

PostgreSQL状态变迁

1.事出有因:

在webserver操作中发现一个神奇的现象:

在创建实例,从agent获取实例状态时,刚好(定时任务)得到一个**正在启动**的中间状态,过一会才变成了关闭。一个很直观的感觉就是:实例列表的实例状态由**正在启动->正常关闭**

正常的情况应该只是**关闭状态**才对,因为webserver端在创建实例时并没有启动。

2.如何获取状态

```
[uxdb@rsbmaster bin]$ ux_controldata --help
ux_controldata 显示 UXsinoDB 数据库簇控制信息.
```

使用方法:

```
ux_controldata [选项][DATADIR]
```

选项:

```
-Z          Uxsino File System(UXFS)
[-D] DATADIR 数据目录
-V, --version 输出版本信息,然后退出
-?, --help   显示帮助信息,然后退出
```

如果没有指定数据目录(DATADIR),将使用环境变量UXDATA.

报告错误至 <uxdb_support@uxsino.com>.

2.1执行命令获取状态

ux_controldata -D 实例名称 (全路径or相对路径)

2.2内容

```
ux_controldata -D /home/uxdb/mpptest/uxdbinstall/dbsql/bin/11
```

```
ux_control 版本:          1201
Catalog 版本:          201909212
数据库系统标识符:      6946444487253594995
数据库簇状态:          在运行中
ux_control 最后修改:      2021年04月02日 星期五 14时46分43秒
最新检查点位置:        8/8B33E50
优先检查点位置:        7/46939430
最新检查点的 REDO 位置:  1/1
最新检查点的重做日志文件: 000000000000001000000000
最新检查点的 TimeLineID: 0
最新检查点的PrevTimeLineID: 0
最新检查点的full_page_writes: 开启
最新检查点的NextXID:    0:24576
```

```

最新检查点的 NextOID: 1
最新检查点的NextMultiXactId: 0
最新检查点的NextMultiOffsetD: 480
最新检查点的oldestXID: 1
最新检查点的oldestXID所在的数据库: 1
最新检查点的oldestActiveXID: 0
最新检查点的oldestMultixid: 1
最新检查点的oldestMulti所在的数据库: 0
最新检查点的oldestCommitTsXid:0
最新检查点的newestCommitTsXid:0
最新检查点的时间: 2021年04月02日 星期五 14时45分51秒
不带日志的关系: 0/3E8使用虚假的LSN计数器
最小恢复结束位置: 0/0
最小恢复结束位置时间表: 0
开始进行备份的点位置: 0/0
备份的最终位置: 0/0
需要终止备份的记录: 否
wal_level设置: minimal
wal_log_hints设置: 关闭
max_connections设置: 11000
max_worker_processes设置: 128
max_prepared_xacts设置: 0
max_locks_per_xact设置: 11000
track_commit_timestamp设置: 开启
最大数据校准: 0
数据库块大小: 32768
大关系的每段块数: 32768
WAL的块大小: 8192
每一个 WAL 段字节数: 16777216
标识符的最大长度: 64
在索引中可允许使用最大的列数: 32
TOAST区块的最大长度: 8140
大对象区块的大小: 8192
日期/时间 类型存储: 64位整数
正在传递Fllloat4类型的参数: 由值
正在传递Fllloat8类型的参数: 由值
数据页校验和版本: 0
Mock authentication nonce:
ca3a3f42b1f4d2ecfd2ad96dc784df0ce3a43ebd2f297d7034d44c188a94cfea
Full Database encryption: 关闭

```

3.PostgreSQL的各种状态

首先：枚举结构定义各种状态

```
typedef enum DBState
{
    DB_STARTUP = 0,
    DB_SHUTDOWNED,
    DB_SHUTDOWNED_IN_RECOVERY,
    DB_SHUTDOWNING,
    DB_IN_CRASH_RECOVERY,
    DB_IN_ARCHIVE_RECOVERY,
    DB_IN_PRODUCTION
} DBState;
```

PostgreSQL启动以及关闭或运行过程中的状态包括以上七种。

(在pg_controldata获取的内容Database cluster state一栏显示的是DB的状态。)

其中：

DB_STARTUP：表示数据库正在启动状态，实际上没有使用该状态。

DB_SHUTDOWNED：数据库实例正常关闭（非standby）控制文件写入的状态就是这个状态

DB_SHUTDOWNED_IN_RECOVERY：standby实例正常关闭，控制文件写入的状态是这个状态。是由CreateRestartPoint修改该状态。

DB_SHUTDOWNING：非standby实例在关闭时，做checkpoint：CreateCheckPoint，开始做时修改为该状态，做完后修改为DB_SHUTDOWNED状态。

DB_IN_CRASH_RECOVERY：实例异常关闭，重启后，恢复时需要将实例先置为该状态

DB_IN_ARCHIVE_RECOVERY：standby实例重启后置为该状态。

DB_IN_PRODUCTION：非standby实例正常重启后就是这个状态，standby是DB_IN_ARCHIVE_RECOVERY

分析

1、DB_STARTUP

```
initdb->BootstrapXLOG:
    memset(ControlFile, 0, sizeof(ControlFileData));
    ...
    ControlFile->state = DB_SHUTDOWNED;
    ...
    writeControlFile();
```

初始化时，首先将其状态初始化为DB_STARTUP，然后立即置成DB_SHUTDOWNED并将其刷写到磁盘。

2、StartupXLOG

```
StartupXLOG->
    ReadControlFile();
    ...
    readRecoveryCommandFile();->
    |--...
    |   for (item = head; item; item = item->next){
    |       if (strcmp(item->name, "restore_command") == 0){
    |           ...
    |       }...
    |
```

```

|         else if (strcmp(item->name, "standby_mode") == 0){
|             if (!parse_bool(item->value, &StandbyModeRequested))
|                 ...
|         }
|         ...
|-- ArchiveRecoveryRequested = true;
...
if (ArchiveRecoveryRequested &&
    (ControlFile->minRecoveryPoint != InvalidXLogRecPtr ||
     ControlFile->backupEndRequired ||
     ControlFile->backupEndPoint != InvalidXLogRecPtr ||
     ControlFile->state == DB_SHUTDOWNED)){
    InArchiveRecovery = true;
    if (StandbyModeRequested)
        StandbyMode = true;
}
...
record = ReadCheckpointRecord(xlogreader, checkPointLoc, 1, true);
...
if (InRecovery){
    if (InArchiveRecovery)//何时?
        ControlFile->state = DB_IN_ARCHIVE_RECOVERY;
    else
        ControlFile->state = DB_IN_CRASH_RECOVERY;
    ...
    UpdateControlFile();
    replay...
}
...
LWLockAcquire(ControlFileLock, LW_EXCLUSIVE);
ControlFile->state = DB_IN_PRODUCTION;
UpdateControlFile();
LWLockRelease(ControlFileLock);
...

```

只要有recovery.conf文件，ArchiveRecoveryRequested即为TRUE->InArchiveRecovery = true，配置了standby_mode=on,那么StandbyMode=TRUE。这样standby启动后，ControlFile->state为DB_IN_ARCHIVE_RECOVERY状态。

3、checkpoint

```

CheckpointMain->
for (;;) {
    ...
    if (shutdown_requested) {
        ShutdownXLOG(0, 0); ->
        |--if (RecoveryInProgress()) {
            CreateRestartPoint(CHECKPOINT_IS_SHUTDOWN |
CHECKPOINT_IMMEDIATE);
        } else {
            CreateCheckPoint(CHECKPOINT_IS_SHUTDOWN |
CHECKPOINT_IMMEDIATE);
        }
        |--...
        proc_exit(0);
    }
    ...
}

```

```

    if (do_checkpoint){
        do_restartpoint = RecoveryInProgress();
        ...
        if (flags & CHECKPOINT_END_OF_RECOVERY)//flags从哪来?
            do_restartpoint = false;
        ...
        if (!do_restartpoint){
            CreateCheckPoint(flags);
            ckpt_performed = true;
        }
        else
            ckpt_performed = CreateRestartPoint(flags);
    }
}

```

备机上做checkpoint调用CreateRestartPoint, 主机做checkpoint调用CreateCheckPoint

```

CreateCheckPoint(int flags)->
    if (flags & (CHECKPOINT_IS_SHUTDOWN | CHECKPOINT_END_OF_RECOVERY))
        shutdown = true;
    else
        shutdown = false;
    ...
    if (shutdown){
        LWLockAcquire(ControlFileLock, LW_EXCLUSIVE);
        ControlFile->state = DB_SHUTDOWNING;
        ControlFile->time = (pg_time_t) time(NULL);
        UpdateControlFile();
        LWLockRelease(ControlFileLock);
    }
    ...
    LWLockAcquire(ControlFileLock, LW_EXCLUSIVE);
    if (shutdown)
        ControlFile->state = DB_SHUTDOWNED;
    ...
    UpdateControlFile();
    LWLockRelease(ControlFileLock);

```

shutdown时, 先将状态置为DB_SHUTDOWNING, 最后将状态置为DB_SHUTDOWNED

```

CreateRestartPoint(int flags)->
    LWLockAcquire(CheckpointLock, LW_EXCLUSIVE);
    SpinLockAcquire(&XLogCtl->info_lck);
    lastCheckPointRecPtr = XLogCtl->lastCheckPointRecPtr;
    lastCheckPointEndPtr = XLogCtl->lastCheckPointEndPtr;
    lastCheckPoint = XLogCtl->lastCheckPoint;
    SpinLockRelease(&XLogCtl->info_lck);
    if (!RecoveryInProgress()){
        LWLockRelease(CheckpointLock);
        return false;
    }
    ...
    if (XLogRecPtrIsValid(lastCheckPointRecPtr) || lastCheckPoint.redo <=
ControlFile->checkpointCopy.redo){
        UpdateMinRecoveryPoint(InvalidXLogRecPtr, true);
        if (flags & CHECKPOINT_IS_SHUTDOWN){

```

```

        LWLockAcquire(ControlFileLock, LW_EXCLUSIVE);
        ControlFile->state = DB_SHUTDOWNED_IN_RECOVERY;
        ControlFile->time = (pg_time_t) time(NULL);
        UpdateControlFile();
        LWLockRelease(ControlFileLock);
    }
    LWLockRelease(CheckpointLock);
    return false;
}

...
LWLockAcquire(ControlFileLock, LW_EXCLUSIVE);
if (ControlFile->state == DB_IN_ARCHIVE_RECOVERY && ControlFile->
    >checkPointCopy.redo < lastCheckpoint.redo){
    ...
    if (flags & CHECKPOINT_IS_SHUTDOWN)
        ControlFile->state = DB_SHUTDOWNED_IN_RECOVERY;
    UpdateControlFile();
}
LWLockRelease(ControlFileLock);
...

```

备机shutdown, 将状态置为DB_SHUTDOWNED_IN_RECOVERY

4.分布式事务

4.1背景

数据膨胀, 业务扩展, 微服务的普及, 单机版服务应用已经无法满足需求。所以我们会基于数据水平分表, 根据业务垂直分库。单个事务操作就需要在多个数据库节点进行。分布式事务由此而生

4.2解决的问题

事务发生在的多个数据库节点, 如何保证一致性。

4.3实现的方式

分布式事务实现方案从类型上去分刚性事务、柔性事务。

刚性事务: 通常无业务改造, 强一致性, 原生支持回滚/隔离性, 低并发, 适合短事务。(基于ACID)

刚性事务从字面上就可以看出来, 这个方案很刚, 它是基于ACID理论的, 要求强一致性。

柔性事务: 有业务改造, 最终一致性, 实现补偿接口, 实现资源锁定接口, 高并发, 适合长事务。(基于BASE,从CAP发展而来), 柔性事务从字面上就可以看出来, 这个方案很柔, 它是基于BASE理论的, 要求最终一致性。

| | 刚性事务 | 柔性事务 |
|------|------------|------------------------|
| 分类 | XA、2PC、3PC | TCC、Saga、事务消息、最大努力通知事务 |
| 一致性 | 强一致 | 最终一致 |
| 隔离性 | 原生支持 | 实现资源锁定接口 |
| 适合场景 | 短事务，并发较低 | 长事务，高并发 |
| 并发性能 | 严重衰退 | 略微衰退 |
| 业务改造 | 无 | 有 |

刚性事务

我们说的刚性事务，基本上就可以认为是XA，它基于2PC协议，各数据库厂商都实现了XA规范（实际上干的就是两阶段的事）

它是根据 XA 接口做真正的写入操作，但不提交，最后有一个事务管理器去协调通知它们提交，在此期间数据就被锁住了

其它事务或人就不能用了，性能损耗极大，并且刚性事务的时间大部分都耗在数据库上了，也就不太适合互联网

不过，改造起来比较简单：把以前调用普通数据源的地方，改成调用 XA 的数据源就行了

另外，XA的事务管理器会将事务执行状态记录在 local-log，即它是有状态的，若机器崩了那状态也就没了，故其不支持高可用

4.3刚性事务-XA/JTA规范/DTP模型

XA是由X/Open组织提出的分布式事务的规范。XA规范主要定义了(全局)事务管理器(Transaction Manager)和(局部)资源管理器(Resource Manager)之间的接口。主流的关系型 数据库产品都是实现了XA接口的。

作为java平台上事务规范JTA（java Transaction API）也定义了对XA事务的支持，实际上，JTA是基于XA架构上建模的，在JTA 中，事务管理器抽象为javax.transaction.TransactionManager接口，并通过底层事务服务（即JTS）实现。

全局事务（DTP模型）--标准分布式事务



XA需要两阶段提交: prepare 和 commit.

第一阶段为 准备 (prepare) 阶段。即所有的参与者准备执行事务并锁住需要的资源。参与者ready时，向transaction manager报告已准备就绪。

第二阶段为提交阶段 (commit) 。当transaction manager确认所有参与者都ready后，向所有参与者发送commit命令。

首先有一个XA规范：XA是由X/Open组织提出的分布式事务的规范。XA规范主要定义了(全局)事务管理器(Transaction Manager)和(局部)资源管理器(Resource Manager)之间的接口。主流的关系型 数据库产品都是实现了XA接口的。基于XA规范有两种协议，一种是2pc,一种是3pc。2pc两阶段提交，

```
package com.uxdb.webserver.myshirotest.xa;
```

```

import com.uxsino.uxdb.UxConnection;
import com.uxsino.uxdb.core.BaseConnection;
import com.uxsino.uxdb.jdbc.AutoSave;
import com.uxsino.uxdb.xa.UXXAConnection;
import com.uxsino.uxdb.xa.UXXADataSource;

import javax.transaction.xa.XAException;
import javax.transaction.xa.XAResource;
import javax.transaction.xa.Xid;
import java.sql.*;
import java.util.Arrays;

/**
 * @author dufz
 * @version 1.0
 * <p>
 * Copyright 2021 Beijing Uxsino Software Co., Ltd./Branch Of Xi'an
 * All right reserved.
 * @date 2021/4/6 14:08
 */

/**
 * XA是由X/Open组织提出的分布式事务的规范。
 * XA规范主要定义了(全局)事务管理器(Transaction Manager)和(局部)资源管理器(Resource Manager)之间的接口。
 *
 * 作为Java平台上事务规范JTA（Java Transaction API）也定义了对XA事务的支持，实际上，JTA是基于XA架构上建模的，
 * 在JTA 中，事务管理器抽象为javax.transaction.TransactionManager接口，并通过底层事务服务（即JTS）实现
 *
 * max_prepared_transactions uxsino.conf需要配置，不然报错
 * Caused by: com.uxsino.uxdb.util.USQLException: ERROR: prepared transactions are disabled
 * Hint: Set max_prepared_transactions to a nonzero value.
 */
public class XaTest4 {

    public static void main(String[] args) {
        //两个数据库数据源

        //两个数据库RM 资源管理器
        UXXAConnection uxxaConnection1=null;
        Connection connection1=null;
        PreparedStatement preparedStatement1=null;
        Statement statement1=null;
        ResultSet resultSet1=null;
        XAResource rm1=null;
        UXXAConnection uxxaConnection2=null;
        Connection connection2=null;
        PreparedStatement preparedStatement2=null;
        Statement statement2=null;
        ResultSet resultSet2=null;
        XAResource rm2=null;
    }
}

```



```

CustomXid customXid1=null;
CustomXid customXid2=null;

try {
    //这种方式创建连接,可以正常创建分布式事务, rollback不报错
    //1.创建rm资源
    Class.forName("com.uxsino.uxdb.Driver");
    connection1 =
DriverManager.getConnection("jdbc:uxdb://localhost:5432/uxdb","Lenovo", "1");
    connection1.setAutoCommit(false);
    connection2 =
DriverManager.getConnection("jdbc:uxdb://localhost:1234/uxdb","Lenovo", "1");
    connection2.setAutoCommit(false);
    uxxaConnection1=new UXXAConnection((BaseConnection) connection1);
    uxxaConnection2=new UXXAConnection((BaseConnection) connection2);

    //2.app 请求TM执行一个分布式事务, TM生成全局Xid
    //全局事务id是1, 各自有个子id
    customXid1=new CustomXid("gl1", "bre1");
    customXid2=new CustomXid("gl1", "bre2");

    //3.rm 注册到TM, 获取全局事务id和子事务id
    rm1=uxxaConnection1.getXAResource();
    rm1.start(customXid1, XAResource.TMNOFLAGS);
    preparedStatement1=connection1.prepareStatement("insert into test1
(id,name) values (1,\'conn1\')");
    int i=preparedStatement1.executeUpdate();
    System.out.println("添加"+i+"条");
    preparedStatement1=connection1.prepareStatement("select * from
test1");
    resultSet1=preparedStatement1.executeQuery();
    while(resultSet1.next()){
        System.out.println("test id="+resultSet1.getInt("id")+----,test
name"+resultSet1.getString("name"));
    }
    rm1.end(customXid1, XAResource.TMSUCCESS);

    //phase1 第一阶段: 预提交
    int rm1prestatus=rm1.prepare(customXid1);//资源管理器1预提交--成功
    // select * from ux_prepared_xacts; 会存储一条准备提交的事务

    rm2=uxxaConnection2.getXAResource();
    rm2.start(customXid2, XAResource.TMNOFLAGS);
    preparedStatement2=connection2.prepareStatement("insert into test1
(id,name) values (\'1x\',\'conn2\')");//制造异常---可以看到数据库不添加, 都回滚
    int j=preparedStatement2.executeUpdate();
    System.out.println("添加"+j+"条");
    preparedStatement2=connection2.prepareStatement("select * from
test1");
    resultSet2=preparedStatement2.executeQuery();
    while(resultSet2.next()){

```

```

        System.out.println("test id="+resultSet2.getInt("id")+"----,test
name"+resultSet2.getString("name"));
    }
    rm2.end(customXid2, XAResource.TMSUCCESS);

    //phase1 第一阶段：预提交

    int rm2prestatus=rm2.prepare(customXid2); //资源管理器2预提交

    //phase1 第二阶段：正式提交
    if(rm1prestatus==XAResource.XA_OK&&rm2prestatus==XAResource.XA_OK)
    { //所有分支预提交都成功
        //TM判断有两个事务，所以第一阶段并不真实提交
        boolean pnePhase = false;
        rm1.commit(customXid1, false);
        rm2.commit(customXid2, false);
    } else { //如果有一个未成功，则回滚
        rm1.rollback(customXid1);
        rm2.rollback(customXid2);
    }

    } catch (SQLException | XAException | ArithmeticException |
ClassNotFoundException e) {
        System.out.println("捕获异常");
        try {
            assert rm1 != null;
            rm1.rollback(customXid1); //select * from ux_prepared_xacts;
            之前预提交的事务删除
            System.out.println("rm1回滚");
        } catch (XAException ex) {
            ex.printStackTrace();
        }
        try {
            assert rm2 != null;
            rm2.rollback(customXid2);
            System.out.println("rm2回滚");
        } catch (XAException ex) {
            ex.printStackTrace();
        }
        // e.printStackTrace();
    } finally {
        /*
        try {
            if(resultSet1!=null) {
                resultSet1.close();
            }
        } catch (SQLException e) {
            e.printStackTrace();
        }
        try {
            if(preparedStatement1!=null) {
                preparedStatement1.close();
            }
        } catch (SQLException e) {
            e.printStackTrace();
        }
        try {

```

```

        if(connection1!=null) {
            connection1.close();
        }
    } catch (SQLException e) {
        e.printStackTrace();
    }
    try {
        if(resultSet2!=null) {
            resultSet2.close();
        }
    } catch (SQLException e) {
        e.printStackTrace();
    }
    try {
        if(preparedStatement2!=null) {
            preparedStatement2.close();
        }
    } catch (SQLException e) {
        e.printStackTrace();
    }
    try {
        if(connection2!=null) {
            connection2.close();
        }
    } catch (SQLException e) {
        e.printStackTrace();
    }
}*/
}
}

}

class CustomXid implements Xid {

    byte[] gtrid = new byte[Xid.MAXGTRIDSIZE];
    byte[] bqqual = new byte[Xid.MAXBQUALSIZE];

    CustomXid(String globalTransactionId,String branchExec) {
        gtrid=globalTransactionId.getBytes();
        bqqual=branchExec.getBytes();
    }

    @Override
    public int getFormatId() {
        return 0;
    }

    @Override
    public byte[] getGlobalTransactionId() {
        return gtrid;
    }

    @Override
    public byte[] getBranchQualifier() {
        return bqqual;
    }
}

```

```

@Override
public boolean equals(Object o) {
    if (!(o instanceof Xid)) {
        return false;
    }

    Xid other = (Xid) o;
    if (other.getFormatId() != this.getFormatId()) {
        return false;
    }
    if (!Arrays.equals(other.getBranchQualifier(),
this.getBranchQualifier())) {
        return false;
    }
    if (!Arrays.equals(other.getGlobalTransactionId(),
this.getGlobalTransactionId())) {
        return false;
    }

    return true;
}

@Override
public int hashCode() {
    final int prime = 31;
    int result = 1;
    result = prime * result + Arrays.hashCode(getBranchQualifier());
    result = prime * result + getFormatId();
    result = prime * result + Arrays.hashCode(getGlobalTransactionId());
    return result;
}
}

```

4.4 刚性事务-基于XA/JTA规范的2pc 3pc

2pc

3pc

4.5 柔性事务-TCC

4.6 柔性事务-Saga

4.7 柔性事务-Seata