(_id,rec.id,line);
  END LOOP;
END; $$
LANGUAGE plpgsql;
```

8.3.3 PostGIS的PL/pgSQL存储过程

- 存储过程调用语句如下：

```
CALL sp_CreateMST();
```



8.4 PostGIS桌面应用开发

- 桌面应用开发是使用高级开发语言研发运行于桌面操作系统之上的应用程序的开发方法，GIS桌面应用开发方法大致包括
  - 独立开发模式
  - 宿主式开发模式
  - 组件式开发模式
  - 等等
- QGIS桌面开发
  - QGIS的Python控制台开发模式
  - 基于Python的QGIS插件开发模式

中南大學
CENTRAL SOUTH UNIVERSITY

感谢您的观看!