聊聊PostgreSQL的WAL日志系统

作者	时间	QQ技术交流群
perrynzhou@gmail.com	2022/01/01	672152841

存储内核技术交流

微信扫描二维码，关注我的公众号

开源存储问题解答社区:https://github.com/perrynzhou/deep-dive-storage-in-china

wal日志格式

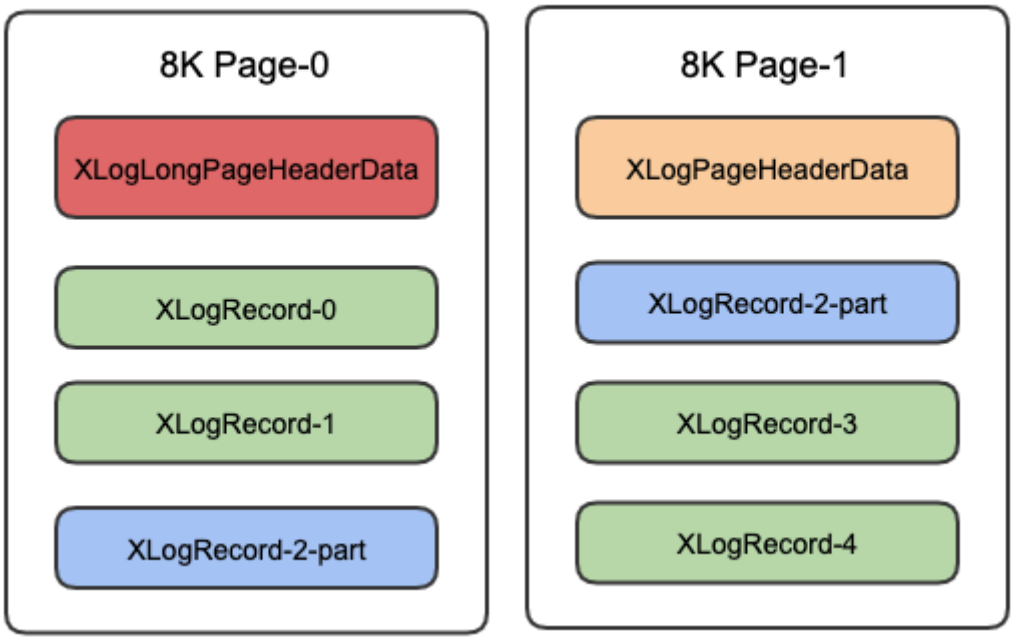
- wal 日志是由多个固定的段组成，每个段都是单独的wal日志文件。日志文件内部划分类型的page,每个page是有page header(页面头)和 log record(日志记录)组成，每个page默认是8K大小。page中的header分为两类，日志文件中第一个page的header记录了日志文件长度，和page大小，这个是由`XLogLongPageHeaderData`来描述。日志文件其他类型的page header则是使用`XLogPageHeaderData`描述，它包含了日志对应版本和时间线信息.

```
// 日志的版本信息
#define XLOG_PAGE_MAGIC 0xD10D

typedef uint64 XLogRecPtr;
typedef struct XLogPageHeaderData
{
        // 校验当前事务的版本信息
```

```
        uint16          xlp_magic;
        // 事务中的一些flag信息
        uint16          xlp_info;
        // 当前页中第一条记录的时间线信息
        TimeLineID      xlp_tli;
        // 当前日主记录的地址
        XLogRecPtr      xlp_pageaddr;
        // 日志记录跨页面保存时候使用，保存日志长度
        uint32          xlp_rem_len;
} XLogPageHeaderData;

typedef struct XLogLongPageHeaderData
{
        // 标准的page header
        XLogPageHeaderData std;
        // pg_control中的系统标识ID
        uint64          xlp_sysid;
        // 段大小
        uint32          xlp_seg_size;
        uint32          xlp_xlog_blcksz;
} XLogLongPageHeaderData;
```

- `XLogRecord` 日志是每条日志中的日志数据的表达形式，日志是记录整个数据库每一次变更的动作。wal日志从实现的角度分析,wal日志page中存储page header和log record。每个page header都存储在每个page的头部，紧接着头部后面都是存储log record的数据。

```
typedef struct XLogRecord
{
        // 日志记录的长度
        uint32          xl_tot_len;
        // 事务的ID
        TransactionId xl_xid;
        // 上一条redord的指针
        XLogRecPtr      xl_prev;
        uint8           xl_info;
        // 日志对应的resource manager的ID
        RmgrId          xl_rmid;

        // 日志数据的crc的数据校验
        pg_crc32c       xl_crc;                 /* CRC for this record */

        /* XLogRecordBlockHeaders and XLogRecordDataHeader follow, no
padding */

} XLogRecord;
```



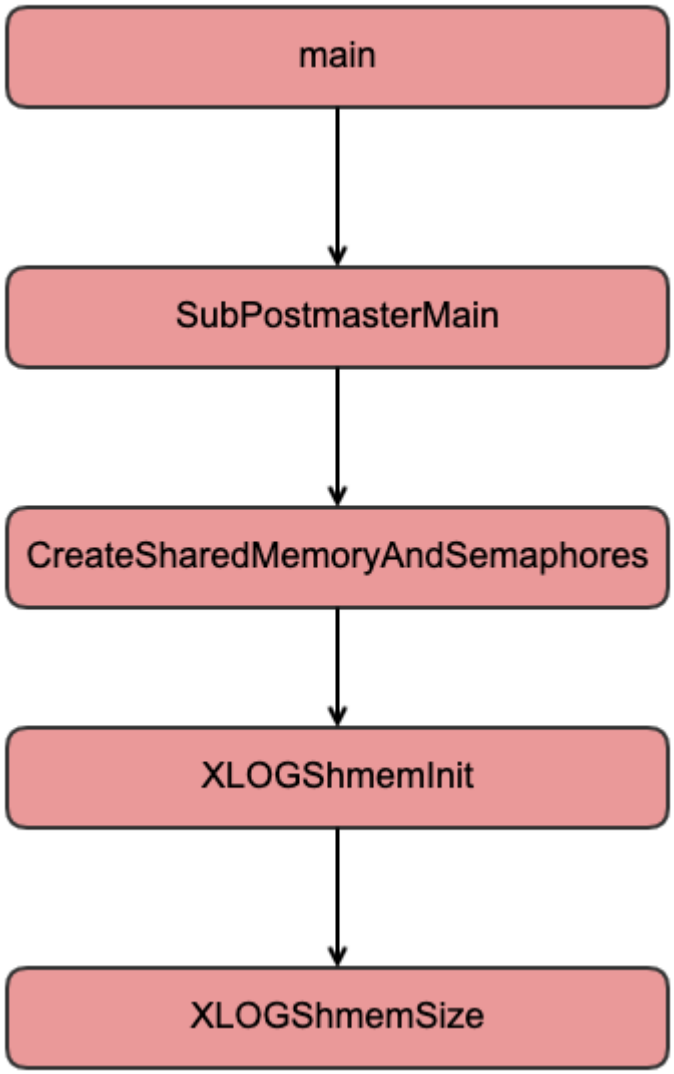
wal日志系统初始化

- 每次变更事务提交时候,需要将变更事务日志落盘，在PG中为了提高性能，并非采用实时flush到磁盘,而是在PG中提供`XLog Buffer`空间临时存储提交的事务日志，然后定期flush到磁盘。`XLog Buffer`的大小是有参数`wal_buffers`设定，当这个参数设置为-1时候，PG会根据`shared_buffers`和`wal_segment_size`参数自动计算而得到。
- 日志系统初始化流程,流程是从`main`开始，计算需要共享的内存大小，然后通过`XLOGShmemInit`调用`XLOGShmemSize`进行初始化Log需要的内存和Log控制信息结构初始化。Log子系统中比较核心的控制字段有`struct XLogCtlData`和`struct XLogCtlInsert`,其中`XLogCtlData`结构中存储了当前WAL的写入状态、flush状态以及Buffer Page的状态信息；`XLogCtlInsert`结构存储往WAL日志中写入log record需要各种结构

```
int main(int argc, char *argv[])
{
#ifdef EXEC_BACKEND
        if (argc > 1 && strncmp(argv[1], "--fork", 6) == 0)
                SubPostmasterMain(argc, argv);  /* does not return */
#endif
}
void SubPostmasterMain(int argc, char *argv[])
{
        CreateSharedMemoryAndSemaphores();
}
void CreateSharedMemoryAndSemaphores(void)
{
        XLOGShmemInit()
}
void XLOGShmemInit(void)
{
```

```
                XLogCtl = (XLogCtlData *)
                ShmemInitStruct("XLOG Ctl", XLOGShmemSize(), &foundXLog);
}


        {
                {"wal_buffers", PGC_POSTMASTER, WAL_SETTINGS,
                        gettext_noop("Sets the number of disk-page buffers
in shared memory for WAL."),
                        NULL,
                        GUC_UNIT_XBLOCKS
                },
                &XLOGbuffers,
                -1, -1, (INT_MAX / XLOG_BLCKSZ),
                check_wal_buffers, NULL, NULL
        },

// 初始胡Log系统的内部数据结构和Log的共享内存
Size XLOGShmemSize(void)
{
        Size            size;

        // 如果XLOGbuffers=-1，则表示没有设置log_buffer大小，则需要自动计算出一个合
理的值。
        // XLOGChooseNumBuffers 逻辑，初始化一个val=1000/32,如果val <8，val=8;
如果val > (16*1024*1024)/xlog块大小，则val = (16*1024*1024)/xlog块大小,返回val
作为计算出来的XLOGBuffer的大小
        if (XLOGbuffers == -1)
        {
                char            buf[32];

                snprintf(buf, sizeof(buf), "%d", XLOGChooseNumBuffers());
                SetConfigOption("wal_buffers", buf, PGC_POSTMASTER,
PGC_S_OVERRIDE);
        }
        Assert(XLOGbuffers > 0);

        /* xlog的控制头，也是共享信息 */
        size = sizeof(XLogCtlData);

        // 日志插入时需要的共享锁需要的空间大小
        size = add_size(size, mul_size(sizeof(WALInsertLockPadded),
NUM_XLOGINSERT_LOCKS + 1));
        // 日志文件块起始lsn需要的空间大小
        size = add_size(size, mul_size(sizeof(XLogRecPtr), XLOGbuffers));
        // log buffer io的空间大小
        size = add_size(size, XLOG_BLCKSZ);
        // XLOGBuffers大小，xlog块大小 和XLOGbuffers 乘机
        size = add_size(size, mul_size(XLOG_BLCKSZ, XLOGbuffers));
```

```
        return size;
}
```

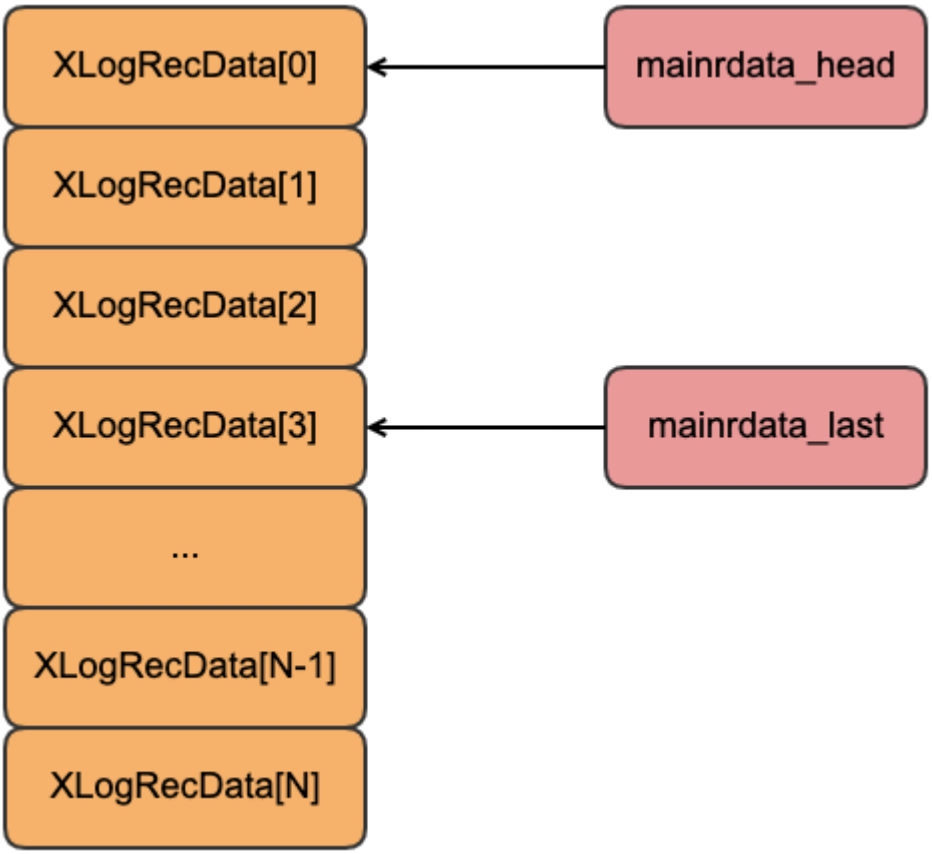


wal日志写入

-PostgreSQL高版本中(>9.5)事务的日志不是直接写入到Wal Buffer中，而是先组成`XLogRecData`链表，然后在转换为一个log record.PG中默认定义了XLogRecData链表数组`XLogRecData *rdatas`，这个数组长度由`XLR_NORMAL_RDATAS=20`.另外日志写入中定义了`registered_buffer *registered_buffers`，用来注册已经被修改的page,数组中每个元素都占用一个槽位。

```
// 日志链表的数组的定义
static XLogRecData *rdatas;
// 当前已经使用的数组下标
static int      num_rdatas;
// 最大的数组下标
static int      max_rdatas;
```

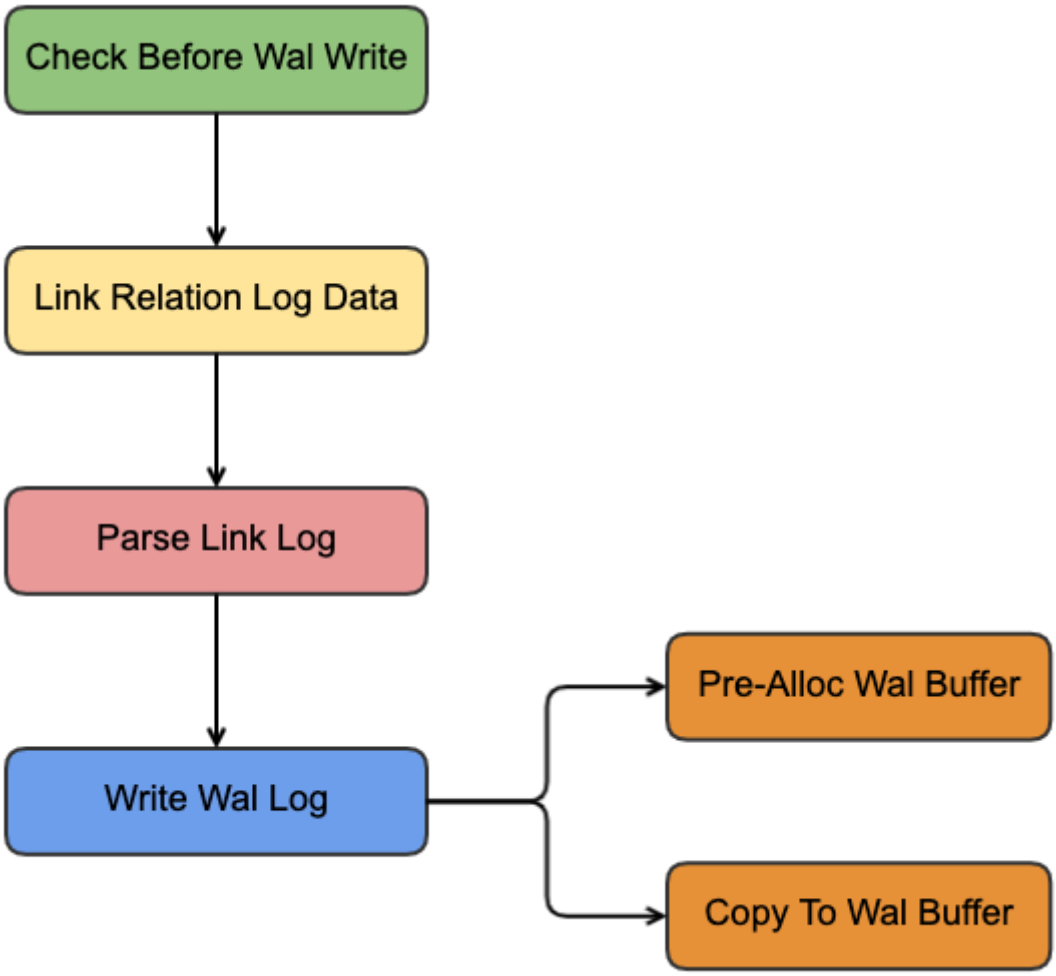
```
typedef struct XLogRecData
{
        // 下一个日志data的指针
        struct XLogRecData *next;
        // 日志数据
        char       *data;
        // 数据长度
        uint32          len;
} XLogRecData;


// 当页面被修改buffer page会被注册到registered_buffers数组
static registered_buffer *registered_buffers;
// registered_buffers最大长度
static int      max_registered_buffers;
// 起始长度
static int      max_registered_block_id = 0;

typedef struct
{
        //  槽位使用标志
        bool            in_use;
        // 和buffer page相关的flag信息
        uint8           flags;
        // buffer page对应的那个表
        RelFileNode rnode;
        ForkNumber      forkno;
        BlockNumber block;
        // buffer page的指针
        Page            page;
        // XLogRegisterBufData注册的长度
        uint32          rdata_len;
        // XLogRegisterBufData注册数据到这个链表
        XLogRecData *rdata_head;
        // XLogRegisterBufData注册数据到链表尾部
        XLogRecData *rdata_tail;

        XLogRecData bkp_rdatas[2];      /* temporary rdatas used to hold
references to
                                                                        * backup
block data in XLogRecordAssemble() */

        // 压缩page的使用临时空间
        char            compressed_page[PGLZ_MAX_BLCKSZ];
} registered_buffer;
```



- 日志首先是写入日志前的检查，组装日志相关数据，这个阶段日志相关的数据写入到链表，最后根据链表中的日志相关数据转换wal 日志物理条目，通过申请预留wal_buffer空间和日志数据复制来完成wal的写入。接下来以`heap_insert`函数中为例，简要分析下其过程

```
void
heap_insert(Relation relation, HeapTuple tup, CommandId cid,
                        int options, BulkInsertState bistate)
{
        /* XLOG stuff */
        if (RelationNeedsWAL(relation))
        {
                // 写入日志前的准备
                XLogBeginInsert();

                // 注册生成日志相关的数据
                XLogRegisterData((char *) &xlrec, SizeOfHeapInsert);

                xlhdr.t_infomask2 = heaptup->t_data->t_infomask2;
                xlhdr.t_infomask = heaptup->t_data->t_infomask;
                xlhdr.t_hoff = heaptup->t_data->t_hoff;

                //注册一个buffer,占用了0号槽位buffer
                XLogRegisterBuffer(0, buffer, REGBUF_STANDARD | bufflags);

                // 在0号槽位的buffer注册相关的数据
```

```
                XLogRegisterBufData(0, (char *) &xlhdr, SizeOfHeapHeader);
                XLogRegisterBufData(0,
                                    (char *) heaptup->t_data + SizeofHeapTupleHeader,
                                    heaptup->t_len - SizeofHeapTupleHeader);

                XLogSetRecordFlags(XLOG_INCLUDE_ORIGIN);

                // 执行日志插入
                recptr = XLogInsert(RM_HEAP_ID, info);
                // 设置page的LSN
                PageSetLSN(page, recptr);
        }
}
```



针对机械磁盘一般是512个字节的扇区，而os的一般数据block为4K，PG的数据库的每次数据写入是page是8K。这三者的大小完全不对等，所以PG的Page刷新到磁盘并不是原子操作，因此一个Page写入到磁盘很可能是写了一部分，这会导致Page损坏。不同的数据都有对应的解决方案，MSQL是double write机制保证；PG则是Full Page Write.如果在数据更新的某个Page时，这个Page最近修改的LSN小于全局Checkpoint Lsn，修改之前会把这个Page全部写到Wal

日志中，然后进行修改操作；PG是默认是通过参数`full_page_writes`参数来禁用或者启用Full Page Write机制。