GREENPLUM
DATABASE®

# 揭秘Greenplum恢复（Recovery）系统

钉钉直播 | 1月29日 16：00 - 17：00

GREENPLUM DATABASE®

# Greenplum中文社区

https://cn.greenplum.org

博文 · 资料 · 文档 · 项目

GREENPLUM DATABASE®

# 全新的问答论坛

https://cn.greenplum.org/askgp

GREENPLUM DATABASE®

微信技术讨论群
添加入群小助手：gp_assistant

微信公众号
技术干货、行业热点、活动预告

欢迎访问Greenplum中文社区：cn.greenplum.org

GREENPLUM DATABASE

# 提纲

- 恢复系统概述
- 预写式日志简介
- 单节点系统恢复
- 分布式系统恢复

GREENPLUM DATABASE®

GREENPLUM DATABASE

# 常见故障：

- 硬件问题（异常断电，磁盘访问，网络等）
- 系统问题（内核崩溃，库函数bug等）
- Postgres进程异常退出（比如OOM， bug，保护性PANIC) –> Postgres重启

# 数据库恢复系统

GREENPLUM DATABASE

- Postgres进程异常退出重启后集群的恢复。
- 基于WAL (预写式日志):事务先写日志，其他数据修改通常可以滞后，事务提交时日志同步到从节点后才成功。
- 单节点上:恢复系统通常从Checkpoint日志中redo值所代表的位置开始重放日志。
- 分布式:Master充当分布式事务管理器(也负责分布式恢复)保证整个集群的数据全局一致性。
- 必要时候需要人工干预
- 后面内容除非特别说明都是基于Greenplum 6

GREENPLUM DATABASE®

# 一个简单Insert的预写式日志例子

GREENPLUM DATABASE

postgres=# insert into t1 select generate_series(1,10);
INSERT 0 10

以完成时间顺序看这个两阶段事务日志写入情况（下为pg_xlogdump解析日志输出结果）

- 第一个Primary节点（其余 Primary节点类似）

```
rmgr: Heap        len (rec/tot):     31/    63, tx:        721, lsn: 0/0C0F02A0, prev 0/0C0F0238, bkp: 0000, desc: insert: rel 1663/12812/16391; tid 0/6
rmgr: Heap        len (rec/tot):     31/    63, tx:        721, lsn: 0/0C0F02E0, prev 0/0C0F02A0, bkp: 0000, desc: insert: rel 1663/12812/16391; tid 0/7
rmgr: Heap        len (rec/tot):     31/    63, tx:        721, lsn: 0/0C0F0320, prev 0/0C0F02E0, bkp: 0000, desc: insert: rel 1663/12812/16391; tid 0/8
rmgr: Heap        len (rec/tot):     31/    63, tx:        721, lsn: 0/0C0F0360, prev 0/0C0F0320, bkp: 0000, desc: insert: rel 1663/12812/16391; tid 0/9
rmgr: Heap        len (rec/tot):     31/    63, tx:        721, lsn: 0/0C0F03A0, prev 0/0C0F0360, bkp: 0000, desc: insert: rel 1663/12812/16391; tid 0/10
rmgr: Transaction len (rec/tot):    376/   408, tx:        721, lsn: 0/0C0F03E0, prev 0/0C0F03A0, bkp: 0000, desc: prepare
```

- Master节点

```
rmgr: Transaction len (rec/tot):     84/   116, tx:          0, lsn: 0/0C0FC9C8, prev 0/0C0FC988, bkp: 0000, desc: distributed commit 2021-01-12
16:29:12.085955 CST gid = 4129264608-0000032766 gid = 1610432654-0000000012, gxid = 12
```

- 第一个Primary节点（其余priamry节点类似）

```
rmgr: Transaction len (rec/tot):     72/   104, tx:          0, lsn: 0/0C0F0578, prev 0/0C0F03E0, bkp: 0000, desc: commit prepared 721: 2021-01-12
16:29:12.086621 CST gid = 139787424-0000000000 gid = 1610432654-0000000012 gxid = 12
```

- Master 节点

```
rmgr: Transaction len (rec/tot):     28/    60, tx:          0, lsn: 0/0C0FCA40, prev 0/0C0FC9C8, bkp: 0000, desc: distributed forget  gid =
1610432654-0000000012, gxid = 12
```

GREENPLUM DATABASE

# 思考：

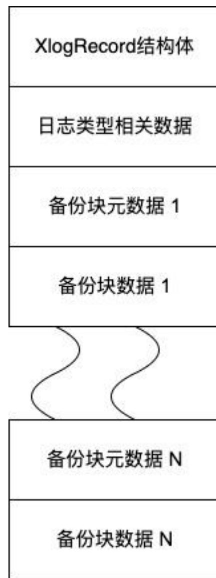
- 为什么master先做distributed commit再发起segment上两阶段提交？
- PREPARE日志需要同步到从端才告诉master PREPARE成功吗？
- 为什么有一个distributed forget日志? 需要吗？

  *Tip: 从数据库恢复的角度考虑, 如果没有这么做, 某一步出错了, 系统应该如何处理？*

# 预写式日志

- 通常:事务提交前写日志,提交时候刷盘日志。
- Greenplum基于redo日志。
- 系统自动删除/回收无用的日志文件(主动或者被动做checkpoint的时候)。
- 应用于:单机恢复以及流式复制(主从方案高可用)。
- Greenplum日志
  - 64MB大小一个日志文件。
  - 每个文件内部以32K大小为一个日志页面(不要和Buffer页面大小混淆)。
  - 日志位置以LSN (Log Sequence Number)记录位置 (例如:0/57920DE8 实际是一个64bit数)。
  - 日志文件内容除了日志还包含一些元数据(比如日志页面头信息)。

# 预写式日志格式




备份块数据：Full Page Write相关（后面会介绍到）。

# 日志头信息

GREENPLUM DATABASE®

| 类型 | 字段 | 说明 |
| --- | --- | --- |
| uint32 | xl_tot_len | 整个日志数据的长度（包括header） |
| TransactionId | xl_xid | 和日志相关的本地事务id（注意不是每个日志产生都会有xid。比如xlog switch日志通常就不会有xid） |
| uint32 | xl_len | 日志类型相关数据的总长度（不包含header以及full page write写的页面也即备份块数据） |
| uint8 | xl_info | 日志info |
| RmgrId | xl_rmid | 资源管理id |
| XLogRecPtr | xl_prev | 上一个日志的RecPtr。这一项一可用于日志头检查ValidXLogRecordHeader()，二是有时候需要查找一些前面的日志，比如一个checkpoint日志。 |
| pg_crc32 | xl_crc | 所有数据的crc校验。 |

# 预写式日志文件

```
drwx------ 2 pguo        6 Mar 15 22:30 archive_status
-rw------- 1 pguo 67108864 Mar 15 22:35 000000010000000000000002
-rw------- 1 pguo 67108864 Mar 15 22:35 000000010000000000000003
-rw------- 1 pguo       41 Mar 15 22:35 00000002.history
-rw------- 1 pguo 67108864 Mar 15 22:36 000000020000000000000003
-rw------- 1 pguo       83 Mar 15 22:37 00000003.history
-rw------- 1 pguo 67108864 Mar 15 22:44 000000030000000000000003
-rw------- 1 pguo 67108864 Mar 15 22:45 000000030000000000000004
```

History文件：Greenplum上记录了每次TimeLine ID切换时候的LSN信息。

TimeLine ID: 记录各种“分叉”，比如主从切 换的时候。

长数字串文件是日志文件。

日志文件名格式：

GREENPLUM DATABASE

# 预写式日志操作

XLogInsert(): 日志数据写入到日志buffer

XLogFlush(): 刷盘日志buffer到存储

Greenplum在segment上事务提交使用同步复制策略 ：SyncRepWaitForLSN()

Background wal write进程：周期性地刷盘日志buffer到存储
pguo 66053 66038 0 14:24 ? 00:00:00 postgres: 6003, wal writer process

# Greenplum支持的资源管理列表
## (src/include/access/ rmgrlist.h)

```
PG_RMGR(RM_XLOG_ID, "XLOG", xlog_redo, xlog_desc, NULL, NULL)
PG_RMGR(RM_XACT_ID, "Transaction", xact_redo, xact_desc, NULL, NULL)
PG_RMGR(RM_SMGR_ID, "Storage", smgr_redo, smgr_desc, NULL, NULL)
PG_RMGR(RM_CLOG_ID, "CLOG", clog_redo, clog_desc, NULL, NULL)
PG_RMGR(RM_DBASE_ID, "Database", dbase_redo, dbase_desc, NULL, NULL)
PG_RMGR(RM_TBLSPC_ID, "Tablespace", tblspc_redo, tblspc_desc, NULL, NULL)
PG_RMGR(RM_MULTIXACT_ID, "MultiXact", multixact_redo, multixact_desc, NULL, NULL)
PG_RMGR(RM_RELMAP_ID, "RelMap", relmap_redo, relmap_desc, NULL, NULL)
PG_RMGR(RM_STANDBY_ID, "Standby", standby_redo, standby_desc, NULL, NULL)
PG_RMGR(RM_HEAP2_ID, "Heap2", heap2_redo, heap2_desc, NULL, NULL)
PG_RMGR(RM_HEAP_ID, "Heap", heap_redo, heap_desc, NULL, NULL)
PG_RMGR(RM_BTREE_ID, "Btree", btree_redo, btree_desc, NULL, NULL)
PG_RMGR(RM_HASH_ID, "Hash", hash_redo, hash_desc, NULL, NULL)
PG_RMGR(RM_GIN_ID, "Gin", gin_redo, gin_desc, gin_xlog_startup, gin_xlog_cleanup)
PG_RMGR(RM_GIST_ID, "Gist", gist_redo, gist_desc, gist_xlog_startup, gist_xlog_cleanup)
PG_RMGR(RM_SEQ_ID, "Sequence", seq_redo, seq_desc, NULL, NULL)
PG_RMGR(RM_SPGIST_ID, "SPGist", spg_redo, spg_desc, spg_xlog_startup, spg_xlog_cleanup)
PG_RMGR(RM_BITMAP_ID, "Bitmap", bitmap_redo, bitmap_desc, NULL, NULL)
PG_RMGR(RM_DISTRIBUTEDLOG_ID, "DistributedLog", DistributedLog_redo, DistributedLog_desc, NULL, NULL)
PG_RMGR(RM_APPEND_ONLY_ID, "Appendonly", appendonly_redo, appendonly_desc, NULL, NULL)
```

GREENPLUM DATABASE®

# 单节点恢复 (Postgres进程异常退出)

- 确保单节点上数据库“正确”工作（事务提交或者回滚的事务修改能确保落地，不影响数据 库一致性）。
- Postmaster退出：外部组件重新启动单点服务。比如FTS自动标记节点为’d’后通过gprecoverseg完成。
- Postgres后端进程异常退出，Postmaster自动重启，segment上如果重启足够快FTS不会标记这个节点为’d’。否则gprecoverseg恢复。
- 不论如何，postgres重启后，恢复系统会尽力先启动以确保数据库在正确状态，方便gprecoverseg增量恢复。
- 增量恢复失败（不一定是产品bug）可以改用全量恢复 (gprecoverseg –F，慢，但是基本不会失败)
- 全量恢复失败：<br>仔细看清原因，修复，绕过，必要时候找对应的商业支持（硬件、OS、数据库）。

GREENPLUM DATABASE

# Master/standby

Master/standby：

- gpdb 7开始支持master (master改叫coordinator）节点自动切换。
- gpdb6或者以前版本用外部工具管理切换和激活standby。
- 也可以用相关工具重建一个新的standby。

# 单节点恢复过程实现：

GREENPLUM DATABASE

Startup进程是恢复进程。

Postmaster最早启动（除了logger进程）startup进程，不允许libpq连接（通常模式或者utility mode）。

pguo 67006 66996 0 14:24 ? 00:00:00 postgres: 6007, startup process recovering 000000010000000000000003

依赖：pg_control文件 + 日志文件

核心函数：StartupXLOG()

- **基本原理是从一个日志起点循环读取日志和重放。**

  典型的：读取pg_control文件中的checkpoint LSN，在根据这个值在日志中读取checkpoint日志的redo值作为重放起点。

- **循环什么时候退出？**

  对主(master/primary)节点来说做完所有日志重放就可以结束循环最后startup进程退出。
  对从节点循环不会退出，一直等待主节点的日志流数据传输，除非被promote（主从切换）或者出错。

- **整个函数还有大量各种细节分支不赘述：**比如：unlogged表truncate，replication slot处理等。
- **包含Greenplum特有的分布式相关日志的处理。**
- **startup进程出错:** 一般会比较复杂，需要小心处理。如果其对应的主或对应的从节点数据库没问题，即使这个节点有问题也可以gprecoverseg恢复。
- **慎用pg_resetxlog工具!!!**

# 单节点恢复过程(续)

GREENPLUM DATABASE

怎么定义重放循环结束:

- 日志格式检查发现已无合法日志(参见日志头数据结构), 比如在主节点。
- 如果日志来源是在进行中的流复制, 循环一直不结束, 比如从节点。直到出错或者从主切换, 跳出循环, 从变成主。

Q: 怎么定义mirror节点? Data目录下文件recovery.conf

$ cat recovery.conf
standby_mode = 'on'
primary_conninfo = 'user=pguo host=host67 port=6002 sslmode=prefer sslcompression=1 krbsrvname=postgres application_name=gp_walreceiver'
primary_slot_name = 'internal_wal_replication_slot‘

# Full Page Write

GREENPLUM DATABASE

机器断电或者内核崩溃可能会造成数据 页面包含了部分修改后的新数据。

重放日志操作数据 页面依赖于文件上数据 页面正确性（比如数据 页面checksum不对页面装载会失败）。

如果一个日志是 Checkpoint.redo后第一次修改了某个 页面的日志，先把老的整个 页面存一份在日志中，确保checkpoint.redo位置开始的重放一定能基于正确的数据 页面。

相关判断逻辑：

```
page = BufferGetPage(rdata->buffer);

/*
* We assume page LSN is first data on *every* page that can be passed to
* XLogInsert, whether it has the standard page layout or not. We don't
* need to take the buffer header lock for PageGetLSN if we hold an
* exclusive lock on the page and/or the relation.
*/
if (holdsExclusiveLock)
*lsn = PageGetLSN(page);
else
*lsn = BufferGetLSNAtomic(rdata->buffer);

if (*lsn <= RedoRecPtr)
{
/* 页面需要被backup */
```

# 日志回收和删除

GREENPLUM DATABASE

Checkpoint时会回收或者删除老旧日志文件。保留日志文件原则:

- 不能删除Checkpoint.redo LSN后的日志。
- 最老的Prepared但是还没有Commit/Abort的位置开始的日志不能删。
- Replication Slot上记录的日志最新同步位置之后的日志不能删。
- guc wal_keep_segments (缺省5) 条件要满足。
- guc max_slot_wal_keep_size_mb (缺省不限制. 6.10.0及以上)

慎用pg_resetxlog工具!

# gprecoverseg

- Python应用用于集群节点自动恢复
- 并发多节点恢复
- 增量恢复基于pg_rewind工具，全量恢复基于pg_basebackup工具
- 缺省增量恢复，如果失败可以再用-F做全量恢复。前提：
    1. primary/mirror有一个是好的，
    2. 有问题的节点系统没问题（比如不能是文件系统坏了，磁盘坏了，操作系统启动不起来了等）。

GREENPLUM DATABASE

GREENPLUM DATABASE

# 分布式恢复

- 分布式数据库：需要分布式事务以保证全局可见性。（Query在不同节点上事务启动顺序可能不同）
- 事务提交：广泛使用两阶段提交协议（一阶段提交先不表）。
- 两阶段：
  - 第一阶段完成事务写后发起PREPARE请求并完成，代表如果事务最后如果提交肯定和必须得成功。
  - PREPARE完成：日志落盘并同步到从节点（否则发生failover后master发起PREPARE提交会失败。）
  - 第二阶段提交：有一个节点提交失败，其他节点提交成功。如果不干预，对可见性影响。
  - 比如不能insert 1000个数据报告客户端成功，但是最后查询只能返回900个数据。
  - 分布式恢复：主要指master（负责全局分布式事务管理）的dtx recovery进程，确保两阶段提交或者回滚能在所有受影响节点“落地”。
- 分布式恢复的前提是所有segment的节点都已经能连接访问。
- Greenplum：不支持显示的PREPARE等Postgres上两阶段相关的Query命令，虽然内部实现还是使用了其相关代码。不要utility mode运行这些命令！

GREENPLUM DATABASE

# Greenplum分布式事务相关日志处理

- Checkpoint日志本身会记录还没有处理(是指日志重放)完毕的分布式事务相关的元数据(包括master和segment节点上的相关日志)

- Greenplum 6早期版本有一些bug会导致某些场景下各种问题，例如
  - 因为orphaned事务导致日志文件累积无法被自动删除或者回收
  - requested WAL segment *** has already been removed
  - Primary的startup进程hang
  - Standby报overflow的PANIC

这些问题已经被修复。请至少升级到Greenplum 6.12.1或者更高版本。多关注新版本的release note.

# Greenplum分布式事务相关恢复



Primary如果在两阶段提交 /回滚的阶段时候无法完成，master会重试两阶段提交/回滚事务，等候Primary节点恢复或者主从切换，如果重试超过一定次数：

- 如果是两阶段提交，会PANIC。
- 如果是两阶段回滚，会转交dtx recovery进程继续尝试（避免了PANIC）

为什么两阶段回滚不需要PANIC?

- 早期实现其实是直接PANIC。
- PANIC不友好（所有连接断开，单机恢复也需要时间影响服务可用性）
- 两阶段回滚失败不处理坏处：
  - 占用两阶段PREPARE后的资源（锁等），但不影响数据全局一致性。
- 转交dtx recovery进程继续回滚。

# 复习前面提到的两阶段提交日志类型

GREENPLUM DATABASE

```
postgres=# insert into t1 select generate_series(1,10);
INSERT 0 10
```

- 第一个Primary节点（其他Primary节点类似）

```
rmgr: Heap        len (rec/tot):     31/    63, tx:        721, lsn: 0/0C0F02A0, prev 0/0C0F0238, bkp: 0000, desc: insert: rel 1663/12812/16391; tid 0/6
rmgr: Heap        len (rec/tot):     31/    63, tx:        721, lsn: 0/0C0F02E0, prev 0/0C0F02A0, bkp: 0000, desc: insert: rel 1663/12812/16391; tid 0/7
rmgr: Heap        len (rec/tot):     31/    63, tx:        721, lsn: 0/0C0F0320, prev 0/0C0F02E0, bkp: 0000, desc: insert: rel 1663/12812/16391; tid 0/8
rmgr: Heap        len (rec/tot):     31/    63, tx:        721, lsn: 0/0C0F0360, prev 0/0C0F0320, bkp: 0000, desc: insert: rel 1663/12812/16391; tid 0/9
rmgr: Heap        len (rec/tot):     31/    63, tx:        721, lsn: 0/0C0F03A0, prev 0/0C0F0360, bkp: 0000, desc: insert: rel 1663/12812/16391; tid 0/10
rmgr: Transaction len (rec/tot):    376/   408, tx:        721, lsn: 0/0C0F03E0, prev 0/0C0F03A0, bkp: 0000, desc: prepare
```

- Master节点

```
rmgr: Transaction len (rec/tot):     84/   116, tx:          0, lsn: 0/0C0FC9C8, prev 0/0C0FC988, bkp: 0000, desc: distributed commit 2021-01-12
16:29:12.085955 CST gid = 4129264608-0000032766 gid = 1610432654-0000000012, gxid = 12
```

- 第一个Primary节点（其他Primary节点类似）

```
rmgr: Transaction len (rec/tot):     72/   104, tx:          0, lsn: 0/0C0F0578, prev 0/0C0F03E0, bkp: 0000, desc: commit prepared 721: 2021-01-12
16:29:12.086621 CST gid = 139787424-0000000000 gid = 1610432654-0000000012 gxid = 12
```

- Master 节点

```
rmgr: Transaction len (rec/tot):     28/    60, tx:          0, lsn: 0/0C0FCA40, prev 0/0C0FC9C8, bkp: 0000, desc: distributed forget  gid =
1610432654-0000000012, gxid = 12
```

# Master dtx recovery进程

- Master启动：先自身单节点恢复过程(startup process)，其后启动dtx recovery进程，其功能如下：
  - 如果检测到master上有两阶段事务只是DISTRIBUTED_COMMIT但是还没有DISTRIBUTED_FORGET，在segment上发起两阶段提交请求。
  - 为什么master先提交本地DISTRIBUTED_COMMIT事务再在segment上发起两阶段提交，作用之一在于此，另外一个原因是如果这个两阶段提交还包括master修改，master不需要做PREPARE直接提交事务。
  - 写入DISTRIBUTED_FORGET日志作用也在于此，这个日志当时写入时候不调用XLogFlush()刷盘（想想为什么？）所以代价不大。
- 如果在segment上 (通过相关的UDF)检测到还有Prepared的事务但是还没有commit/abort的（所谓orphaned两阶段事务)，对这种事务发起两阶段回滚请求。
- 这些步骤完成后master才会接受新的客户端连接。
- Greenplum6上对上述第二种场景会进一步周期性的检查（因为这种残留事务影响大比如日志文件累积无法删除，资源占用
- orphaned两阶段事务发生的场景：
  - master/standby切换或者master异常重启可能会有（虽然发生概率不那么高）。
  - 前面提到的两阶段回滚（Abort Prepared）重试失败了，dtx recovery需要继续尝试回滚这些orphan事务。
  - 产品bug。
- Dtx recovery进程会一直存在，不会退出。

GREENPLUM DATABASE

# 准备工作

# 从源代码开始：下载编译Greenplum源代码

greenplum-db/gpdb: Greenplu ×

github.com/greenplum-db/gpdb

应用 Pivotal Network :: pulse :: projects 已导入 GPDB-Postgres-... dev:mm_gpexpan... CMake PostgresMergeDe... postgres_merge -... 修行 xkcd: Nighttime St...

Learn Git and GitHub without any code!

Using the Hello World guide, you'll start a branch, write comments, and open a pull request.

Read the guide

greenplum-db / gpdb

Unwatch 432 | Star 3.9k | Fork 1.1k

Code | Issues 308 | Pull requests 104 | Actions | Projects 6 | Wiki | Security | Insights | Settings

Greenplum Database http://greenplum.org

Edit

Manage topics

55,652 commits | 33 branches | 0 packages | 85 releases | 223 contributors | View license

Branch: master | New pull request | Create new file | Upload files | Find file | Clone or download

dh-cloud Fix a bug when setting DistributedLogShared->oldestXmin ... Latest commit 2759b6d yesterday

| | | |
|---|---|---|
| .github | CONTRIBUTING.md: Add guidelines to run pgindent | 3 years ago |
| concourse | Increase instance memory for gpexpand tests | 6 days ago |
| config | Remove ORCA specific configure checks | 6 days ago |
| contrib | Enable external table's error log to be persistent for ETL. (#9757) | 25 days ago |

GREENPLUM
DATABASE®

微信技术讨论群
添加入群小助手：gp_assistant

微信公众号
技术干货、行业热点、活动预告

欢迎访问Greenplum中文社区：cn.greenplum.org