第 5 章

事务机制源码解析

事务是数据库操作的执行单位，需要满足最基本的 ACID（原子性、一致性、隔离性、持久性）属性。

（1）原子性：一个事务提交之后要么全部执行，要么全部不执行。

（2）一致性：事务的执行不能破坏数据库的完整性和一致性。

（3）隔离性：事务的隔离性是指在并发中，一个事务的执行不能被其他事务干扰。

（4）持久性：一旦事务完成提交，那么它对数据库的状态变更就会永久保存在数据库中。

本章主要介绍 openGauss 事务模块如何实现数据库事务的基本属性，使用户数据不丢不错、修改不乱、查询无错误。

5.1 事务整体架构与代码

事务模块总体结构如图 5-1 所示。

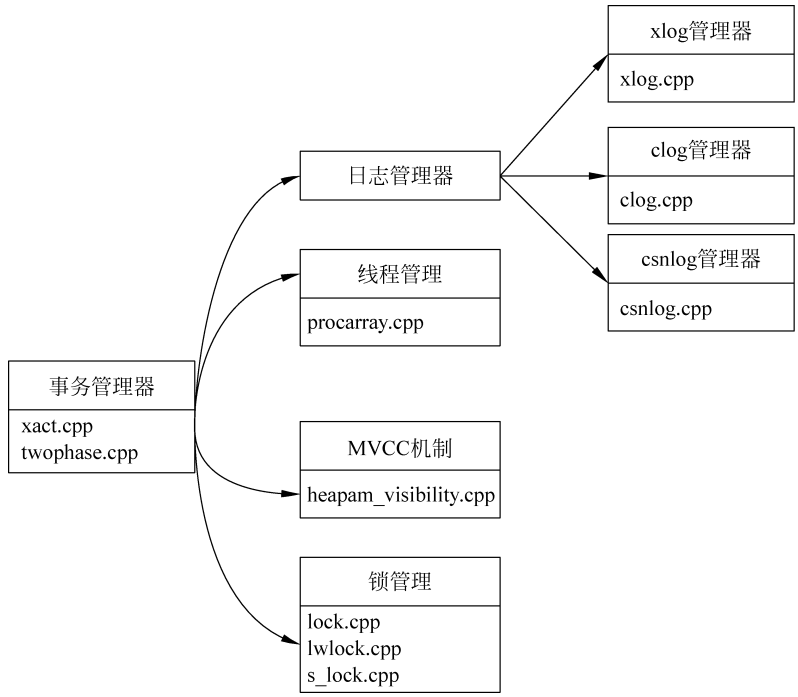

图 5-1 事务模块总体结构

在 openGauss 中，事务的实现与存储引擎的实现有很强关联，代码主要集中在 src/gausskernel/storage/access/transam 及 src/gausskernel/storage/lmgr 下，关键文件如图 5-1 所示。

(1) 事务管理器：事务系统的中枢，它的实现是一个有限循环状态机，通过接收外部系统的命令并根据当前事务所处的状态决定事务的下一步执行过程。

(2) 日志管理器：用来记录事务执行的状态及数据变化的过程，包括事务提交日志(CLOG)、事务提交序列日志(CSNLOG)及事务日志(XLOG)。其中，CLOG 只用来记录事务执行的结果状态；CSNLOG 记录日志提交的顺序，用于可见性判断；XLOG 是数据的 redo 日志，用于恢复及持久化数据。

(3) 线程管理：通过一片内存区域记录所有线程的事务信息，任何一个线程可以通过访问该区域获取其他事务的状态信息。

(4) MVCC 机制：openGauss 系统中，事务执行读流程结合各事务提交的 CSN 序列号，采用了多版本并发控制机制，实现了元组的读和写互不阻塞。详细可见性判断方法见本书 5.2 节相关内容。

(5) 锁管理：实现系统的读写并发控制，通过锁机制来保证事务读写流程的隔离性。

5.2　事务并发控制

事务并发控制机制用来保证并发执行事务的情况下 openGauss 的 ACID 特性。下面将逐一介绍事务并发控制的各组成部分。

5.2.1　事务状态机

openGauss 将事务系统分为上层(事务块 TBlockState)和底层(TransState)两个层次。

通过分层的设计，在处理上层业务时可以屏蔽具体细节，灵活支持客户端各类事务执行语句(BEGIN/START TRANSACTION/COMMIT/ROLLBACK/END)。

(1) TBlockState：客户端 query 的状态，用于提高用户操作数据的灵活性，用事务块的形式支持在一个事务中执行多条 query 语句。

(2) TransState：内核端视角，记录了整个事务当前所处的具体状态。

1. 事务上层状态机

事务上层状态机结构体代码如下：

```
typeset enum TBlockState
{
/* 不在事务块中的状态:单条 SQL 语句 */
TBLOCK_DEFAULT,          /* 事务块默认状态 */
TBLOCK_STARTED,          /* 执行单条 query 语句 */

/* 处于事务块中的状态:一个事务包含多条语句 */
TBLOCK_BEGIN,            /* 遇到事务开始命令 BEGIN/START TRANSACTION */
TBLOCK_INPROGRESS,       /* 表明正在事务块处理过程中 */
TBLOCK_END,              /* 遇到事务结束命令 END/COMMIT */
TBLOCK_ABORT,            /* 事务块内执行报错,等待客户端执行 ROLLBACK */
```

```
TBLOCK_ABORT_END,       /* 在事务块内执行报错后,接收客户端执行 ROLLBACK */
TBLOCK_ABORT_PENDING,   /* 事务块内执行成功,接收客户端执行 ROLLBACK(期望事务回滚) */
TBLOCK_PREPARE,         /* 两阶段提交事务,收到 PREPARE TRANSACTION 命令 */

/* 子事务块状态,与上述事务块状态类似 */
TBLOCK_SUBBEGIN,        /* 遇到子事务开始命令 SAVEPOINT */
TBLOCK_SUBINPROGRESS,   /* 表明正在子事务块处理过程中 */
TBLOCK_SUBRELEASE,      /* 遇到子事务结束命令 RELEASE SAVEPOINT */
TBLOCK_SUBCOMMIT,       /* 遇到事务结束命令 END/COMMIT 从底层的子事务递归提交到顶层事务 */
TBLOCK_SUBABORT,        /* 子事务块内执行报错,等待客户端 ROLLBACK TO/ROLLBACK */
TBLOCK_SUBABORT_END,    /* 在子事务块内执行报错后,接收到客户端 ROLLBACK TO 上层子事务/ROLLBACK */
TBLOCK_SUBABORT_PENDING,/* 子事务块内执行成功,接收客户端执行的 ROLLBACK TO 上层子事务/ROLLBACK */
TBLOCK_SUBRESTART,      /* 子事务块内执行成功,收到 ROLLBACK TO 当前子事务 */
TBLOCK_SUBABORT_RESTART         /* 子事务块内执行报错后,接收到 ROLLBACK TO 当前子事务 */
} TBlockState;
```

为了便于理解,可以先不关注子事务块的状态。当理解了主事务的状态机行为后,子事务块的状态机转换同父事务类似。父子事务的关系类似于一个栈的实现,子事务相较于父事务后开始先结束。

显式事务块的状态机及相应的转换函数如图 5-2 所示。

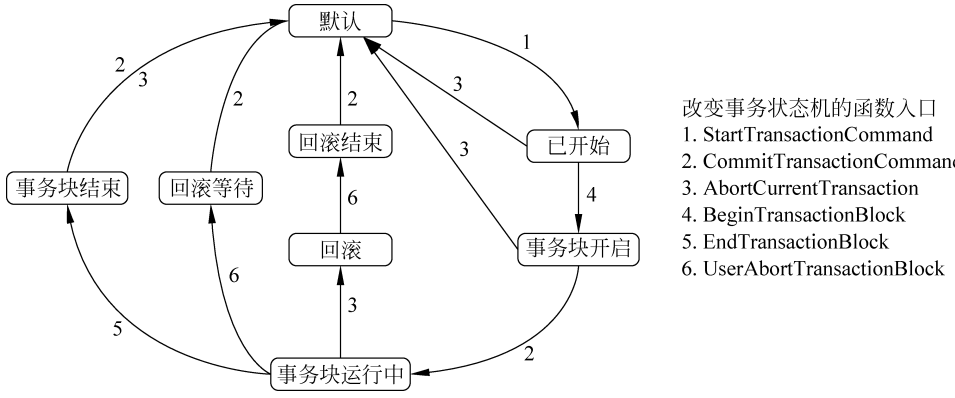

图 5-2　显式事务块状态机及相应的转换函数

图 5-2 中的事务状态相对应的事务状态机结构体中的值如表 5-1 所示。

表 5-1　事务块状态相对应的事务状态机结构体中的值

事务状态	事务状态机结构体
默认	TBLOCK_DEFAULT
已开始	TBLOCK_STARTED
事务块开启	TBLOCK_BEGIN
事务块运行中	TBLOCK_INPROGRESS
事务块结束	TBLOCK_END
回滚	TBLOCK_ABORT

续表

事务状态	事务状态机结构体
回滚结束	TBLOCK_ABORT_END
回滚等待	TBLOCK_ABORT_PENDING

在无异常情形下，一个事务块的状态机如图 5-2 所示按照默认(TBLOCK_DEFAULT)→已开始(TBLOCK_STARTED)→事务块开启(TBLOCK_BEGIN)→事务块运行中(TBLOCK_INPROGRESS)→事务块结束(TBLOCK_END)→默认(TBLOCK_DEFAULT)循环。剩余的状态机是在上述正常场景下的各个状态点的异常处理分支。

(1) 在进入事务块运行(TBLOCK_INPROGRESS)前出错，因为事务还没有开启，直接报错并回滚，清理资源回到默认(TBLOCK_DEFAULT)状态。

(2) 在事务块运行中(TBLOCK_INPROGRESS)出错分为两种情形。事务执行失败：事务块运行中(TBLOCK_INPROGRESS)→回滚(TBLOCK_ABORT)→回滚结束(TBLOCK_ABORT_END)→默认(TBLOCK_DEFAULT)；用户手动回滚执行成功的事务：事务块运行中(TBLOCK_INPROGRESS)→回滚等待(TBLOCK_ABORT_PENDING)→默认(TBLOCK_DEFAULT)。

(3) 在用户执行 COMMIT 语句时出错：事务块结束(TBLOCK_END)→默认(TBLOCK_DEFAULT)。由图 5-2 可以看出，事务开始后离开默认(TBLOCK_DEFAULT)状态，事务完全结束后回到默认(TBLOCK_DEFAULT)状态。

(4) openGauss 同时还支持隐式事务块，当客户端执行单条 SQL 语句时可以自动提交，其状态机相对比较简单：按照默认(TBLOCK_DEFAULT)→已开始(TBLOCK_STARTED)→默认(TBLOCK_DEFAULT)循环。

2. 事务底层状态机

TransState 结构体代码如下：

```
typedef enum TransState
{
TRANS_DEFAULT,          /* 当前为空闲默认状态,无事务开启 */
TRANS_START,            /* 事务正在开启 */
TRANS_INPROGRESS,       /* 事务开启完毕,进入事务运行中 */
TRANS_COMMIT,           /* 事务正在提交 */
TRANS_ABORT,            /* 事务正在回滚 */
TRANS_PREPARE           /* 两阶段提交事务进入 PREPARE TRANSACTION 阶段 */
} TransState;
```

内核内部事务底层状态如图 5-3 所示，底层状态机的描述见结构体 TransState。

(1) 在事务开启前事务状态为 TRANS_DEFAULT。

(2) 事务开启过程中事务状态为 TRANS_START。

(3) 事务成功开启后一直处于 TRANS_INPROGRESS。

(4) 事务结束/回滚的过程中状态为 TARNS_COMMIT/ TRANS_ABORT。

(5) 事务结束后事务状态回到 TRANS_DEFAULT。

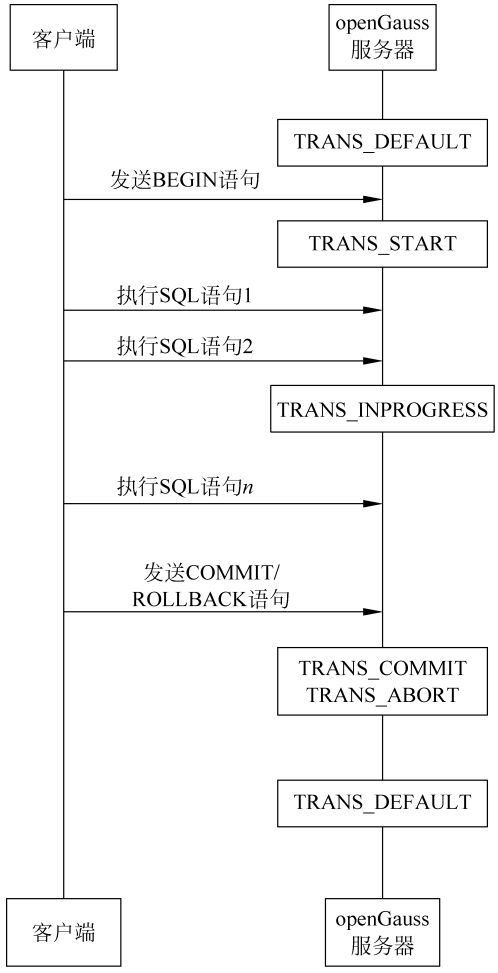

图 5-3　内核内部事务底层状态

3. 事务状态机系统实例

这里给出一条 SQL 的状态机运转实例，有助于更好地理解内部事务如何运作。在客户端执行 SQL 语句：

```
BEGIN;
    SELECT * FROM TABLE1;
END;
```

1）整体流程

整体执行过程如图 5-4 所示，任何语句的执行总是先进入事务处理接口事务块中，然后调用事务底层函数处理具体命令，最后返回事务块中。

2）BEGIN 执行流程

BEGIN 执行流程如图 5-5 所示。

（1）入口函数 exec_simple_query 处理 begin 语句。

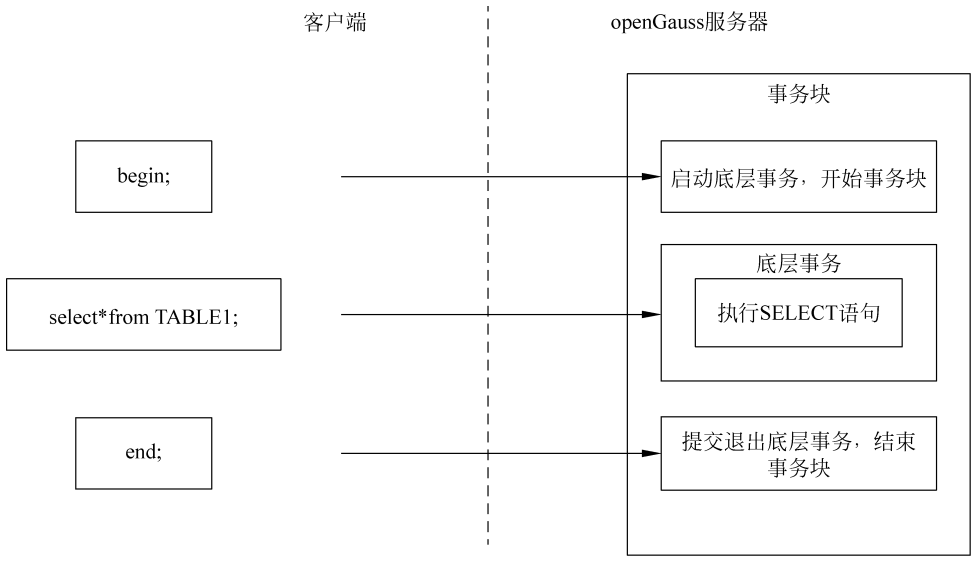

图 5-4　整体执行过程

（2）start_xact_command 函数开始一个 query 命令，调用 StartTransactionCommand 函数，此时事务块上层状态为 TBLOCK_DEFAULT，继续调用 StartTransaction 函数，设置事务底层状态 TRANS_START，完成内存、缓存区、锁资源的初始化后将事务底层状态设为 TRANS_INPROGRESS，最后在 StartTransactionCommand 函数中设置事务块上层状态为 TBLOCK_STARTED。

（3）PortalRun 函数处理 begin 语句，依次向下调用函数，最后调用 BeginTransactionBlock 函数转换事务块上层状态为 TBLOCK_BEGIN。

（4）finish_xact_command 函数结束一个 query 命令，调用 CommitTransactionCommand 函数设置事务块上层状态从 TBLOCK_BEGIN 变为 TBLOCK_INPROGRESS，并等待读取下一条命令。

3）SELECT 执行流程

SELECT 执行流程如图 5-6 所示。

（1）入口函数 exec_simple_query 处理“SELECT * FROM TABLE1;”命令。

（2）start_xact_command 函数开始一个 query 命令，调用 StartTransactionCommand 函数，由于当前上层事务块状态为 TBLOCK_INPROGRESS，说明已经在事务块内部，则直接返回，不改变事务上层及底层的状态。

（3）PortalRun 执行 SELECT 语句，依次向下调用函数 ExecutorRun，根据执行计划执行最优路径查询。

（4）finish_xact_command 函数结束一条 query 命令，调用 CommitTransactionCommand 函数，当前事务块上层状态仍为 TBLOCK_INPROGESS，不改变当前事务上层及底层的状态。

4）END 执行流程

END 执行流程如图 5-7 所示。

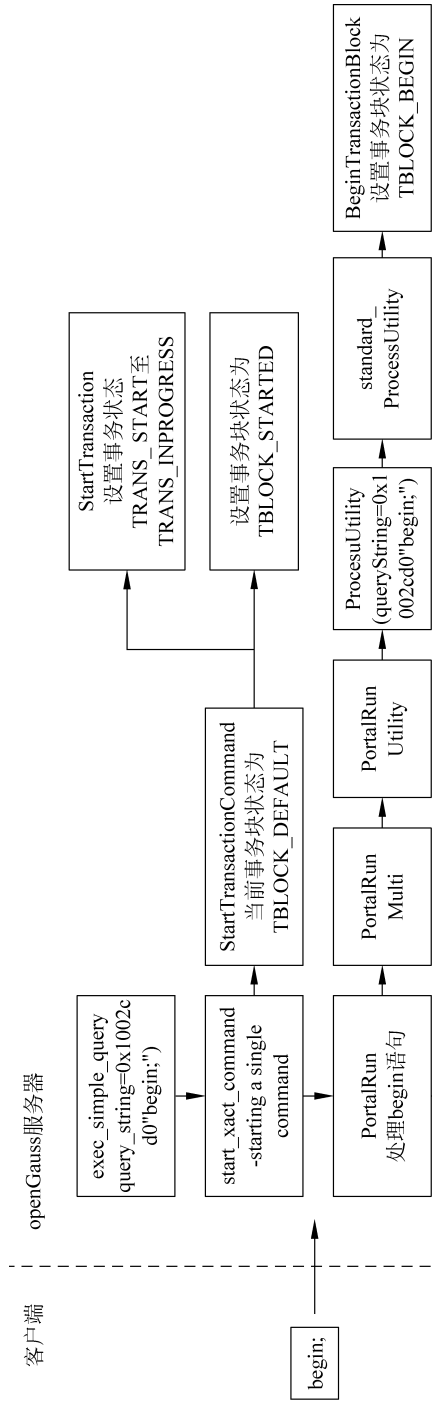

图 5-5 BEGIN 执行流程

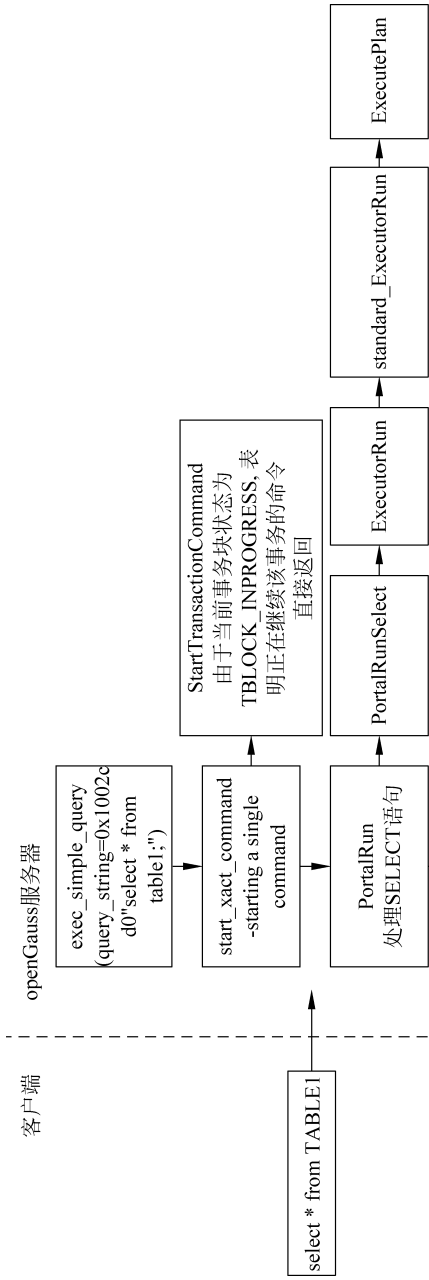

图 5-6　SELECT 执行流程

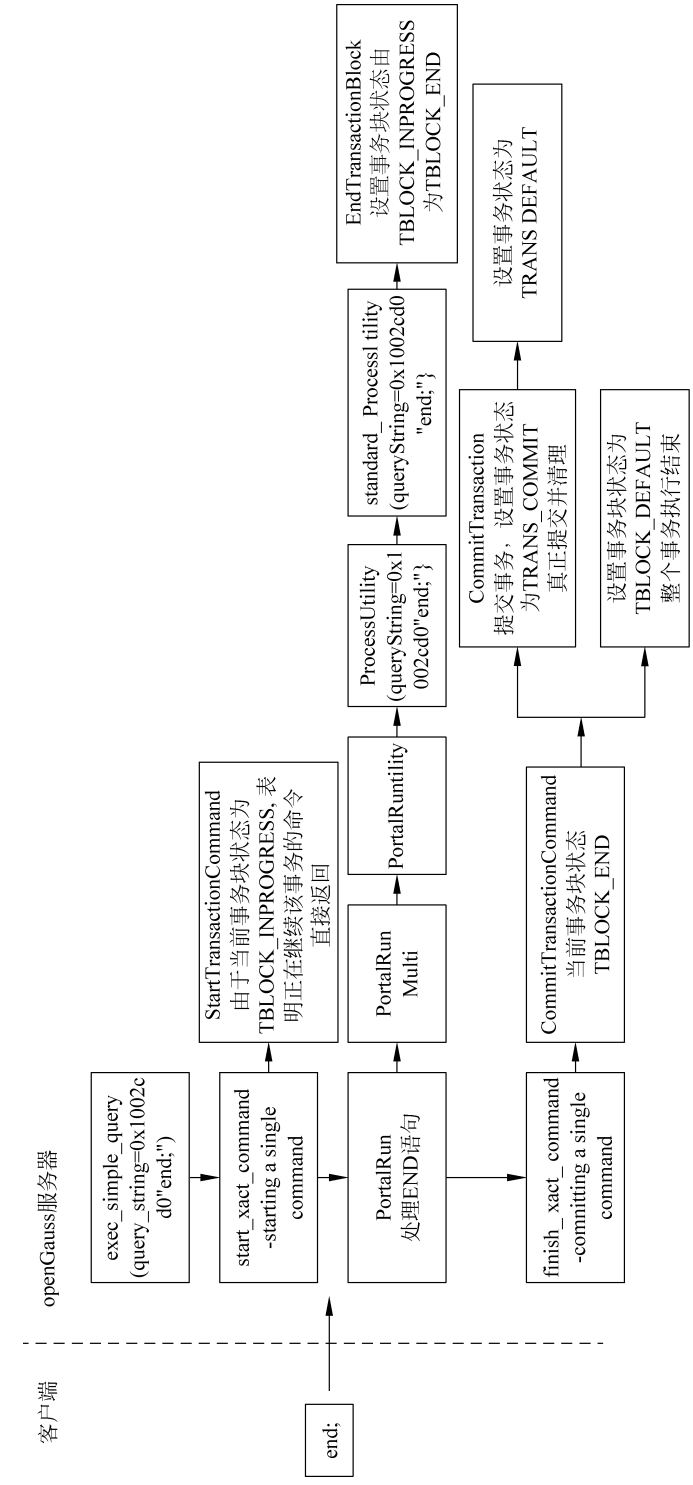
客户端
openGauss服务器
end;
exec_simple_query
(query_string=0x1002cd0"end;")
start_xact_command
-starting a single command
StartTransactionCommand
由于当前事务块状态为TBLOCK_INPROGRESS, 表明正在继续该事务的命令直接返回
PortalRun
处理END语句
PortalRun
Multi
PortalRuntility
ProcessUtility
(queryString=0x1002cd0"end;")
standard_ProcessUtility
(queryString=0x1002cd0"end;")
EndTransactionBlock
设置事务块状态由TBLOCK_INPROGRESS为TBLOCK_END
finish_xact_command
-committing a single command
CommitTransactionCommand
当前事务块状态TBLOCK_END
CommitTransaction
提交事务，设置事务状态为TRANS_COMMIT
真正提交并清理
设置事务状态为TRANS DEFAULT
设置事务块状态为TBLOCK_DEFAULT
整个事务执行结束

图 5-7　END 执行流程

（1）入口函数 exec_simple_query 处理 end 命令。

（2）start_xact_command 函数开始一个 query 命令，调用 StartTransactionCommand 函数，当前上层事务块状态为 TBLOCK_INPROGESS，表明事务仍然在进行，此时也不改变任何上层及底层事务状态。

（3）PortalRun 函数处理 END 语句，依次调用 processUtility 函数，最后调用 EndTransactionBlock 函数对当前上层事务块状态机进行转换，设置事务块上层状态为 TBLOCK_END。

（4）finish_xact_command 函数结束 query 命令，调用 CommitTransactionCommand 函数，当前事务块状态为 TBLOCK_END；继续调用 CommitTransaction 函数提交事务，设置事务底层状态为 TRANS_COMMIT，进行事务提交流程并且清理事务资源；清理后设置底层事务状态为 TRANS_DEFAULT，返回 CommitTansactionCommand 函数；设置事务块上层状态为 TBLOCK_DEFAULT，整个事务块结束。

4. 事务状态转换相关函数简述

（1）事务处理子函数：根据当前事务上层状态机，对事务的资源进行相应的申请、回收及清理。具体介绍如表 5-2 所示。

表 5-2　事务处理子函数

子　函　数	说　　明
StartTransaction	开启事务，对内存及变量进行初始化操作，完成后将底层事务状态置为 TRANS_INPROGRESS
CommitTransaction	当前的底层状态机为 TRANS_INPROGRESS，然后设置为 TRANS_COMMIT，本地持久化 CLOG 及 XLOG 日志，并清空相应的事务槽位信息，最后将底层状态机置为 TRANS_DEFAULT
PrepareTransaction	当前底层状态机为 TRANS_INPROGRESS，同前面描述的 CommitTransaction 函数类似处理，设置底层状态机为 TRANS_PREPARE，构造两阶段 GXACT 结构并创建两阶段文件，加入 dummy 的槽位信息，将线程的锁信息转移到 dummy 槽位中，释放资源，最后将底层状态机置为 TRANS_DEFAULT
AbortTransaction	释放 LWLock、UnlockBuffers、LockErrorCleanup，当前底层状态为 TRANS_INPROGRESS，设置为 TRANS_ABORT，记录相应的 CLOG 日志，清空事务槽位信息，释放各类资源
CleanupTransaction	当前底层状态机应为 TRANS_ABORT，继续清理一些资源，一般紧接着 AbortTransaction 调用
FinishPreparedTransaction	结束两阶段提交事务
StartSubTransaction	开启子事务
CommitSubTransaction	提交子事务
AbortSubTransaction	回滚子事务
CleanupSubTransaction	清理子事务的一些资源信息，类似于 CleanupTransaction
PushTransaction/PopTransaction	子事务类似于一个栈式的信息，开启和结束子事务时使用 Push/Pop 函数

（2）事务执行函数：根据相应的状态机调用子函数。具体介绍如表 5-3 所示。

表 5-3 事务执行函数

函　数	说　明
StartTransactionCommand	事务开始时根据上层状态机调用相应的事务执行函数
CommitTransactionCommand	事务结束时根据上层状态机调用相应的事务执行函数
AbortCurrentTransaction	事务内部出错，长跳转 longjump 调用，提前清理掉相应的资源，并将事务上层状态机置为 TBLOCK_ABORT

（3）上层事务状态机控制函数。具体介绍如表 5-4 所示。

表 5-4 上层事务状态机控制函数

函　数	说　明
BeginTransactionBlock	显式开启一个事务时，将上层事务状态机变为 TBLOCK_BEGIN
EndTransactionBlock	显式提交一个事务时，将上层事务状态机变为 TBLOCK_END
UserAbortTransactionBlock	显式回滚一个事务时，将上层事务状态机变为 TBLOCK_ABORT_PENDING/ TBLOCK_ABORT_END
PrepareTransactionBlock	显式执行 PREPARE 语句，将上层事务状态机变为 TBLOCK_PREPARE
DefineSavepoint	执行 SAVEPOINT 语句，通过调用 PushTransaction 将子事务上层事务状态机变为 TBLOCK_SUBBEGIN
ReleaseSavepoint	执行 RELEASE SAVEPOING 语句，将子事务上层状态机转变为 TBLOCK_SUBRELEASE
RollbackToSavepoint	执行 ROLLBACK TO 语句，将所有子事务上层状态机转变为 TBLOCK_SUBABORT_PENDING/ TBLOCK_SUBABORT_END，顶层事务的上层状态机转变为 TBLOCK_SUBABORT_RESTART

5.2.2 事务 ID 分配及 CLOG/CSNLOG

为了在数据库内部区别不同的事务，openGauss 会为它们分配唯一的标识符，即事务 ID(XID)，XID 是 uint64 单调递增的序列。当事务结束后，使用 CLOG 记录是否提交，使用 CSNLOG(Commit Sequence Number Log)记录该事务提交的序列，用于可见性判断。

1. 64 位 XID 及其分配

openGauss 对每个写事务均会分配一个唯一标识。当事务插入时，会将事务信息写到元组头部的 xmin，代表插入该元组的 XID；当事务进行更新和删除时，会将当前事务信息写到元组头部的 xmax，代表删除该元组的 XID。当前事务 ID 的分配采用的是 uint64 单调递增序列，为了节省空间并兼容老的版本，当前设计是将元组头部的 xmin/xmax 分成两部分存储，元组头部的 xmin/xmax 均为 uint32 的数字，页面的头部存储 64 位的 xid_base，为当前页面的 xid_base。

元组结构如图 5-8 所示，页面头结构如图 5-9 所示，那么对于每条元组真正的 xmin、xmax 计算公式即为：元组头中 xmin/xmax ＋ 页面 xid_base。

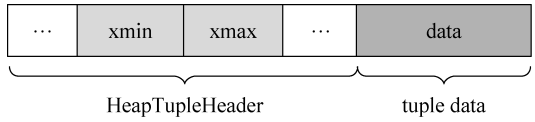

图 5-8　元组结构

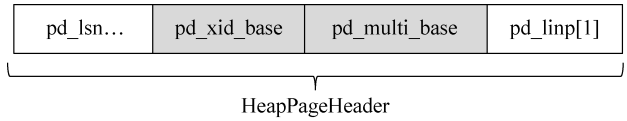

图 5-9　页面头结构

当页面不断有更大的 XID 插入时，可能超过“xid_base + 2^{32}”，此时需要通过调节 xid_base 来满足所有元组的 xmin/xmax 都可以通过该值及元组头部的值计算出来，详细逻辑见“3. 关键函数”。

为了使 XID 不消耗过快，openGauss 当前只对写事务进行 XID 的分配，只读事务不会额外分配 XID，也就是说并不是任何事务一开始都会分配 XID，只有真正使用 XID 时才会去分配。在分配子事务 XID 时，如果父事务还未分配 XID，则会先给父事务分配 XID，再给子事务分配 XID，确保子事务的 XID 比父事务大。理论上 64 位 XID 已经足够使用：假设数据库的 tps 为 1000 万，即 1 秒处理 1000 万个事务，64XID 可以使用 58 万年。

2. CLOG、CSNLOG

CLOG 和 CSNLOG 分别维护事务 ID→CommitLog 及事务 ID→CommitSeqNoLog 的映射关系。由于内存的资源有限，并且系统中可能会有长事务存在，内存中可能无法存放所有的映射关系，此时需要将这些映射写盘成物理文件，所以产生了 CLOG（XID→CommitLog Map）、CSNLOG（XID→CommitSeqNoLog Map）文件。CSNLOG 和 CLOG 均采用了 SLRU（Simple Least Recently Used，简单最近最少使用）机制来实现文件的读取及刷盘操作。

1）CLOG

CLOG 用于记录事务 ID 的提交状态。openGauss 中对于每个事务 ID 使用 2 个 bit 位 4 种状态来标识它的状态。CLOG 定义代码如下：

```
#define CLOG_XID_STATUS_IN_PROGRESS 0x00 表示事务未开始或还在运行中(故障场景可能是crash)
#define CLOG_XID_STATUS_COMMITTED 0x01 表示该事务已经提交
#define CLOG_XID_STATUS_ABORTED 0x02 表示该事务已经回滚
#define CLOG_XID_STATUS_SUB_COMMITTED 0x03 表示子事务已经提交而父事务状态未知
```

CLOG 页面的物理组织形式如图 5-10 所示。

图 5-10 表示事务 1、4、5 还在运行中，事务 2 已经提交，事务 3 已经回滚。

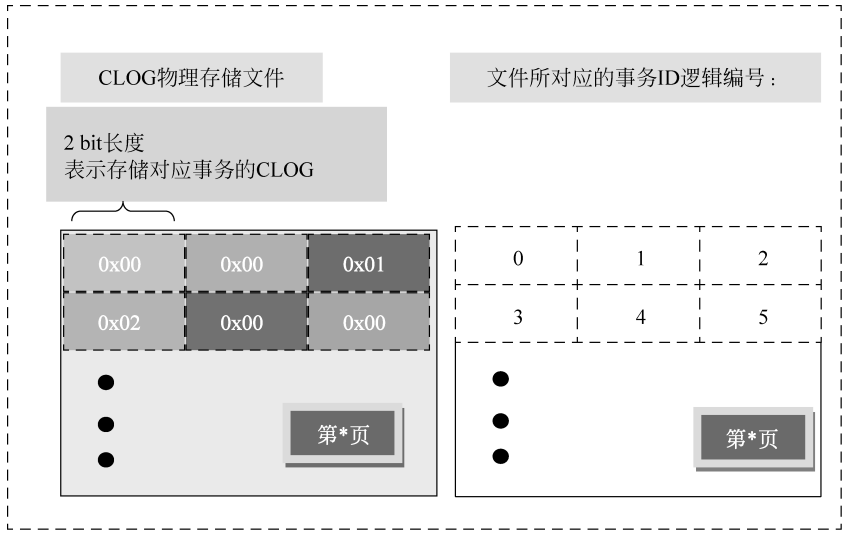

图 5-10　CLOG 页面的物理组织形式

2）CSNLOG

CSNLOG 用于记录事务提交的序列号。openGauss 为每个事务 ID 分配 8 字节 uint64 的 CSN 号，所以一个 8K 页面能保存 1K 个事务的 CSN 号。CSNLOG 达到一定大小后会分块，每个 CSNLOG 文件块的大小为 256KB。同 XID 号类似，CSN 号预留了几个特殊的号。CSNLOG 定义代码如下：

```
#define COMMITSEQNO_INPROGRESS UINT64CONST(0x0) 表示该事务还未提交或回滚
#define COMMITSEQNO_ABORTED UINT64CONST(0x1) 表示该事务已经回滚
#define COMMITSEQNO_FROZEN UINT64CONST(0x2) 表示该事务已提交，且对任何快照可见
#define COMMITSEQNO_FIRST_NORMAL UINT64CONST(0x3) 事务正常的 CSN 号起始值
#define COMMITSEQNO_COMMIT_INPROGRESS (UINT64CONST(1) << 62) 事务正在提交中
```

同 CLOG 相似，CSNLOG 的物理结构体如图 5-11 所示。

事务 ID 2048、2049、2050、2051、2052、2053 的对应的 CSN 号依次是 5、4、7、10、6、8，也就是说事务提交的次序依次是 2049→2048→2052→2050→2053→2051。

3. 关键函数

64 位 XID 页面 xid_base 的计算函数：

（1）heap_page_prepare_for_xid 函数：在对页面有写入操作时调用，用来调节 xid_base。

① 新来 XID 在“xid_base + FirstNormalxid”与“xid_base + MaxShortxid (0xFFFFFFFF)”之间时，当前的 xid_base 不需要调整。

② 新来 XID 在“xid_base + FirstNormalxid”左侧（XID 小于该值）时，需要减小 xid_base。

③ 新来 XID 在“xid_base + MaxShortxid”右侧（XID 大于该值）时，需要增加 xid_base。

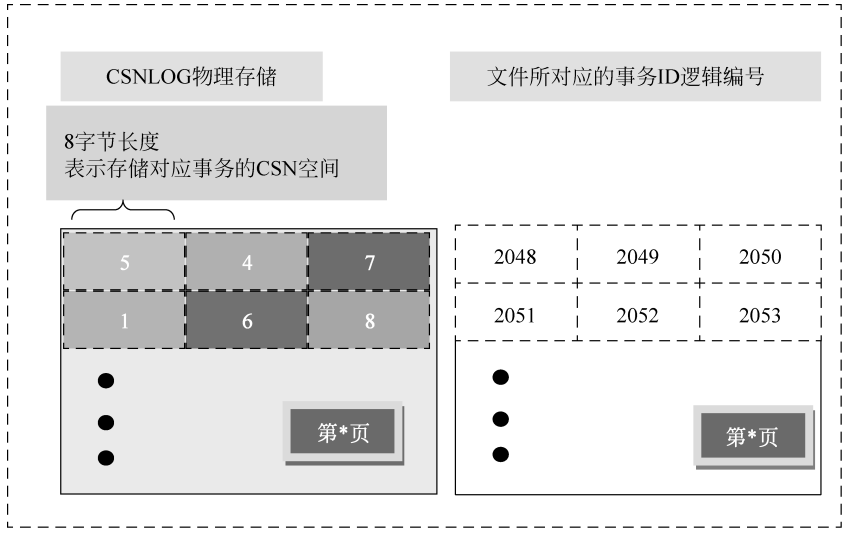

图 5-11　CSNLOG 的物理结构体

④ 特殊情况下，页面的 XID 跨度大于 32 位能表示的范围时，就需要冻结本页面上较小的 XID，即将提交的 XID 设为 FrozenTransactionId(2)，该值对所有事务均可见；将回滚的 XID 设为 InvalidTransactionId(0)，该值对所有的事务均不可见。

(2) freeze_single_heap_page 函数：对该页面上较小的 XID 进行冻结操作。

① 计算 oldestxid，比该值小的事务已经无任何事务访问更老的版本，此时可以将提交的 XID 直接标记为 FrozenTransactionId，即对所有事务可见；将回滚的 XID 标记为 InvalidTransactionId，即对所有事务不可见。

② 页面整理，清理 hot update 链，重定向 itemid，整理页面空间。

③ 根据 oldestxid 处理各个元组。

(3) heappage_shift_base 函数：更新 xid_base，调整页面中各个元组头中的 xmin/xmax。

(4) GetNewTransactionId 函数：获取最新的事务 ID。

5.2.3　MVCC 可见性判断机制

openGauss 利用多版本并发控制来维护数据的一致性。当扫描数据时，每个事务看到的只是拿快照那一刻的数据，而不是数据当前的最新状态。这样就可以避免一个事务看到其他并发事务的更新而导致不一致的场景。使用多版本并发控制的主要优点是，读取数据的锁请求与写数据的锁请求不冲突，以此来实现读不阻塞写，写也不阻塞读。下面介绍事务隔离级别及 CSN 机制。

1. 事务隔离级别

SQL 标准考虑了并行事务间应避免的现象，定义了几种隔离级别，如表 5-5 所示。

表 5-5 事务隔离级别

隔离级别	P0：脏写	P1：脏读	P2：不可重复读	P3：幻读
读未提交	不可能	可能	可能	可能
读已提交	不可能	不可能	可能	可能
可重复读	不可能	不可能	不可能	可能
可串行化	不可能	不可能	不可能	不可能

(1) 脏写(dirty write)：两个事务分别写入，两个事务分别提交或回滚，则事务的结果无法确定，即一个事务可以回滚另一个事务的提交。

(2) 脏读(dirty read)：一个事务可以读取另一个事务未提交的修改数据。

(3) 不可重复读(fuzzy read)：一个事务重复读取前面读取过的数据，数据的结果被另外的事务修改。

(4) 幻读(phantom)：一个事务重复执行范围查询，返回一组符合条件的数据，每次查询的结果集因为其他事务的修改发生改变(条数)。

在各类数据库实现的过程中，并发事务产生了一些新的现象，在原来的隔离级别的基础上，有了一些扩展，如表 5-6 所示。

表 5-6 事务隔离级别扩展

隔离级别	P0：脏写	P1：脏读	P4：更新丢失	P2：不可重复读	P3：幻读	A5A：读偏斜	A5B：写偏斜
读未提交	不可能	可能	可能	可能	可能	可能	可能
读已提交	不可能	不可能	可能	可能	可能	可能	可能
可重复读	不可能	不可能	不可能	不可能	可能	不可能	不可能
快照一致性读	不可能	不可能	不可能	不可能	偶尔	不可能	可能
可串行化	不可能	不可能	不可能	不可能	不可能	不可能	不可能

(5) 更新丢失(lost update)：一个事务在读取元组并更新该元组的过程中，有另一个事务修改了该元组的值，导致最终这次修改丢失。

(6) 读偏斜(read skew)：假设数据 x、y 有隐式的约束 $x+y=100$；事务一读取 $x=50$；事务二写 $x=25$ 并更新 $y=75$ 保证约束成立，事务二提交，事务一再读取 $y=75$，导致事务一中读取 $x+y=125$，不满足约束。

(7) 写偏斜(write skew)：假设数据 x、y 有隐式的约束 $x+y\leqslant 100$；事务一读取 $x=50$，并写入 $y=50$；事务二读取 $y=30$ 并写入 $x=70$，并提交；事务一再提交；最终导致 $x=70$，$y=50$ 不满足 $x+y\leqslant 100$ 的约束。

openGauss 提供读已提交隔离级别和可重复读隔离级别：在实现上可重复读隔离级别

无幻读问题，有 A5B 写偏斜问题。

2. CSN 机制

1）CSN 原理

CSN 原理如图 5-12 所示。

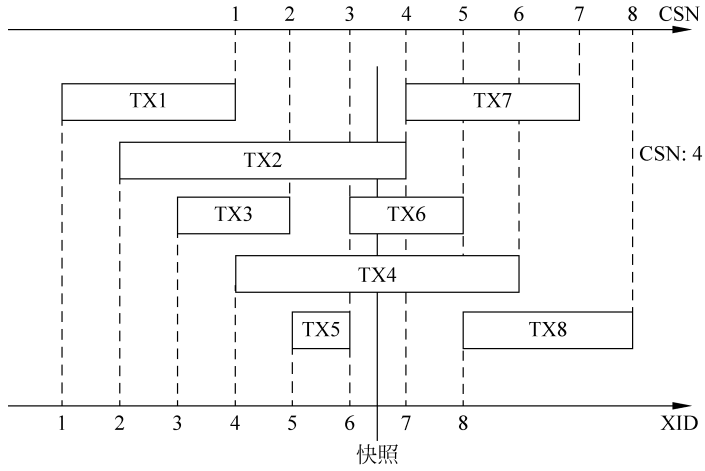

图 5-12　CSN 原理

每个非只读事务在运行过程中会取得一个 XID 号，在事务提交时会推进 CSN，同时会将当前 CSN 与事务的 XID 映射关系保存起来(CSNLOG)。在图 5-12 中，实心竖线表示取 snapshot(快照)时刻，会获取最新提交 CSN(3)的下一个值 4。TX1、TX3、TX5 已经提交，对应的 CSN 号分别是 1、2、3。TX2、TX4、TX6 正在运行，TX7、TX8 是还未开启的事务。对于当前快照而言，严格小于 CSN 号 4 的事务提交结果均可见；其余事务提交结果在获取快照时刻还未提交，不可见。

2）MVCC 快照可见性判断的流程

获取快照时记录当前活跃的最小的 XID，记为 snapshot. xmin；当前最新提交的“事务 ID(latestCompleteXid) + 1”记为 snapshot. xmax；当前最新提交的“CSN 号 + 1”(NextCommitSeqNo)记为 snapshot. csn。MVCC 快照可见性判断的简易流程如图 5-13 所示。

(1) XID 大于或等于 snapshot. xmax 时，该事务 ID 不可见。

(2) XID 比 snapshot. xmin 小时，说明该事务 ID 在本次事务启动前已经结束，需要查询事务的提交状态，并在元组头上设置相应的标记位。

(3) XID 处于 snapshot. xmin 和 snapshot. xmax 之间时，需要从 CSN-XID 映射中读取事务结束的 CSN；如果 CSN 有值且比 snapshot. csn 小，表示该事务可见，否则不可见。

3）提交流程

事务提交流程如图 5-14 所示。

(1) 设置 CSN-XID 映射 commit-in-progress 标记。

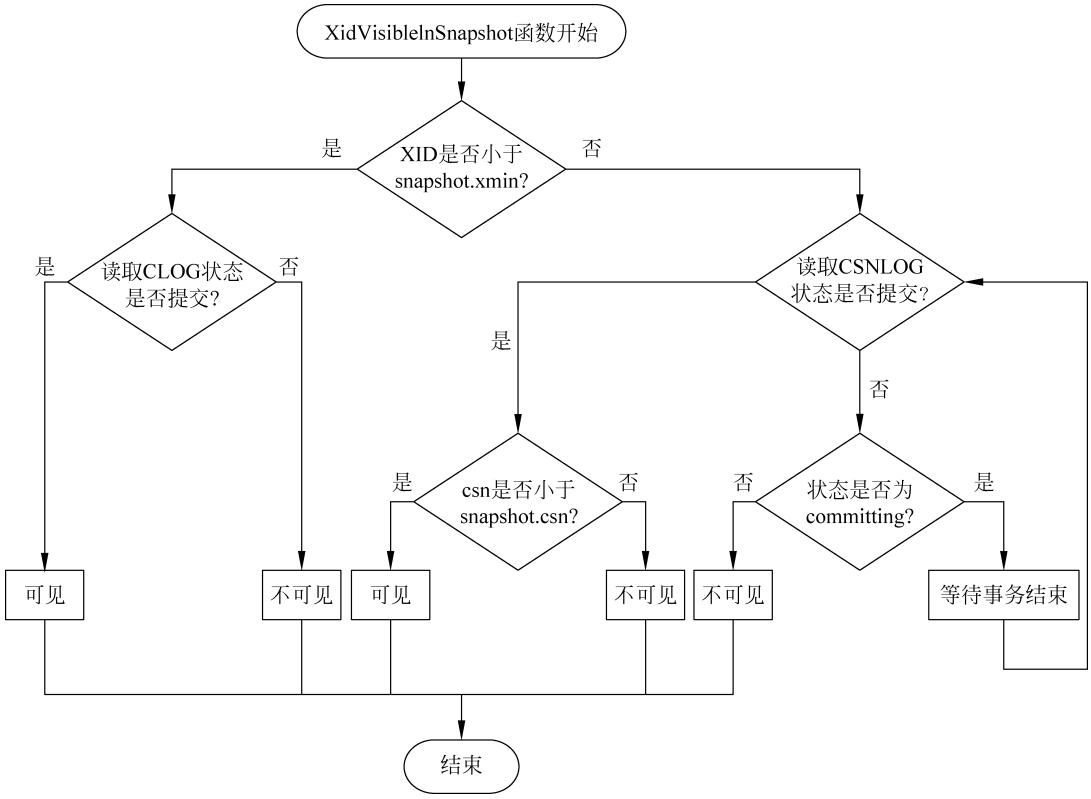

图 5-13　MVCC 快照可见性判断的简易流程

(2) 原子更新 NextCommitSeqNo 值。

(3) 生成 redo 日志,写 CLOG,写 CSNLOG。

(4) 更新 PGPROC,将对应的事务信息从 PGPROC 中移除,XID 设置为 InvalidTransactionId,xmin 设置为 InvalidTransactionId。

4) 热备支持

在事务的提交流程步骤(1)与(2)之间,增加 commit-in-progress 的 XLOG 日志。备机在读快照时,首先获取轻量锁 ProcArrayLock,并计算当前快照。如果使用当前快照中的 CSN,碰到 XID 对应的 CSN 号有 COMMITSEQNO_COMMIT_INPROGRESS 标记,则必须等待相应的事务提交 XLOG 回放结束后再读取相应的 CSN 判断是否可见。为了实现上述等待操作,备机在对 commit-in-progress 的 XLOG 日志做 redo 操作时,会调用 XactLockTableInsert 函数获取相应 XID 的事务排他锁;其他的读事务如果访问到该 XID,会等待在此 XID 的事务锁上,直到相应的事务提交 XLOG 回放结束后再继续运行。

3. 关键数据结构及函数

1) 快照

快照相关代码如下:

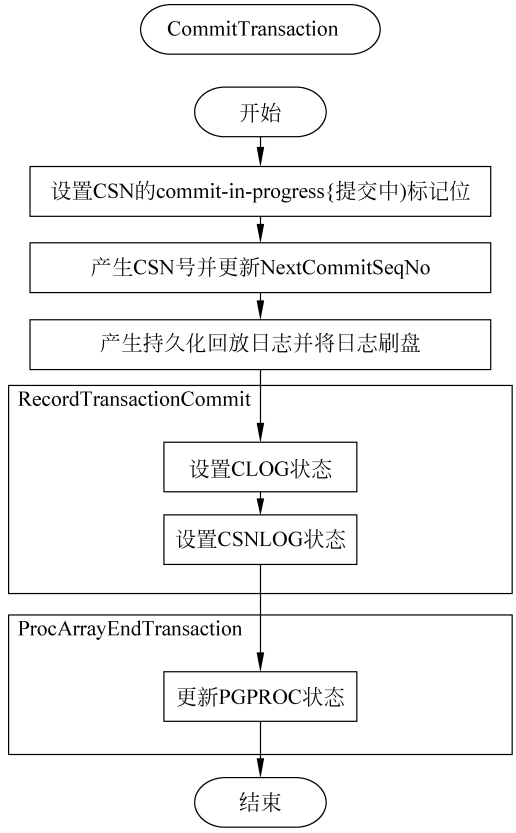

图 5-14　提交流程

```
typedef struct SnapshotData {
     SnapshotSatisfiesFunc  satisfies;       /* 判断可见性的函数;通常使用 MVCC,即
HeapTupleSatisfiesMVCC */
    TransactionId xmin;      /* 当前活跃事务最小值,小于该值的事务说明已结束 */
    TransactionId xmax;      /* 最新提交事务 ID(latestCompeleteXid) + 1,大于或等于该值说明
事务还未开始,该事务 ID 不可见 */
    TransactionId *  xip;    /* 记录当前活跃事务链表,在 CSN 版本中该值无用 */
    TransactionId *  subxip; /* 记录缓存子事务活跃链表,在 CSN 版本中该值无用 */
    uint32 xcnt;             /* 记录活跃事务的个数(xip 中元组数),在 CSN 版本中该值无用 */
    GTM_Timeline timeline;   /* openGauss 单机中无用 */
    uint32 max_xcnt;         /* xip 的最大个数,CSN 版本中该值无用 */
    int32 subxcnt;         /* 缓存子事务活跃链表的个数,在 CSN 版本中该值无用 */
    int32 maxsubxcnt;      /* 缓存子事务活跃链表最大个数,在 CSN 版本中该值无用 */
    bool suboverflowed;    /* 子事务活跃链表是否已超过共享内存中预分配的上限,在 CSN 版本中
无用 */

    CommitSeqNo snapshotcsn;       /* 快照的 CSN 号,一般为最新提交事务的 CSN 号 + 1
(NextCommitSeqNo),CSN 号严格小于该值的事务可见 */

    int prepared_array_capacity;      /* 单机 openGauss 无用 */
```

```
    int prepared_count;              /* 单机 openGauss 无用 */
    TransactionId* prepared_array;   /* 单机 openGauss 无用 */

    bool takenDuringRecovery;        /* 是否 Recovery 过程中产生的快照 */
    bool copied;      /* 该快照是会话级别静态的,还是新分配内存复制的 */

    CommandId curcid;      /* 事务块中的命令序列号,即同一事务中,前面插入的数据后面可
见 */
    uint32 active_count;             /* ActiveSnapshot stack 的 refcount */
    uint32 regd_count;               /* RegisteredSnapshotList 的 refcount */
    void* user_data;      /* 本地多版本快照使用,标记该快照还有线程使用,不能直接释放 */
    SnapshotType snapshot_type;      /* openGauss 单机无用 */
} SnapshotData;
```

2）HeapTupleSatisfiesMVCC

用于一般读事务的快照扫描，基于 CSN 的大体逻辑，详细代码如下：

```
bool HeapTupleSatisfiesMVCC(HeapTuple htup, Snapshot snapshot, Buffer buffer)
{
    ... /* 初始化变量 */

    if (!HeapTupleHeaderXminCommitted(tuple)) {    /* 此处先判断用 1bit 记录的 hint bit(提示
比特位. openGauss 判断可见性时,通常需要知道元组 xmin 和 xmax 对应的 CLOG 的提交状态; 为了避
免重复访问 CLOG, openGauss 内部对可见性判断进行了优化. hint bit 把事务状态直接记录在元组头
中,用 1bit 来表示提交和回滚状态. openGauss 不会在事务提交或回滚时主动更新元组上的 hint
bit,而是等到访问该元组并进行可见性判断时,如果发现 hint bit 没有设置,则在 CLOG 中读取并设
置,否则直接读取 hint bit 值. ),防止同一条 tuple 反复获取事务最终提交状态. 如果一次扫描发现
该元组的 xmin/xmax 已经提交,就会打上相应的标记,加速扫描;如果没有标记则继续判断 */
        if (HeapTupleHeaderXminInvalid(tuple))   /* 同样判断 hint bit. 如果 xmin 已经标记为
invalid 说明插入该元组的事务已经回滚,直接返回不可见 */
            return false;

         if (TransactionIdIsCurrentTransactionId(HeapTupleHeaderGetXmin(page, tuple))) {
/* 如果是一个事务内部,需要去判断该元组的 CID,即同一个事务内,后面可以查到当前事务之前插
入的扫描结果 */
           ....
        } else {      /* 如果扫描其他事务,需要根据快照判断事务是否可见 */
            visible = XidVisibleInSnapshot(HeapTupleHeaderGetXmin(page, tuple), snapshot,
&hintstatus); /* 通过 CSNLOG 判断事务是否可见,并且返回该事务的最终提交状态 */
            if (hintstatus == XID_COMMITTED)     /* 如果该事务提交,则打上提交的 hint bit
用于加速判断 */
                SetHintBits(tuple, buffer, HEAP_XMIN_COMMITTED, HeapTupleHeaderGetXmin
(page, tuple));

            if (hintstatus == XID_ABORTED) {
                ... /* 如果事务回滚,则打上回滚标记 */
                SetHintBits(tuple, buffer, HEAP_XMIN_INVALID, InvalidTransactionId);
            }
            if (!visible) { /* 如果 xmin 不可见,则该元组不可见,否则表示插入该元组的事务对
于该次快照已经提交,继续判断删除该元组的事务是否对该次快照提交 */
```

```
                    return false;
                }
            }
        }
    } else { /* 如果该条元组的 xmin 已经被打上提交的 hint bit,则通过函数接口
CommittedXidVisibleInSnapshot 判断是否对本次快照可见 */
        /* xmin is committed, but maybe not according to our snapshot */
        if (!HeapTupleHeaderXminFrozen(tuple) &&
                ! CommittedXidVisibleInSnapshot ( HeapTupleHeaderGetXmin ( page,  tuple ),
snapshot)) {
            return false;
        }
    }
   ... /* 后续 xmax 的判断与 xmin 类似,如果 xmax 对于本次快照可见,则说明删除该条元组的事务
已经提交 */
    if (!(tuple->t_infomask & HEAP_XMAX_COMMITTED)) {
        if (TransactionIdIsCurrentTransactionId(HeapTupleHeaderGetXmax(page, tuple))) {
            if (HeapTupleHeaderGetCmax(tuple, page) >= snapshot->curcid)
                return true;      /* 在扫描前该删除事务已经提交 */
            else
                return false;     /* 扫描开始后删除操作的事务才提交 */
        }

         visible = XidVisibleInSnapshot(HeapTupleHeaderGetXmax(page, tuple), snapshot,
&hintstatus);
        if (hintstatus == XID_COMMITTED) {
            /* 设置 xmax 的 hint bit */
             SetHintBits(tuple, buffer, HEAP_XMAX_COMMITTED, HeapTupleHeaderGetXmax(page,
tuple));
        }
        if (hintstatus == XID_ABORTED) {
            /* 回滚或者故障 */
            SetHintBits(tuple, buffer, HEAP_XMAX_INVALID, InvalidTransactionId);
        }
        if (!visible) {
            return true; /* 快照中 xmax 对应的事务不可见,则认为该元组仍然活跃 */
        }
    } else {
        /* xmax 对应的事务已经提交,但是快照中该事务不可见,认为删除该元组的操作未完成,
仍然认为该元组可见 */
            if  (! CommittedXidVisibleInSnapshot ( HeapTupleHeaderGetXmax ( page,  tuple ),
snapshot)) {
            return true; /* 认为元组可见 */
        }
    }
    return false;
}
```

3）HeapTupleSatisfiesNow

该函数的逻辑与 MVCC 类似,只是此时并没有统一快照,而仅仅是判断当前 xmin/xmax

的状态，而不再继续调用 XidVisibleInSnapshot 函数、CommittedXidVisibleInSnapshot 函数来判断是否对快照可见。

4）HeapTupleSatisfiesVacuum

根据传入的 OldestXmin 值返回相应的状态。死亡元组（openGauss 多版本机制中不可见的旧版本元组）且没有任何其他未结束的事务可能访问该元组（xmax＜oldestXmin），可以被 VACUUM 清理。本函数具体代码如下：

```
HTSV_Result HeapTupleSatisfiesVacuum(HeapTuple htup, TransactionId OldestXmin, Buffer
buffer)
{
    … /* 初始化变量 */
    if (!HeapTupleHeaderXminCommitted(tuple)) {      /* hint bit 标记加速，与 MVCC 的逻辑相
同 */
        if (HeapTupleHeaderXminInvalid(tuple))      /* 如果 xmin 未提交，则返回该元组死亡，
可以清理 */
            return HEAPTUPLE_DEAD;
         xidstatus = TransactionIdGetStatus(HeapTupleGetRawXmin(htup), false); /* 通过
CSNLOG 来获取当前的事务状态 */
        if (xidstatus == XID_INPROGRESS) {
            if (tuple->t_infomask & HEAP_XMAX_INVALID) /* 如果 xmax 还没有，说明没有人删
除，此时判断该元组正在插入过程中，否则在删除过程中 */
                return HEAPTUPLE_INSERT_IN_PROGRESS;
            return HEAPTUPLE_DELETE_IN_PROGRESS; /* 返回正在删除的过程中 */
        } else if (xidstatus == XID_COMMITTED) { /* 如果 xmin 提交了，打上 hint bit，后面继
续看 xmax 是否提交 */
            SetHintBits(tuple, buffer, HEAP_XMIN_COMMITTED, HeapTupleGetRawXmin(htup));
        } else {
   …. /* 事务结束了且未提交，可能是回滚(abort)或者是宕机(crash)的事务，一般返回死亡，可
删除；单机情形 t_thrd.xact_cxt.useLocalSnapshot 没有作用，恒为 false */
            SetHintBits(tuple, buffer, HEAP_XMIN_INVALID, InvalidTransactionId);
             return ((!t_thrd.xact_cxt.useLocalSnapshot || IsInitdb) ? HEAPTUPLE_DEAD :
HEAPTUPLE_LIVE);
        }
    }
  /* 接着判断 xmax. 如果还没有设置 xmax 说明没有人删除该元组，返回元组存活，不可删除 */
    if (tuple->t_infomask & HEAP_XMAX_INVALID)
        return HEAPTUPLE_LIVE;
…
    if (!(tuple->t_infomask & HEAP_XMAX_COMMITTED)) {      /* 如果 xmax 提交，则看 xmax 是否
比 oldesxmin 小. 小的话说明没有未结束的事务会访问该元组，可以删除 */
        xidstatus = TransactionIdGetStatus(HeapTupleGetRawXmax(htup), false);
        if (xidstatus == XID_INPROGRESS)
            return HEAPTUPLE_DELETE_IN_PROGRESS;
         else if (xidstatus == XID_COMMITTED)
            SetHintBits(tuple, buffer, HEAP_XMAX_COMMITTED, HeapTupleGetRawXmax(htup));
        else {
… /* xmax 对应的事务回滚或者宕机 */
            SetHintBits(tuple, buffer, HEAP_XMAX_INVALID, InvalidTransactionId);
```

```
                return HEAPTUPLE_LIVE;
            }
        }

    /* 判断该元组是否可以删除,xmax < OldestXmin 可以删除 */
        if (!TransactionIdPrecedes(HeapTupleGetRawXmax(htup), OldestXmin))
            return ((!t_thrd.xact_cxt.useLocalSnapshot || IsInitdb) ? HEAPTUPLE_RECENTLY_DEAD
 : HEAPTUPLE_LIVE);

        /* 该元组可以认为已经死亡,不被任何活跃事务访问,可以删除 */
        return ((!t_thrd.xact_cxt.useLocalSnapshot || IsInitdb) ? HEAPTUPLE_DEAD : HEAPTUPLE_
LIVE);
    }
```

5）SetXact2CommitInProgress

设置 XID 对应 CSNLOG 的标记位 COMMITSEQNO_COMMIT_INPROGRESS(详见 5.2.2 节),表示此 XID 对应的事务正在提交过程中。该操作是为了保证可见性判断时的原子性,即防止并发读事务在 CSN 号设置的过程中读到不一致的数据。

6）CSNLogSetCommitSeqNo

给对应的 XID 设置相应的 CSNLOG。

7）RecordTransactionCommit

记录事务提交,主要是写 CLOG、CSNLOG 及它们的 XLOG 日志。

5.2.4　进程内多线程管理机制

本节简述进程内多线程管理机制相关数据结构及多版本快照计算机制。

1. 事务信息管理

数据库启动时维护了一段共享内存,每个线程初始化时都会从这个共享内存中获取一个槽位并将其线程信息记录到槽位中。获取快照时,需要在共享内存数组中更新槽位信息,事务结束时,需要从槽位中将事务信息清除。计算快照时,通过遍历该全局数组,获取当前所有并发线程的事务信息,并计算出快照信息(xmin、xmax、snapshotcsn 等)。事务信息管理的关键数据结构代码如下:

```
typedef struct PGXACT {
    GTM_TransactionHandle handle;       /* 单机模式无用参数 */
    TransactionId xid;                  /* 该线程持有的 XID 号,如果没有则为 0 */
    TransactionId prepare_xid;          /* 准备阶段的 XID 号 */

TransactionId xmin; /* 当前事务开启时最小的活跃 XID,vaccum 操作不会删除那些 XID 大于或等于
                       xmin 的元组 */
    CommitSeqNo csn_min;                /* 当前事务开启时最小的活跃 CSN 号 */
    TransactionId next_xid;             /* 单机模式无用参数 */
    int nxids;                          /* 子事务个数 */
    uint8 vacuumFlags;                  /* vacuum 操作相关的 flag */
```

```
bool needToSyncXid;                    /* 单机模式无用参数 */
bool delayChkpt;                   /* 如果该线程需要 checkpoint 线程延迟等待,此值为 true */
#ifdef __aarch64__
    char padding[PG_CACHE_LINE_SIZE - PGXACT_PAD_OFFSET]; /* 为了性能考虑的结构体对齐 */
#endif
} PGXACT;

struct PGPROC {
    SHM_QUEUE links;                       /* 链表中的指针 */

    PGSemaphoreData sem;                   /* 休眠等待的信号量 */
    int waitStatus;                        /* 等待状态 */

    Latch procLatch;                       /* 线程的通用闩锁 */

    LocalTransactionId lxid;               /* 当前线程本地顶层事务 ID */
    ThreadId pid;                          /* 线程的 PID */

    ThreadId sessMemorySessionid;
    uint64 sessionid;                      /* 线程池模式下当前的会话 ID */
    int logictid;                          /* 逻辑线程 ID */
    TransactionId gtt_session_frozenxid;   /* 会话级全局临时表的冻结 XID */

    int pgprocno;
    int nodeno;

    /* 线程启动时下面这些数据结构为 0 */
    BackendId backendId;                   /* 线程的后台 ID */
    Oid databaseId;                        /* 当前访问数据库的 OID */
    Oid roleId;                            /* 当前用户的 OID */

    /* 版本号,用于升级过程中新老版本的判断 */
    uint32 workingVersionNum;

    /* 热备模式下,标记当前事务是否收到冲突信号.设置该值时需要持有 ProcArray 锁 */
    bool recoveryConflictPending;

    /* 线程等待的轻量级锁信息. */
    bool lwWaiting;                    /* 当等待轻量级锁时,为真 */
    uint8 lwWaitMode;                  /* 预获取锁的模式 */
    bool lwIsVictim;                   /* 强制放弃轻量级锁 */
    dlist_node lwWaitLink;             /* 等待在相同轻量级锁对象的下一个等待者 */

    /* 线程等待的常规锁信息 */
    LOCK * waitLock;                   /* 等待的常规锁对象 */
    PROCLOCK * waitProcLock;           /* 等待常规锁对象的持有者 */
    LOCKMODE waitLockMode;             /* 预获取常规锁对象的模式 */
    LOCKMASK heldLocks;                /* 本线程获取锁对象模式的位掩码 */

    /* 等待主备机回放日志同步的信息 */
```

```
    XLogRecPtr waitLSN;                 /* 等待的 lsn */
    int syncRepState;                   /* 等待主备同步的状态 */
    bool syncRepInCompleteQueue;        /* 是否等待在完成队列中 */
    SHM_QUEUE syncRepLinks;             /* 指向同步队列的指针 */

    DataQueuePtr waitDataSyncPoint;     /* 数据页复制的数据同步点 */
    int dataSyncRepState;               /* 数据页复制的同步状态 */
    SHM_QUEUE dataSyncRepLinks;         /* 指向数据页同步队列的指针 */

    MemoryContext topmcxt;              /* 本线程的顶层内存上下文 */
    char myProgName[64];
    pg_time_t myStartTime;
    syscalllock deleMemContextMutex;

    SHM_QUEUE myProcLocks[NUM_LOCK_PARTITIONS];

    /* 以下结构为了实现 XID 批量提交 */
    /* 是否为 XID 批量提交中的成员 */
    bool procArrayGroupMember;
    /* XID 批量提交中的下一个成员 */
    pg_atomic_uint32 procArrayGroupNext;
    /* 父事务 XID 和子事物 XID 中的最大者 */
    TransactionId procArrayGroupMemberXid;

    /* 提交序列号 */
    CommitSeqNo commitCSN;
    /* 以下结构为了实现 CLOG 批量提交 */
    bool clogGroupMember;                       /* 是否为 CLOG 批量提交中的成员 */
    pg_atomic_uint32 clogGroupNext;             /* CLOG 批量提交中的下一个成员 */
    TransactionId clogGroupMemberXid;           /* CLOG 批量提交的事务 ID */
    CLogXidStatus clogGroupMemberXidStatus;     /* CLOG 批量提交的事务状态 */
    int64 clogGroupMemberPage;                  /* CLOG 批量提交对应的 CLOG 页面 */
    XLogRecPtr clogGroupMemberLsn;  /* CLOG 批量提交成员的提交回放日志位置 */
#ifdef __aarch64__
    /* 以下结构体是为了实现 ARM 架构下回放日志批量插入 */
    bool xlogGroupMember;
    pg_atomic_uint32 xlogGroupNext;
    XLogRecData * xlogGrouprdata;
    XLogRecPtr xlogGroupfpw_lsn;
    XLogRecPtr * xlogGroupProcLastRecPtr;
    XLogRecPtr * xlogGroupXactLastRecEnd;
    void * xlogGroupCurrentTransactionState;
    XLogRecPtr * xlogGroupRedoRecPtr;
    void * xlogGroupLogwrtResult;
    XLogRecPtr xlogGroupReturntRecPtr;
    TimeLineID xlogGroupTimeLineID;
    bool * xlogGroupDoPageWrites;
    bool xlogGroupIsFPW;
    uint64 snap_refcnt_bitmap;
#endif
```

```
    LWLock * subxidsLock;
    struct XidCache subxids;      /* 子事务 XID */

    LWLock * backendLock;         /* 每个线程的轻量级锁,用于保护以下数据结构的并发访问 */

    /* Lock manager data, recording fast-path locks taken by this backend. */
    uint64 fpLockBits;                          /* 快速路径锁的持有模式 */
    FastPathTag fpRelId[FP_LOCK_SLOTS_PER_BACKEND];     /* 表对象的槽位 */
    bool fpVXIDLock;                            /* 是否获得本地 XID 的快速路径锁 */
    LocalTransactionId fpLocalTransactionId;    /* 本地的 XID */
};
```

如图 5-15 所示,proc_base_all_procs 和 proc_base_all_xacts 为全局的共享区域,每个线程启动时都会在这个共享区域中注册一个槽位,并且将线程级指针变量 t_thrd.proc 和 t_thrd.pgxact 指向该区域。当该线程有事务开始时,会将对应事务的 xmin、XID 等信息填写到 pgxact 结构体中。关键函数及接口如下:

(1) GetOldestXmin:返回当前多版本快照缓存的 oldestXmin(多版本快照机制见后续章节)。

(2) ProcArrayAdd:线程启动时在共享区域中注册一个槽位。

(3) ProcArrayRemove:将当前线程从 ProcArray 数组中移除。

(4) TransactionIdIsInProgress:判断 XID 是否还在运行中。

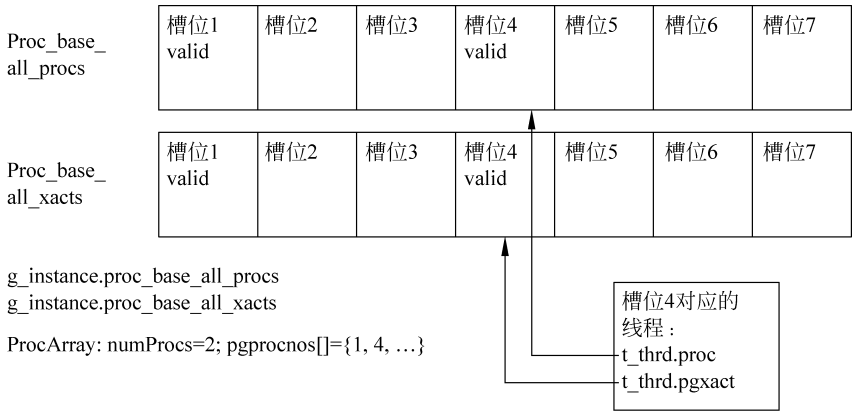

图 5-15 事务信息

2. 多版本快照机制

因为 openGauss 使用一段共享内存来实现快照的获取及各线程事务信息的管理,计算快照持有共享锁及事务结束持有排他锁有严重的锁争抢问题。为了解决该冲突,openGauss 引入了多版本快照机制解决锁冲突。每当事务结束时,持有排他锁、计算快照的一个版本,记录到一个环形缓冲区队列内存里;当其他线程获取快照时,并不持有共享锁去重新计算,而是通过原子操作到该环形队列顶端获取最新快照并将其引用计数加 1;待复

制完快照信息后，将引用计数减 1；当槽位引用计数为 0 时，表示可以被新的快照复用。

1）多版本快照数据结构

多版本快照数据结构代码如下：

```
typedef struct _snapxid {
    TransactionId xmin;
    TransactionId xmax;
    CommitSeqNo snapshotcsn;
    TransactionId localxmin;
    bool takenDuringRecovery;
    ref_cnt_t ref_cnt[NREFCNT];      /* 该快照的引用计数,如果为 0 则可复用 */
} snapxid_t; /* 多版本快照内容,在 openGauss CSN 方案下,仅需要记录 xmin、xmax、snapshotcsn 等
关键信息即可 */

static snapxid_t* g_snap_buffer = NULL;      /* 缓冲区队列内存区指针 */
static snapxid_t* g_snap_buffer_copy = NULL; 缓冲区队列内存的浅复制 */
static size_t g_bufsz = 0;
static bool g_snap_assigned = false;/* 多版本快照缓冲区队列是否已初始化 */

#define SNAP_SZ sizeof(snapxid_t)   /* 每个多版本快照的大小 */
#define MaxNumSnapVersion 64        /* 多版本快照队列的大小,64 个版本 */
static volatile snapxid_t* g_snap_current = NULL;        /* 当前的快照指针 */
static volatile snapxid_t* g_snap_next = NULL;           /* 下一个可用槽位的快照指针 */
```

2）缓冲区队列创建流程

在创建共享内存时，根据 MaxNumSnapVersion 函数的大小生成“MaxNumSnapVersion * SNAP_SZ”大小的共享内存区，并将 g_snap_current 设置为 0 号偏移，g_snap_next 设置为“1 * SNAP_SZ”偏移。

3）多版本快照的计算

（1）获取当前 g_snap_next。

（2）保证当前已持有 Proc 数组的排他锁，进行 xmin、xmax、CSN 等关键结构的计算，并存放到 g_snap_next 中。

（3）寻找下一个 refcount 为 0 可复用的槽位，将 g_snap_current 赋值为 g_snap_next，g_snap_next 赋值为可复用的槽位偏移。

4）多版本快照的获取

（1）获取 g_snap_current 指针并将当前快照槽位的引用计数加 1，防止并发更新快照时被复用。

（2）将当前快照中的信息复制到当前连接的静态快照内存中。

（3）释放当前多版本快照，并将当前快照槽位的引用计数减 1。

5）关键函数

（1）CreateSharedRingBuffer：创建多版本快照共享内存信息。

（2）GetNextSnapXid：获取下一个多版本快照位置。函数代码如下：

```
static inline snapxid_t * GetNextSnapXid()
{
    return g_snap_buffer ? (snapxid_t *)g_snap_next : NULL;
}
```

(3) SetNextSnapXid：获取下一个可用的槽位，并且将当前多版本快照更新到最新。函数代码如下：

```
static void SetNextSnapXid()
{
    if (g_snap_buffer != NULL) {
        g_snap_current = g_snap_next;      /* 将多版本快照更新到最新 */
        pg_write_barrier();              /* 此处是防止 buffer ring 初始化时的 ARM 乱序问题 */
        g_snap_assigned = true;
        snapxid_t * ret = (snapxid_t *)g_snap_current;
        size_t idx = SNAPXID_INDEX(ret);
    loop:      /* 主循环，整体思路是不停遍历多版本槽位信息，一直找到一个 refcout 为 0 的可重用槽位 */
        do {
            ++idx;
            /* 如果发生回卷，则从头再找 */
            if (idx == g_bufsz)
                idx = 0;
            ret = SNAPXID_AT(idx);
            if (IsZeroRefCount(ret)) {
                g_snap_next = ret;
                return;
            }
        } while (ret != g_snap_next);
        ereport(WARNING, (errmsg("snapshot ring buffer overflow.")));
/* 当前多版本快照个数为 64 个，理论上可能是会出现槽位被占满，如果没有空闲槽位，重新遍历即可 */
        goto loop;
    }
}
```

(4) CalculateLocalLatestSnapshot：计算多版本快照信息。函数代码如下：

```
void CalculateLocalLatestSnapshot(bool forceCalc)
{
    …/* 初始化变量 */

    snapxid_t * snapxid = GetNextSnapXid();      /* 设置下一个空闲多版本快照槽位信息 */

    /* 初始化 xmax 为 latestCompletedXid + 1 */
    xmax = t_thrd.xact_cxt.ShmemVariableCache->latestCompletedXid;
    TransactionIdAdvance(xmax);

     /* 并不是每个事务提交都会重新计算 xmin 和 oldestxmin，只有每 1000 个事务或者每隔 1s 才会计算，此时 xmin 及 oldestxmin 一般偏小，但是不影响可见性判断 */
    currentTimeStamp = GetCurrentTimestamp();
    if (forceCalc || ((++snapshotPendingCnt == MAX_PENDING_SNAPSHOT_CNT) ||
```

```
                        (TimestampDifferenceExceeds(snapshotTimeStamp, currentTimeStamp,
CALC_SNAPSHOT_TIMEOUT))) {
        snapshotPendingCnt = 0;
        snapshotTimeStamp = currentTimeStamp;

        /* 初始化 xmin */
        globalxmin = xmin = xmax;

        int * pgprocnos = arrayP->pgprocnos;
        int numProcs;

        /*
          循环遍历 proc 并计算快照相应值
         */
        numProcs = arrayP->numProcs;
         /* 主要流程,遍历 proc_base_all_xacts,将其中 pgxact->xid 的最小值记为 xmin,
pgxact->xmin 的最小值记为 oldestxmin */
        for (index = 0; index < numProcs; index++) {
            int pgprocno = pgprocnos[index];
            volatile PGXACT * pgxact = &g_instance.proc_base_all_xacts[pgprocno];
            TransactionId xid;

            if (pgxact->vacuumFlags & PROC_IN_LOGICAL_DECODING)
                continue;

            /* 对于 autovacuum 的 xmin,跳过,避免长 VACUUM 阻塞脏元组回收 */
            if (pgxact->vacuumFlags & PROC_IN_VACUUM)
                continue;

            /* 用最小的 xmin 来更新 globalxmin */
            xid = pgxact->xmin;

            if (TransactionIdIsNormal(xid) && TransactionIdPrecedes(xid, globalxmin))
                globalxmin = xid;

            xid = pgxact->xid;

            if (!TransactionIdIsNormal(xid))
                xid = pgxact->next_xid;

            if (!TransactionIdIsNormal(xid) || !TransactionIdPrecedes(xid, xmax))
                continue;

            if (TransactionIdPrecedes(xid, xmin))
                xmin = xid;
        }

        if (TransactionIdPrecedes(xmin, globalxmin))
            globalxmin = xmin;

        t_thrd.xact_cxt.ShmemVariableCache->xmin = xmin;
```

```
        t_thrd.xact_cxt.ShmemVariableCache->recentLocalXmin = globalxmin;
    }
    /* 此处给多版本快照信息赋值,xmin、oldestxmin 因为不是及时计算故可能偏小,xmax、CSN 号都是当前的准确值,注意计算快照时必须持有排他锁 */
    snapxid->xmin = t_thrd.xact_cxt.ShmemVariableCache->xmin;
    snapxid->xmax = xmax;
    snapxid->localxmin = t_thrd.xact_cxt.ShmemVariableCache->recentLocalXmin;
    snapxid->snapshotcsn = t_thrd.xact_cxt.ShmemVariableCache->nextCommitSeqNo;
    snapxid->takenDuringRecovery = RecoveryInProgress();
    SetNextSnapXid();      /* 设置当前多版本快照 */
}
```

(5) GetLocalSnapshotData：获取最新的多版本快照供事务使用。函数代码如下：

```
Snapshot GetLocalSnapshotData(Snapshot snapshot)
{
    /* 检查是否有多版本快照.在 recover 启动之前,是没有计算出多版本快照的,此时直接返回 */
    if (!g_snap_assigned || (g_snap_buffer == NULL)) {
        ereport(DEBUG1, (errmsg("Falling back to origin GetSnapshotData: not assigned yet or during shutdown\n")));
        return NULL;
    }
    pg_read_barrier();      /* 为了防止 ringBuffer 初始化时的 ARM 乱序问题 */
    snapxid_t* snapxid = GetCurrentSnapXid(); /* 将当前的多版本快照 refcount++,避免被并发计算新快照的事务重用 */

    snapshot->user_data = snapxid;

    … /* 此处将多版本快照 snapxid 中的信息赋值给快照,注意此处是深复制,因为多版本快照仅有几个变量的关键信息,直接赋值即可,之后就可以将相应的多版本快照 refcount 释放 */
    u_sess->utils_cxt.RecentXmin = snapxid->xmin;
    snapshot->xmin = snapxid->xmin;
    snapshot->xmax = snapxid->xmax;
    snapshot->snapshotcsn = snapxid->snapshotcsn;
    …
    ReleaseSnapshotData(snapshot);      /* 将多版本快照的 refcount 释放,以便可以被重用 */
    return snapshot;
}
```

5.3 锁机制

数据库对公共资源的并发控制是通过锁实现的,根据不同用途,通常可以将锁分为 3 种：自旋锁(spinlock)、轻量级锁(Light Weight Lock,LWLock)和常规锁(或基于这 3 种锁的进一步封装)。使用锁的一般操作流程可以简述为 3 步：加锁、临界区操作、放锁。在保证正确性的情况下,锁的使用及争抢成为制约性能的重要因素,下面先简单介绍 openGauss 中的 3 种锁,最后再着重介绍 openGauss 基于鲲鹏架构所做的锁相关性能优化。

5.3.1　自旋锁

自旋锁一般是使用 CPU 的原子指令 TAS(Test-And-Set)实现的。自旋锁只有两种状态：锁定和解锁。自旋锁最多只能被一个进程持有。自旋锁与信号量的区别在于，当进程无法得到资源时，信号量使进程处于睡眠阻塞状态，而自旋锁使进程处于忙等待状态。自旋锁主要用于加锁时间非常短的场合，比如修改标志或者读取标志字段，在几十个指令之内。在编写代码时，自旋锁的加锁和解锁要保证在一个函数内。自旋锁由编码保证不会产生死锁，没有死锁检测，并且没有等待队列。由于自旋锁消耗 CPU，当使用不当长期持有时会触发内核 core dump(核心转储)，openGauss 中将许多 32/64/128 位变量的更新改用 CAS 原子操作，避免或减少使用自旋锁。

与自旋锁相关的操作主要有下面几个：

(1) SpinLockInit：自旋锁的初始化。

(2) SpinLockAcquire：自旋锁加锁。

(3) SpinLockRelease：自旋锁释放锁。

(4) SpinLockFree：自旋锁销毁并清理相关资源。

5.3.2　轻量级锁

轻量级锁是使用原子操作、等待队列和信号量实现的。轻量级锁存在两种类型：共享锁和排他锁。多个进程可以同时获取共享锁，但排他锁只能被一个进程拥有。当进程无法得到资源时，轻量级锁会使进程处于睡眠阻塞状态。轻量级锁主要用于内部临界区操作比较久的场合，加锁和解锁的操作可以跨越函数，但使用完后要立即释放。轻量级锁应由编码保证不会产生死锁。由于代码复杂度及各类异常处理，openGauss 提供了轻量级锁的死锁检测机制，避免各类异常场景产生的轻量级锁死锁问题。

与轻量级锁相关的函数有如下几个。

(1) LWLockAssign：申请一个轻量级锁。

(2) LWLockAcquire：加锁。

(3) LWLockConditionalAcquire：条件加锁，如果没有获取锁则返回 false，并不一直等待。

(4) LWLockRelease：释放锁。

(5) LWLockReleaseAll：释放拥有的所有锁。当事务过程中出错了，会将持有的所有轻量级锁全部回滚释放，避免残留阻塞后续操作。

相关结构体代码如下：

```
#define LW_FLAG_HAS_WAITERS ((uint32)1 << 30)
#define LW_FLAG_RELEASE_OK ((uint32)1 << 29)
#define LW_FLAG_LOCKED ((uint32)1 << 28)

#define LW_VAL_EXCLUSIVE ((uint32)1 << 24)
```

```
#define LW_VAL_SHARED 1                    /* 用于标记轻量级锁的状态,实现锁的获取和释放 */

typedef struct LWLock {
    uint16 tranche;                        /* 轻量级锁的 ID 标识 */
    pg_atomic_uint32 state;                /* 锁的状态位 */
    dlist_head waiters;                    /* 等锁线程的链表 */
#ifdef LOCK_DEBUG
    pg_atomic_uint32 nwaiters;             /* 等锁线程的个数 */
    struct PGPROC * owner;                 /* 最后独占锁的持有者 */
#endif
#ifdef ENABLE_THREAD_CHECK
    pg_atomic_uint32 rwlock;
    pg_atomic_uint32 listlock;
#endif
} LWLock;
```

5.3.3 常规锁

常规锁是使用哈希表实现的。常规锁支持多种锁模式(lock modes),这些锁模式之间的语义和冲突是通过冲突表定义的。常规锁主要用于业务访问的数据库对象加锁。常规锁的加锁遵守数据库的两阶段加锁协议,即访问过程中加锁,事务提交时释放锁。

常规锁有等待队列并提供了死锁检测机制,当检测到死锁发生时选择一个事务进行回滚。

openGauss 提供了 8 个锁级别分别用于不同的语句并发:1 级锁一般用于 SELECT 查询操作;3 级锁一般用于基本的 INSERT、UPDATE、DELETE 操作;4 级锁用于 VACUUM、ANALYZE 等操作;8 级锁一般用于各类 DDL 语句,具体宏定义及命名代码如下:

```
#define AccessShareLock 1          /* SELECT 语句 */
#define RowShareLock 2             /* SELECT FOR UPDATE/FOR SHARE 语句 */
#define RowExclusiveLock 3 /* INSERT, UPDATE, DELETE 语句 */
#define ShareUpdateExclusiveLock \
    4 /* VACUUM (non-FULL),ANALYZE, CREATE INDEX CONCURRENTLY 语句 */
#define ShareLock 5 /* CREATE INDEX (WITHOUT CONCURRENTLY)语句 */
#define ShareRowExclusiveLock              \
    6 /* 类似于独占模式, 但是允许 ROW SHARE 模式并发 */
#define ExclusiveLock              \
    7 /* 阻塞 ROW SHARE,如 SELECT...FOR UPDATE 语句 */
#define AccessExclusiveLock              \
    8 /* ALTER TABLE, DROP TABLE, VACUUM FULL, LOCK TABLE 语句 */
```

这 8 个级别的锁冲突及并发控制如表 5-7 所示,其中“√”表示两个锁操作可以并发。

表 5-7 锁冲突及并发控制

锁 级 别	1	2	3	4	5	6	7	8
1. ACCESS SHARE	√	√	√	√	√	√	√	—
2. ROW SHARE	√	√	√	√	√	√	—	—

续表

锁　级　别	1	2	3	4	5	6	7	8
3. ROW EXCLUSIVE	√	√	√	√	—	—	—	—
4. SHARE UPDATE EXCLUSIVE	√	√	√	—	—	—	—	—
5. SHARELOCK	√	√	—	—	√	—	—	—
6. SHARE ROW EXCLUSIVE	√	√	—	—	—	—	—	—
7. EXCLUSIVE	√	—	—	—	—	—	—	—
8. ACCESS EXCLUSIVE	—	—	—	—	—	—	—	—

加锁对象数据结构：对 field1 到 field5 赋值标识不同的锁对象，使用 locktag_type 标识锁对象类型，如 relation 表级对象、tuple 行级对象、事务对象等，对应的代码如下：

```
typedef struct LOCKTAG {
    uint32 locktag_field1;                  /* 32 比特位 */
    uint32 locktag_field2;                  /* 32 比特位 */
    uint32 locktag_field3;                  /* 32 比特位 */
    uint32 locktag_field4;                  /* 32 比特位 */
    uint16 locktag_field5;                  /* 32 比特位 */
    uint8 locktag_type;                     /* 详情见枚举类 LockTagType */
    uint8 locktag_lockmethodid;             /* 锁方法类型 */
} LOCKTAG;

typedef enum LockTagType {
    LOCKTAG_RELATION,                       /* 表关系 */
    /* LOCKTAG_RELATION 的 ID 信息为所属库的 OID + 表 OID;如果库的 OID 为 0 表示此表是共享表,
其中 OID 为 openGauss 内核通用对象标识符 */
    LOCKTAG_RELATION_EXTEND,                /* 扩展表的优先权 */
    /* LOCKTAG_RELATION_EXTEND 的 ID 信息 */
    LOCKTAG_PARTITION,                      /* 分区 */
    LOCKTAG_PARTITION_SEQUENCE,             /* 分区序列 */
    LOCKTAG_PAGE,                           /* 表中的页 */
    /* LOCKTAG_PAGE 的 ID 信息为 RELATION 信息 + BlockNumber(页面号) */
    LOCKTAG_TUPLE,                          /* 物理元组 */
    /* LOCKTAG_TUPLE 的 ID 信息为 PAGE 信息 + OffsetNumber(页面上的偏移量) */
    LOCKTAG_TRANSACTION,                            /* 事务 ID (为了等待相应的事务结束) */
    /* LOCKTAG_TRANSACTION 的 ID 信息为事务 ID 号 */
    LOCKTAG_VIRTUALTRANSACTION,             /* 虚拟事务 ID */
    /* LOCKTAG_VIRTUALTRANSACTION 的 ID 信息为它的虚拟事务 ID 号 */
    LOCKTAG_OBJECT,                         /* 非表关系的数据库对象 */
    /* LOCKTAG_OBJECT 的 ID 信息为数据 OID + 类 OID + 对象 OID + 子 ID */
    LOCKTAG_CSTORE_FREESPACE,       /* 列存储空闲空间 */
    LOCKTAG_USERLOCK,               /* 预留给用户锁的锁对象 */
    LOCKTAG_ADVISORY,               /* 用户顾问锁 */
    LOCK_EVENT_NUM
} LockTagType;
```

常规锁 LOCK 结构：tag 是常规锁对象的唯一标识，LOCK 结构的成员变量 procLocks 是将该锁所有的持有、等待线程串联起来的结构体指针，对应的代码如下：

```
typedef struct LOCK {
    /* 哈希键 */
    LOCKTAG tag;                        /* 锁对象的唯一标识 */

    /* 数据 */
    LOCKMASK grantMask;                 /* 已经获取锁对象的位掩码 */
    LOCKMASK waitMask;                  /* 等待锁对象的位掩码 */
    SHM_QUEUE procLocks;                /* 与锁关联的 PROCLOCK 对象链表 */
    PROC_QUEUE waitProcs;               /* 等待锁的 PGPROC 对象链表 */
    int requested[MAX_LOCKMODES];       /* 请求锁的计数 */
    int nRequested;                     /* requested 数组总数 */
    int granted[MAX_LOCKMODES];         /* 已获取锁的计数 */
    int nGranted;                       /* granted 数组总数 */
} LOCK;
```

PROCLOCK 结构：主要是将同一锁对象等待和持有者的线程信息串联起来的结构体，对应的代码如下：

```
typedef struct PROCLOCK {
    /* 标识 */
    PROCLOCKTAG tag;              /* PROCLOCK 对象的唯一标识 */

    /* 数据 */
    LOCKMASK holdMask;            /* 已获取锁类型的位掩码 */
    LOCKMASK releaseMask;         /* 预释放锁类型的位掩码 */
    SHM_QUEUE lockLink;           /* 指向锁对象链表的指针 */
    SHM_QUEUE procLink;           /* 指向 PGPROC 链表的指针 */
} PROCLOCK;
```

t_thrd.proc 结构体里 waitLock 字段记录了该线程等待的锁，该结构体中 procLocks 字段将所有跟该锁有关的持有、等待线程串起来，其队列关系如图 5-16 所示。

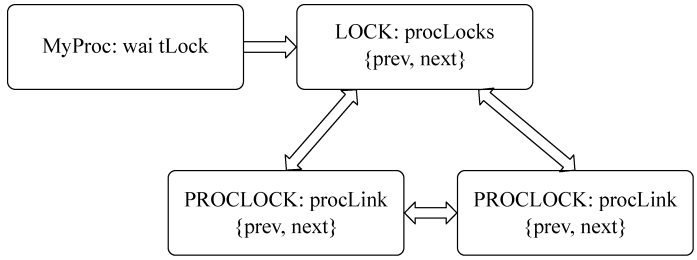

图 5-16　t_thrd.proc 结构体队列关系图

常规锁的主要函数如下。

(1) LockAcquire：对锁对象加锁。

(2) LockRelease：对锁对象释放锁。

(3) LockReleaseAll：释放所有锁资源。

5.3.4　死锁检测机制

死锁主要是指进程 B 要访问进程 A 所在的资源，而进程 A 又由于种种原因不释放掉其锁占用的资源，从而数据库一直处于阻塞状态的情况。如图 5-17 中，T1 使用资源 R1 去请求 R2，而 T2 事务持有 R2 的资源去申请 R1。

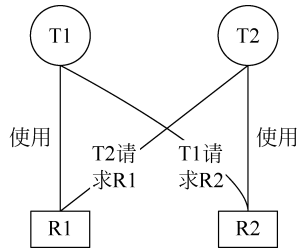

图 5-17　死锁状态

形成死锁的必要条件是：资源的请求与保持。每个进程都可以在使用一个资源的同时去申请访问另一个资源。打破死锁的常见处理方式：中断其中的一个事务的执行，打破环状的等待。openGauss 提供了轻量级锁的死锁检测及常规锁的死锁检测机制，下面简单介绍相关原理及代码。

1. 轻量级锁死锁检测

openGauss 使用一个独立的监控线程来完成轻量级锁的死锁探测、诊断和解除。工作线程在请求轻量级锁成功之前会写入一个时间戳数值，成功获得锁后置该时间戳为 0。监测线程可以通过快速对比时间戳数值来发现长时间获得不到锁资源的线程，这一过程是快速轻量的。只有发现长时间的锁等待，死锁检测的诊断才会触发。这样做的目的是防止频繁诊断影响业务正常执行。一旦确定了死锁环的存在，监控线程首先会将死锁信息记录到日志中去，然后采取恢复措施使得死锁自愈，即选择死锁环中的一个线程报错退出。轻量级锁死锁检测机制如图 5-18 所示。

因为检测死锁是否真正发生是一个重 CPU 操作，为了不影响数据库性能和运行稳定性，轻量级死锁检测使用了一种轻量式的探测，用来快速判断是否可能发生了死锁，采用看门狗(watchdog)的方法，利用时间戳来探测。工作线程在锁请求进入时会在全局内存上写入开始等待的时间戳；在锁请求成功后，将该时间戳清零。对于一个发生死锁的线程，它的锁请求是等待状态，时间戳也不会清零，且与当前运行时间戳数值的差值越来越大。GUC 参数 fault_mon_timeout 控制检测间隔时间，默认为 5s。轻量级锁死锁检测每隔 fault_mon_timeout 去进行检测，如果当前发现有同样线程、同样锁 ID，且时间戳等待时间超过检测间隔时间值，则触发真正死锁检测。时间统计及轻量级检测函数如下。

(1) pgstat_read_light_detect：从统计信息结构体中读取线程及锁 ID 相关的时间戳，并记录到指针队列中。

(2) lwm_compare_light_detect：跟几秒检测前的时间对比，如果找到可能发生死锁的线程及锁 ID 则返回 true，否则返回 false。

真正的轻量级锁死锁检测是一个有向无环图的判定过程，它的实现跟常规锁类似，这部分会在下面详细介绍。死锁检测需要两部分信息：锁，包括请求和分配的信息；线程，包括等待和持有的信息，这些信息记录到相应的全局变量中，死锁监控线程可以访问并进行判断，相关的函数如下。

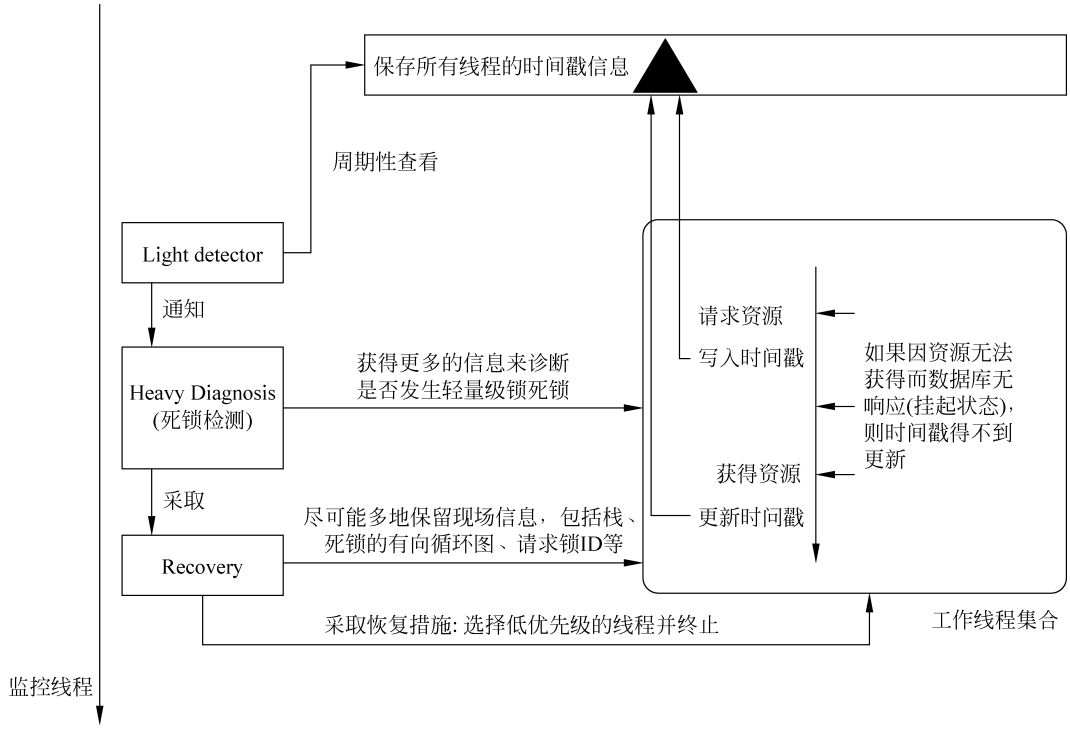

图 5-18　轻量级锁死锁检测机制

(1) lwm_heavy_diagnosis：检测是否有死锁。

(2) lwm_deadlock_report：报告死锁详细信息，方便定位诊断。

(3) lw_deadlock_auto_healing：治愈死锁，选择环中一个线程退出。

用于死锁检测的锁和线程相关数据结构如下。

(1) lock_entry_id 记录线程信息，有 thread_id 及 sessionid 是为了适配线程池框架，可以准确地从统计信息中找到相应的信息，对应的代码如下：

```
typedef struct {
    ThreadId thread_id;
    uint64 st_sessionid;
} lock_entry_id;
```

(2) lwm_light_detect 记录可能出现死锁的线程，最后用一个链表的形式将当前所有信息串联起来，对应的代码如下：

```
typedef struct {
    /* 线程 ID */
    lock_entry_id entry_id;

    /* 轻量级锁检测引用计数 */
    int lw_count;
```

```
} lwm_light_detect;
```

(3) lwm_lwlocks 记录线程相关的锁信息,持有锁数量及等锁信息,对应的代码如下:

```
typedef struct {
    lock_entry_id be_tid;            /* 线程 ID */
    int be_idx;                      /* 后台线程的位置 */
    LWLockAddr want_lwlock;          /* 预获取锁的信息 */
    int lwlocks_num;                 /* 线程持有的轻量级锁个数 */
    lwlock_id_mode * held_lwlocks;   /* 线程持有的轻量级锁数组 */
} lwm_lwlocks;
```

2. 常规锁死锁检测

openGauss 在获取锁时如果没有冲突可以直接上锁,如果有冲突则设置一个定时器(timer),并进入等待,过一段时间会被定时器唤起进行死锁检测。如果在某个锁的等锁队列中,进程 T2 排在进程 T1 后面,且进程 T2 需要获取的锁与 T1 需要获取的锁资源冲突,则 T2 到 T1 会有一条软等待边(soft edge);如果进程 T2 的加锁请求与 T1 进程所持有的锁冲突,则有一条硬等待边(hard edge)。那么整体思路就是通过递归调用,从当前顶点等锁的顶点出发,沿着等待边向前走,看是否存在环,如果环中有软等待边,说明环中两个进程都在等锁,重新排序,尝试解决死锁冲突。如果没有软等待边,那么只能终止当前等锁的事务,解决死锁等待环。如图 5-19 所示,虚线代表软等待边,实线代表硬等待边。线程 A 等待线程 B,线程 B 等待线程 C,线程 C 等待线程 A,因为线程 A 等待线程 B 的是软等待边,进行一次调整成为图 5-19 右边的等待关系,此时发现线程 A 等待线程 C,线程 C 等待线程 A,没有软等待边,检测到死锁。

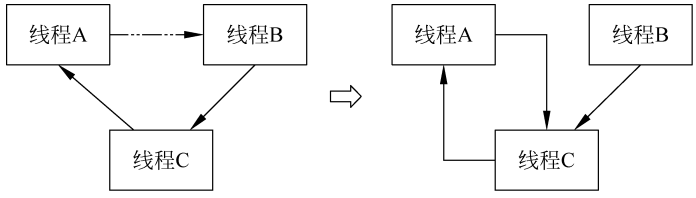

图 5-19　常规锁死锁检测示意图

主要函数如下。

(1) DeadLockCheck:死锁检测函数。

(2) DeadLockCheckRecurse:如果死锁则返回 true,如果有软等待边,返回 false 并且尝试解决死锁冲突。

(3) check_stack_depth:openGauss 会检查用于死锁递归检测的堆栈,防止堆栈过长,导致死锁检测时,轻量级锁分区因长期持有锁而阻塞后面所有的业务。

(4) CheckDeadLockRunningTooLong:openGauss 会检查死锁检测时间,防止死锁检测时间过长阻塞后面所有业务。对应的代码如下:

```
static void CheckDeadLockRunningTooLong(int depth)
```

```
{ /* 每 4 层检测一下 */
    if (depth > 0 && ((depth % 4) == 0)) {
        TimestampTz now = GetCurrentTimestamp();
        long secs = 0;
        int usecs = 0;

        if (now > t_thrd.storage_cxt.deadlock_checker_start_time) {
             TimestampDifference(t_thrd.storage_cxt.deadlock_checker_start_time, now,
&secs, &usecs);
            if (secs > 600) {      /* 如果从死锁检测开始超过 10min,则报错处理 */
#ifdef USE_ASSERT_CHECKING
                DumpAllLocks();  /* 在 debug 版本时,导出所有的锁信息,便于定位问题 */
#endif

                ereport(defence_errlevel(), (errcode(ERRCODE_INTERNAL_ERROR),
                                              errmsg("Deadlock checker runs too long and is
greater than 10 minutes.")));
            }
        }
    }
}
```

(5) FindLockCycle：检查是否有死锁环。

(6) FindLockCycleRecurse：死锁检测内部递归调用函数。

相应的数据结构有：

(1) 死锁检测中最核心最关键的有向边数据结构,对应的代码如下：

```
typedef struct EDGE {
    PGPROC * waiter;        /* 等待的线程 */
    PGPROC * blocker;       /* 阻塞的线程 */
    int pred;               /* 拓扑排序的工作区 */
    int link;               /* 拓扑排序的工作区 */
} EDGE;
```

(2) 可重排的一个等待队列,对应的代码如下：

```
typedef struct WAIT_ORDER {
    LOCK * lock;        /* 描述其等待队列的锁 */
    PGPROC ** procs;    /* 按新等待顺序排列的 PGPROC 数组 */
    int nProcs;
} WAIT_ORDER;
```

(3) 死锁检测最后打印的相应信息,对应的代码如下：

```
typedef struct DEADLOCK_INFO {
    LOCKTAG locktag;        /* 等待锁对象的唯一标识 */
    LOCKMODE lockmode;      /* 等待锁对象的锁类型 */
    ThreadId pid;           /* 阻塞线程的线程 ID */
} DEADLOCK_INFO;
```

5.3.5　无锁原子操作

openGauss 封装了 32、64、128 位的原子操作，主要用于取代自旋锁，实现简单变量的原子更新操作。

(1) gs_atomic_add_32：32 位原子加，并且返回加之后的值，对应的代码如下：

```
static inline int32 gs_atomic_add_32(volatile int32 * ptr, int32 inc)
{
    return __sync_fetch_and_add(ptr, inc) + inc;
}
```

(2) gs_atomic_add_64：64 位原子加，并且返回加之后的值，对应的代码如下：

```
static inline int64 gs_atomic_add_64(int64 * ptr, int64 inc)
{
    return __sync_fetch_and_add(ptr, inc) + inc;
}
```

(3) gs_compare_and_swap_32：32 位 CAS 操作，如果 dest 地址的值在更新前没有被其他线程更新，则将 newval 写到 dest 地址并返回 true，否则返回 false，对应的代码如下：

```
static inline bool gs_compare_and_swap_32(int32 * dest, int32 oldval, int32 newval)
{
    if (oldval == newval)
        return true;

    volatile bool res = __sync_bool_compare_and_swap(dest, oldval, newval);

    return res;
}
```

(4) gs_compare_and_swap_64：64 位 CAS 操作，如果 dest 地址的值在更新前没有被其他线程更新，则将 newval 写到 dest 地址并返回 true，否则返回 false，对应的代码如下：

```
static inline bool gs_compare_and_swap_64(int64 * dest, int64 oldval, int64 newval)
{
    if (oldval == newval)
        return true;

    return __sync_bool_compare_and_swap(dest, oldval, newval);
}
```

(5) arm_compare_and_swap_u128：openGauss 提供跨平台的 128 位 CAS 操作，在 ARM 平台下，使用单独的指令集汇编了 128 位原子操作，用于提升内核测锁并发的性能，对应的代码如下：

```
static inline uint128_u arm_compare_and_swap_u128(volatile uint128_u * ptr, uint128_u
oldval, uint128_u newval)
```

```
{
#ifdef __ARM_LSE
    return __lse_compare_and_swap_u128(ptr, oldval, newval);
#else
    return __excl_compare_and_swap_u128(ptr, oldval, newval);
#endif
}
#endif
```

(6) atomic_compare_and_swap_u128：128 位 CAS 操作，如果 dest 地址的值在更新前没有被其他线程更新，则将 newval 写到 dest 地址。dest 地址的值没有被更新，就返回新值，否则返回被别人更新后的值。需要注意必须由上层的调用者保证传入的参数是 128 位对齐的。对应的代码如下：

```
static inline uint128_u atomic_compare_and_swap_u128(
    volatile uint128_u * ptr,
    uint128_u oldval = uint128_u{0},
    uint128_u newval = uint128_u{0})
{
#ifdef __aarch64__
    return arm_compare_and_swap_u128(ptr, oldval, newval);
#else
    uint128_u ret;
    ret.u128 = __sync_val_compare_and_swap(&ptr->u128, oldval.u128, newval.u128);
    return ret;
#endif
}
```

5.3.6 基于鲲鹏服务器的性能优化

1. WAL Group Inset 优化

数据库 redo 日志缓存系统指的是数据库 redo 日志持久化的写缓存，数据库 redo 日志落盘前会写入日志缓存中再写到磁盘进行持久化。日志缓存的写入效率是决定数据库整体吞吐量的主要因素，而各个线程之间写日志时为了保证日志顺序写存在锁争抢，锁的争抢就成为性能的主要瓶颈点。openGauss 针对鲲鹏服务器 ARM CPU 的特点，通过 group 的方式进行日志的插入，减少锁的争抢，提升 WAL 日志的插入效率，从而提升整个数据库的吞吐性能。group 方式日志插入的主要流程如图 5-20 所示。

(1) 不需要所有线程都竞争锁。

(2) 在同一时间窗口所有线程在争抢锁前先加入一个 group 中，第一个加入 group 的线程作为 leader，通过 CAS 原子操作来实现队列的管理。

(3) leader 线程代表整个 group 去争抢锁。group 中的其他线程(follower)开始睡眠，等待 leader 唤醒。

(4) 争抢到锁后，leader 线程将 group 里的所有线程想要插入的日志遍历一遍得到需要空间总大小。leader 线程只执行一次 reserve space 操作。

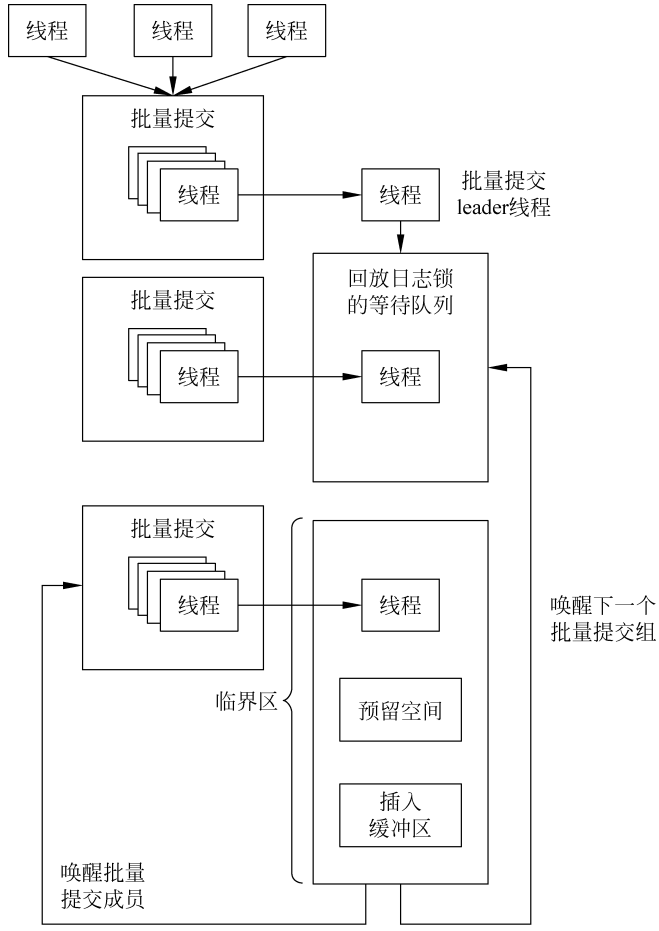

图 5-20　group 方式日志插入的主要流程

(5) leader 线程将 group 中所有线程想要写入的日志都写入日志缓冲区中。

(6) 释放锁,唤醒所有 follower 线程。

(7) follower 线程由于需要写入的日志已经被 leader 写入,不需要再争抢锁,直接进入后续流程。

关键函数代码如下:

```
static XLogRecPtr XLogInsertRecordGroup(XLogRecData * rdata, XLogRecPtr fpw_lsn)
{
    …/ * 初始化变量及简单校验 * /
    START_CRIT_SECTION();          / * 开启临界区 * /
    proc - > xlogGroupMember = true;
     …
    proc - > xlogGroupDoPageWrites = &t_thrd.xlog_cxt.doPageWrites;

    nextidx = pg_atomic_read_u32(&t_thrd.shemem_ptr_cxt.LocalGroupWALInsertLocks
```

```
[groupnum].l.xlogGroupFirst);

    while (true) {
        pg_atomic_write_u32(&proc->xlogGroupNext, nextidx); /* 将上一个成员记录到 proc
结构体中 */
        /* 防止 ARM 乱序:保证所有前面的写操作都可见 */
        pg_write_barrier();

        if (pg_atomic_compare_exchange_u32(&t_thrd.shemem_ptr_cxt.LocalGroupWALInsertLocks
[groupnum].l.xlogGroupFirst,
                &nextidx,
                (uint32)proc->pgprocno)) {
            break;
        } /* 这一步原子操作获取上一个成员的 proc no,如果是 invalid,说明是 leader */
    }
     /* 非 leader 成员不去获取 WAL Insert 锁,仅仅进行等待,直到被 leader 唤醒 */
    if (nextidx != INVALID_PGPROCNO) {
        int extraWaits = 0;

        for (;;) {
            /* 充当读屏障 */
            PGSemaphoreLock(&proc->sem, false);
            /* 充当读屏障 */
            pg_memory_barrier();
            if (!proc->xlogGroupMember) {
                break;
            }
            extraWaits++;
        }

        while (extraWaits-- > 0) {
            PGSemaphoreUnlock(&proc->sem);
        }
        END_CRIT_SECTION();
        return proc->xlogGroupReturntRecPtr;
    }
    /* leader 成员持有锁 */
    WALInsertLockAcquire();
    /* 计算每个成员线程的 xlog record size */
    …
    /* leader 线程将所有成员线程的 xlog record 插入缓冲区 */
    while (nextidx != INVALID_PGPROCNO) {
        localProc = g_instance.proc_base_all_procs[nextidx];

        if (unlikely(localProc->xlogGroupIsFPW)) {
            nextidx = pg_atomic_read_u32(&localProc->xlogGroupNext);
            localProc->xlogGroupIsFPW = false;
            continue;
        }
        XLogInsertRecordNolock(localProc->xlogGrouprdata,
```

```
            localProc,
            XLogBytePosToRecPtr(StartBytePos),
            XLogBytePosToEndRecPtr(
                StartBytePos + MAXALIGN(((XLogRecord *)(localProc->xlogGrouprdata->
data))->xl_tot_len)),
            XLogBytePosToRecPtr(PrevBytePos));
        PrevBytePos = StartBytePos;
        StartBytePos += MAXALIGN(((XLogRecord *)(localProc->xlogGrouprdata->data))->
xl_tot_len);
        nextidx = pg_atomic_read_u32(&localProc->xlogGroupNext);
    }

    WALInsertLockRelease();        /*完成工作放锁并唤醒所有成员线程*/
    while (wakeidx != INVALID_PGPROCNO) {
        PGPROC * proc = g_instance.proc_base_all_procs[wakeidx];

        wakeidx = pg_atomic_read_u32(&proc->xlogGroupNext);
        pg_atomic_write_u32(&proc->xlogGroupNext, INVALID_PGPROCNO);
        proc->xlogGroupMember = false;
        pg_memory_barrier();

        if (proc != t_thrd.proc) {
            PGSemaphoreUnlock(&proc->sem);
        }
    }

    END_CRIT_SECTION();
    return proc->xlogGroupReturntRecPtr;
}
```

2. Cache align 消除伪共享

CPU 在访问主存时一次会获取整个缓存行的数据，其中 x86 典型值是 64 字节，而 ARM 1620 芯片 L1 和 L2 缓存都是 64 字节，L3 缓存是 128 字节。这种数据获取方式本身可以大大提升数据访问的效率，但是假如同一个缓存行中不同位置的数据频繁被不同的线程读取和写入，由于写入时会造成其他 CPU 下的同一个缓存行失效，从而会使得 CPU 按照缓存行来获取主存数据的努力不但白费，反而成为性能负担。伪共享就是指这种不同的 CPU 同时访问相同缓存行的不同位置的性能低效的行为。

以轻量级锁为例，代码如下所示：

```
#ifdef __aarch64__
#define LWLOCK_PADDED_SIZE PG_CACHE_LINE_SIZE(128)
#else
#define LWLOCK_PADDED_SIZE (sizeof(LWLock) <= 32 ? 32 : 64)
#endif
typedef union LWLockPadded
{
LWLocklock;
```

```
charpad[LWLOCK_PADDED_SIZE];
} LWLockPadded;
```

当前锁逻辑中轻量级锁的访问仍然是最突出的热点之一。如果 LWLOCK_PADDED_SIZE 是 32 字节的，且轻量级锁是按照一个连续的数组来存储的，64 字节的缓存行可以同时容纳两个 LWLockPadded，128 字节的缓存行则可以同时含有 4 个 LWLockPadded。当系统中对轻量级锁竞争激烈时，对应的缓存行不停地获取和失效，浪费大量 CPU 资源。故在 ARM 机器的优化下将 padding_size 直接设置为 128，消除伪共享，提升整体轻量级锁的使用性能。

3. WAL INSERT 128CAS 无锁临界区保护

在目前数据库或文件系统中，WAL 需要把内存中生成的日志信息插入日志缓存中。为了实现日志高速缓存，日志管理系统会并发插入，通过预留全局位置来完成，一般使用两个 64 位的全局数据位置索引分别表示存储插入的起始和结束位置，最大能提供 16EB (Exabyte)的数据索引的支持。为了保护全局的位置索引，WAL 引入了一个高性能的原子锁实现每个日志缓存位置的保护，在 NUMA 架构中，特别是 ARM 架构中，由于原子锁退避和高跨 CPU 访问延迟，缓存一致性性能差异导致 WAL 并发的缓存保护成为瓶颈。

优化的主要思想是将两个 64 位的全局数据位置信息通过 128 位原子操作替换原子锁，消除原子锁本身在跨 CPU 访问、原子锁退避(backoff)、缓存一致性代价，如图 5-21 所示。

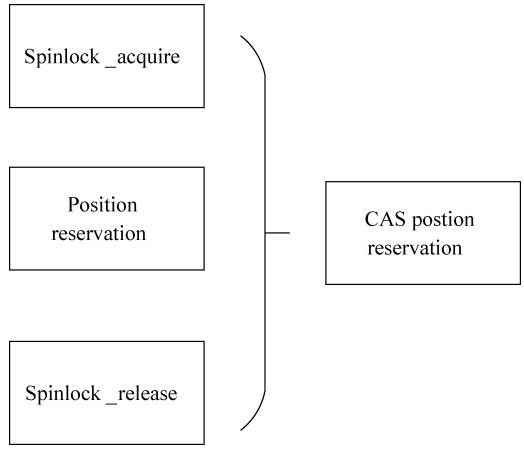

图 5-21　128CAS 无锁临界区保护示意图

全局位置信息包括一个 64 位起始地址和一个 64 位的结束地址，将这两个地址合并成为一个 128 位信息，通过 CAS 原子操作完成免锁位置信息的预留。在 ARM 平台中没有实现 128 位的原子操作库，openGauss 通过 exclusive 命令加载两个 ARM64 位数据来实现，ARM64 汇编指令为 LDXP/STXP。

关键数据结构及函数 ReserveXLogInsertLocation 的代码如下：

```
typedef union {
```

```
    uint128  u128;
    uint64  u64[2];
    uint32   u32[4];
} uint128_u; /* 为了代码可读及操作,将 u128 设计成 union 的联合结构体,内存位置进行 64 位数值的赋值 */
static void ReserveXLogInsertLocation(uint32 size, XLogRecPtr *  StartPos, XLogRecPtr * EndPos, XLogRecPtr* PrevPtr)
{
    volatile XLogCtlInsert* Insert = &t_thrd.shemem_ptr_cxt.XLogCtl->Insert;
    uint64 startbytepos;
    uint64 endbytepos;
    uint64 prevbytepos;

    size = MAXALIGN(size);

#if defined(__x86_64__) || defined(__aarch64__)
    uint128_u compare;
    uint128_u exchange;
    uint128_u current;

    compare = atomic_compare_and_swap_u128((uint128_u*)&Insert->CurrBytePos);

loop1:
    startbytepos = compare.u64[0];
    endbytepos = startbytepos + size;

    exchange.u64[0] = endbytepos; /* 此处为了代码可读,将 uint128 设置成一个 union 的联合结构体,将起始和结束位置写入 exchange 中 */
    exchange.u64[1] = startbytepos;

    current = atomic_compare_and_swap_u128((uint128_u*)&Insert->CurrBytePos, compare, exchange);
    if (!UINT128_IS_EQUAL(compare, current)) {     /* 如果被其他线程并发更新,重新循环 */
        UINT128_COPY(compare, current);
        goto loop1;
    }
    prevbytepos = compare.u64[1];

#else
    SpinLockAcquire(&Insert->insertpos_lck); /* 其余平台使用自旋锁来保护变量更新 */
    startbytepos = Insert->CurrBytePos;
    prevbytepos = Insert->PrevBytePos;
    endbytepos = startbytepos + size;
    Insert->CurrBytePos = endbytepos;
    Insert->PrevBytePos = startbytepos;

    SpinLockRelease(&Insert->insertpos_lck);
#endif /* __x86_64__|| __aarch64__ */
    *StartPos = XLogBytePosToRecPtr(startbytepos);
    *EndPos = XLogBytePosToEndRecPtr(endbytepos);
```

```
        * PrevPtr = XLogBytePosToRecPtr(prevbytepos);
    }
```

4. CLOG Partition 优化

CLOG 日志即事务提交日志(详情可参考 5.2.2 节相关内容)。每个事务存在 4 种状态：IN_PROGRESS、COMMITED、ABORTED、SUB_COMMITED，每条日志占 2 bit。CLOG 日志需要存储在磁盘上，一个页面(8KB)可以包含 2^{15} 条，每个日志文件(段=256×8K)2^{26} 条。当前 CLOG 的访问通过缓冲池实现，代码中使用统一的 SLRU 缓冲池算法。

如图 5-22 所示，CLOG 的日志缓冲池在共享内存中且全局唯一，名称为 GlogCtl，为各工作线程共享该资源。在高并发的场景下，该资源的竞争成为性能瓶颈，优化分区后如图 5-23 所示。按页面号进行取模运算(求两个数相除的余数)，将日志均分到多个共享内存的缓冲池中，由线程局部对象的数组 ClogCtlData 来记录，名称为 ClogCtl，同步增加共享内存中的缓冲池对象及对应的全局锁，即通过打散的方式提高整体吞吐。

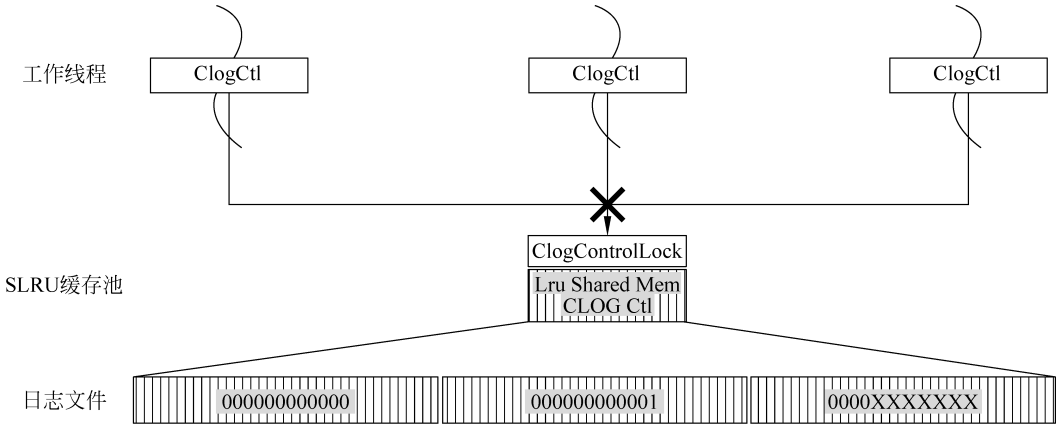

图 5-22　CLOG 的日志缓冲池优化前

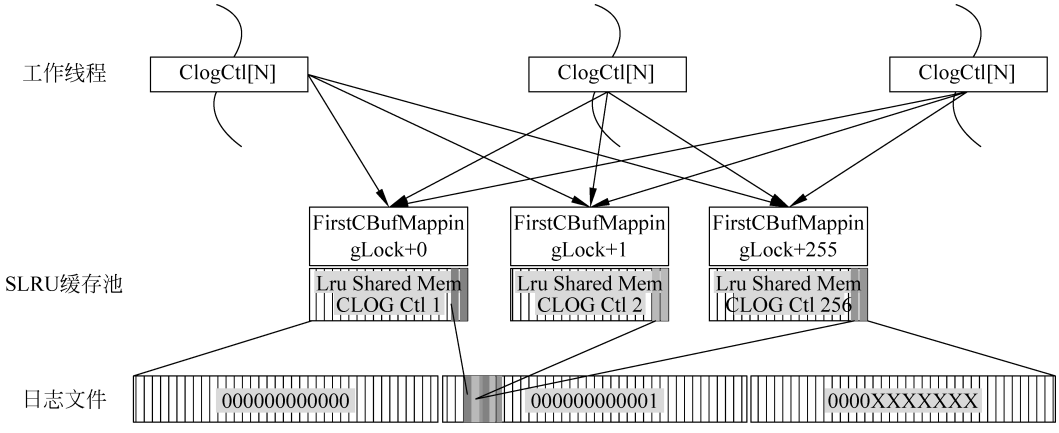

图 5-23　CLOG 的日志缓冲池优化后

CLOG 分区优化需要将源代码中涉及原缓冲池的操作进行修改，改为操作对应的分区的缓冲池，而通过事务 ID、页面号能方便地找到对应的分区，与此同时，对应的控制锁也从原来的一把锁改为多把锁，涉及的结构体代码如下，涉及的函数如表 5-8 所示。

```
/* CLOG 分区 */
#define NUM_CLOG_PARTITIONS 256      /* 分区打散的个数 */
/* CLOG 轻量级分区锁 */
#define CBufHashPartition(hashcode) \
    ((hashcode) % NUM_CLOG_PARTITIONS)
#define CBufMappingPartitionLock(hashcode) \
    (&t_thrd.shemem_ptr_cxt.mainLWLockArray[FirstCBufMappingLock + CBufHashPartition
(hashcode)].lock)
#define CBufMappingPartitionLockByIndex(i) \
    (&t_thrd.shemem_ptr_cxt.mainLWLockArray[FirstCBufMappingLock + i].lock)
```

表 5-8　CLOG 分区优化函数

函　数　名	简　　述
CLOGShmemInit	调用 SimpleLruInit 初始化共享内存中的 CLOG 缓冲区
ZeroCLOGPage	CLOG 日志页面的初始化为 0
BootStrapCLOG	创建数据库时，在缓冲区中创建初始可用的 CLOG 日志页面，并调用 ZeroCLOGPage 初始化页面为 0，写回到磁盘，并返回页面
CLogSetTreeStatus	设置事务提交的最终状态
CLogGetStatus	查询事务状态
ShutdownCLOG	关闭缓冲区，刷新到磁盘中
ExtendCLOG	为新分配的事务创建 CLOG 页面
TruncateCLOG	日志检查点的建立使得部分事务的日志过期，可删除以节省空间
WriteZeroPageXlogRec	新建 XLOG 页面时，写“CLOG_ZEROPAGE”XLOG 日志，以便将来恢复使用
clog_redo	CLOG 日志相关的 redo 操作，含 CLOG_ZEROPAGE 及 CLOG_TRUNCATE

5. 支持 NUMA-aware 数据和线程访问分布

NUMA 远端访问：内存访问涉及访问线程和被访问内存的物理位置。只有两者在同一个 NUMA Node 中时，内存访问才是本地的，否则就会涉及跨 Node 远端访问，此时性能开销较大。

Numactl 开源软件提供了 libnuma 库允许应用程序方便地将线程绑定在特定的 NUMA Node 或者 CPU 列表，可以在指定的 NUMA Node 上分配内存。下面对 openGauss 代码可能涉及的 API 进行描述。

(1) “int numa_run_on_node(int node);”将当前任务及子任务运行在指定的 Node 上。该 API 对应函数如下。

numa_run_on_node 函数：在特定节点上运行当前任务及其子任务。在使用 numa_run_on_node_mask 函数重置节点关联之前，这些任务不会迁移到其他节点的 CPU 上。传递 −1 让内核再次在所有节点上调度。成功时返回 0；错误时返回 −1，错误码记录在

errno 中。

(2)“void numa_set_localalloc(void);”将调用者线程的内存分配策略设置为本地分配,即优先从本节点进行内存分配。该 API 对应函数如下。

numa_set_localalloc 函数:设置调用任务的内存分配策略为本地分配。在此模式下,内存分配的首选节点为内存分配时任务正在执行的节点。

(3)“void numa_alloc_onnode(void);”在指定的 NUMA Node 上申请内存。该 API 对应函数如下。

numa_alloc_onnode 函数:在特定节点上分配内存。分配大小为系统页的倍数并向上取整。如果指定的节点在外部拒绝此进程,则此调用将失败。与函数系列 Malloc(3)相比,此函数相对较慢,必须使用 numa_free 函数释放内存。错误时返回 NULL。

openGauss 基于 NUMA 架构进行了内部数据结构优化。

1) 全局 PGPROC 数组优化

如图 5-24 所示,对每个客户端连接系统都会分配一个专门的 PGPROC 结构来维护相关信息。ProcGlobal→allProcs 原本是一个 PGPROC 结构的全局数组,但是其物理内存所在的 NUMA Node 是不确定的,造成每个事务线程访问自己的 PGPROC 结构时,线程可能由于操作系统的调度在多个 NUMA Node 间,而对应的 PGPROC 结构的物理内存位置也是无法预知,大概率会是远端访存。

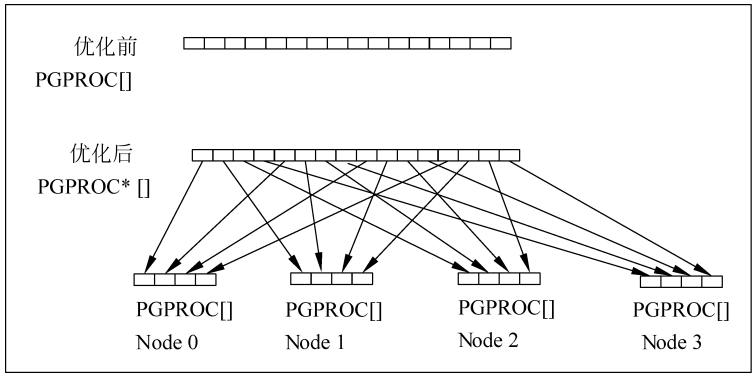

图 5-24 全局 PGPROC 数组优化

由于 PGPROC 结构的访问较为频繁,根据 NUMA Node 的个数将这个全局结构数组分为多份,每份分别使用 numa_alloc_onnode 来固定 NUMA Node 分配内存。为了尽量减少对当前代码的结构性改动,将 ProcGlobal→allProcs 由 PGPROC * 改为 PGPROC **。对应所有访问 ProcGlobal→allProcs 的地方均需要做相应调整(多了一层间接指针引用)。相关代码如下:

```
#ifdef __USE_NUMA
    if (nNumaNodes > 1) {
        ereport(INFO, (errmsg("InitProcGlobal nNumaNodes: %d, inheritThreadPool: %d,
groupNum: %d",
```

```
            nNumaNodes, g_instance.numa_cxt.inheritThreadPool,
            (g_threadPoolControler ? g_threadPoolControler->GetGroupNum() : 0))));

        int groupProcCount = (TotalProcs + nNumaNodes - 1) / nNumaNodes;
        size_t allocSize = groupProcCount * sizeof(PGPROC);
        for (int nodeNo = 0; nodeNo < nNumaNodes; nodeNo++) {
            initProcs[nodeNo] = (PGPROC *)numa_alloc_onnode(allocSize, nodeNo);
            if (!initProcs[nodeNo]) {
                ereport(FATAL, (errcode(ERRCODE_OUT_OF_MEMORY),
                                errmsg("InitProcGlobal NUMA memory allocation in node %d
failed.", nodeNo)));
            }
            add_numa_alloc_info(initProcs[nodeNo], allocSize);
             int ret = memset_s(initProcs[nodeNo], groupProcCount * sizeof(PGPROC), 0,
groupProcCount * sizeof(PGPROC));
            securec_check_c(ret, "\0", "\0");
        }
    } else {
#endif
```

2）全局 WALInsertLock 数组优化

WALInsertLock 用来对 WAL Insert 操作进行并发保护，可以配置多个，比如 16。优化前，所有的 WALInsertLock 都在同一个全局数组，并通过共享内存进行分配。事务线程运行时在整个全局数组中分配其中的一个 Insert Lock 进行使用，因此大概率会涉及远端访存，即多个线程会进行跨 Node、跨 P 竞争。WALInsertLock 也可以按 NUMA Node 单独分配内存，并且每个事务线程仅使用本 Node 分组内的 WALInsertLock，这样就可以将数据竞争限定在同一个 NUMA Node 内部。全局 WALInsertLock 数组优化基本原理如图 5-25 所示。

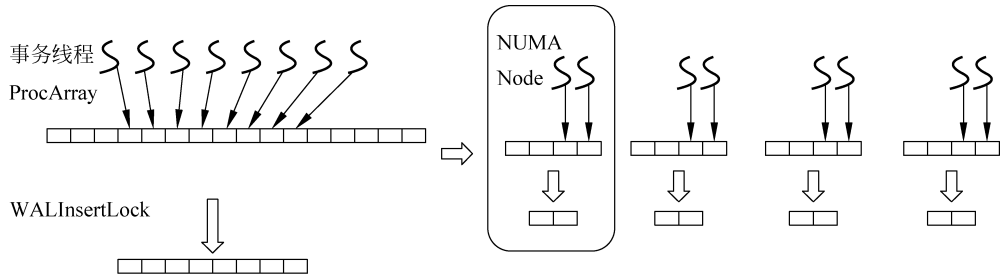

图 5-25　全局 WALInsertLock 数组优化基本原理

假如系统配置了 16 个 WALInsertLock，同时 NUMA Node 配置为 4 个，则原本长度为 16 的数组将会被拆分为 4 个数组，每个数组长度为 4。全局结构体为“WALInsertLockPadded ** GlobalWALInsertLocks”，线程本地 WALInsertLocks 将指向本 Node 内的 WALInsertLock，不同的 NUMA Node 拥有不同地址的 WALInsertLock 子数组。GlobalWALInsertLocks 则用于跟踪多个 Node 下的 WALInsertLock 数组，以方便遍历。WALInsertLock 分组方式

示意图如图 5-26 所示。

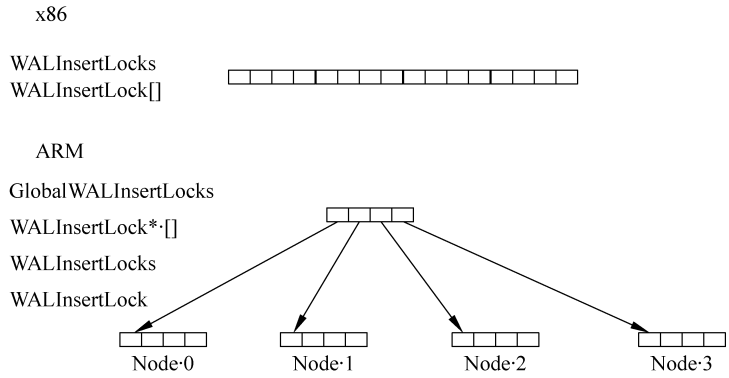

图 5-26　WALInsertLock 分组方式示意图

初始化 WALInsertLock 结构体的代码如下：

```
    WALInsertLockPadded ** insertLockGroupPtr =
            ( WALInsertLockPadded * * ) CACHELINEALIGN ( palloc0 ( nNumaNodes * sizeof
(WALInsertLockPadded * ) + PG_CACHE_LINE_SIZE));
#ifdef __USE_NUMA
    if (nNumaNodes > 1) {
        size_t allocSize = sizeof(WALInsertLockPadded) * g_instance.xlog_cxt.num_locks_in
_group + PG_CACHE_LINE_SIZE;
        for (int i = 0; i < nNumaNodes; i++) {
            char * pInsertLock = (char * )numa_alloc_onnode(allocSize, i);
            if (pInsertLock == NULL) {
                ereport(PANIC, (errmsg("XLOGShmemInit could not alloc memory on node %d",
i)));
            }
            add_numa_alloc_info(pInsertLock, allocSize);
            insertLockGroupPtr[i] = (WALInsertLockPadded * )(CACHELINEALIGN(pInsertLock));
        }
    } else {
#endif
        char * pInsertLock = (char * )CACHELINEALIGN(palloc(
            sizeof(WALInsertLockPadded) * g_instance.attr.attr_storage.num_xloginsert_
locks + PG_CACHE_LINE_SIZE));
        insertLockGroupPtr[0] = (WALInsertLockPadded * )(CACHELINEALIGN(pInsertLock));
#ifdef __USE_NUMA
    }
#endif
```

在 ARM 平台下，访问 WALInsertLock 需遍历 GlobalWALInsertLocks 两维数组，第一层遍历 NUMA Node，第二层遍历 Node 内部的 WALInsertLock 数组。

WALInsertLock 引用的 LWLock 内存结构在 ARM 平台下也进行相应的优化适配，代码如下：

```
typedef struct
{
LWLock lock;
#ifdef __aarch64__
pg_atomic_uint32xlogGroupFirst;
#endif
XLogRecPtrinsertingAt;
} WALInsertLock;
```

这里的 lock 成员变量将引用共享内存中的全局 LWLock 数组中的某个元素，在 WALInsertLock 优化之后，尽管 WALInsertLock 已经按照 NUMA Node 分布了，但是其引用的 LWLock 却无法控制其物理内存位置，因此在访问 WALInsertLock 的 lock 时仍然涉及了大量的跨 Node 竞争。因此，将 LWLock 直接嵌入 WALInsertLock 内部，从而将使用的 LWLock 一起进行 NUMA 分布，同时还减少了一次缓存访问。

5.4　本章小结

本章主要介绍了 openGauss 事务及并发控制的机制。

事务系统将 SQL、执行及存储模块串联起来，是数据库的重要角色（收到外部命令，根据当前内部系统状态，决定执行走向），保证了事务处理的连贯性及正确性。

本章除介绍 openGauss 最基础与最核心的事务系统外，还详细描述了 openGauss 是如何基于鲲鹏服务器做出性能优化的。

总而言之，用“疾如闪电，稳如泰山”来形容 openGauss 的事务及并发控制模块是最适合不过了。