

# 《完整性约束》 实验报告



学院： 计算机学院（国家示范性软件学院）

---

班级： 2019211308 2019211308 2019211308

---

姓名： 顾天阳 曾世茂 庞仕泽

---

学号： 2019211539 2019211532 2019211509

---

# 目录

一、利用 Create table/Alter table 语句建立完整性约束.....	3
二、主键/空值/check/默认值约束验证 .....	3
1、主键约束 .....	3
2、空值 .....	5
3、Check 约束 .....	7
4、默认值 .....	8
三、外键/参照完整性约束验证.....	8
1、参照完整性约束验证.....	8
2、级联外键关联下数据访问.....	10
四、函数依赖分析验证 .....	13
方案一： .....	13
方案二： .....	13
五、触发器约束.....	14
实验一： .....	14
实验二： .....	15
六、问题及解决.....	17
问题一： .....	17
问题二： .....	17
问题三： .....	18

## 一、利用 Create table/Alter table 语句建立完整性约束

针对数据库表 tbHandOver，创建其副本 tbHandOver\_Copy，并在 create table 中定义主键，默认值，空值和 check 约束。

执行如下 SQL 语句：

```
1. create table "tbHandOver_Copy"
2. (
3.     "CITY" text,
4.     "SCCELL" varchar(50),
5.     "NCELL" varchar(50),
6.     "HOATT" int,
7.     "HOSUCC" int,
8.     "HOSUCCRATE" float default null,
9.     primary key ("SCCELL", "NCELL"),
10.    check (("HOATT" >= 0)),
11.    check (("HOSUCC" >= 0))
12. );
```

将 tbHandOver 的数据复制到 tbHandOver\_Copy 中，发现 tbHandOver\_Copy 内容和 tbHandOver 一致。

	CITY	SCCELL	NCELL	HOATT	HOSUCC	HOSUCCRATE
1	sanxia	124711-0	15290-128	797	791	.992500000000000049
2	sanxia	124711-0	124711-1	435	435	1
3	sanxia	124711-0	253932-0	146	144	.986299999999999955
4	sanxia	124711-0	124687-2	141	141	1
5	sanxia	124711-0	124711-2	111	111	1
6	sanxia	124711-0	124687-1	6	4	.666699999999999959
7	sanxia	124711-1	7201-128	7234	7227	.998999999999999999
8	sanxia	124711-1	253901-0	3157	3147	.996800000000000019
9	sanxia	124711-1	253932-0	1889	1884	.997399999999999953
10	sanxia	124711-1	124711-0	399	399	1
11	sanxia	124711-1	5691-128	311	311	1
12	sanxia	124711-1	253904-0	147	147	1
13	sanxia	124711-1	253932-2	89	89	1
14	sanxia	124711-1	15290-128	70	70	1
15	sanxia	124711-1	124711-2	63	63	1

## 二、主键/空值/check/默认值约束验证

### 1、主键约束

使用分组聚集运算语句，判断是否满足主键约束，可以看出没有重复主键的数据行。执行如下 SQL 语句：

```
1. select "SCCELL", "NCELL", count(*)
2. from "tbHandOver_Copy"
3. group by ("SCCELL", "NCELL")
4. having count(*) > 1;
```

查看表中并无主键为空的数据，说明满足主键约束。

```

1 select "SCell", "NCell", count(*)
2 from "tbHandOver_Copy"
3 group by ("SCell", "NCell")
4 having count(*) > 1;

```

SQL执行记录 消息 结果集1 X

以下是select "SCell", "NCell", count(\*) from "tbHandOver\_Copy" group by ("SCell... 该表不可编辑。

SCell	NCell	count
暂无数据		

查看表中是否有主键为空的数据。

执行如下 SQL 语句：

```

1. select *
2. from "tbHandOver_Copy"
3. where "SCell" = null and "NCell" = null;

```

结果显示表中无主键为空的数据。

```

1 select *
2 from "tbHandOver_Copy"
3 where "SCell" = null and "NCell" = null;

```

SQL执行记录 消息 结果集1 X

以下是select \* from "tbHandOver\_Copy" where "SCell" = null and "NCell" = null的... 该表不可编辑。

CITY	SCell	NCell	HOATT	HOSUCC	HOSUCC
暂无数据					

插入主键为空数据，观察 DBMS 反应。

执行如下 SQL 语句：

```

1. insert into "tbHandOver_Copy" ("SCell", "NCell", "HOATT", "HOSUCC", "
   HOSUCCRATE")
2. values (null, null, 435, 435, 1);

```

结果显示违背了主键的非空约束，插入失败。

```

1 insert into "tbHandOver_Copy" ("SCell", "NCell", "HOATT", "HOSUCC", "HOSUCCRATE")
2 values (null, null, 435, 435, 1);

```

SQL执行记录 消息

-----开始执行-----

【拆分SQL完成】：将执行SQL语句数量：（1条）

【执行SQL：（1）】

```

insert into "tbHandOver_Copy" ("SCell", "NCell", "HOATT", "HOSUCC", "HOSUCCRATE")
values (null, null, 435, 435, 1);

```

执行失败，失败原因：ERROR: null value in column "SCell" violates not-null constraint  
Detail: Failing row contains (null, null, null, 435, 435, 1).

修改原有数据行 SCell，NCell 字段为空。

执行如下 SQL 语句：

```

1. update "tbHandOver_Copy"
2. set "SCell" = null and "NCell" = null;

```

结果显示违背了主键的非空约束，更新失败。

```
1 update "tbHandOver_Copy"
2 set "SCELL" = null and "NCELL" = null;
```

SQL执行记录 消息

-----开始执行-----

【拆分SQL完成】：将执行SQL语句数量：（1条）

【执行SQL：（1）】

update "tbHandOver\_Copy"

set "SCELL" = null and "NCELL" = null;

执行失败，失败原因：ERROR: null value in column "SCELL" violates not-null constraint

Detail: Failing row contains (sanxia, null, 15290-128, 797, 791, .992500000000000049).

更新表中 SCELL 字段值为 124711-0 和 NCELL 字段值为 124711-1 的数据行，将其字段值分别修改为 124711-0 和 15290-128。

执行如下 SQL 语句：

```
1. update "tbHandOver_Copy"
2. set "SCELL" = '124711-0', "NCELL" = '15290-128'
3. where "SCELL" = '124711-0' and "NCELL" = '124711-1';
```

结果显示主键重复，违背了主键唯一性约束。

```
1 update "tbHandOver_Copy"
2 set "SCELL" = '124711-0', "NCELL" = '15290-128'
3 where "SCELL" = '124711-0' and "NCELL" = '124711-1';
```

SQL执行记录 消息

-----开始执行-----

【拆分SQL完成】：将执行SQL语句数量：（1条）

【执行SQL：（1）】

update "tbHandOver\_Copy"

set "SCELL" = '124711-0', "NCELL" = '15290-128'

where "SCELL" = '124711-0' and "NCELL" = '124711-1';

执行失败，失败原因：ERROR: duplicate key value violates unique constraint "tbHandOver\_Copy\_pkey"

Detail: Key ("SCELL", "NCELL")=(124711-0, 15290-128) already exists.

同样地，我们插入主键重复的数据。

执行如下 SQL 语句：

```
1. insert into "tbHandOver_Copy"
2. values ('city', '124711-0', '124711-1', 1, 1, 1);
```

结果显示主键重复，插入失败。

```
1 insert into "tbHandOver_Copy"
2 values ('city', '124711-0', '124711-1', 1, 1, 1);
```

SQL执行记录 消息

-----开始执行-----

【拆分SQL完成】：将执行SQL语句数量：（1条）

【执行SQL：（1）】

insert into "tbHandOver\_Copy" values ('city', '124711-0', '124711-1', 1, 1, 1);

执行失败，失败原因：ERROR: duplicate key value violates unique constraint "tbHandOver\_Copy\_pkey"

Detail: Key ("SCELL", "NCELL")=(124711-0, 124711-1) already exists.

## 2、空值

首先通过 alter table 添加 not null 约束。

```
1 alter table "tbHandOver_Copy"|
2 alter "CITY" set not null;
```

SQL执行记录 消息

```
-----开始执行-----
【拆分SQL完成】：将执行SQL语句数量：（1条）
【执行SQL：（1）】
alter table "tbHandOver_Copy"
alter "CITY" set not null;
执行成功，耗时：[44ms.]
```

插入一数据行，其 CITY 字段为空。

执行如下 SQL 语句：

```
1. insert into "tbHandOver_Copy"
2.     values (null, '12', '13', 1, 1, 1);
```

结果显示违背 CITY 非空约束，插入失败。

```
1 insert into "tbHandOver_Copy"
2     values (null, '12', '13', 1, 1, 1);
```

SQL执行记录 消息

```
-----开始执行-----
【拆分SQL完成】：将执行SQL语句数量：（1条）
【执行SQL：（1）】
insert into "tbHandOver_Copy"
    values (null, '12', '13', 1, 1, 1);
执行失败，失败原因：ERROR: null value in column "CITY" violates not-null constraint
Detail: Failing row contains (null, 12, 13, 1, 1, 1).
```

更新其中一行数据，修改其 CITY 字段为空。

执行如下 SQL 语句：

```
1. update "tbHandOver_Copy"
2.     set "CITY" = null
3.     where "SCell" = '124711-0' and "NCell" = '15290-128';
```

结果显示违背 CITY 非空约束，插入失败。

```

1 update "tbHandOver_Copy"
2 set "CITY" = null
3 where "SCELL" = '124711-0' and "NCELL" = '15290-128';

```

SQL执行记录 消息

```

-----开始执行-----

【拆分SQL完成】：将执行SQL语句数量：（1条）

【执行SQL: (1)】
update "tbHandOver_Copy"
set "CITY" = null
where "SCELL" = '124711-0' and "NCELL" = '15290-128';
执行失败, 失败原因: ERROR: null value in column "CITY" violates not-null constraint
Detail: Failing row contains (null, 124711-0, 15290-128, 797, 791, .992500000000000049).

```

### 3、Check 约束

针对 HOATT 属性上的 check 约束关系，修改 SCCELL 和 NCELL 字段值为 124711-0 和 15290-128 的数据，将其 HOATT 改为-1。

执行 SQL 语句如下：

```

1. UPDATE "tbHandOver_Copy"
2. SET "HOATT" = -1
3. WHERE "SCELL" = '124711-0'
4. AND "NCELL" = '15290-128';

```

结果显示违背 HOATT\_check 约束，更新失败。

```

1 UPDATE "tbHandOver_Copy"
2 SET "HOATT" = -1
3 WHERE "SCELL" = '124711-0'
4 AND "NCELL" = '15290-128';

```

SQL执行记录 消息

```

-----开始执行-----

【拆分SQL完成】：将执行SQL语句数量：（1条）

【执行SQL: (1)】
UPDATE "tbHandOver_Copy"
SET "HOATT" = -1
WHERE "SCELL" = '124711-0'
AND "NCELL" = '15290-128';
执行失败, 失败原因: ERROR: new row for relation "tbHandOver_Copy" violates check constraint "tbHandOver_Copy_HOATT_check"
Detail: Failing row contains (sanxia, 124711-0, 15290-128, -1, 791, .992500000000000049).

```

同样的，插入不满足 HOATT>=0 的数据行。

执行 SQL 语句如下：

```

1. insert into "tbHandOver_Copy"
2. values ('bj', '1', '2', -1, 1, 1);

```

结果显示违背 HOATT\_check 约束，插入失败。

```

1 insert into "tbHandOver_Copy"
2 values ('bj', '1', '2', -1, 1, 1);

```

SQL执行记录 消息

```

-----开始执行-----
【拆分SQL完成】：将执行SQL语句数量：（1条）

【执行SQL: (1)】
insert into "tbHandOver_Copy"
    values ('bj', '1', '2', -1, 1, 1);
执行失败，失败原因: ERROR: new row for relation "tbHandOver_Copy" violates check constraint "tbHandOver_Copy_HOATT_check"
Detail: Failing row contains (bj, 1, 2, -1, 1, 1).

```

## 4、默认值

字段 HOSUCCRATE 默认值为 null，向表中插入一行数据，不给出 HOSUCCRATE 字段。  
执行 SQL 语句如下：

```

1. insert into "tbHandOver_Copy"
2.     values ('bj', '1', '2', 1, 1);

```

结果显示插入成功。

```

1 insert into "tbHandOver_Copy"
2 values ('bj', '1', '2', 1, 1);

```

SQL执行记录 消息

```

-----开始执行-----
【拆分SQL完成】：将执行SQL语句数量：（1条）

【执行SQL: (1)】
insert into "tbHandOver_Copy"
    values ('bj', '1', '2', 1, 1);
执行成功，耗时: [12ms.]

```

查询刚插入的数据，发现其 HOSUCCRATE 被设成默认值空值。

	CITY	SCELL	NCELL	HOATT	HOSUCC	HOSUCCRATE
1	bj	1	2	1	1	

## 三、外键/参照完整性约束验证

### 1、参照完整性约束验证

首先，判断 tbHandOver\_Copy 在属性 NCELL 上的取值是否都出现在 tbCell 表的 SECTOR\_ID 列中。

执行如下 SQL 语句：

```

1. select "NCELL"
2. from "tbHandOver_Copy"
3. where "NCELL" not in (select "SECTOR_ID"
4.                          from "tbcell_invariable");

```



结果不为空，说明两张表间不满足参照完整性约束。

```
1 select "NCELL"
2 from "tbHandOver_Copy"
3 where "NCELL" not in (select "SECTOR_ID"
4                        from "tbcell_invariable");
```

SQL执行记录 消息 结果集1 X

以下是select "NCELL" from "tbHandOver\_Copy" where "NCELL" not in (select "SECT... ① 该表不可编辑。

	NCELL
1	253930-0
2	253930-1
3	2

去除上述 SQL 语句中查出的 NCELL 所在元组，使得两表间参照完整性约束关系成立。  
执行如下 SQL 语句：

```
1. delete from "tbHandOver_Copy" as t1
2. where t1."NCELL" not in (select "SECTOR_ID"
3.                        from "tbcell_invariable");
```

执行成功！

```
1 delete from "tbHandOver_Copy" as t1
2 where t1."NCELL" not in (select "SECTOR_ID"
3                        from "tbcell_invariable");
```

SQL执行记录 消息

-----开始执行-----

【拆分SQL完成】：将执行SQL语句数量：（1条）

【执行SQL：（1）】

```
delete from "tbHandOver_Copy" as t1
where t1."NCELL" not in (select "SECTOR_ID"
                        from "tbcell_invariable");
```

执行成功，耗时：[10ms.]

再次执行查询：

```
1. select "NCELL"
2. from "tbHandOver_Copy"
3. where "NCELL" not in (select "SECTOR_ID"
4.                        from "tbcell_invariable");
```

结果显示为空，说明两表已经满足参照完整性约束。

```

1 select "NCELL"
2 from "tbHandOver_Copy"
3 where "NCELL" not in (select "SECTOR_ID"
4                       from "tbcell_invariable");

```

SQL执行记录 消息 结果集1 X

以下是select "NCELL" from "tbHandOver\_Copy" where "NCELL" not in (select "SECT... ① 该表不可编辑。

NCELL
暂无数据

定义 tbHandOver 和 tbCell 之间的级联关联。

执行如下 SQL 语句：

```

1. alter table "tbHandOver_Copy"
2. add constraint FK_NCELL
3. foreign key ("NCELL") references "tbcell_invariable" ("SECTOR_ID")
4. on delete cascade
5. on update cascade;

```

执行成功，表明级联外键定义成功。

```

1 alter table "tbHandOver_Copy"
2 add constraint FK_NCELL
3 foreign key ("NCELL") references "tbcell_invariable" ("SECTOR_ID")
4 on delete cascade
5 on update cascade;

```

SQL执行记录 消息

-----开始执行-----

【拆分SQL完成】：将执行SQL语句数量：（1条）

【执行SQL：（1）】

```

alter table "tbHandOver_Copy"
add constraint FK_NCELL
foreign key ("NCELL") references "tbcell_invariable" ("SECTOR_ID")
on delete cascade
on update cascade;
执行成功，耗时：[15ms.]

```

## 2、级联外键关联下数据访问

建立好级联关系后，向 tbHandOver 表中插入一行数据，其 NCELL 值设为 124。

执行如下 SQL 语句：

```

1. insert into "tbHandOver_Copy"
2. values ('bj', '123', '124', 1, 1, 1);

```

结果显示，由于 tbCell 表中不存在 SECTOR\_ID 为 124 的数据行，违反了外键约束，因此插入失败。

```
1 insert into "tbHandOver_Copy"
2   values ('bj', '123', '124', 1, 1, 1);
3 |
```

SQL执行记录 消息

-----开始执行-----

【拆分SQL完成】：将执行SQL语句数量：（1条）

【执行SQL: (1)】

insert into "tbHandOver\_Copy" values ('bj', '123', '124', 1, 1, 1);

执行失败, 失败原因: ERROR: insert or update on table "tbHandOver\_Copy" violates foreign key constraint "fk\_ncell"  
Detail: Key (NCELL)=(124) is not present in table "tbcell\_invariable".

再向 tbHandOver 表中插入一行数据，其 NCELL 值设为 124672-0。

执行如下 SQL 语句：

```
1. insert into "tbHandOver_Copy"
2.   values ('bj', '123', '124672-0', 1, 1, 1);
```

结果显示，由于 tbCell 表中存在 SECTOR\_ID 为 124672-0 的数据行，不违反外键约束，因此插入成功。

```
1 insert into "tbHandOver_Copy"
2   values ('bj', '123', '124672-0', 1, 1, 1);
3 |
```

SQL执行记录 消息

-----开始执行-----

【拆分SQL完成】：将执行SQL语句数量：（1条）

【执行SQL: (1)】

insert into "tbHandOver\_Copy"  
values ('bj', '123', '124672-0', 1, 1, 1);

执行成功, 耗时: [9ms.]

向 tbCell 中插入一行 SECTOR\_ID 为 12 的数据，而 tbHandOver 中不存在 NCELL 为 12 的数据，插入成功。

```
1 insert into "tbcell"
2   values ('bj', '12', 'sc', '14', 'en', 12, 13, 14, 15, 16, '17', 18, 19, '20', 21, 22, 23, 24, 25);
3 |
```

SQL执行记录 消息

-----开始执行-----

【拆分SQL完成】：将执行SQL语句数量：（1条）

【执行SQL: (1)】

insert into "tbcell"  
values ('bj', '12', 'sc', '14', 'en', 12, 13, 14, 15, 16, '17', 18, 19, '20', 21, 22, 23, 24, 25);  
执行成功, 耗时: [9ms.]

将 tbHandOver 表中一行 NCELL 值为 124711-1 的数据的 NCELL 值修改为 13。

执行如下 SQL 语句：

```
1. update "tbHandOver_Copy"
2.   set "NCELL" = '13'
3.   where "NCELL" = '124711-1';
```

结果显示，tbCell 表中并没有 SECTOR\_ID 为 13 的数据行，因此违反了外键约束，更新失败。

将 tbHandOver 表中一行 NCELL 值为 124711-1 的数据的 NCELL 值修改为 124672-0。

执行如下 SQL 语句：

```
1. update "tbHandOver_Copy"
2. set "NCELL" = '124672-0'
3. where "NCELL" = '124711-1';
```

结果显示，tbCell 表中已有 SECTOR\_ID 值为 124672-0 的数据行，因此执行成功。  
删除 tbHandOver 表中 NCELL 字段值为 124711-1 的数据行，执行成功。

```
1 start transaction;
2 delete from "tbHandOver_Copy"
3 where "NCELL" = '253932-0';
4 rollback;
```

SQL执行记录 消息

```
-----开始执行-----

【拆分SQL完成】：将执行SQL语句数量：（3条）

【执行SQL: (1)】
start transaction;
执行成功, 耗时: [5ms.]

【执行SQL: (2)】
delete from "tbHandOver_Copy"
where "NCELL" = '253932-0';
执行成功, 耗时: [6ms.]

【执行SQL: (3)】
rollback;
执行成功, 耗时: [6ms.]
```

从 tbCell 表中删除 SECTOR\_ID 为 12 的数据行，表 tbHandOver 中不存在 NCELL 为 12 的数据行，执行成功。

```
1 start transaction;
2 delete from "tbcell"
3 where "SECTOR_ID" = '12';
4 rollback;
```

SQL执行记录 消息

```
-----开始执行-----

【拆分SQL完成】：将执行SQL语句数量：（3条）

【执行SQL: (1)】
start transaction;
执行成功, 耗时: [6ms.]

【执行SQL: (2)】
delete from "tbcell"
where "SECTOR_ID" = '12';
执行成功, 耗时: [6ms.]

【执行SQL: (3)】
rollback;
执行成功, 耗时: [6ms.]
```

从 tbCell 表中删除 SECTOR\_ID 值为 124672-1 的数据行, 表 tbHandOver 存在 NCELL 值为 124672-0 的数据行，执行成功。

```
1 start transaction;
2 delete from "tbcell_invariable"
3 where "SECTOR_ID" = '124672-1';
4 rollback;
```

SQL执行记录 消息

-----开始执行-----  
  
【拆分SQL完成】：将执行SQL语句数量：（3条）  
  
【执行SQL: (1)】  
start transaction;  
执行成功, 耗时: [6ms.]  
  
【执行SQL: (2)】  
delete from "tbcell\_invariable"  
where "SECTOR\_ID" = '124672-1';  
执行成功, 耗时: [6ms.]  
  
【执行SQL: (3)】  
rollback;  
执行成功, 耗时: [6ms.]

#### 四、函数依赖分析验证

在 PCI 优化分配表 tbPCIAssignment 中，验证函数依赖 ENODEB\_ID->PCI 是否成立。

方案一：

直接选取 ENODEB\_ID 值相等，但 PCI 字段不同的元组进行查看；若存在结果不为空，则其之间不满足函数依赖。

执行如下 SQL 语句：

```
1. select *
2. from "tbPCIAssignment" as t1, "tbPCIAssignment" as t2
3. where (t1."ENODEB_ID" = t2."ENODEB_ID") and (t1."PCI" <> t2."PCI");
```

结果不为空，说明该函数依赖不成立。

1 select \*  
2 from "tbPCIAssignment" as t1, "tbPCIAssignment" as t2  
3 where (t1."ENODEB\_ID" = t2."ENODEB\_ID") and (t1."PCI" <> t2."PCI");  
4

SQL执行记录 消息 结果集1 X 覆盖模式

以下是select \* from "tbPCIAssignment" as t1, "tbPCIAssignment" as t2 where (t1."EN... 该表不可编辑。

SECTOR_NAME	ENODEB_ID	PCI	PSS	SSS	LONGITUDE
三峡?	124711	185	2	61	112.84897999999999
三峡?	124711	185	2	61	112.84897999999999
三峡?	124711	183	0	61	112.84897999999999
三峡?	124711	183	0	61	112.84897999999999
三峡?	124711	184	1	61	112.84897999999999
...					

方案二：

首先对 ENODEB\_ID 分组，对于有相同 ENODEB\_ID 的元组，统计去重之后的 PCI 值，对于每组结果，只要不同的 PCI 值的数量大于 1，则说明相同的 ENODEB\_ID 对于不同得 PCI，不满足函数依赖，若所有分组结果的最大值都为 1，则满足函数依赖，否则不满足。

执行如下 SQL 语句：

```
1. select max(c)
2. from (select count(distinct "PCI") as c
3.       from "tbPCIAssignment"
```

```
4.         group by "ENODEB_ID");
```

结果显示最大值为 4，所以不满足函数依赖。

```
1 select max(c)
2 from (select count(distinct "PCI") as c
3       from "tbPCIAssignment"
4       group by "ENODEB_ID");
5
```

SQL 执行记录 消息 结果集1 X

以下是select max(c) from (select count(distinct "PCI") as c from "tbPCIAssignment" ..... ① 该表不可编辑。

max	
1	4

## 五、触发器约束

实验一：

在 tbHandOver 上定义触发器，用于在插入数据时根据 HOATT 和 HOSUCC 字段的值进而计算出 HOSUCCRATE 的值。

首先定义触发器函数，如果 HOATT 字段为 0，则 HOSUCCRATE 赋值为空；否则 HOSUCCRATE=HOSUCC/HOATT。

执行如下 SQL 语句：

```
1. create function insert_trig_func_1() returns trigger as $$
2.     begin
3.         if new."HOATT" = 0 then
4.             new."HOSUCCRATE" := null;
5.         else
6.             new."HOSUCCRATE" := new."HOSUCC" / new."HOATT";
7.         end if;
8.         return new;
9.     end;
10. $$ language plpgsql;
```

接着定义触发器，在 PostgreSQL 中，触发器对每一行调用一个过程。

执行如下 SQL 语句：

```
1. create trigger insert_trig_before_1 before insert on "tbHandOver_Copy
  1"
2.     for each row execute procedure insert_trig_func_1();
```

下面向 tbHandOver 里插入一行数据，HOATT 的值不为 0。

执行如下 SQL 语句：

```
1. insert into "tbHandOver_Copy1"
2.     values ('bj', '520', '124672-0', 4, 3);
```

结果显示插入成功。

```
1 insert into "tbHandOver_Copy1"
2 values ('bj', '520', '124672-0', 4, 3);
```

SQL执行记录 消息

-----开始执行-----

【拆分SQL完成】：将执行SQL语句数量：（1条）

【执行SQL: (1)】

```
insert into "tbHandOver_Copy1"
values ('bj', '520', '124672-0', 4, 3);
```

执行成功, 耗时: [9ms.]

打开表 tbHandOver, 我们发现数据已经成功插入, 且 HOSUCCRATE 的值已经被计算。

	CITY	SCell	NCell	HOATT	HOSUCC	HOSUCCRATE
1	bj	510	124672-0	4	3	.75
2	bj	520	124672-0	4	3	.75

下面, 继续插入一个 HOATT 字段值为 0 的数据行。

执行如下 SQL 语句:

```
1. insert into "tbHandOver_Copy1"
2. values ('bj', '1028', '124672-0', 0, 0);
```

插入成功!

```
1 insert into "tbHandOver_Copy1"
2 values ('bj', '1028', '124672-0', 0, 0);
```

SQL执行记录 消息

-----开始执行-----

【拆分SQL完成】：将执行SQL语句数量：（1条）

【执行SQL: (1)】

```
insert into "tbHandOver_Copy1"
values ('bj', '1028', '124672-0', 0, 0);
```

执行成功, 耗时: [9ms.]

打开 tbHandOver 表, 我们发现数据已经成功插入, HOSUCCRATE 为空值, 完美符合预期结果!

	CITY	SCell	NCell	HOATT	HOSUCC	HOSUCCRATE
1	bj	510	124672-0	4	3	.75
2	bj	520	124672-0	4	3	.75
3	bj	1028	124672-0	0	0	

## 实验二:

在 tbHandOver 上定义触发器, 用于在修改数据时根据 HOATT 和 HOSUCC 字段的值进而计算出 HOSUCCRATE 的值。

首先定义触发器函数, 如果 HOATT 字段为 0, 则 HOSUCCRATE 赋值为空; 否则 HOSUCCRATE=HOSUCC/HOATT。

执行如下 SQL 语句:

```
1. create function update_trig_func() returns trigger as $$
2. begin
3.     if new."HOATT" = 0 then
```

```

4.         new."HOSUCCRATE" := null;
5.     else
6.         new."HOSUCCRATE" := new."HOSUCC" / new."HOATT";
7.     end if;
8.     return new;
9. end;
10. $$ language plpgsql;

```

结果显示，定义触发器函数成功。

```

1 create function update_trig_func() returns trigger as $$
2 begin
3     if new."HOATT" = 0 then
4         new."HOSUCCRATE" := null;
5     else
6         new."HOSUCCRATE" := new."HOSUCC" / new."HOATT";
7     end if;
8     return new;
9 end;
10 $$ language plpgsql;

```

SQL执行记录 消息

```

-----开始执行-----
【拆分SQL完成】：将执行SQL语句数量：（1条）
【执行SQL：（1）】
create function update_trig_func() returns trigger as $$
begin
    if new."HOATT" = 0 then
        new."HOSUCCRATE" := null;
    else
        new."HOSUCCRATE" := new."HOSUCC" / new."HOATT";
    end if;
    return new;
end;
$$ language plpgsql;
执行成功，耗时：[12ms.]

```

接着定义触发器，在 PostgreSQL 中，触发器对每一行调用一个过程。

执行如下 SQL 语句：

```

1. create trigger update_trig_before before update on "tbHandOver_Copy1"
2.     for each row execute procedure update_trig_func();

```

执行成功！

```

1 create trigger update_trig_before before update on "tbHandOver_Copy1"
2     for each row execute procedure update_trig_func();

```

SQL执行记录 消息

```

-----开始执行-----
【拆分SQL完成】：将执行SQL语句数量：（1条）
【执行SQL：（1）】
create trigger update_trig_before before update on "tbHandOver_Copy1"
    for each row execute procedure update_trig_func();
执行成功，耗时：[10ms.]

```

下面在 tbHandOver 里修改数据，HOATT 的值不为 0。

执行如下 SQL 语句：

```

1. update "tbHandOver_Copy1"

```



2. `set "HOATT" = 5, "HOSUCC" = 2`
3. `where "CITY" = 'bj';`

打开 tbHandOver 表，发现数据已经成功更新，HOSUCCRATE 的值被自动计算。

	CITY	SCell	NCELL	HOATT	HOSUCC	HOSUCCRATE
1	bj	510	124672-0	5	2	.400000000000000022
2	bj	520	124672-0	5	2	.400000000000000022
3	bj	1028	124672-0	5	2	.400000000000000022

继续修改数据，令 HOATT 字段为 0。

执行如下 SQL 语句：

1. `update "tbHandOver_Copy1"`
2. `set "HOATT" = 0, "HOSUCC" = 0`
3. `where "SCell" = '124711-0';`

打开 tbHandOver 表，发现数据已经成功更新，HOSUCCRATE 的值为空。

	CITY	SCell	NCELL	HOATT	HOSUCC	HOSUCCRATE
1	saxia	124711-0	124687-1	0	0	
2	saxia	124711-0	124687-2	0	0	
3	saxia	124711-0	124711-1	0	0	
4	saxia	124711-0	124711-2	0	0	
5	saxia	124711-0	15290-128	0	0	
6	saxia	124711-0	253932-0	0	0	

## 六、问题及解决

问题一：

在主键约束部分，首先出现了类似 column "124711-0" does not exist 问题，仔细检查并无语法问题。后来试着改成了单引号，便运行成功，查阅资料发现，PostgreSQL 会把双引号认为是“名称”。

试验几次，发现这里改成单引号就可以了~~

```
INSERT INTO user ( user_id, user_name )  
VALUES ( 1, 'Smart'); --> 这里
```

就可以成功插入了~想了想，原因可能是被双引号括起来的，PostgreSQL 都会认为是“名称”，如表名，字段名等~而被单引号括起来的就表示值了~

问题二：

在修改数据时，出现了如下 SET 问题：

```
1 update "tbHandOver_Copy"
2 set "SCELL" = '124711-0' and "NCELL" = '15290-128'
3 where "SCELL" = '124711-0' and "NCELL" = '124711-1';
```

SQL执行记录 消息

-----开始执行-----

【拆分SQL完成】：将执行SQL语句数量：（1条）

【执行SQL：（1）】

```
update "tbHandOver_Copy"
set "SCELL" = '124711-0' and "NCELL" = '15290-128'
where "SCELL" = '124711-0' and "NCELL" = '124711-1';
执行失败，失败原因： ERROR: invalid input syntax for type boolean: "124711-0"
Position: 41
Where: referenced column: SCELL
```

因为这本就是主键重复的试验，执行并不会成功，但我却困惑于失败原因，没有显示重复键相关的信息。后来查阅了相关资料，得知需要把 and 改为逗号，便出现了预期的结果。

You must separate fields in SET part with ", ", not with " AND "

Share Improve this answer Follow

answered Nov 21 '12 at 15:21



Ihor Romanchenko

24k ● 7 ● 44 ● 41

2 Common mistake indeed; using AND inside the SET part of an UPDATE query. Gives unusual errors like this. – pyrocumulul Nov 21 '12 at 15:25

mysql was the bad teacher :) though is a bit weird that we don't use "AND" after SET but we use after "WHERE" – themihai Nov 23 '12 at 18:39

4 @mihai It isnt weird, if you think about it. AND is an operator. Like +, -, or \*. In WHERE clause it is true AND false like 3 + 5, but in SET it becomes weird var1 = true AND var2 = 3. – Ihor Romanchenko Nov 23 '12 at 19:43

### 问题三：

在做触发器实验时，首先按照教材上的方法编写 SQL 语句，发现执行失败。后来得知，PostgreSQL 里触发器对每一行调用一个过程，需要添加 execute procedure 语句。这样依然执行失败，一直在 if 附近出现语法错误：

```

1 create trigger insert_trig_before before insert on "tbHandOver_Copy"
2   for each row
3   execute procedure
4   begin
5     if (new."HOATT" = 0)
6     then
7       set new."HOSUCCRATE" = null;
8     else
9       set new."HOSUCCRATE" = new."HOSUCC" / new."HOATT";
10    end if;
11  end;

```

SQL执行记录 消息

-----开始执行-----

【拆分SQL完成】：将执行SQL语句数量：（4条）

【执行SQL: (1)】

```

create trigger insert_trig_before before insert on "tbHandOver_Copy"
  for each row
  execute procedure
  begin
    if (new."HOATT" = 0)

```

```

    then
      set new."HOSUCCRATE" = null;
    else
      set new."HOSUCCRATE" = new."HOSUCC" / new."HOATT";
    end if;
  end;

```

执行失败，失败原因: ERROR: syntax error at or near "if"  
Position: 127

后来经过查阅 Stack Overflow 得知，在 PostgreSQL 里，if 一般需要放在触发器函数里：

64 IF and other PL/pgSQL features are only available inside PL/pgSQL functions. You need to wrap your code in a function if you want to use `IF`. If you're using 9.0+ then you can do use [DO](#) to write an inline function:

```

do $$
begin
  -- code goes here
end
$$

```

If you're using an earlier version of PostgreSQL then you'll have to write a named function which contains your code and then execute that function.

终于，等我编写了触发器函数之后，再通过定义触发器调用触发器函数，执行成功！  
此时的触发器函数代码：

```

1 create function insert_trig_func1() returns trigger as $$
2   begin
3     if new."HOATT" = 0 then
4       new."HOSUCCRATE" := null;
5     else
6       new."HOSUCCRATE" := new."HOSUCC" / new."HOATT";
7     end if;
8     insert into "tbHandOver_Copy" values (new."CITY", new."SCCELL", new."NCELL", new."HOATT", new."HOSUCC", new."HOSUCCRATE");
9     return null;
10  end;
11 $$ language plpgsql;

```

但不幸的事情又发生了，当我试图去插入一个数据时，出现了爆栈错误，即 Error: stack limit exceeded，我考虑在触发器函数调用过程中，可能发生了死循环。但思考了一会，仍然无法解决这个问题，在 Stack Overflow 上，我再次获得了解答，在手动调用 insert 时，当前函数会不断进行循环，将其替换为 return new 即可。

▲  
3 You `INSERT` again in an `AFTER INSERT` trigger, causing the trigger to be fired again for this second `INSERT` which again `INSERT` s and fires the trigger anew and so on and so on. At some point the stack is exhausted from all that function calls and you get the error.

▼  
✓ Remove the `INSERT` from the trigger functions and just `RETURN new`. Returning `new` will cause the original `INSERT` to be completed. There's no need for a manual `INSERT` in the trigger function for `AFTER INSERT` triggers.

🕒  
Like:

至此，再插入数据时，便得到了预期中的结果。