HOW2025

IvorySQL 2025生态大会
暨PostgreSQL高峰论坛

PostgreSQL的逻辑复制
spill溢出案例和启停库逻辑

目录
CONTENTS

HOW2025

IvorySQL 2025生态大会
暨PostgreSQL高峰论坛

walsender、archiver
阻止停库，
如何优雅停库

walsender阻止停库

walsender阻止停库时，数据库的停库状态：

```
    1 110402 110402 110402 ?           -1 Ss    6001   0:00 postgres -D /myhost/pg/data
110402 110599 110599 110599 ?           -1 Ss    6001   0:00  \_ postgres: lzlpg: logger
110402 117803 117803 117803 ?           -1 Ss    6001   0:00  \_ postgres: lzlpg: checkpointer
110402 117807 117807 117807 ?           -1 Ss    6001   0:00  \_ postgres: lzlpg: stats collector
110402 118563 118563 118563 ?           -1 Rs    6001   3:29  \_ postgres: lzlpg: walsender lzl
127.0.0.1(62971) idle
110402 222918 222918 222918 ?           -1 Rs    6001   2:59  \_ postgres: lzlpg: walsender logicaluser
30.181.46.203(57218) idle
```

此时的控制文件：

```
 pg_controldata|grep -i state
Database cluster state:               in production
```

walsender阻止停库

```
pstack 117803
#0  0x00002b879fe0b983 in __select_nocancel () from /lib64/libc.so.6
#1  0x00000000008fd04a in pg_usleep (microsec=microsec@entry=10000) at pgsleep.c:56
#2  0x00000000007610c8 in WalSndWaitStopping () at walsender.c:3209
#3  0x000000000051fa86 in ShutdownXLOG (code=code@entry=0, arg=arg@entry=0) at xlog.c:8596
#4  0x00000000007215ff in HandleCheckpointerInterrupts () at checkpointer.c:566
#5  CheckpointerMain () at checkpointer.c:343
...
```

此时的checkpointer堵在WalSndWaitStopping函数，代表checkpointer在等待walsender进程进入stopping状态

此时库是停一半的状态，如果强行kill -9，会导致数据库是**非一致性停库**

怎么优雅的停库

walsender阻止停库该怎么办?

- 方案1：关闭下游进程
- 方案2：发送SIGTERM给walsender

怎么优雅的停库

1.关闭下游进程

```
static void
ProcessRepliesIfAny(void)
{...
                                        /*
                                         * 'X' means that the standby is closing down the socket.
                                         */
                                case 'X':
                                        proc_exit(0);
```

1.alter SUBSCRIPTION sub_lzl disable;---需要提前找到所有关联的下游PG库
2.停同步工具----同步工具可能无法及时维护

怎么优雅的停库

pg_terminate_backend()本质上
就是在发送SIGTERM给子进程

2.发送SIGTERM给walsender

running状态：
select pg_terminate_backend($walsender_pid)

停一半的状态：
kill -SIGTERM $walsender_pid
#同 kill -15 $walsender_pid
#同 kill $walsender_pid

archiver阻止停库

除了walsender外，常见的还有archiver进程也可能阻止停库。

reaper checkpointer会发送SIGUSR2给archiver让其最后一次归档并退出：

```
static void
reaper(SIGNAL_ARGS)
{...
                        if (pid == CheckpointerPID)
                        {
                                CheckpointerPID = 0;
                                if (EXIT_STATUS_0(exitstatus) && pmState == PM_SHUTDOWN)
                                {...

                                        /* Waken archiver for the last time */
                                        if (PgArchPID != 0)
                                                signal_child(PgArchPID, SIGUSR2);
        ...
                                }
...
```

archiver阻止停库

pm也依赖归档进程的退出：

```
if (pmState == PM_WAIT_DEAD_END)
{
        if (dlist_is_empty(&BackendList) &&
                PgArchPID == 0 && PgStatPID == 0)
        {
                /* These other guys should be dead already */
                Assert(StartupPID == 0);
                Assert(WalReceiverPID == 0);
                Assert(BgWriterPID == 0);
                Assert(CheckpointerPID == 0);
                Assert(WalWriterPID == 0);
                Assert(AutoVacPID == 0);
                /* syslogger is not considered here */
                pmState = PM_NO_CHILDREN;
        }
}
```

archiver阻止停库

pm也依赖归档进程的退出：

```
$  ps -axjf|grep  61470
    1  61470  61470  61470 ?           -1 Ss    6001   0:00 /myhost/bin/postgres -D /data
 61470  61772  61772  61772 ?          -1 Ss    6001   0:00  \_ postgres: lzlpg: logger
 61470  63880  63880  63880 ?          -1 Ss    6001   0:00  \_ postgres: lzlpg: archiver archiving
000000010000018800000007
```

可能的原因：归档延迟较多，归档盘写入较慢
不可能的原因：归档失败，NUM_ARCHIVE_RETRIES限制

此时暴力停库是否有问题?
如果只是归档进程阻止停库，那么checkpointer已经停止，shutdown checkpoint条目已写入WAL，controldata的状态也为shut down，即表示一致性停库。所以此时哪怕archiver在running，kill -9对数据库本身没有影响

怎么优雅的停库

在停库的时候，可以添加的动作：

1. 锁定逻辑同步用户
2. pg_terminate_backend($logical_walsender)
3. 临时关闭归档（可选,archive_command置空)
4. 手动checkpoint
5. stop fast
6. 如果仅archiver阻止停库，可以考虑暴力停库

PostgreSQL停库逻辑

信号

linux可以通过信号进行进程通信，并且定义了很多信号。
PG常用的信号：

- -1 或 -SIGHUP：挂起信号，在PG中通常是通知进程重新加载配置。
- -2 或 -SIGINT：中断信号（通常是Ctrl+C），在pg中通常对应取消命令。
- -3 或 -SIGQUIT：pg中通常是强制退出die
- -9 或 -SIGKILL：无条件终止信号。
- -15或 -SIGTERM：终止信号，pg_terminate_backend使用的信号，pg中通常是合理退出
- -10或 -SIGUSR1：自定义信号
- -12或 -SIGUSR2：自定义信号
- -17或 SIGCHLD：pm进程使用的信号，一般是子进程退出后pm接受该信号触发回收子进程任务

信号

pg_ctl通过信号管理停库方式：

signal	pg_ctl	含义
`SIGTERM`	*Smart Shutdown*	不允许新的连接，但允许现有会话正常结束其工作。只有在所有会话终止后，它才会关闭
`SIGINT`	*Fast Shutdown*	服务器不允许新的连接，并向所有现有的子进程发送**SIGTERM**，中止当前的事务并迅速退出。等待几乎所有子进程（有几个子进程不需要）退出，最后关闭
`SIGQUIT`	*Immediate Shutdown*	将向所有子进程发送**SIGQUIT**，并等待它们终止。如果在5秒内有子进程没有终止，它们将被发送**SIGKILL**

KILL -9和shutdown -m i是不同的
pg_ctl没有发送SIGKILL（kill -9）的参数，但其实是可以直接对pm发送SIGKILL的，但肯定不推荐这么做，SIGKILL pm时pm不会对子进程、共享内存、信号做任何处理。因为SIGQUIT pm有兜底逻辑做子进程的SIGKILL，所以SIGQUIT pm基本能保证pm能停下来。

信号

pm注册的信号：

```
void
PostmasterMain(int argc, char *argv[])
{...
        pqsignal_pm(SIGHUP, SIGHUP_handler);       /* reread config file and have

         * children do same */
        pqsignal_pm(SIGINT, pmdie);      /*这里是fast shutdown*/
        pqsignal_pm(SIGQUIT, pmdie);       /* send SIGQUIT and die */
        pqsignal_pm(SIGTERM, pmdie);       /* wait for children and shut down */
        pqsignal_pm(SIGALRM, SIG_IGN);    /* ignored */
        pqsignal_pm(SIGPIPE, SIG_IGN);      /* ignored */
        pqsignal_pm(SIGUSR1, sigusr1_handler);       /* message from child process */
        pqsignal_pm(SIGUSR2, dummy_handler);       /* unused, reserve for children */
        pqsignal_pm(SIGCHLD, reaper);       /* handle child termination */
```

信号

子进程注册的信号，示例checkpointer：

```
void
CheckpointerMain(void)
{
...
    //checkpointer会屏蔽SIGTERM，真正的停止是SIGUSR2
            pqsignal(SIGHUP, SignalHandlerForConfigReload);
            pqsignal(SIGINT, ReqCheckpointHandler); /* request checkpoint */
            pqsignal(SIGTERM, SIG_IGN); /* ignore SIGTERM */
            pqsignal(SIGQUIT, SignalHandlerForCrashExit);
            pqsignal(SIGALRM, SIG_IGN);
            pqsignal(SIGPIPE, SIG_IGN);
            pqsignal(SIGUSR1, procsignal_sigusr1_handler);
            pqsignal(SIGUSR2, SignalHandlerForShutdownRequest);
```

每个子进程都会注册信号，大部分内容都差不多，根据子进程负责不同的功能略微有差别

reaper函数

reaper是进程回收函数，子进程退出后会发送pm SIGCHLD信号，pm通过reaper函数清理进程。 如backend、startup、checkpointer等进程都有自己的清理流程。
以checkpointer进程退出，reaper回收为例：

```
if (pid == CheckpointerPID)
{
        CheckpointerPID = 0;
        if (EXIT_STATUS_0(exitstatus) && pmState == PM_SHUTDOWN)
        {...
                //
                if (PgArchPID != 0)
                        signal_child(PgArchPID, SIGUSR2); //最后一次唤醒archiver
                SignalChildren(SIGUSR2);//最后一次唤醒walsender
 pmState = PM_SHUTDOWN_2;
                ...
        }
        ...
PostmasterStateMachine();//最后reaper还是会进入状态机函数
```

pmdie函数

pmdie函数用于处理不同的postmaster signals，包括子进程给pm发送的SIGCHILD和pg_ctl发送的停库信号。
pm信号处理主体逻辑是根据signal转换pmState状态机状态，并进入状态机PostmasterStateMachine处理。

```
pmdie(SIGNAL_ARGS)
{...
        switch (postgres_signal_arg)
        {
                case SIGTERM://Smart Shutdown...
                        if (pmState == PM_RUN)
                                connsAllowed = ALLOW_SUPERUSER_CONNS;
                        PostmasterStateMachine();  //smart shutdown直接交给状态机处理
                case SIGINT://Fast Shutdown
                        ...
                                pmState = PM_STOP_BACKENDS;
                        PostmasterStateMachine();
                case SIGQUIT://Immediate Shutdown
                        ...
                        TerminateChildren(SIGQUIT);//abort all children with SIGQUIT
        pmState = PM_WAIT_BACKENDS;
                        PostmasterStateMachine();
                        break;
```

PostmasterStateMachine函数

PostmasterStateMachine函数是处理pm退出的主体函数，主要是处理pm在不同停库状态下做什么。比如一般进程退出、walsender、归档进程退出、异常退出等等状态

```
    if (pmState == PM_RUN || pmState == PM_HOT_STANDBY)
  //当所有normal backend都退出后，pmState转换PM_STOP_BACKENDS
    if (pmState == PM_STOP_BACKENDS)
  //除了checkpointer, archiver, stats, and syslogger，其他子进程都发生SIGTERM
  //pmState转换为PM_WAIT_BACKENDS;
    if (pmState == PM_WAIT_BACKENDS)
  //signal_child(CheckpointerPID, SIGUSR2);
  //pmState转换为PM_SHUTDOWN;
    if (pmState == PM_SHUTDOWN_2) //reaper在处理checkpointer的退出时，会设置pmState = PM_SHUTDOWN_2
  //PM_SHUTDOWN_2本质上是等待walsender和archiver
  //pmState转换为PM_WAIT_DEAD_END
    if (pmState == PM_WAIT_DEAD_END)
  //PM_WAIT_DEAD_END判断BackendList已经完全清空
  //pmState 转换为PM_NO_CHILDREN
    if (Shutdown > NoShutdown && pmState == PM_NO_CHILDREN) //PM_NO_CHILDREN是停库的最后一个状态，表示可以正常停库了
      //异常退出ExitPostmaster(1);正常退出ExitPostmaster(0);
```

checkpointer和walsender的退出

checkpointer进程在pm退出时会被唤醒做最后一次shutdown checkpoint等工作，但是创建shutdown checkpoint需要等walsender全部进入退出态

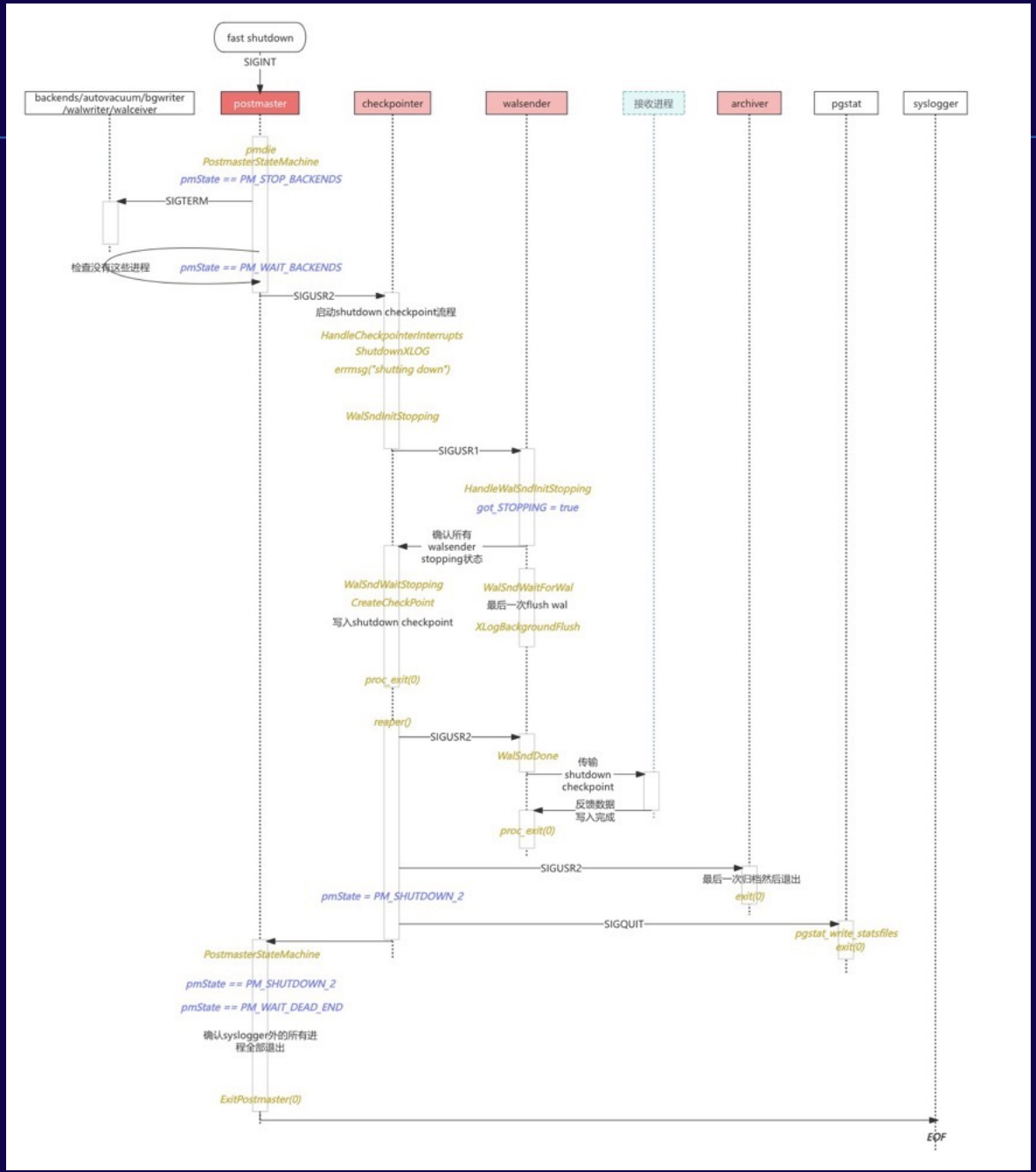
```
HandleCheckpointerInterrupts(void)  //checkpointer的退出，最最重要的就是做ShutdownXLOG
{
...
                ShutdownXLOG(0, 0);//这里会写shutdown checkpoint
                proc_exit(0);//正常退出态为0
}

ShutdownXLOG(int code, Datum arg)
{
...
        WalSndInitStopping();//walsender初始化stopping

        WalSndWaitStopping();//等待所有walsender处于stopping状态
        {
                //这里就是shutdown checkpoint的创建函数
                CreateCheckPoint(CHECKPOINT_IS_SHUTDOWN | CHECKPOINT_IMMEDIATE);
        }
```

checkpointer和walsender的退出

WalSndWaitStopping会等待walsender退出，如果不退出是死循环等待

```
WalSndWaitStopping(void)
{
        for (;;)
        {...//省略很多
                if (all_stopped)
                        return;
        }
}
```

停库逻辑


fast shutdown
SIGINT
backends/autovacuum/bgwriter /walwriter/walceiver
postmaster
checkpointer
walsender
接收进程
archiver
pgstat
syslogger
pmdie
PostmasterStateMachine
pmState == PM_STOP_BACKENDS
SIGTERM
检查没有这些进程
pmState == PM_WAIT_BACKENDS
SIGUSR2
启动shutdown checkpoint流程
HandleCheckpointerInterrupts
ShutdownXLOG
errmsg("shutting down")
WalSndInitStopping
SIGUSR1
HandleWalSndInitStopping
got_STOPPING = true
确认所有walsender stopping状态
WalSndWaitStopping
CreateCheckPoint
写入shutdown checkpoint
WalSndWaitForWal
最后一次flush wal
XLogBackgroundFlush
proc_exit(0)
reaper()
SIGUSR2
WalSndDone
传输shutdown checkpoint
反馈数据写入完成
proc_exit(0)
SIGUSR2
最后一次归档然后退出
exit(0)
pmState = PM_SHUTDOWN_2
SIGQUIT
pgstat_write_statsfiles
exit(0)
PostmasterStateMachine
pmState == PM_SHUTDOWN_2
pmState == PM_WAIT_DEAD_END
确认syslogger外的所有进程全部退出
ExitPostmaster(0)
EOF

Spill阻止起库和加速起库

spill阻止起库

现象：数据库启动缓慢，startup进程在读取spill文件，文件名在变化。查看spill文件也很慢，ls -l最后跑出来有1000w个文件spill文件。
- 数据库正在启动，没有直接卡住。不过看上去启动会花很多时间
- 1000w个spill文件直接导致linux操作系统很卡。在主机上做任何操作都很费劲了

疑问？？？
1.1000w个spill哪里来的?
2.库起不来，怎么办?

spill怎么来的，怎么定位到wal文件

```
ReorderBufferSerializedPath(char *path, ReplicationSlot *slot, TransactionId xid,
                            XLogSegNo segno)
{
    XLogRecPtr    recptr;

    XLogSegNoOffsetToRecPtr(segno, 0, wal_segment_size, recptr);

    snprintf(path, MAXPGPATH, "pg_replslot/%s/xid-%u-lsn-%X-%X.spill",
             NameStr(MyReplicationSlot->data.name),
             xid,
             (uint32) (recptr >> 32), (uint32) recptr);
}
```

spill怎么来的，怎么定位到wal文件

```
#define XLogSegNoOffsetToRecPtr(segno, offset, wal_segsz_bytes, dest) \
        (dest) = (segno) * (wal_segsz_bytes) + (offset)
```

XLogRecPtr就是LSN
(uint32) (recptr >> 32)表示取LSN前32位。因为传入的offset=0，也传入了segno，那么根本不需要wal日志段内偏移量信息。wal_segsz_bytes的真实值是128M*1014*1024，将XLogSegNoOffsetToRecPtr中的式子转化下为（以xid-407989064-lsn-42D1E-20000000.spill为例）：

```
segno= dest/(128*1024*1024)
-- 再把16进制20000000转化下
segno= x'20000000'::int/(128*1024*1024)
segno= 4

ls 42D1E*04
000000020042D1E00000004
```

spill怎么来的

spill文件生成规则如下：
- 同一个事务id，如果跨wal就会产生多个spill。如：一个不含子事务的大事务跨越3个wal，就会对应3个spill文件
- 不同的事务id对应不同的spill。如：1000w个子事务对应1000w个spill

例如：spill文件名结构xid-407989064-lsn-42D1E-20000000.spill

xid	lsn前32位；即wal逻辑日志号	由wal日志段号换算；不等于段号
xid-407989064	lsn-42D1E	20000000

spill怎么来的

spill溢出逻辑的版本差异：

- PG12及以前是写死的4096条changes
- PG13新增logical_decoding_work_mem参数，可调整内存大小以减少spill概率
- PG14及以后支持流式复制Streaming
- 触发流式复制也需要一定的条件，所以即使有流式复制也可能会发生spill
- PG17新增debug_logical_replication_streaming参数以强制触发流式传输

如何加速起库

库起不来?
起库中最关键的进程是startup进程。startup进程无论如何都在起库时启动。对于一致性停库和非一致性停库来说，非一致性停库会走跟多的逻辑。
其一便是sync data目录：

```
if (ControlFile->state != DB_SHUTDOWNED &&
        ControlFile->state != DB_SHUTDOWNED_IN_RECOVERY)
{
        RemoveTempXlogFiles();
        SyncDataDirectory();
}
```

因为控制文件记录的状态不是正常停库的，所以走到if中调用SyncDataDirectory()做fsync持久化。这个动作是为了保证在数据库运行前data目录是完全持久化的

如何加速起库

Startup进程做了很多很多事，其中跟spill相关的除了fsync data外，还有启动reorderbuffer。启动的时候会清理所有slot目录的spill文件

```
StartupReorderBuffer(void)
{...
        while ((logical_de = ReadDir(logical_dir, "pg_replslot")) != NULL)
        {
...
                /* if it cannot be a slot, skip the directory */
                if (!ReplicationSlotValidateName(logical_de->d_name, DEBUG2))
                        continue;

                /*
                 * ok, has to be a surviving logical slot, iterate and delete
                 * everything starting with xid-*
                 */
                ReorderBufferCleanupSerializedTXNs(logical_de->d_name);
        }
        FreeDir(logical_dir);
}
```

如何加速起库

起库逻辑小结：
- pg会启动一个辅助进程startup以协助起库，不同于在常见的childprocess（walwriter、bgwriter、checkpointer等等）进程，它是起库过程中必定会启动的进程，它会做很多事情
- StartupXLOG起库时一定会被调用，无论数据库是否一致性停库
- 只有非正常停库状态下，才会触发SyncDataDirectory
- SyncDataDirectory会fsync持久化所有data文件，并查看所有data文件的stat信息
- fsync是为了在库启动前保证data文件都一致；stat应该是为了验证文件是否正常和可读（在startup进程启动前只验证过datadir目录可读性）
- 无论停库状态，StartupReorderBuffer一定会被调用并清理所有复制槽的spill文件。

如何加速起库

思考如何去加速起库?
1000w个spill删除起来肯定是很慢的，直接mv目录的话就非常快。但是直接mv需要注意mv后的名称和state文件，以及需要知道mv到底跳过了哪一个源码步骤。
由于是异常停库，startup进程会执行SyncDataDirectoryfsync和stat所有data文件，这一点是比较难绕过的。SyncDataDirectory做完以后，才开始处理复制槽。处理复制槽时会调用StartupReorderBuffer()->ReorderBufferCleanupSerializedTXNs全量清理spill文件。

在进入清理前，会调用ReplicationSlotValidateName校验复制槽名称的有效性，我们可以在ReplicationSlotValidateName上做文章，以骗过startup进程跳过ReorderBufferCleanupSerializedTXNs的过程。

如何加速起库

```
ReplicationSlotValidateName(const char *name, int elevel)
{...
                    if (!((*cp >= 'a' && *cp <= 'z')
                          || (*cp >= '0' && *cp <= '9')
                          || (*cp == '_')))
...
}
```

有效slot name只包含a-z;0-9;_。所以rename时建议加个点.，

- 建议slotname.bak,slotname.20241215等。
- 不建议slotnamebackup,slotname20241215,slotname_bak等等
- 不建议.tmp后缀，slotname有.tmp后缀有特殊含义

最后rename后，要创建目录和拷贝state，不然启动的slot会表现的很反常（比如重复的slotname、自动生产一个slotname、删不到slot、下游起不来链路等等）。

```
cd pg_replslot
mv slotname slotname.bak
mkdir slotname
cp slotname.bak/state slotname/
```

如何加速起库

伪造2000w个spill测试起库时间：

编号	测试方案	起库时间
1	正常停库；起库不做fsync和stat，不做unlink	0.1秒
2	正常停库，无效mv；起库不做fsync和stat，做unlink	11分41秒
3	异常停库，有效mv；起库做fsync和stat，不做unlink	4分35秒
4	异常停库，无效mv；起库做fsync和stat，做unlink	32分2秒
5	异常停库，rm（创建slot目录并保留state）	13分04秒

HOW2025

IvorySQL 2025生态大会
暨PostgreSQL高峰论坛

Thanks

谢谢