第六章 数据库管理

在这一部分我们将介绍 Greenplum 管理部分的内容。用户（DBA）在使用 Greenplum 中，会对数据库进行一些管理，例如创建多个用户登陆数据库、创建新的数据库等等。

5.1 用户和角色

5.1.1 角色、组、用户介绍

在 Greenplum 中，有 role，group，user 三个概念，要注意这里的概念和其他数据库中的概念略有不同。用户可以使用 create role、create group、create user 分别向它们中添加新的成员。事实上，这三个概念区别较为微妙，都被用作 Greenplum 的权限管理之中。

首先对三者的语法做一下介绍。无论是 create role 还是其他，语法都是一样的。在 create role 的 with 字句中，可以指定 role 的权限，权限分为以下几种：

1. 登录权限

只有具有 LOGIN 属性的角色才可以用于数据库连接，因此我们可以将具有该属性的角色视为登录用户，创建方法有如下两种：

```
CREATE ROLE name LOGIN PASSWORD '123456';
CREATE USER name PASSWORD '123456';
```

2. 超级用户

最高用户权限，不受资源队列控制，拥有所有的权限，可以对数据库进行任何操作。创建超级用户必须是以超级用户的身份执行以下命令：

CREATE ROLE name SUPERUSER;

3. 创建数据库

角色要想创建数据库，必须明确赋予 CREATEDB 的权限，见如下命令：

CREATE ROLE name CREATEDB;

4. 创建角色

一个角色要想创建更多角色，必须明确给予创建角色的权限，见如下命令：

CREATE ROLE name CREATEROLE;

5.创建外部表权限

属性配置中也可以对外部表有更细的权限控制，如只读，可写外部表权限等：

CREATE ROLE name CREATEEXTTABLE;

6.用户继承

子用户可以拥有父用户的所有权限：

CREATE ROLE name INHERIT;

7.资源队列控制

限制系统中的 SQL 对使用资源的消耗，如 CPU，内存等的消耗。

8.密码控制

CREATE ROLE name password ‘123456’ valid until ‘2016-6-31 00:00:00’;

5.1.2 三者的区别

这里对这三个概念进行一下讲解。role 包含 user 和 group。当用户创建一个 user 或者 group 之中，都会被添加到 role 的成员之中。如：

```
testdb=# create user test;
NOTICE:  resource queue required -- using default resource queue "pg_default"
CREATE ROLE
```

在这里我们创建了一个 user，user 是可以登录数据库的。

```
testdb=# select usename from pg_user;
 usename
---------
 gpadmin
  test
(2 rows)
```

在系统中表 pg_user 里面，我们可以看到刚刚创建的用户。

```
testdb=# select rolname from pg_roles;
 rolname
```

---

gpadmin

test

(2 rows)

而在系统表 pg_roles 中，也有我们刚才创建的 user 成员。这说明 user 是被包含在 role 中的。

注意：gpadmin 是系统初始化时创建的超级用户。

那么，在我们执行 create role，数据库是如何判断它是一个 user 还是一个 group？Greenplum 是这样判断呢的，如果我们给一个 role 以 LOGIN 权限，该 role 被添加到 user 组里，否则，用户默认没有登录权限，成员被添加到 group 组里。从个人的理解来看，user 和 group 的区别仅在于有无 LOGIN 权限，两者其他的属性都是类似的。

当我们执行 create user 命令时，默认有登录权限；同样，当执行 create group 命令时，默认没有登录权限。当然，我们可以在 create user 时指定 NOLOGIN，这样一条命令会创建一个 group 成员。create group 命令也类似。

5.1.3 在 Greenplum 中使用新用户

当创建一个 user 之后并不意味着我们就可以使用该 user 登录数据库，还需要在 pg_hba.conf 中对用户进行认证。配置文件在 masterdata/gpseg-1（master 数据根目录）下。认证的信息如下：

```
    TYPE    DATABASE        USER            CIDR-ADDRESS                MET
HOD
    local       all             gpadmin             ident
    host        all             gpadmin             127.0.0.1/28        trust
    host        all             gpadmin             192.168.80.129/32
  trust
    host        all             gpadmin             ::1/128             trust
    host        all             gpadmin             fe80::20c:29ff:feb5:9af6/
128             trust
    local       replication  gpadmin            ident
    host        replication  gpadmin            samenet             trust
    local       testdb          test                trust
```

每一行代表一条认证信息，可以理解为一张表，共有 5 个字段，从左到右依次为：登录类型、数据库权限、用户名、ip 掩码、验证方式。

登录类型定义了多种连接 Greenplum 的方式：

“local”使用本地 unix 套接字，

“host”使用 TCP/IP 连接（包括 SSL 和非 SSL），“host”结合“IPv4 地址”使用 IPv4 方式，结合“IPv6 地址”则使用 IPv6 方式，

“hostssl”只能使用 SSL TCP/IP 连接，

“hostnossl”不能使用 SSL TCP/IP 连接。

数据库权限限制了用户可以登录哪些数据库，需要注意的是 all 并不代表所有的数据库，仅仅代表除该条认证信息之上的数据库权限

条目的其他数据库。如：

local testdb test reject

local all gpadmin ident

此时 gpadmin 将不能登录 testdb 数据库。

第四个字段掩码在 TYPE 为 local 的时候将不用填写。

METHOD 指定如何处理客户端的认证，常用的有 ident，md5，password，trust，reject。说明如下：

1. ident 需要配置另一个文件 pg_ident.conf，这种认证方式要求在操作系统中有对应的用户；
2. md5 是密文验证方式；
3. password 是明文验证方式；
4. trust 方式登录不需要密码，建议不要在生产环境中使用；
5. reject 方式指定该用户不能登录数据库。

对创建的 test 用户的认证信息如下,添加到最后：

local testdb test md5 ‘123456’

5.1.4 权限与回收

当前登录用户可以对其他用户进行赋权，前提是该用户拥有该权限。赋权命令如下：

GRANT 权限类型 ON relation TO role;

具体用户可以使用\h GRANT 命令查看，这里对常用用法进行举例

说明。

testdb=# grant select on test01 to test; --给 test 用户赋予 test01 表的 select 权限

testdb=# grant all on test01 to test; --给 test 用户赋予 test01 表的 insert、select、update、delete 权限。

testdb=# grant create on database testdb to test; --给 test 用户赋予在 testdb 库创建模式的权限。

testdb=# grant test to gpadmin; --使 test 成为 gpadmin 的 member，一个角色拥有其成员的全部权限，回收成员的权限将使该角色失去权限。

可以通过\du 命令查看数据库中的所有用户：

```
testdb=> \du
                                   List of roles
 Role name |                         Attributes                         | Member of
-----------+------------------------------------------------------------+-----------
 gpadmin   | Superuser, Create role, Create DB                          | {test}
 test      |
```

回收权限的语法 revoke 与 grant 的使用类似。

revoke select on test01 from test;--回收 test 用户对 test01 表的 select 权限

注意：表的 owner 拥有该表的全部权限。

5.4 数据目录

现在我们来看一下 Greenplum 在操作系统文件系统上的组织方式，表 7-4 是 Greenplum 每个节点上的主要的数据目录：

项	描述
PG_VERSION	一个包含 PostgreSQL 主版本号的文件
base	包含每个数据库对应的子目录的子目录
global	包含集群范围的表的子目录，比如 pg_database
pg_clog	包含事务提交状态数据的子目录
项	**描述**
pg_multixact	包含多重事务（multi-transaction）状态数据的子目录（用于共享的行锁）
pg_subtrans	包含子事务状态数据的子目录
pg_tblspc	包含指向表空间的符号链接的子目录
pg_twophase	包含用于准备好事务状态文件的子目录
pg_xlog	包含 WAL （预写日志）文件的子目录
postmaster.opts	一个记录 postmaster 最后一次启动时使用的命令行参数的文件
postmaster.pid	一个锁文件，记录着当前的 postmaster PID 和共享内存段 ID （在 postmaster 关闭之后不存在

表 7-4 数据目录

5.4.1 数据库

在这里主要介绍一下 base 文件夹。首先查看一下 base 下的文件结构：

```
$ ls
1   12021   12022   24606
```

这里的数字都是蓝色表示，说明每一个数字都代表着一个文件夹。实际上，每一个文件夹代表一个数据库，文件夹的名字代表数据库的 oid，可以通过 pg_database 表查询对应的数据库：

```
testdb=# select oid,datname from pg_database;
  oid   |   datname
-------+-----------
 12022  | postgres
 24606  | testdb
     1  | template1
 12021  | template0
(4 rows)
```

5.4.2 表

接下来我们进入一个具体的数据库，以我们建立的 testdb 数据库为例：

$ ls

11764 1250 2620 …

这里的数字都是黑色字体，表示一个文件。在这里的文件可能是表，索引或者是序列。跟数据库一样，也是用数字标示文件名，不过这里的数字跟 relfilenode 有关。一般情况下 relfilenode 跟表的 oid 相同，但是对表的一些特殊的操作会改变表的 relfilenode。

当在数据库中创建一个表后，在所有的节点上都会创建一个数据文件，即使是 maste 节点也是如此，只是 maste 节点一般只会存储跟索引有关的数据。

如果是列存储的话，每个数据文件会被拆分成小文件。这里的命名方式是 oid+数字，数字以 128 为间隔递增。master 不会进行拆分，这里查看一下 segment 节点上的数据文件：

1） 在数据库中查看一张列表（为测试目的，有 2 个字段）的 relfilenode：

```
testdb=# select relfilenode from pg_class where relname = 'test_col';
 relfilenode
-------------
       90476
(1 row)
```

2） 在 linux 终端中查看有关的文件：

```
$ find *90476*
```

90476

90476.1

90476.129

5.4.3 表空间及其创建实例

类似于 Oracle，Greenplum 也有表空间（TableSpace）的概念，但是 Oracle 中表空间是对数据库的逻辑划分，而 Greenplum 是对数据库的物理划分（Greenplum 中 schema 是对数据库的逻辑划分）。表空间允许数据库管理员在文件系统里定义那些代表数据库对象的文件存放的位置。

表空间与文件空间（FileSpace）有关。一个文件空间可以有多个表空间，但是一个表空间只能属于一个文件空间，即一对多的关系。同样的，一个表文件也只能属于一个表空间。

Greenplum 默认的表空间是 pg_default，保存不同数据库的信息。这个表空间对应的文件目录是 base 目录。系统初始化的另外一个表空间是 pg_global，保存各个数据库的通用信息，对应的文件目录是 global 目录。这两个初始化创建的表空间都从属于 pg_system 文件空间。

如果数据库中的表很多，文件数就会很多，给文件系统带来很大压力，因此，在文件数增长到一定程度时就要使用表空间，将数据存放在多个目录下。下面演示在 Greenplum 中创建新的文件空间和表空间：

1） 在 master 节点和每个 segment 节点上创建文件目录，文件

目录分布如下：

master：/home/gpadmin/masterdata/fspc

primary segment: /home/gpadmin/segmentdata/fspc

mirror segment: /home/gpadmin/segdatamirror/fspc

2） 运行 gpfilespace 命令，输入文件系统的名字，以及每个节点上对应的目录，命令执行完之后会生成一个配置文件：

$ gpfilespace

3） 根据生成的文件，创建文件空间：

$ gpfilespace --config /home/gpadmin/segdatamirror/gpfilespace_config_20160726_154251

4） 创建好文件空间之后就可以在上面创建表空间以及表了：

testdb=# create tablespace tbs_test1 filespace fs_test;

CREATE TABLESPACE

testdb=# create table test_01(id int)

testdb-# tablespace tbs_test1 distributed by(id);

在 FileSpace 目录下，可以看到刚刚看到的表的文件以及其他信息：

$ tree

.

|-- gpseg0

| `-- 90555

| `-- 24606

| |-- 90567

```
|                `-- PG_VERSION
`-- gpseg1
    `-- 90555
        `-- 24606
            |-- 90567
            `-- PG_VERSION
6 directories, 4 files
```

默认的表空间是 pg_default，我们可以在当前窗口下使用如下命令修改配置，也可以在配置文件中修改这个参数：

```
set default_tablespace = ‘tbs_test1’;
```

使用下面的命令为每一个用户设置不同的表空间：

```
testdb=# alter role test set default_tablespace = 'tbs_test1';
ALTER ROLE
testdb=# grant ALL on tablespace tbs_test1 to test;
GRANT
```

使用下面的命令更换表所在的表空间（relfilenode 会发生变化）：

```
testdb=# alter table test_col set tablespace tbs_test1;
ALTER TABLE
```

5.5 备份机制

分布式数据库中，进行数据的备份也是提交数据库可靠性的有效手段，当有节点出现故障时，可以及时启用备份节点代替主节点提供

服务，从而对于用户来说并没有感觉到因为节点失效造成整个系统崩溃。

在 Greenplum 中也配置了镜像节点，当主节点不可用时自动切换到镜像节点，集群依然保持可用。当主节点恢复后，会自动恢复宕机期间的变更。只要 Master 节点连接不上 segment 节点就会在系统表中将此实例表示为不可用，并用镜像节点代替。Greenplum 提供两种备份策略，一种是 Grouped Mirror，一台 primary segment 对应的 Mirror Segment 同样放在同一台机器上，如图 3-2 所示

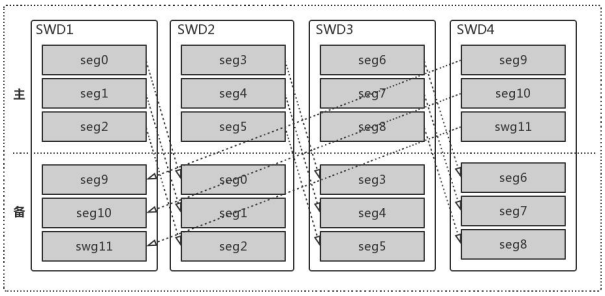

图 3-2 Grouped Mirror 备份策略

另外一种备份策略，则是将一台 primary segment 节点对应的所有节点按顺序打散到临接的机器上进行备份，如图 3-3 所示。

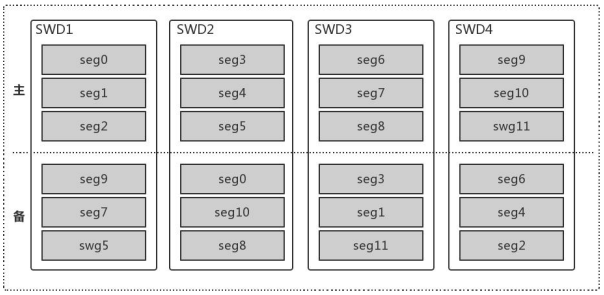

图 3-3 spread mirror 备份策略

两种备份策略各有优劣：

Grouped Mirror：假设 swd1 死机，那么 swd2 将会作为 primary 继续提供服务，这样 swd2 上会有 6 个 primary segment，由于 primary 的消耗要比 mirror 多很多,所以 swd2 容易成为系统的性能瓶颈。而 spread mirror 则可以将 sdw1 的 mirror segment 分不到另外 3 台机器上，这样就相当于剩下的 3 台机器有 4 个 primary，刚好做到负载均衡。

Spread Mirror：从图 10 中也可以看出，如果 sdw1 死机，那么这个时候系统不能再有任何一台机器死机，否则没有完整的 Primary Segment，整个数据库就会死机。而对于 Grouped Mirror，此时 swd3 再死机，系统仍然可以继续工作。

在一般情况下，在有限小组规模下，同时出现两台机器死机的情况比较小，因而采用 Spread Mirror 方式会更好一点。

有了备份机制，为了维护主备的一致性，需要引入同步机制，在最新的 Greenplum 中，主备同步采用文件同步的方式，并且同步是强一致性的，也就是不论当前业务是否繁忙，同步工作都会进行。除非 Primary 节点出现故障，否则备份节点一直处于一种非活动状态。当 Mirror 失败时，Primary 会记录这段时间的文件变更，等 mirror 恢复之后开始同步。当 primary 失败时，Mirror 会被唤醒代替 Primary，此时两者的角色会发生变化，系统会变为修改追踪模式，等 primary 恢复之后变为 mirror 节点，并将这段时间文件同步恢复。

5.6 数据库管理实验导引

5.6.1 数据库管理实战

1. 锁机制

在第一个实验中，我们分别打开两个终端，其中一个进行 select 操作，获取 access share lock，另一个进行 alter 操作，获取 access exclusive lock。

1）新建表 test_05，开启一个事务，进行一次 select 操作

```
testdb=# begin;
BEGIN
testdb=# select * from test_05;
 id
----
(0 rows)
```

2）另建一个终端，再次开启一个事务，进行一次 select 操作和 alt 操作

```
testdb=# begin;
BEGIN
testdb=# select * from test_05;
 id
----
(0 rows)

testdb=# alter table test_05 add column uname varchar(10);
```

可以发现，对于同种锁，并不一定冲突，这也符合表中所列的冲突情况。但是对于冲突的锁，如 ACCESS SHARE 和 ACCESS EXCLUSIVE，sql 语句一直在等待，直到第一个事务进行了提交操作。

3）获取 test_05 的 oid，然后在 pg_lock 里查询到锁信息

```
testdb=# select oid from pg_class where relname = 'test_05';
  oid
-------
 57462
(1 row)

testdb=# select locktype,database,relation,pid,mode from pg_locks where relation
='57462';
 locktype | database | relation | pid  |        mode
----------+----------+----------+------+---------------------
 relation |    24606 |    57462 | 3520 | AccessShareLock
 relation |    24606 |    57462 | 3520 | AccessExclusiveLock
 relation |    24606 |    57462 | 3530 | AccessShareLock
 relation |    24606 |    57462 | 3553 | AccessShareLock
 relation |    24606 |    57462 | 3523 | AccessShareLock
 relation |    24606 |    57462 | 3555 | AccessShareLock
 relation |    24606 |    57462 | 3525 | AccessShareLock
(7 rows)

testdb=# select datname,procpid,usename,current_query from pg_stat_activity wher
e procpid in (3520,3530,3553,3523,3555,3525);
 datname | procpid | usename |
  current_query
---------+---------+---------+-----------------------------------------------
---------------------------------------------------------------
 testdb  |    3520 | gpadmin | alter table test_05 add column uname varchar(10);
 testdb  |    3530 | gpadmin | select datname,procpid,usename,current_query from
 pg_stat_activity where procpid in (3520,3530,3553,3523,3555,3525);
```

4）下面对显式加锁进行试验

```
testdb=# lock test_05 in ROW SHARE mode;
LOCK TABLE
testdb=# select locktype,database,relation,pid,mode from pg_locks where relation='57462';
 locktype | database | relation | pid  |        mode
----------+----------+----------+------+---------------------
 relation |    24606 |    57462 | 3520 | AccessShareLock
 relation |    24606 |    57462 | 3520 | AccessExclusiveLock
 relation |    24606 |    57462 | 3530 | AccessShareLock
 relation |    24606 |    57462 | 3530 | RowShareLock
 relation |    24606 |    57462 | 3553 | AccessShareLock
 relation |    24606 |    57462 | 3553 | RowShareLock
 relation |    24606 |    57462 | 3523 | AccessShareLock
 relation |    24606 |    57462 | 3555 | AccessShareLock
 relation |    24606 |    57462 | 3555 | RowShareLock
 relation |    24606 |    57462 | 3525 | AccessShareLock
(10 rows)
```

在加锁之后再次查看锁信息，可以看到与原来相比多了 RowShar eLock

2. MVCC

下面我们来对 MVCC 的特性做一些实验：

1）首先创建一张表：

```
testdb=# create table test01(id int,name varchar(20)) distributed by(id);
CREATE TABLE
```

2）在事务①进行一次事务插入操作，暂时不进行提交：

```
testdb=# begin;
BEGIN
testdb=# insert into test01 values(1,'Mike');
INSERT 0 1
testdb=#
```

3）在事务②中设置事务隔离级别为缺省的 READ COMMITTED 级别，对表中的数据进行查询：

```
testdb=# begin;
BEGIN
testdb=# SET TRANSACTION ISOLATION LEVEL READ COMMITTED;
SET
testdb=# select * from test01;
 id | name
----+------
(0 rows)
```

4）将事务①进行提交：

```
testdb=# begin;
BEGIN
testdb=# insert into test01 values(1,'Mike');
INSERT 0 1
testdb=#  commit;
COMMIT
```

5）在事务②再次进行一次查询：

```
testdb=# begin;
BEGIN
testdb=# SET TRANSACTION ISOLATION LEVEL READ COMMITTED;
SET
testdb=# select * from test01;
 id | name
----+------
(0 rows)

testdb=# select * from test01;
 id | name
----+------
  1 | Mike
(1 row)
```

可以发现，事务②中进行的两次查询出现了不同的结果，出现了幻读。这是由于我们设定的隔离级别造成的。数据库会根据我们设定的隔离级别调用不同的快照，用于不同形式的可见性判断。

3. 角色

在本次实验中，分别创建了 undergraduate 和 class1 两个角色，并将其加入到配置文件中。由于将 class1 置为 undergraduate 的 member，并给 class1 赋上 t1 表的 select 权限，那么 undergraduate 应该会自动拥有同样的权限。我们在实验中进行了验证。

1） 创建 undergrade 和 class1 角色

```
testdb=# create role undergraduate LOGIN password '123456';
NOTICE:  resource queue required -- using default resource queue "pg_default"
CREATE ROLE
testdb=# create role class1 LOGIN password '123456';
NOTICE:  resource queue required -- using default resource queue "pg_default"
CREATE ROLE
```

2） 在 pg_hba.conf 加上新的 role 的信息

```
local    testdb      undergraduate            trust
local    testdb      class1            trust
```

3） 将 class 1 设为 undergraduate 的 member

```
testdb=# grant class1 to undergraduate;
GRANT ROLE
```

4） 给 class1 赋上 select on t1 的权限（t1 是建立的一张临时表）

```
testdb=# grant select on t1 to class1;
GRANT
```

5） 使用 undergraduate 登录数据库，查看 t1 表（需要重启数据库使修改生效）

```
[gpadmin@localhost ~]$ psql -U undergraduate
psql (8.3devel)
Type "help" for help.

testdb=> select * from t1 where id = 1;
 id |         name
----+----------------------
  1 | test
(1 row)
```

从结果可以看出，undergraduate 是拥有 t1 表的 select 权限的。这说明一个成员是拥有它的子成员的全部权限的。

5.6.2 数据库管理实验要求

1、使用 create role 命令创建一个拥有创建数据库、创建用户、登录权限， 且有效期到 2016 年 12 月 19 日， 密码为‘test’的用户； 使用 grant 和 revoke 命令添加和回收权限；

2、申请锁，通过 pg_locks 系统表查询表的锁信息

3、 查看 Greenplum 的数据目录结构，理清楚 base、 global、 pg_log、 pg_clog、 pg_xlog 等目录存储的内容。

4、了解 greenplum 中的 toast 机制，通过不断增大字段占用空间，探索在何种情况下会触发 toast 机制，生成 toast 表，是否有 toast 表可以从 pg_class 中进行查询。