第 12 章　并发控制与锁机制

12.1 实验介绍

与日志和恢复机制相同，并发控制与锁机制是 openGauss 数据库实现事务处理 ACID 特性的另一重要部分。目前数据库原理教学中对于并发控制与锁机制模块缺乏行之有效的实践教学手段，部分原因归咎于并发控制与锁机制在数据库管理系统中涉及到的相关代码复杂繁复。本实验旨在借助 openGauss 的开源代码，一方面对于并发控制与锁机制的原理进行落地的代码工程实践，另一方面最大限度地减小源代码繁复程度的影响。本章实验原理主要包括 openGauss 的事务处理和锁机制。首先，通过实验查看 openGauss 数据库中表上的加锁信息；然后，通过复现的方式分别验证 Share 锁、Access Share 锁、Row Exclusive 锁和 Access Exclusive 锁。

并发控制与锁机制也是数据库管理系统实现中最为复杂和精妙部分之一。与日志和恢复部分类似，会涉及到若干系统层面的实现代码。希望通过本实验的实践内容，提升对于并发控制与锁机制的理解程度，更好地认识系统软件在工程实现方面的复杂性。

12.2 实验目的

1. 理解 openGauss 事务处理原理。
2. 了解 openGauss 事务处理模块的实现机制。
3. 理解 openGauss 锁机制原理。
4. 了解 openGauss 锁机制的若干实现过程。
5. 掌握 openGauss 中 Share 锁的复现方法。
6. 掌握 openGauss 中 Access Share 锁的复现方法。
7. 掌握 openGauss 中 Row Exclusive 锁的复现方法。
8. 掌握 openGauss 中 Access Exclusive 锁的复现方法。
9. 了解与本实验相关的函数与结构体的源代码。

12.3 实验原理

本章实验的原理包括：openGauss 事务处理和锁机制。

12.3.1　openGauss 事务处理

1. 事务的概念

事务是数据库操作的执行单位，需要满足最基本的 ACID 属性。

- 原子性（Atomicity）：一个事务提交之后要么全部执行，要么全部不执行。
- 一致性（Consistency）：事务的执行不能破坏数据库的完整性和一致性。
- 隔离性（Integrity）：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。

- 持久性（Durability）：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。

openGauss 事务模块实现了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。

2. openGauss 事务处理模块

openGauss 事务处理模块的总体结构如图 9.1 所示。

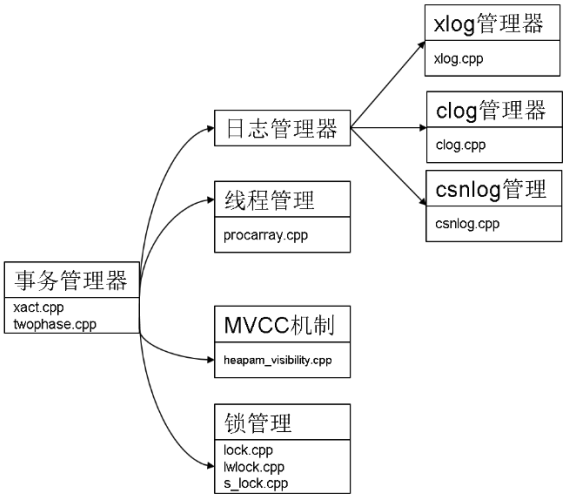

图 9.1 openGauss 事务处理模块的总体架构

在 openGauss 中，事务处理模块主要包括：

- 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，通过接受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。
- 日志管理器：用来记录事务执行的状态以及数据变化的过程，包括事务提交日志(CLOG)、事务提交序列日志（CSNLOG）以及事务日志（XLOG）。其中 CLOG 日志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。XLOG 日志已在第 7 章中探讨和实践了。
- 线程管理机制：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。
- 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。
- 锁管理器：实现系统的写并发控制，通过锁机制来保证事务写流程的隔离性。

3. 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数据库的 ACID 特性。

openGauss 将事务系统分为上层（事务块 TBlockState）以及底层（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，实现灵活支持客户端各类事务执行语句（BEGIN/START TRANSACTION/COMMIT/ROLLBACK/END）。

- 事务块 TBlockState：客户端查询的状态，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查询语句。
- 底层事务 TransState：内核端视角，记录了整个事务当前处于的具体状态。

事务块上层状态机 TBlockState 定义为枚举结构：

【源码】src/gausskernel/storage/access/transam/xact.cpp

```
/*
 *   transaction block states - transaction state of client queries
 *
 * Note: the subtransaction states are used only for non-topmost
 * transactions; the others appear only in the topmost transaction.
 */
typedef enum TBlockState {
    /* not-in-transaction-block states */
    TBLOCK_DEFAULT, /* idle */
    TBLOCK_STARTED, /* running single-query transaction */

    /* transaction block states */
    TBLOCK_BEGIN,         /* starting transaction block */
    TBLOCK_INPROGRESS,    /* live transaction */
    TBLOCK_END,           /* COMMIT received */
    TBLOCK_ABORT,         /* failed xact, awaiting ROLLBACK */
    TBLOCK_ABORT_END,     /* failed xact, ROLLBACK received */
    TBLOCK_ABORT_PENDING, /* live xact, ROLLBACK received */
    TBLOCK_PREPARE,       /* live xact, PREPARE received */
    TBLOCK_UNDO,          /* Need rollback to be executed for this topxact */

    /* subtransaction states */
    TBLOCK_SUBBEGIN,         /* starting a subtransaction */
    TBLOCK_SUBINPROGRESS,    /* live subtransaction */
    TBLOCK_SUBRELEASE,       /* RELEASE received */
    TBLOCK_SUBCOMMIT,        /* COMMIT received while TBLOCK_SUBINPROGRESS */
    TBLOCK_SUBABORT,         /* failed subxact, awaiting ROLLBACK */
    TBLOCK_SUBABORT_END,     /* failed subxact, ROLLBACK received */
    TBLOCK_SUBABORT_PENDING, /* live subxact, ROLLBACK received */
    TBLOCK_SUBRESTART,       /* live subxact, ROLLBACK TO received */
    TBLOCK_SUBABORT_RESTART, /* failed subxact, ROLLBACK TO received */
    TBLOCK_SUBUNDO           /* Need rollback to be executed for this subxact */
} TBlockState;
```

事务块的状态机通过函数调用实现状态转换，其状态转换图如图 9.2 所示。

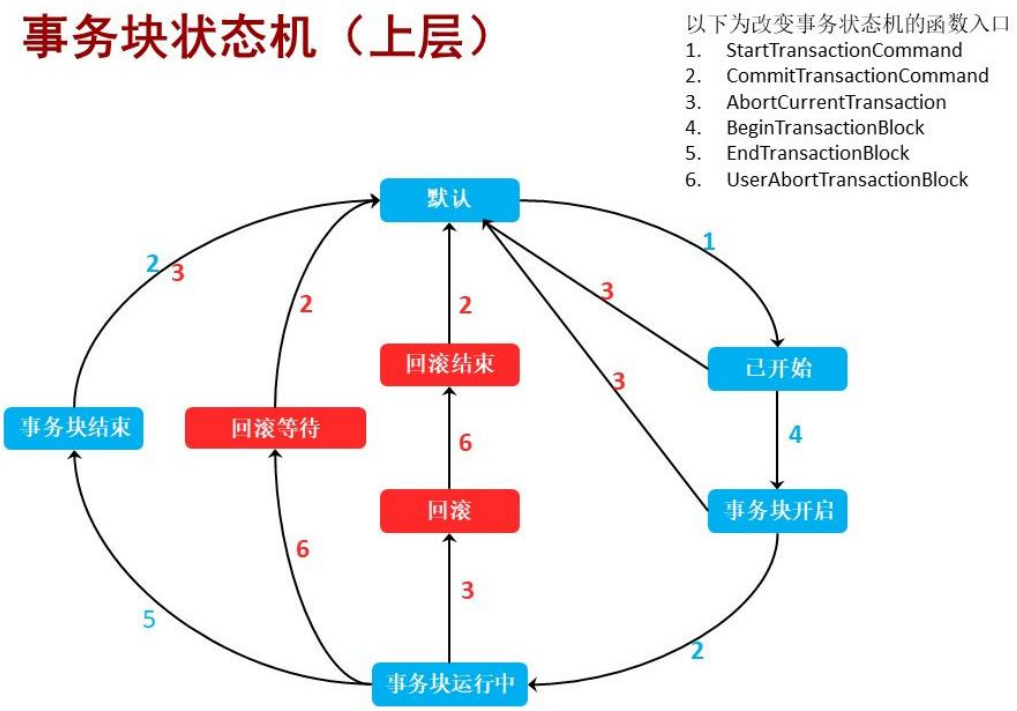

图 9.2 openGauss 事务块状态机的转换

为了便于理解，这里我们不关注子事务块的状态转换。

事务块状态机在正常情况下按照以下状态切换循环：

(1) 默认（TBLOCK_DEFAULT）

(2) 已开始（TBLOCK_STARTED）

(3) 事务块开启（TBLOCK_BEGIN）

(4) 事务块运行中（TBLOCK_INPROGRESS）

(5) 事务块结束（TBLOCK_END）

(6) 默认（TBLOCK_DEFAULT）

事务块状态机的剩余转换是在上述正常流程下各个状态点的异常处理分支：

- 在进入事务块运行中（TBLOCK_INPROGRESS）之前出错，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。
- 在事务块运行中（TBLOCK_INPROGRESS）出错分为 2 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）。
- 在用户执行 COMMIT 语句时出错：事务块结束（TBLOCK_END）->默认（TBLOCK_DEFAULT）。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。
- 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。

事务底层状态是从 openGauss 内核视角的事务状态，是真正意义上的事务状态。

事务底层状态机 TransState 定义为枚举结构：

【源码】src/gausskernel/storage/access/transam/xact.cpp

```
/*
 *   transaction states - transaction state from server perspective
 */
typedef enum TransState {
    TRANS_DEFAULT,    /* idle */
    TRANS_START,      /* transaction starting */
    TRANS_INPROGRESS, /* inside a valid transaction */
    TRANS_COMMIT,     /* commit in progress */
    TRANS_ABORT,      /* abort in progress */
    TRANS_PREPARE,     /* prepare in progress */
    TRANS_UNDO        /* applying undo */
} TransState;
```

事务底层状态机的转换如图 9.3 所示。

(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/回滚的过程中为 TARNS_COMMIT/ TRANS_ABORT

(5) 事务结束后事务状态回到 TRANS_DEFAULT

4. 事务状态机处理详细过程

下面使用一条 SQL 语句作为实例来说明事务状态机的运转详细过程。

在 gsql 客户端执行 SQL 语句:

创建 users 表

```
CREATE TABLE users
(
   u_id varchar(20),                  -- 用户 id，用于系统登录账户名（主键）
   u_passwd varchar(20),              -- 密码，用于系统登录密码
   u_name varchar(10),                -- 真实姓名
   u_idnum varchar(20),               -- 证件号码
   u_regtime timestamp,               -- 注册时间
   CONSTRAINT pk_users PRIMARY KEY (u_id)
);
```

插入数据

```
INSERT INTO users VALUES(1,'qweasd','张三','123456789', '2000-06-23 12:00:00');
INSERT INTO users VALUES(2,'qweasd','李四','123456712', '2000-06-24 13:00:00');
INSERT INTO users VALUES(3,'qweasd','王五','123456754', '2000-06-25 14:00:00');
INSERT INTO users VALUES(4,'qweasd','赵六','123456709', '2000-06-26 15:00:00');
INSERT INTO users VALUES(5,'qweasd','小明','123423709', '2000-06-27 15:00:00');
INSERT INTO users VALUES(6,'qweasd','小李','123423709', '2000-06-27 16:00:00');
INSERT INTO users VALUES(7,'qweasd','小赵','123423712', '2000-06-27 10:00:00');
INSERT INTO users VALUES(8,'qweasd','小红','142423709', '2000-06-27 11:00:00');
INSERT INTO users VALUES(9,'qweasd','小秦','163423709', '2000-06-27 19:00:00');
```

执行下面语句，开启一个事务，该事务中只有一条 SELECT 语句。

```
BEGIN;
```

```
   SELECT * FROM users;
END;
```

执行结果如下：

```
railway=# BEGIN;
BEGIN
railway=# SELECT * FROM users;
u_id | u_passwd | u_name |  u_idnum  |      u_regtime
------+----------+--------+-----------+---------------------
 1    | qweasd   | 张三   | 123456789 | 2000-06-23 12:00:00
 2    | qweasd   | 李四   | 123456712 | 2000-06-24 13:00:00
 3    | qweasd   | 王五   | 123456754 | 2000-06-25 14:00:00
 4    | qweasd   | 赵六   | 123456709 | 2000-06-26 15:00:00
 5    | qweasd   | 小明   | 123423709 | 2000-06-27 15:00:00
 6    | qweasd   | 小李   | 123423709 | 2000-06-27 16:00:00
 7    | qweasd   | 小赵   | 123423712 | 2000-06-27 10:00:00
 8    | qweasd   | 小红   | 142423709 | 2000-06-27 11:00:00
 9    | qweasd   | 小秦   | 163423709 | 2000-06-27 19:00:00
(9 rows)

railway=# END;
COMMIT
```

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，最后返回到事务块中。上述事务语句的整体执行流程如图 9.3 所示。

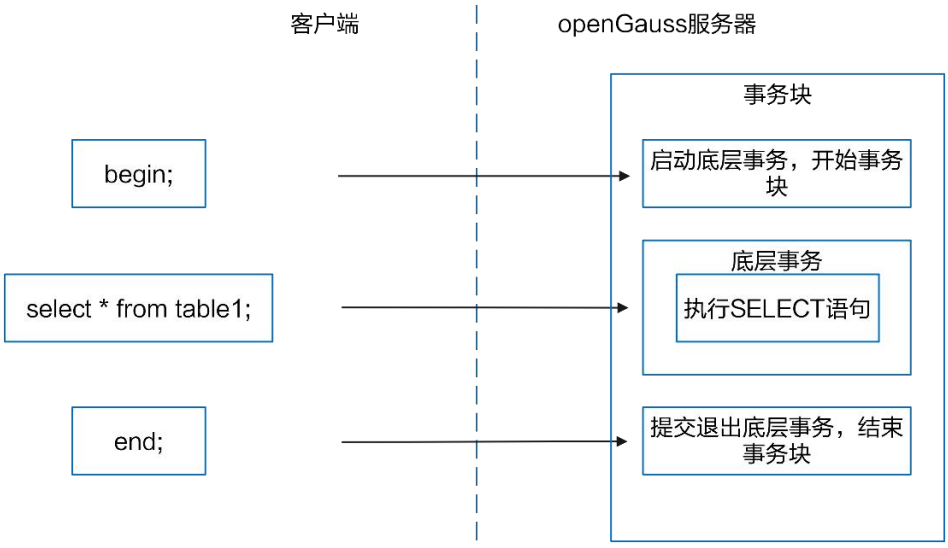

图 9.3 SQL 事务块语句执行整体流程

- BEGIN 语句执行流程

如图 9.4 所示。

(1) 入口函数 exec_simple_query 处理 BEGIN 命令。

(2) start_xact_command 函数开始一个查询命令，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 函数结束一个查询命令，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。

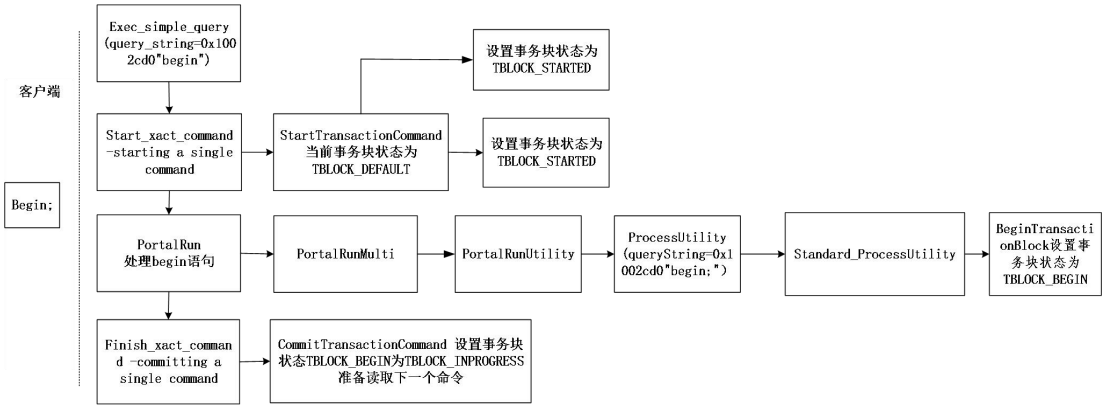

图 9.4 BEGIN 语句执行流程

- SELECT 语句执行流程

如图 9.5 所示。

(1) 入口函数 exec_simple_query 处理“SELECT * FROM users；”语句。

(2) start_xact_command 函数开始一个查询命令，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，不改变事务上层以及底层的状态。

(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 函数结束一条查询命令，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以及底层的状态。

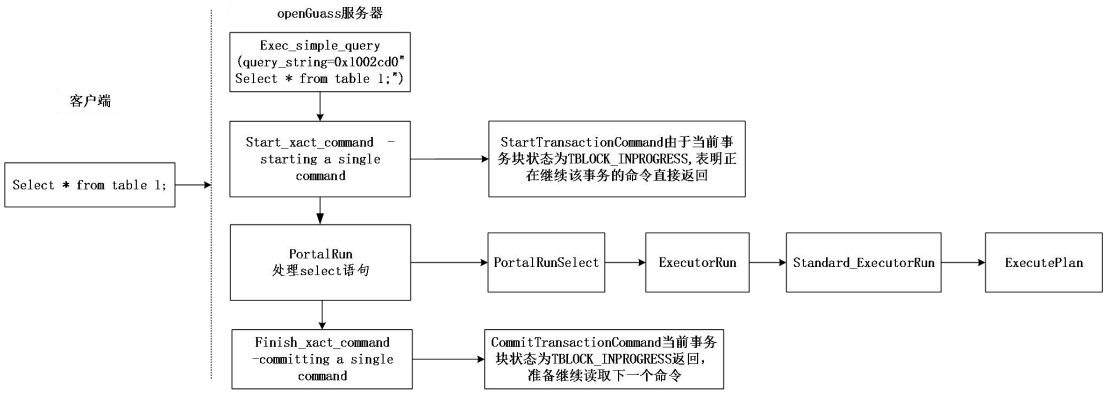

图 9.5 SELECT 语句执行流程

- END 语句执行流程

如图 9.5 所示。

(1) 入口函数 exec_simple_query 处理“END;”语句。

(2) start_xact_command 函数开始一个查询命令，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 函数结束查询命令，调用 CommitTransactionCommand 函数，当前事务块状态 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。

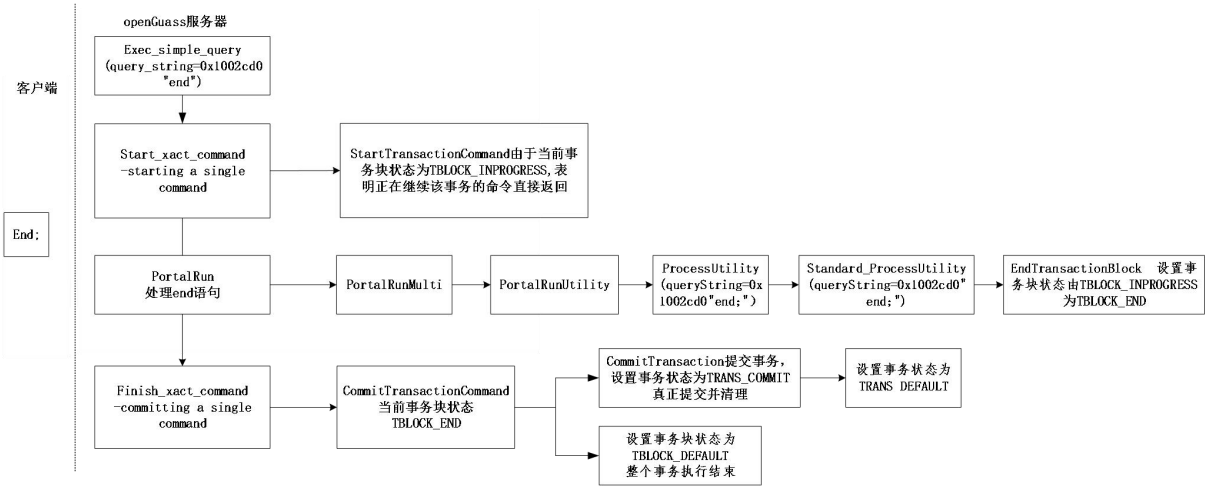

图 9.6 SELECT 语句执行流程

5. 事务状态转换相关函数

- 事务处理函数：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。表 8.1 给出了事务处理函数的介绍。

表 8.1 事务处理函数

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

- 事务执行函数，根据相应的状态机调用函数。表 8.2 给出了事务执行函数的介绍。

表 8.2 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
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

- 上层事务状态机控制函数。表 8.3 给出了上层事务状态机控制函数的介绍。

表 8.3 上层事务状态机控制函数

函数	**说明**
BeginTransactionBlock	开启一个显式事务时，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

6. 事务 ID 分配

openGauss 数据库会为每个事务分配唯一的标识符，即事务 id（transaction id，缩写 xid），xid 是 uint64 单调递增的序列。当事务结束后，使用 CLOG 日志记录事务是否提交，使用 CSNLOG（commit sequence number log）日志记录该事务提交的序列，用于可见性判断。

openGauss 对每一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，为了节省空间以及兼容老的版本，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。如图 9.7 所示。对于每一条元组真正的 xmin、xmax 计算公式即为：

```
页面 xid_base + 元组头中 xmin/xmax
```

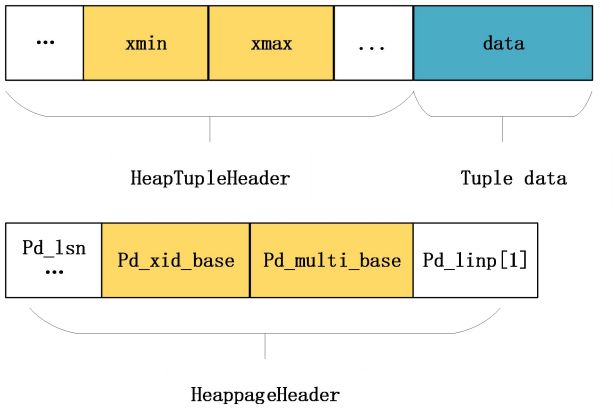

图 9.7 页面中与事务处理有个的字段结构

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。理论上 64 位 xid 已经足够使用：假设数据库的 tps 为 1000 万，即 1 秒钟处理 1000 万个事务，xid 可以使用 58 万年。

7. CLOG 和 CSNLOG

CLOG 日志维护事务 ID 到 CommitLog 的映射关系；CSNLOG 日志维护事务 ID 到 CommitSeqNoLog 的映射关系。由于内存资源有限，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。CLOG 和 CSNLOG 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。

- CLOG 用于记录事务 id 的提交状态。openGauss 中对于每个事务 id 使用 4 个 bit 位来标识它的状态。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
```

CLOG 文件的物理组织形式如图 9.8 所示。事务 0、1、4、5 还在运行中，事务 2 已经提交，事务 3 已经回滚。

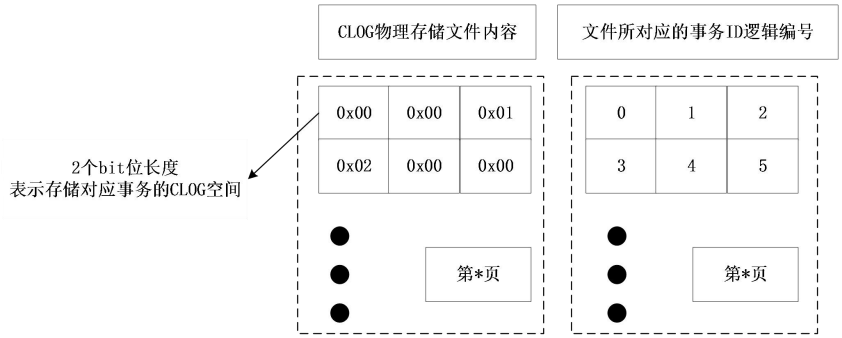

图 9.8 CLOG 文件的物理组织形式

- CSNLOG 用于记录事务提交的序列号。openGauss 为每个事务 id 分配 8 个字节 uint64 的 CSN 号，所以一个 8kB 页面能保存 1k 个事务的 CSN 号。同 xid 号类似，CSN 号预留了几个特殊的号。CLOG 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 文件的物理组织形式如图 9.9 所示。

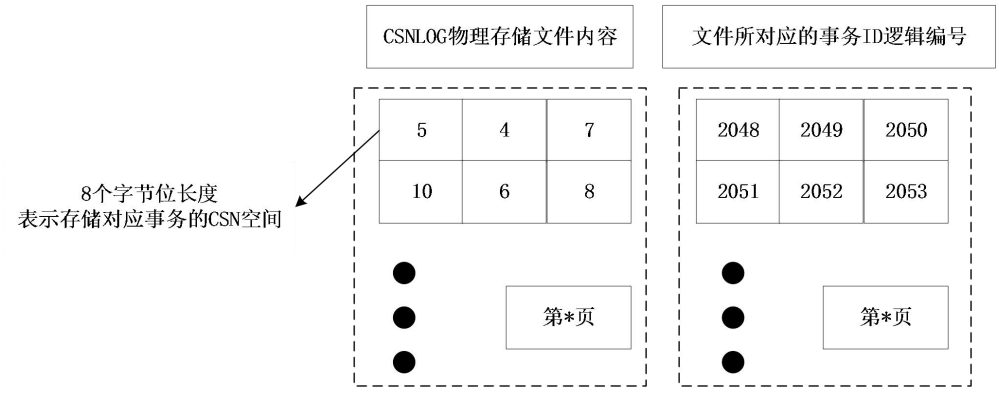

图 9.9 CSNLOG 文件的物理组织形式

事务 id 2048、2049、2050、2051、2052、2053 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。

8. MVCC 可见性判断机制

openGauss 利用多版本并发控制（MVCC）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。

下面介绍事务的隔离级别以及 openGauss 可见性判断 CSN 机制。

SQL 标准考虑了并行事务间应避免的现象，定义了以下几种隔离级别，如表 8.4 所示。

表 8.4 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

- 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。
- 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。
- 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。
- 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。

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。

9. CSN 机制

CSN 原理如图 9.10 所示。

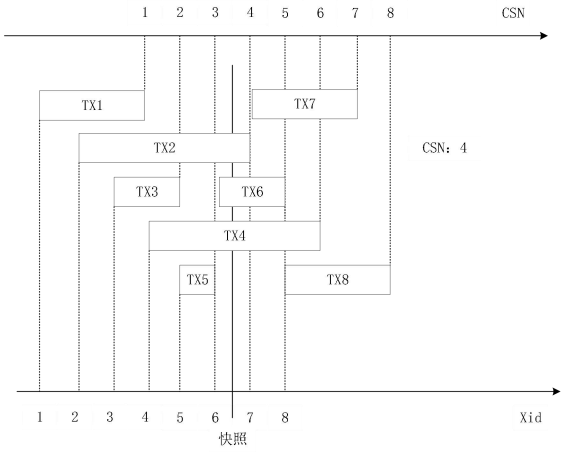

图 9.10 CSN 原理

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 9.10 中，实心竖线标识取 snapshot（快照）时刻，会获取最新提交 CSN（即 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 是未来还未开启的事务。对于当前 snapshot 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。

10. MVCC 快照可见性判断流程

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。例如如图 9.10 给出的快照时刻，事务 TX1、TX3、TX5 已提交，TX2、TX4、TX6 正在运行，当前活跃的最小 xid 即 snapshot.xmin 由 TX2 给出，值为 2；最新提交的事务为 TX5，其 xid 号和 CSN 号分别为 5 和 3，由此得出 snapshot.xmax 和 snapshot.csn 的值分别为 6 和 4。

可见性判断的简易流程如图 9.10 所示。

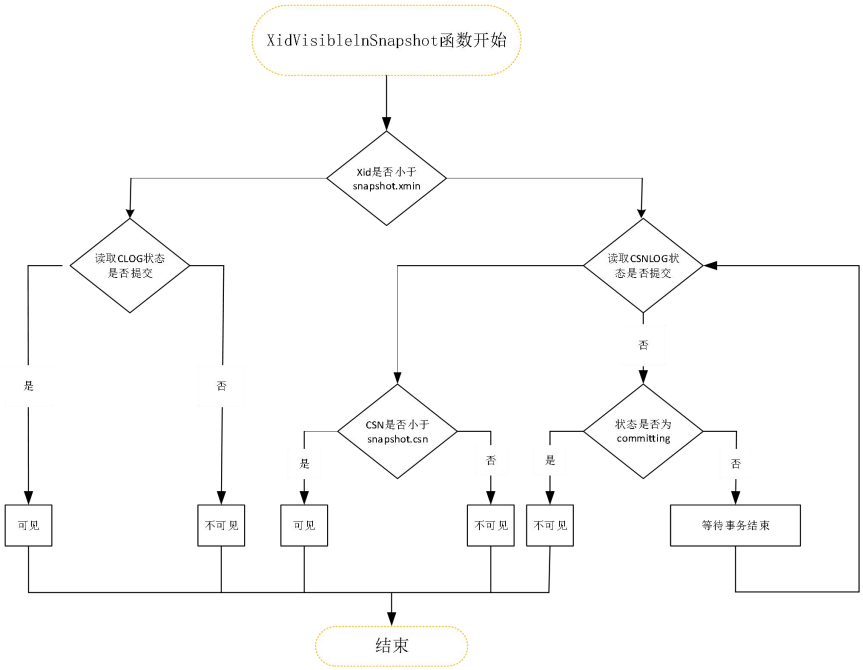

图 9.11 MVCC 快照可见性判断流程

(1) xid 大于等于 snapshot.xmax 时，该事务 id 不可见。
(2) xid 比 snapshot.xmin 小时，说明该事务 id 在本次事务启动以前已经结束，需要去 CLOG 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。

11. 事务提交流程

事务提交过程中涉及到 CLOG 和 CSNLOG 的设置，如图 9.12 所示。

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

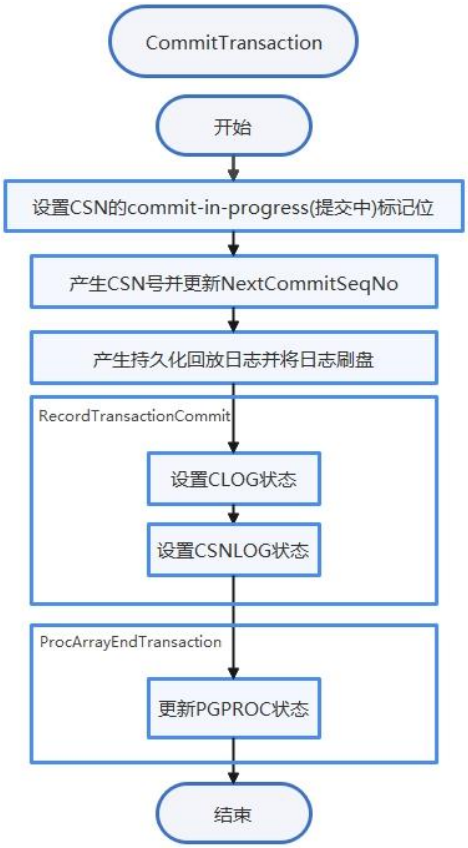

图 9.12 事务提交流程

12.3.2 openGauss 锁机制

数据库对公共资源的并发控制是通过锁来实现的，根据锁的用途不同，通常可以分为 3 种：自旋锁（spinlock）、轻量级锁（LWLock，light weight lock）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。

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。只有 2 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。

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。存在 2 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。但是由于代码复杂度及各类异常处理，openGauss 提供了 LWLock 的死锁检测机制，避免各类异常场景产生的 LWLock 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 LWLock。
(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 LWLock 全部回滚释放，避免残留阻塞后续操作。

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来定义的。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 8.5 所示，其中 √ 表示两个锁操作可以并发。

表 8.5 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	√	√	√	√	√	√	-	-
3. ROW EXCLUSIVE	√	√	√	√	-	-	-	-
4. SHARE UPDATE EXCLUSIVE	√	√	√	-	-	-	-	-
5. SHARELOCK	√	√	-	-	√	-	-	-
6. SHARE ROW EXCLUSIVE	√	√	-	-	-	-	-	-
7. EXCLUSIVE	√	-	-	-	-	-	-	-
8. ACCESS EXCLUSIVE	-	-	-	-	-	-	-	-

openGauss 的锁机制继承于 PostgreSQL，共有八种表级锁，如下：

- Share（S）
- Exclusive（E）
- Access Share（AS）
- Access Exclusive（AE）
- Row Share（RS）
- Row Exclusive（RE）
- Share Update Exclusive（SUE）
- Share Row Exclusive（SRE）

在这八种表级锁中，最简单最初始的就是 Share 锁与 Exclusive 锁，Share 锁就是共享锁，Exclusive 就是排他锁，我们可以把 Share 锁理解为读锁，Exclusive 锁理解为写锁。如果一个表被一个事务施加了读锁，那么其他事务就不能修改这个表。如果一个表被一个事务加上了写锁，这时别的事务不能写也不能读这条数据。

随着数据库技术的发展，出现了多版本功能，就有了 Access Share 和 Access Exclusive 锁，锁中的关键字 Access 是与多版本读特性相关的，有了多版本的功能后，如果修改一行数据，实际并没有改原先那行数据，而是复制了一个新行，修改都在新行上，事务不提交，其他人是看不到修改的这条数据的。由于旧行数据没有变化，在修改过程中，读数据的人仍然可以读到旧的数据。所以一个表被一个事务添加了 Access Share 锁后，仍然可以被修改数据，而一个表被一个事务添加了 Access Exclusive 锁后，仍然可以被读取数据。

由于上述锁的加锁对象都是表，在实际生产环境中这使得加锁范围太大，导致并发量无法提高，于是人们提出了行级锁的概念，并使用更新锁这种机制来描述行级锁与表级锁之间的关系。方法就是当我们要修改表中的某一行数据时，需要先在表上加一种锁，如 Row Share 和 Row Exclusive，表示即将在表的部分行上加共享锁或排他锁。也就是我们常说的更新锁。

剩下的两种锁 Share Update Exclusive 和 Share Row Exclusive 使用频率较低，Share Update Exclusive 用来保护一个表的模式不被并发修改，以及禁止在目标表上执行垃圾回收命令（VACUUM ）。Share Row Exclusive 则禁止对表进行任何的并发修改，是独占锁，一个会话中只能获取一次。

加锁对象数据结构，通过对 field1->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，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者和等着串起来，其队列关系如图 9.13 所示。

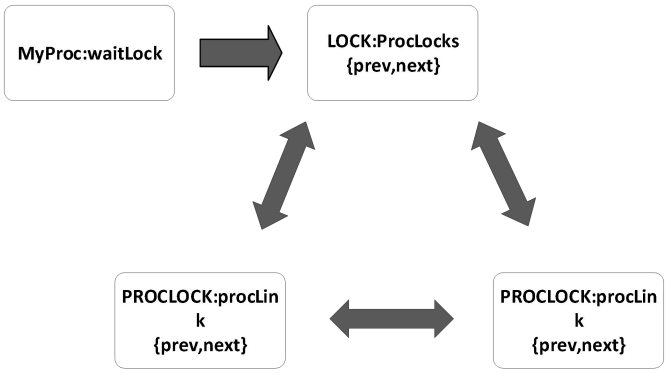

图 9.13 t_thrd.proc 结构体队列关系图

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。

Share、Access Share、Row Exclusive、Access Exclusive 四种锁是最为常见的表级锁。下面介绍一下什么 SQL 语句会在表上施加对应锁：

表 8.6 表级锁介绍

锁模式	**解释**
Share	共享锁、CREATE INDEX 命令会获得这种锁模式。

Access Share	共享锁、SELECT 语句将会在它查询的表上获取 Access Share 锁，一般来说任何一个对表上的只读查询操作都将获取这种类型锁。
Row Exclusive	更新锁、UPDATE/DELETE/INSERT 命令会在目标表上获得这种类型的锁，一般来说更改表数据的命令都将在这张表上获得 Row Exclusive 锁。
Access Exclusive	排他锁、这种模式保证了当前只有一个人访问这张表。ALTER TABLE，DROP TABLE，TRUNCATE 等命令都会获得这种类型锁。

从表中我们可以看到，不同的锁适合不同 SQL 语句，对于 Share 锁来说，添加之后，只能进行读取操作，因此适用于 CREATE INDEX 这种场景下。与上述四种模式中的 Row Exclusive，Access exclusive 锁模式冲突，Share 模式保护一张表数据不被并发的更改。

而其它普通的 SELECT 语句使用 Access Share 语句即可，既可以读取也可以写入，保证了数据库在生产环境中的并发性。它只与 Access Exclusive 锁模式冲突。

Row Exclusive 更新锁适用于更改表数据的语句，它用来对对应行施加行级上的排他锁，与上述四种模式中的 Share，Access Exclusive 模式冲突。

Access Exclusive 排他锁则是表级上的排他锁，与其它所有锁模式均冲突。

4. 死锁检测机制

死锁主要是由于进程 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，从而数据库就会一直处于阻塞状态。如图 9.14 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。

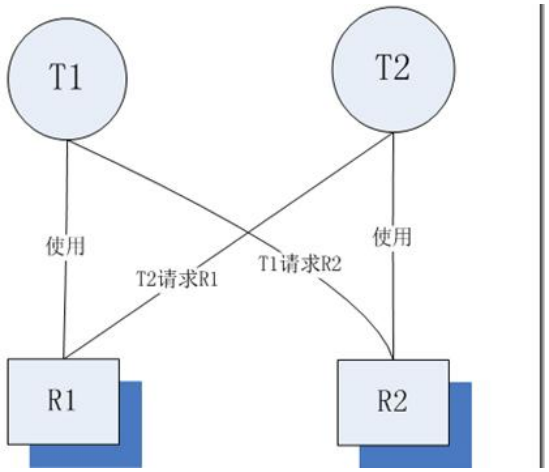

图 9.14 死锁状态

形成死锁的必要条件是：资源的请求与保持。每一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 LWLock 的死锁检测及常规锁的死锁检测机制。

- LWLock 死锁检测自愈

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，成功获得锁之后置该时间戳为 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。这么做的目的是防止频繁诊断影响业务正常执行能。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。如图 9.15 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 0。对于一个发生死锁的线程，它的锁请求是 wait 状态，时间戳也不会清 0，且与当前运行时间戳数值的差值越来越大。由 GUC 参数 fault_mon_timeout 控制检测间隔时间，默认为 5s。LWLock 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：跟几秒检测之前的时间对比，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 LWLock 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，这部分会在下面一小节中详细介绍。死锁检测需要两部分的信息：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。

(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，可以准确的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，以及等锁信息。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;
```

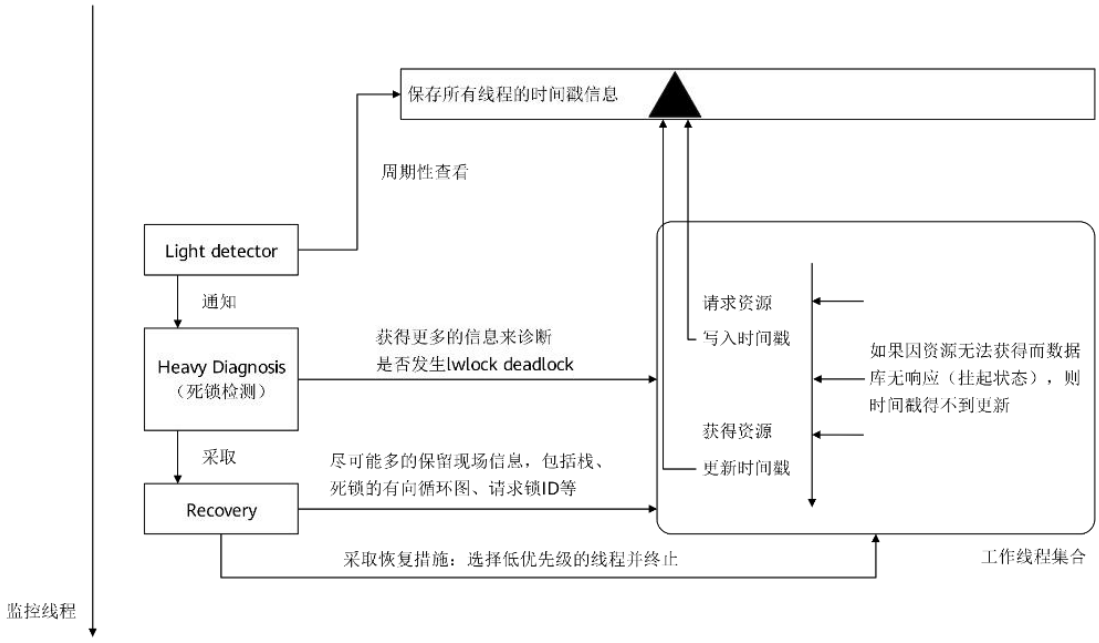

图 9.15 LWLock 死锁检测自愈

- 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 timer 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 soft edge，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 soft edge，那么只能终止当前等锁的事务，解决死锁等待环。如图 9.16 所示，虚线代表 soft edge，实线代表 hard fdge。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 soft edge，进行一次调整成为图 9.16 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 soft edge，检测到死锁。

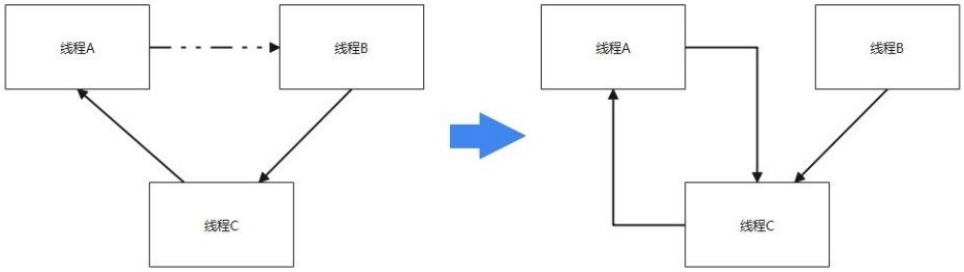

图 9.16 常规锁死锁检测示意图

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 soft edge，返回 false 并且尝试解决死锁冲突。

(3) check_stack_depth：openGauss 会检查死锁递归检测堆栈（死锁检测递归栈过长，会引发在死锁检测时，长期持有所有锁的 LWLock 分区，从而阻塞业务）。

(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。

(5) FindLockCycle：检查是否有死锁环。

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

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十分钟，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.")));
            }
        }
    }
}
```

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;     /* the lock whose wait queue is described */
    PGPROC **procs; /* array of PGPROC *'s in new wait order */
    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. 无锁原子操作

openGauss 封装了 32、64、128 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 在更新前没有被更新，则将 newval 写到 dest 地址。dest 地址的值没有被更新，就返回 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 在更新前没有被更新，则将 newval 写到 dest 地址。dest 地址的值没有被更新，就返回 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，详情可以参考下一小结。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
}
```

12.3.3 关键数据结构及函数

1. 结构体 SnapshotData

表示事务处理时的快照。

【源码】src/include/utils/snapshot.h

```
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，大于等于改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，还是新分配内存拷贝的  */

    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. 函数 HeapTupleSatisfiesMVCC

用于一般读事务的快照扫描，基于 CSN 的大体逻辑。

【源码】src/gausskernel/storage/access/heap/heapam_visibility.cpp

```
/*
 * HeapTupleSatisfiesMVCC
 *       True iff heap tuple is valid for the given MVCC snapshot.
 *
 *   Here, we consider the effects of:
 *       all transactions committed as of the time of the given snapshot
 *       previous commands of this transaction
 *
 *   Does _not_ include:
 *       transactions shown as in-progress by the snapshot
 *       transactions started after the snapshot was taken
 *       changes made by the current command
 *
 * This is the same as HeapTupleSatisfiesNow, except that transactions that
 * were in progress or as yet unstarted when the snapshot was taken will
 * be treated as uncommitted, even if they have committed by now.
 *
 * (Notice, however, that the tuple status hint bits will be updated on the
 * basis of the true state of the transaction, even if we then pretend we
 * can't see it.)
 */
static bool HeapTupleSatisfiesMVCC(HeapTuple htup, Snapshot snapshot, Buffer buffer)
{
bool HeapTupleSatisfiesMVCC(HeapTuple htup, Snapshot snapshot, Buffer buffer)
{
```

```
    …… /*  初始化变量  */

    if (!HeapTupleHeaderXminCommitted(tuple)) {
    /* 此处先判断用一个 bit 位记录的 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 是
把事务状态直接记录在元组头中，用一个 bit 位来表示提交和回滚状态。openGauss 并不会在事务提交或者回滚时主动
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，则从 CLOG 中读
取并设置，否则直接读取 hint bit 的值），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，也即是同一个事务内，后面的查询可以查到当前事务之前插
入的扫描结果  */
          …….
        } else {  /*  如果扫描别的事务，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 的判断同 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，则
不可见，否则可见，此处不再赘述 */
```

```
    if (!(tuple->t_infomask & HEAP_XMAX_COMMITTED)) {
        if (TransactionIdIsCurrentTransactionId(HeapTupleHeaderGetXmax(page, tuple))) {
            if (HeapTupleHeaderGetCmax(tuple, page) >= snapshot->curcid)
                return true; /* 在扫面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. 结构体 LWLock

轻量级锁。

【源码】src/include/storage/lock/lwlock.h

```
#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 LWLock 的 state 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;
```

4. 与锁相关的一些结构体与宏定义

- LOCKMODE 定义

【源码】src/include/storage/lock/lock.h

```
/*
 * LOCKMODE is an integer (1..N) indicating a lock type.  LOCKMASK is a bit
 * mask indicating a set of held or requested lock types (the bit 1<<mode
 * corresponds to a particular lock mode).
 */
typedef int LOCKMASK;
typedef int LOCKMODE;

/* MAX_LOCKMODES cannot be larger than the # of bits in LOCKMASK */
#define MAX_LOCKMODES 10

#define LOCKBIT_ON(lockmode) (1U << (lockmode))
#define LOCKBIT_OFF(lockmode) (~(1U << (lockmode)))
```

- LockMethod 定义

【源码】src/include/storage/lock/lock.h

```
/*
 * This data structure defines the locking semantics associated with a
 * "lock method".  The semantics specify the meaning of each lock mode
 * (by defining which lock modes it conflicts with).
 * All of this data is constant and is kept in const tables.
 *
 * numLockModes -- number of lock modes (READ,WRITE,etc) that
 *       are defined in this lock method.  Must be less than MAX_LOCKMODES.
 *
 * conflictTab -- this is an array of bitmasks showing lock
 *       mode conflicts.  conflictTab[i] is a mask with the j-th bit
 *       turned on if lock modes i and j conflict.  Lock modes are
 *       numbered 1..numLockModes; conflictTab[0] is unused.
 *
 * lockModeNames -- ID strings for debug printouts.
 *
```

```
 * trace_flag -- pointer to GUC trace flag for this lock method.  (The
 * GUC variable is not constant, but we use "const" here to denote that
 * it can't be changed through this reference.)
 */
typedef struct LockMethodData {
    int numLockModes;
    const LOCKMASK* conflictTab;
    const char* const* lockModeNames;
    const bool* trace_flag;
} LockMethodData;

typedef const LockMethodData* LockMethod;

/*
 * Lock methods are identified by LOCKMETHODID.  (Despite the declaration as
 * uint16, we are constrained to 256 lockmethods by the layout of LOCKTAG.)
 */
typedef uint16 LOCKMETHODID;

/*
 * These identify the known lock methods
 * Attention: adding new lock method should also modify MAX_LOCKMETHOD in knl_session.h
 */
#define DEFAULT_LOCKMETHOD 1
#define USER_LOCKMETHOD 2
```

- 不同的 LOCKMODE 值

【源码】src/include/storage/lock/lock.h

```
/*
 * These are the valid values of type LOCKMODE for all the standard lock
 * methods (both DEFAULT and USER).
 */

/* NoLock is not a lock mode, but a flag value meaning "don't get a lock" */
#define NoLock 0

#define AccessShareLock 1  /* SELECT */
#define RowShareLock 2     /* SELECT FOR UPDATE/FOR SHARE */
#define RowExclusiveLock 3 /* INSERT, UPDATE, DELETE */
#define ShareUpdateExclusiveLock                          \
    4               /* VACUUM (non-FULL),ANALYZE, CREATE \
                     * INDEX CONCURRENTLY */
#define ShareLock 5 /* CREATE INDEX (WITHOUT CONCURRENTLY) */
#define ShareRowExclusiveLock               \
    6 /* like EXCLUSIVE MODE, but allows ROW \
```

```
      * SHARE */
#define ExclusiveLock                   \
    7 /* blocks ROW SHARE/SELECT...FOR \
       * UPDATE */
#define AccessExclusiveLock             \
    8 /* ALTER TABLE, DROP TABLE, VACUUM \
       * FULL, and unqualified LOCK TABLE */
```

- LOCKTAG 类型定义

【源码】src/include/storage/lock/lock.h

```
/*
 * LOCKTAG is the key information needed to look up a LOCK item in the
 * lock hashtable. A LOCKTAG value uniquely identifies a lockable object.
 *
 * The LockTagType enum defines the different kinds of objects we can lock.
 * We can handle up to 256 different LockTagTypes.
 */
typedef enum LockTagType {
    LOCKTAG_RELATION, /* whole relation */
    /* ID info for a relation is DB OID + REL OID; DB OID = 0 if shared */
    LOCKTAG_RELATION_EXTEND, /* the right to extend a relation */
    /* same ID info as RELATION */
    LOCKTAG_PARTITION,          /*partition*/
    LOCKTAG_PARTITION_SEQUENCE, /*partition sequence*/
    LOCKTAG_PAGE,               /* one page of a relation */
    /* ID info for a page is RELATION info + BlockNumber */
    LOCKTAG_TUPLE, /* one physical tuple */
    /* ID info for a tuple is PAGE info + OffsetNumber */
    LOCKTAG_TRANSACTION, /* transaction (for waiting for xact done) */
    /* ID info for a transaction is its TransactionId */
    LOCKTAG_VIRTUALTRANSACTION, /* virtual transaction (ditto) */
    /* ID info for a virtual transaction is its VirtualTransactionId */
    LOCKTAG_OBJECT, /* non-relation database object */
    /* ID info for an object is DB OID + CLASS OID + OBJECT OID + SUBID */
    LOCKTAG_CSTORE_FREESPACE, /* cstore free space */

    /*
     * Note: object ID has same representation as in pg_depend and
     * pg_description, but notice that we are constraining SUBID to 16 bits.
     * Also, we use DB OID = 0 for shared objects such as tablespaces.
     */
    LOCKTAG_USERLOCK, /* reserved for old contrib/userlock code */
    LOCKTAG_ADVISORY, /* advisory user locks */
    /* same ID info as spcoid, dboid, reloid */
    LOCKTAG_RELFILENODE, /* relfilenode */
```

```
    LOCKTAG_SUBTRANSACTION, /* subtransaction (for waiting for subxact done) */
    /* ID info for a transaction is its TransactionId + SubTransactionId */
    LOCKTAG_UID,
    LOCK_EVENT_NUM
} LockTagType;

#define LOCKTAG_LAST_TYPE (LOCK_EVENT_NUM - 1)
extern const char* const LockTagTypeNames[];

/*
 * The LOCKTAG struct is defined with malice aforethought to fit into 16
 * bytes with no padding.  Note that this would need adjustment if we were
 * to widen Oid, BlockNumber, or TransactionId to more than 32 bits.
 *
 * We include lockmethodid in the locktag so that a single hash table in
 * shared memory can store locks of different lockmethods.
 */
typedef struct LOCKTAG {
    uint32 locktag_field1;      /* a 32-bit ID field */
    uint32 locktag_field2;      /* a 32-bit ID field */
    uint32 locktag_field3;      /* a 32-bit ID field */
    uint32 locktag_field4;      /* a 32-bit ID field */
    uint16 locktag_field5;      /* a 16-bit ID field */
    uint8 locktag_type;         /* see enum LockTagType */
    uint8 locktag_lockmethodid; /* lockmethod indicator */
} LOCKTAG;
```

5. 获取锁的相关函数

需要获取锁的相关代码会调用 lock.cpp 中的 LockAcquire 函数，该函数会调用 LockAcquireExtended 函数。

【源码】src/gausskernel/storage/lmgr/lock.cpp

```
LockAcquireResult LockAcquire(const LOCKTAG *locktag, LOCKMODE lockmode,
                      bool sessionLock, bool dontWait, bool allow_con_update, int waitSec)
{
    return LockAcquireExtended(locktag, lockmode, sessionLock, dontWait, true, allow_con_update,
waitSec);
}
```

LockAcquireExtended 函数会调用 LockAcquireExtendedXC 函数。

【源码】src/gausskernel/storage/lmgr/lock.cpp

```
LockAcquireResult LockAcquireExtended(const LOCKTAG *locktag, LOCKMODE lockmode,
  bool sessionLock, bool dontWait, bool reportMemoryError, bool allow_con_update, int waitSec)
{
    return LockAcquireExtendedXC(locktag, lockmode, sessionLock, dontWait, reportMemoryError,
           false, allow_con_update, waitSec);
}
```

LockAcquireExtendedXC 函数中实际执行获取锁的具体操作。

【源码】src/gausskernel/storage/lmgr/lock.cpp

```
static LockAcquireResult LockAcquireExtendedXC(const LOCKTAG *locktag, LOCKMODE lockmode, bool
sessionLock,
                                               bool  dontWait,  bool  reportMemoryError,  bool
only_increment,
                                               bool allow_con_update, int waitSec)
{
    LOCKMETHODID lockmethodid = locktag->locktag_lockmethodid;
    LockMethod lockMethodTable;
    LOCALLOCKTAG localtag;
    LOCALLOCK *locallock = NULL;
    LOCK *lock = NULL;
    PROCLOCK *proclock = NULL;
    bool found = false;
    ResourceOwner owner = NULL;
    uint32 hashcode;
    LWLock *partitionLock = NULL;
    int status;
    bool log_lock = false;
... 省略若干行
}
```

12.4 实验步骤

12.4.1 查看锁信息

1. 创建表

```
CREATE TABLE users
(
   u_id varchar(20),               -- 用户 id，用于系统登录账户名（主键）
   u_passwd varchar(20),           -- 密码，用于系统登录密码
   u_name varchar(10),             -- 真实姓名
   u_idnum varchar(20),            -- 证件号码
   u_regtime timestamp,            -- 注册时间
   CONSTRAINT pk_users PRIMARY KEY (u_id)
);
```

2. 插入一些随机数据

```
INSERT INTO users VALUES (
    to_char(generate_series(100, 199)),
    to_char(random() * 100000000, '09999999'),
    to_char(10000 + random() * 10000, '09999'),
    to_char(10000000000 + random() * 10000000000, '09999999999'),
```

```
   CURRENT_DATE + floor((random() * 15))::int);
```

3. 首先通过 SELECT pg_backend_pid();命令，我们可以查看当前会话的会话 id，实验结果如下；

```
openGauss=# SELECT pg_backend_pid( );
pg backend pid
------------------
139628707247872
(1 row)
```

4. 再从系统表中查找到当前会话所添加的锁：

```
SELECT locktype,database,relation,pid,mode FROM pg_locks WHERE pid= 139628707247872;
```

查找结果如下：

```
openGauss=# SELECT locktype,database,relation,pid,mode FROM pg_locks WHERE pid=139628707247872;
 locktype   | database | relation |        pid      |     mode
------------+----------+----------+-----------------+----------------
 relation   |  14553   |   12009  | 139628707247872 | AccessShareLock
 virtualxid |          |          | 139628707247872 | ExclusiveLock
```

可以看到，上面列出了当前会话的锁，locktype 列表示被施加锁的类型，关注为 relation 的即可，relation 列表示对应 relation 的 id。Mode 表示锁类型。

5. 接下来找到我们所创建的表 users 的 relation id。

查找结果如下：

```
openGauss=# SELECT oid FROM pg_class WHERE relname='users';
oid
-------
24582
(1 row)
```

Users 表的对应 id 为 24582。

12.4.2 复现 Share 锁

从实验原理中了解到，CREATE INDEX 命令才会添加 Share 锁，由于命令执行时间太短，所以我们通过 Begin 命令来生成事务，再创建索引，以方便获取锁信息。

1. 生成事务，添加 Share 锁。

```
BEGIN;
CREATE INDEX idx_idnum ON users(u_idnum);
SELECT locktype,database,relation,pid,mode FROM pg_locks WHERE pid= 139628707247872;
```

执行结果如下：

```
openGauss=# BEGIN;
BEGIN
openGauss=# CREATE INDEX idx_idnum ON users(u_idnum);
CREATE INDEX
openGauss=# SELECT locktype, database, relation, pid, mode FROM pg_locks WHERE pid=
139628707247872;
  locktype    |  database |  relation  |     pid         |      mode
```

```
-------------+----------+-----------+---------------+---------------
 relation    |    14553 |     12009 | 139628707247872| AccessSharelock
 Virtualxid  |          |           | 139628707247872| ExclusiveLock
 relation    |    14553 |     24582 | 139628707247872| AccessSharelock
 relation    |    14553 |     24582 | 139628707247872| Sharelock
 transactiontd|          |           | 139628707247872| ExclusiveLock
 relation    |    14553 |     24587 | 139628707247872| AccessExclusiveLock
(6 rows)
```

可以看到，在 users 表上添加了 Share 锁。由于 Share 锁与 Row Exclusive，Access Exclusive 锁模式冲突，所以此时应该无法进行 INSERT、ALTER 或 DROP 等操作。但是可以执行 SELECT 操作。

2. 进行 INSERT 操作

   打开另一个会话窗口，进行 INSERT 操作，发现该命令**被堵塞**，无法执行完成。如下：

```
openGauss=# INSERT INTO users VALUES ('1234','1234','asdzxc','1234','2000-04-20');
```

3. 进行 SELECT 操作。

   打开另一个会话窗口，进行 SELECT 操作，因为 Share 锁和之前第 2 步中的 INSERT 操作都不会阻塞该 SELECT 语句，所以顺利执行完成。执行结果如下：

```
openGauss=# SELECT * FROM users WHERE u_id='100';
u_id |  u_passwd | u_name |   u_idnum     | u_regtime
-----+-----------+--------+---------------+-----------------
 100 |  27850548 | 16514  |   13452643161 | 2022-05-29 00:00:00
(1 row)
```

4. 提交事务，释放 Share 锁

   在第一个会话中提交事务，发现第二个会话中的 INSERT 操作成功执行。执行结果如下：

```
openGauss=# COMMIT;
COMMIT
openGauss=# INSERT INTO USer VALUES ('1234' ,'1234' ,'asdzxc" ,'1234' ,' 2000-04-20');
INSERT 0 1
```

12.4.3 复现 Access Share 锁

从实验原理中了解到，普通的查询命令都会添加 Access Share 锁.

1. 生成锁

```
BEGIN;
SELECT * FROM users WHERE u_id='100';
SELECT locktype,database,relation,pid,mode FROM pg_locks WHERE pid= 139628707247872;
```

可以看到，在 users 表上添加了 Access Share 锁。由于 Access Share 锁与 Access exclusive 锁模式冲突，所以此时无法进行 ALTER 或 DROP 等表级操作。但是可以执行 SELECT、INSERT、UPDATE 等操作。执行结果如下：

```
openGauss=# Begin;
Begin
openGauss=# SELECT * FROM users WHERE u_id="100';
```

```
 u_id  |   u_passwd |  u_name |    u_idnum    |       u_regtime
-------+------------+---------+--------------+----------------------------
  100  |   27371846 |   12400 |   18015805623|   2022-05-18 00:00:00
(1 row)
openGauss=# SELECT locktype,database,relation,pid,mode FROM pg_ locks WHERE pid= 139628707247872;
locktype   |  database |   relation  |     pid          |     mode
-----------+-----------+-------------+-----------------+-------------------
relation   |     14553 |       12009 | 139628707247872 | AccessShareLock
relation   |     14553 |       24587 | 139628707247872 | AccessShareLock
relation   |     14553 |       24585 | 139628707247872 | AccessShareLock
relation   |     14553 |       24582 | 139628707247872 | AccessShareLock
virtualxid |           |             | 139628707247872 | ExclusiveLock
(5 rows)
```

2. 进行 SELECT 操作

   打开另一个会话窗口，进行 SELECT 操作，顺利执行完成。执行结果如下

```
openGauss=# SELECT * FROM users WHERE u_id = '120';
 u_id  |   u_passwd |  u_name |    u_idnum    |       u_regtime
-------+------------+---------+--------------+---------------------
   120 |   88119327 |  17967  | 19510355243  |   2022-05-30 00:00:00
 (1 row)
```

3. 进行 ALTER 操作

   打开另一个会话窗口，进行 ALTER 操作，发现该命令**被堵塞**，无法执行完成。

   执行结果如下：

```
openGauss=# ALTER TABLE users ADD(address varchar(10));
WARNING: Session unused timeout
FATAL: terminating connection due to administrator command
Could not send data to server: Broken pipe
The connection to the server was lost. Attempting reset: Succeeded.
openGauss=# ALTER TABLE users ADD(address warchar(10));
```

4. 提交事务，释放锁

   在第一个会话中提交事务，发现第三个会话中的 ALTER 成功执行。执行结果如下：

```
openGauss=# COMMIT;
COMMIT
openGauss=# ALTER TABLE users ADD(address warchar(10));
ALTER TABLE
```

12.4.4 复现 Row Exclusive 锁

从实验原理中了解到，更改表数据的语句会添加 Row Exclusive 锁。通过 Begin 命令来生成事务，再更改表数据，以获取锁信息，执行结果如下：

1. 生成锁

```
openGauss=# BEGIN;
BEGIN
openGauss=# UPDATE users SET u_passwd='123asd' WHERE u_id='120';
```

```
UPDATE 1
openGauss=# SELECT locktype,database,relation,pid,mode FROM pg_locks WHERE pid= 139628707247872;
locktype     |  database  |   relation  |      pid        |       mode
-------------+------------+-------------+-----------------+--------------------
relation     |      14553 |       12009 | 139628707247872 | AccessshareLock
relation     |      14553 |       24587 | 139628707247872 | RowExcTustyelock
relation     |      14553 |       24595 | 139628707247872 | RowExcTustyelock
relation     |      14553 |       24582 | 139628707247872 | RowExcTustyelock
virtualxid   |            |             | 139628707247872 | ExclusiveLock
transactionid|            |             | 139628707247872 | ExclusiveLock
(6 rows)
```

可以看到，确实在 users 表上添加了 Row Exclusive 锁。由于 Row Exclusive 锁与 Share、Access exclusive 锁模式冲突，所以此时应该无法进行 CREATE INDEX、ALTER 或 DROP 等操作。

2. 此时打开另一个会话窗口，进行 CREATE INDEX 操作，发现该命令**被堵塞**，无法执行完成。执行结果如下：

```
openGauss=# CREATE INDEX idx_name ON users(u_name);
```

3. 再在第一个会话中提交事务，发现第二个会话中的 CREATE INDEX 操作成功执行。第一个会话执行结果如下：

```
openGauss=# BEGIN;
BEGIN
openGauss=# UPDATE users SET u_passwd='123asd' WHERE u_id='120';
UPDATE 1
openGauss=# SELECT locktype,database,relation,pid,mode FROM pg_locks WHERE pid= 139628707247872;
locktype     |  database  |   relation  |      pid        |       mode
-------------+------------+-------------+-----------------+--------------------
relation     |      14553 |       12009 | 139628707247872 | AccessshareLock
relation     |      14553 |       24587 | 139628707247872 | RowExcTustyelock
relation     |      14553 |       24585 | 139628707247872 | RowExcTustyelock
relation     |      14553 |       24582 | 139628707247872 | RowExcTustyelock
virtualxid   |            |             | 139628707247872 | ExclusiveLock
transactionid|            |             | 139628707247872 | ExclusiveLock
(6 rows)

openGauss=# COMMIT;
COMMIT
```

第二个会话执行结果如下：

```
openGauss=# CREATE INDEX idx_name ON users(u_name);
CREATE INDEX
```

同理，ALTER TABLE 等表级操作也将会被堵塞。但是普通的查询语句不会被堵塞。

12.4.5 复现 Access Exclusive 锁

从实验原理中了解到，更改表结构的语句会添加 Access exclusive 锁。通过 Begin 命令来生成事务，再更改表，以获取锁信息。

1. 生成锁执行结果如下：

```
openGauss=# BEGIN;
BEGIN
openGauss=# ALTER TABLE users ADD(sex varchar(10));
ALTER TABLE
openGauss=#SELECT locktype,database,relation,pid,mode FROM pg_locks WHERE pid= 139628707247872;
 locktypo   |  database |  relation |       pid       |     mode
------------+-----------+-----------+-----------------+----------------------
 relation   |    14553  |    12009  | 139628707247872 | AccessshareLock
 virtualxid |    14553  |           | 139628707247872 | ExclusiveLock
 relation   |    14553  |    24582  | 139628707247872 | AccessExclusiveLock
 relation   |           |           | 139628707247872 | ExclusiveLock
(4 rows)
```

可以看到，在 users 表上添加了 Access Exclusive 锁。由于 AccessExclusive 锁与其它所有锁模式都冲突，所以此时应该无法进行 SELECT、INSERT、CREATE INDEX 等操作。

2. 此时打开另一个会话窗口，进行 SELECT 操作，发现该命令**被堵塞**，无法执行完成执行结果如下；

```
openGauss=# SELECT * FROM users;
```

3. 再在第一个会话中提交事务，发现第二个会话中的 SELECT 操作成功执行。
提交第一个会话执行结果如下：

```
openGauss=# BEGIN;
BEGIN
openGauss=# ALTER TABLE users ADD(sex varchar(10));
ALTER TABLE
openGauss=#SELECT locktype,database,relation,pid,mode FROM pg_locks WHERE pid= 139628707247872;
locktypo   |  database |  relation |       pid       |     mode
-----------+-----------+-----------+-----------------+----------------------
relation   |    14553  |    12009  | 139628707247872 | AccessshareLock
virtualxid |    14553  |           | 139628707247872 | ExclusiveLock
relation   |    14553  |    24582  | 139628707247872 | AccessExclusiveLock
relation   |           |           | 139628707247872 | ExclusiveLock
(4 rows)


openGauss=# COMMIT;
COMMIT
```

另一个会话 SELECT 语句执行成功，结果如下：

```
openGauss=# SELECT * FROM users;
u_id  |   u_passwd |  u_name |    u_idnum    |       u_regtime    | address  | sex
```

```
------+------------+---------+-------------+------------------+----------+--------
1234  | 33283598   |  asdzxc |  1234       |2000-04-20 00:00:00|          |
10    | 37949309   |  17275  |  13802157128 |2022-05-28 00:00:00|          |
11    | 38536763   |  18770  |  10338732097 |2022-05-21 00:00:00|          |
12    | 49277871   |  14428  |  15410211668 |2022-05-24 00:00:00|          |
13    | 69054886   |  16443  |  19563251198 |2022-05-16 00:00:00|          |
14    | 72895860   |  13429  |  17649156391 |2022-05-20 00:00:00|          |
15    | 63811959   |  12835  |  15001087133 |2022-05-24 00:00:00|          |
16    | 63811959   |  18873  |  18701889757 |2022-05-27 00:00:00|          |
17    | 71957045   |  15961  |  11624111109 |2022-05-24 00:00:00|          |
18    | 41386669   |  10463  |  19581409139 |2022-05-22 00:00:00|          |
19    |  65930947  |  10408  |  11436991268 |2022-05-26 00:00:00|          |
20    |  20029237  |  12412  |  10669710946 |2022-05-22 00:00:00|          |
```

12.4.6 添加代码：输出获取与释放锁的信息

在本节中，将在 openGauss 源代码中添加语句分别输出获取锁和释放锁的信息。获取锁和释放锁的代码均位于 src/gausskernel/storage/lmgr/lock.cpp 文件中。

1. 在获取锁的相关函数中添加代码，输出获取的锁信息。
   获取锁时调用的函数为 lock.cpp 中的 LockAcquire 函数，而该函数会继续调用 LockAcquireExtended 函数；LockAcquireExtended 函数会继续调用 LockAcquireExtendedXC 函数。
   在 LockAcquireExtendedXC 函数中添加如下代码。此处，“if (locktag->locktag_field2 == 24582) {”表明只会输出获取 oid 为 24582 这个数据库对象上锁的信息，这里“24582” 实际上是 users 表的 oid。需要注意的是，每次创建表都会获得不同的 oid，因而，在添加代码时，要替换为自己机器上的 users 表的 oid。

【源码】src/gausskernel/storage/lmgr/lock.cpp

```
static LockAcquireResult LockAcquireExtendedXC(const LOCKTAG *locktag, LOCKMODE lockmode,
    bool sessionLock, bool dontWait, bool reportMemoryError, bool only_increment,
    bool allow_con_update, int waitSec)
{
... 省略若干行
#ifdef LOCK_DEBUG
    if (LOCK_DEBUG_ENABLED(locktag))
        ereport(LOG, (errmsg("LockAcquire: lock [%u,%u] %s",
                        locktag->locktag_field1, locktag->locktag_field2,
                            lockMethodTable->lockModeNames[lockmode])));
#endif
/* [ DBLAB ============================================================ */
    if (locktag->locktag_field2 == 24582) {
        ereport(INFO, (errmsg("LockAcquire: lock [%u,%u] %s",
                            locktag->locktag_field1, locktag->locktag_field2,
                            lockMethodTable->lockModeNames[lockmode])));
    }
```

```
/* DBLAB ] ========================================================== */

    instr_stmt_report_lock(LOCK_START, lockmode, locktag);
... 省略若干行
}
```

这里所添加的代码会输出 3 个信息：

- locktag->locktag_field1：users 表所在 railway 数据库的 id；
- locktag->locktag_field2：users 表的 oid；
- lockMethodTable->lockModeNames[lockmode]：不同锁类型的信息，如 AccessShareLock、RowExclusiveLock 等。

2. 在释放锁的相关函数中添加代码，输出信息。

释放锁时调用的函数为 lock.cpp 中的 LockRelease 函数和 LockReleaseAll 函数。

在 LockRelease 函数中添加如下代码。也是输出步骤 1 中的 3 个信息。

【源码】src/gausskernel/storage/lmgr/lock.cpp

```
bool LockRelease(const LOCKTAG *locktag, LOCKMODE lockmode, bool sessionLock)
{
... 省略若干行
#ifdef LOCK_DEBUG
    if (LOCK_DEBUG_ENABLED(locktag))
        ereport(LOG,  (errmsg("LockRelease:  lock  [%u,%u]  %s",  locktag->locktag_field1,
locktag->locktag_field2,
                              lockMethodTable->lockModeNames[lockmode])));
#endif

    /* [ DBLAB ========================================================== */
    if (locktag->locktag_field2 == 24582) {
        ereport(INFO, (errmsg("LockRelease: lock [%u,%u] %s",
                              locktag->locktag_field1, locktag->locktag_field2,
                              lockMethodTable->lockModeNames[lockmode])));
    }
    /* DBLAB ] ========================================================== */

    /*
     * Find the LOCALLOCK entry for this lock and lockmode
     */
    errno_t rc = memset_s(&localtag, sizeof(localtag), 0, sizeof(localtag)); /* must clear padding
*/
    securec_check(rc, "", "");
    localtag.lock = *locktag;
    localtag.mode = lockmode;
... 省略若干行
}
```

在 LockReleaseAll 函数中添加如下代码。这里只是输出信息“LockReleaseAll”。

【源码】src/gausskernel/storage/lmgr/lock.cpp

```
void LockReleaseAll(LOCKMETHODID lockmethodid, bool allLocks)
{
... 省略若干行
#ifdef LOCK_DEBUG
    if (*(lockMethodTable->trace_flag))
        ereport(LOG, (errmsg("LockReleaseAll: lockmethod=%d", lockmethodid)));
#endif

    /* [ DBLAB ========================================================== */
    ereport(INFO, (errmsg("LockReleaseAll")));
    /* DBLAB ] ========================================================== */

    /*
     * Get rid of our fast-path VXID lock, if appropriate.    Note that this is
     * the only way that the lock we hold on our own VXID can ever get
     * released: it is always and only released when a toplevel transaction
     * ends.
     */
    if (lockmethodid == DEFAULT_LOCKMETHOD)
        VirtualXactLockTableCleanup();
... 省略若干行
}
```

3. 关闭 openGauss 数据库服务器。
4. 进行 openGauss 编译和安装。
5. 启动 openGauss 数据库服务器。

```
[dblab@eduog openGauss-server-v3.0.0]$ gs_ctl start -D $GAUSSHOME/data -Z single_node -l logfile
[2023-03-26       15:25:54.950][214768][][gs_ctl]:       gs_ctl       started,datadir       is
/home/dblab/opengauss-compile/openGauss-server-v3.0.0/dest/data
[2023-03-26 15:25:54.989][214768][][gs_ctl]: waiting for server to start...
.INFO:  LockReleaseAll
INFO:  LockReleaseAll

[2023-03-26 15:25:56.211][214768][][gs_ctl]:  done
[2023-03-26             15:25:56.211][214768][][gs_ctl]:             server             started
(/home/dblab/opengauss-compile/openGauss-server-v3.0.0/dest/data)
```

可见，输出了两条“INFO: LockReleaseAll”信息。

6. 重新连接 gsql 客户端。

```
[dblab@eduog ~]$ gsql -r postgres
INFO:  LockReleaseAll
INFO:  LockReleaseAll
INFO:  LockReleaseAll
INFO:  LockReleaseAll
INFO:  LockReleaseAll
INFO:  LockReleaseAll
```

```
INFO:  LockReleaseAll
INFO:  LockReleaseAll
gsql ((openGauss 3.0.0 build ) compiled at 2023-02-23 19:56:43 commit 0 last mr  debug)
INFO:  LockReleaseAll
INFO:  LockReleaseAll
INFO:  LockReleaseAll
INFO:  LockReleaseAll
Non-SSL connection (SSL connection is recommended when requiring high-security)
Type "help" for help.


openGauss=#
```

可见，输出了若干条“INFO: LockReleaseAll”信息。

7. 删除 users 表中的全部数据。

```
openGauss=# delete from users;
INFO:  LockAcquire: lock [15557,24582] RowExclusiveLock
LINE 1: delete from users;
                    ^
INFO:  LockAcquire: lock [15557,24582] AccessShareLock
INFO:  LockRelease: lock [15557,24582] AccessShareLock
INFO:  LockAcquire: lock [15557,24582] AccessShareLock
INFO:  LockRelease: lock [15557,24582] AccessShareLock
INFO:  LockAcquire: lock [15557,24582] AccessShareLock
INFO:  LockRelease: lock [15557,24582] AccessShareLock
INFO:  LockAcquire: lock [15557,24582] AccessShareLock
INFO:  LockRelease: lock [15557,24582] AccessShareLock
INFO:  LockAcquire: lock [15557,24582] RowExclusiveLock
INFO:  LockAcquire: lock [15557,24582] AccessShareLock
INFO:  LockRelease: lock [15557,24582] AccessShareLock
INFO:  LockReleaseAll
INFO:  LockReleaseAll
DELETE 100
```

可见，输出了获取和释放 RowExclusiveLock 锁和 AccessShareLock 锁的信息，表明执行删除数据的 DELETE 语句，需要获取这 2 种锁。

8. 再次插入数据到 users 表中。

```
openGauss=# INSERT INTO users VALUES (
openGauss(#     to_char(generate_series(100, 199)),
openGauss(#     to_char(random() * 100000000, '09999999'),
openGauss(#     to_char(10000 + random() * 10000, '09999'),
openGauss(#     to_char(10000000000 + random() * 10000000000, '09999999999'),
openGauss(#     CURRENT_DATE + floor((random() * 15))::int);
INFO:  LockAcquire: lock [15557,90277] RowExclusiveLock
LINE 1: INSERT INTO users VALUES (
                    ^
INFO:  LockAcquire: lock [15557,90277] AccessShareLock
```

```
INFO:  LockRelease: lock [15557,90277] AccessShareLock
INFO:  LockAcquire: lock [15557,90277] RowExclusiveLock
INFO:  LockAcquire: lock [15557,90277] ExclusiveLock
INFO:  LockRelease: lock [15557,90277] ExclusiveLock
INFO:  LockAcquire: lock [15557,90277] ExclusiveLock
INFO:  LockRelease: lock [15557,90277] ExclusiveLock
INFO:  LockAcquire: lock [15557,90277] AccessShareLock
INFO:  LockRelease: lock [15557,90277] AccessShareLock
INFO:  LockReleaseAll
INFO:  LockReleaseAll
INSERT 0 100
```

可见，输出了获取和释放 ExclusiveLock 锁、RowExclusiveLock 锁和 AccessShareLock 锁的信息，表明执行插入数据的 INSERT 语句，需要获取这 3 种锁。

9. 执行 users 表上的查询语句。

```
openGauss=# select * from users limit 10;
INFO:  LockAcquire: lock [15557,90277] AccessShareLock
LINE 1: select * from users limit 10;
                      ^
INFO:  LockAcquire: lock [15557,90277] AccessShareLock
INFO:  LockRelease: lock [15557,90277] AccessShareLock
INFO:  LockAcquire: lock [15557,90277] AccessShareLock
INFO:  LockRelease: lock [15557,90277] AccessShareLock
INFO:  LockAcquire: lock [15557,90277] AccessShareLock
INFO:  LockRelease: lock [15557,90277] AccessShareLock
INFO:  LockAcquire: lock [15557,90277] AccessShareLock
INFO:  LockRelease: lock [15557,90277] AccessShareLock
INFO:  LockAcquire: lock [15557,90277] AccessShareLock
INFO:  LockRelease: lock [15557,90277] AccessShareLock
INFO:  LockAcquire: lock [15557,90277] AccessShareLock
INFO:  LockReleaseAll
INFO:  LockReleaseAll
 u_id | u_passwd  | u_name |   u_idnum    |      u_regtime
------+-----------+--------+--------------+---------------------
 100  |  84744649 |  14187 |  11712828493 | 2023-04-08 00:00:00
 101  |  32110406 |  10451 |  19948854805 | 2023-04-02 00:00:00
 102  |  33926037 |  18168 |  18822833803 | 2023-03-27 00:00:00
 103  |  78584907 |  16159 |  19664696818 | 2023-04-05 00:00:00
 104  |  86179597 |  14966 |  10198638942 | 2023-03-31 00:00:00
 105  |  33821661 |  18035 |  19427528442 | 2023-04-02 00:00:00
 106  |  51520118 |  16392 |  16104313722 | 2023-04-08 00:00:00
 107  |  83419337 |  10959 |  10173469703 | 2023-04-09 00:00:00
 108  |  22461876 |  14669 |  11647198098 | 2023-03-28 00:00:00
 109  |  53118411 |  19305 |  14048828408 | 2023-04-03 00:00:00
(10 rows)
```

可见，输出了获取和释放 AccessShareLock 锁的信息，表明执行查询数据的 SELECT 语句，只需要获取 AccessShareLock 锁。

12.5 实验结果

【请按照要求完成实验操作】

1. 完成实验步骤 9.4.2 节，复现 Share 锁，并验证 Share 锁不会对 SELECT 操作进行堵塞。

2. 完成实验步骤 9.4.3 节，复现 Access Share 锁，并验证 Access Share 锁不会对 SELECT 操作、INSERT 操作进行堵塞。

3. 完成实验步骤 9.4.4 节，复现 Row Exclusive 锁，并验证 Row Exclusive 锁不会对 SELECT 操作、INSERT 操作进行堵塞。

4. 完成实验步骤 9.4.5 节，复现 Access Exclusive 锁。

12.6 讨论与总结

【请将实验中遇到的问题描述、解决办法与思考讨论列在下面，并对本实验进行总结。】