

PaxosStore 源码分析「四、协议日志」

2020.03.08 SF-Zhou

本系列的上一篇里分析了 PaxosStore 中共识协议的实现，本篇将聚焦 Paxos 协议中 PaxosLog 的实现。博文标题为了对齐一下，强行翻译为协议日志，不必太在意 :D

1. 术语解释

PaxosStore 中引入了一些术语，这里笔者会按照自己的理解解释一下。首先是 **Entry**，英文翻译为条目，可以理解为一次 Paxos 过程。Entry 包含一个严格单调递增的编号 iEntry 和一个 Paxos 状态机 poMachine，日常存储于内存中，代码定义于 src/EntryInfoMng.h：

```
struct EntryInfo_t {
    EntityInfo_t *ptEntityInfo;
    uint64_t iEntry; // 编号

    uint32_t iEntrySize;
    uint32_t iInteractCnt;

    bool bCatchUpFlag; // 是否正在 CatchUp，后续博文有 CatchUp 的介绍
    bool bUncertain; // 是否处于 Uncertain 的状态，该 Entry 在内存上的更新未落盘时则处于 U
    bool bRemoteUpdated; // 是否需要更新远端节点上的 Entry
    bool bBroadcast; // 是否需要广播给所有节点
```

```

bool bNotFound;          // 本地对应的 PLog 是否已删除

clsPaxosCmd **apWaitingMsg;

clsEntryStateMachine *poMachine; // Paxos 状态机

// For clsArrayTimer<EntryInfo_t>.
clsArrayTimer<EntryInfo_t>::TimeoutEntry_t tTimeoutEntry;

// For tEntryList in EntityInfo
CIRCLEQ_ENTRY(EntryInfo_t) tListElt;
};

```

然后是 **Entity**，英文翻译为实体，可以理解为执行多次 Paxos 过程的对象，例如用户的银行账户。Entity 包含一个唯一 ID iEntityID、一组 Entry 和一些自身状态，也存储于内存中，代码定义于 src/EntityInfoMng.h：

```

struct EntityInfo_t {
    uint64_t iEntityID; // 唯一 ID

    volatile uint64_t iMaxPLogEntry;          // 最大的 PLog 进度
    volatile uint64_t iMaxChosenEntry;        // 已知的全局最大 Chosen 进度
    volatile uint64_t iMaxContChosenEntry;    // 获取到的最大连续 Chosen 进度，连续表示没有空洞

    uint64_t iLocalPreAuthEntry; // 预授权的 Entry 编号，参见本系列第六篇预授权优化部分
    uint64_t iCatchUpEntry;      // 当前尝试 CatchUp 的最大 Entry
    uint64_t iValueIDGenerator;  // ValueID 生成器，实现上是时间戳 + 自增编号
    uint64_t iNotifiedEntry;     // 已经通知 DBWorker Commit 的最大 Entry
};

```

```
uint64_t iGetAllFinishTimeMS; // 上次 GetAll 完成的时间, 避免短时间重复 GetAll

clsClientCmd *poClientCmd;
clsPaxosCmd **apWaitingMsg;
clsLeasePolicy *poLeasePolicy;

CIRCLEQ_HEAD(EntryList_t, EntryInfo_t) tEntryList; // 当前存储的一组 Entry

uint32_t iLocalAcceptorID;
uint32_t iActiveAcceptorID;
uint32_t iWaitingSize;

volatile int32_t iRefCount;

bool bRangeLoading;
bool bRangeLoaded;

bool bGetAllPending;
};
```

内存中的 Entry 持久化时会存储为 **EntryRecord**, 后者在上一篇中提到过。将所有 **EntryRecord** 按顺序存储起来, 就可以按需恢复出 **Entity** 的状态。PaxosStore 中将这些 **EntryRecord** 称为 **PLog**, 存储 **PLog** 的数据库称为 **PLogDB**。

2. 日志读写

PaxosStore 中使用 PLogWorker 来处理 **PLog** 的读写, 流程基本是请求方将请求塞入队列、PLogWorker 从队列中获取请求、执行读写操作、最后将回复塞入队列。代码见

src/PLogWorker.cpp :

```
// 将 Req 塞到队列里
int clsPLogWorker::EnterPLogReqQueue(clsCmdBase *poCmd) {
    uint64_t iEntityID = poCmd->GetEntityID();
    clsAsyncQueueMng *poQueueMng = clsAsyncQueueMng::GetInstance();
    clsConfigure *poConf = clsCertainWrapper::GetInstance()->GetConf();

    clsPLogReqQueue *poQueue =
        poQueueMng->GetPLogReqQueue(Hash(iEntityID) % poConf->GetPLogWorkerNum());

    poCmd->SetTimestampUS(GetCurrTimeUS());

    int iRet = poQueue->PushByMultiThread(poCmd);
    if (iRet != 0) {
        OSS::ReportPLogQueueErr();
        return -1;
    }

    return 0;
}

// 启动协程处理 PLogReqQueue 中地请求
void clsPLogWorker::Run() {
    // Bind cpu affinity here.
    uint32_t iLocalServerID = m_poConf->GetLocalServerID();
    SetThreadTitle("plog_%u_%u", iLocalServerID, m_iWorkerID);
    CertainLogInfo("plog_%u_%u run", iLocalServerID, m_iWorkerID);

    co_enable_hook_sys();
}
```

```

stCoEpoll_t *ev = co_get_epoll_ct();
s_epoll_stat = (EpollRunStat_t *)calloc(sizeof(EpollRunStat_t), 1);
// co_set_eventloop_stat( OnEpollStart, OnEpollEnd );
for (int i = 0; i < int(m_poConf->GetPLogRoutineCnt()); ++i) {
    PLogRoutine_t *w = (PLogRoutine_t *)calloc(1, sizeof(PLogRoutine_t));
    stCoRoutine_t *co = NULL;
    co_create(&co, NULL, PLogRoutine, w);

    int iRoutineID = m_iStartRoutineID + i;
    w->pCo = (void *)co;
    w->pSelf = this;
    w->pData = NULL;
    w->bHasJob = false;
    w->iRoutineID = iRoutineID;
    co_resume((stCoRoutine_t *) (w->pCo));
}

printf("PLogWorker idx %d %u Routine\n", m_iWorkerID, m_poConf->GetPLogRoutineCnt());
CertainLogImpt("PLogWorker idx %d %u Routine", m_iWorkerID, m_poConf->GetPLogRoutineCnt());

co_eventloop(ev, CoEpollTick, this);
}

// 协程 Tick, 给空闲的协程分配任务
int clsPLogWorker::CoEpollTick(void *arg) {
    clsPLogWorker *pPLogWorker = (clsPLogWorker *)arg;
    stack<PLogRoutine_t *> &IdleCoList = *(pPLogWorker->m_poCoWorkList);

    if (pPLogWorker->CheckIfExiting(0)) {
        return -1;
    }
}

```

```

}

TIMERUS_START(iCoEpollTickTimeUS);
uint64_t iGetFromIdleCoListCnt = 0;

while (!IdleCoList.empty()) {
    clsCmdBase *poCmd = NULL;
    int iRet = pPLogWorker->m_poPLogReqQueue->TakeByOneThread(&poCmd);
    if (iRet == 0 && poCmd) {
        uint64_t iUseTimeUS = GetCurrTimeUS() - poCmd->GetTimestampUS();
        s_poPLogReqQueueWait->Update(iUseTimeUS);

        PLogRoutine_t *w = IdleCoList.top();
        w->pData = (void *)poCmd;
        w->bHasJob = true;
        IdleCoList.pop();
        co_resume((stCoRoutine_t *) (w->pCo));
        iGetFromIdleCoListCnt++;
    } else {
        break;
    }
}
s_poGetFromIdleCoListCnt->Update(iGetFromIdleCoListCnt);

TIMERUS_STOP(iCoEpollTickTimeUS);
s_poCoEpollTick->Update(iCoEpollTickTimeUS);

clsCertainUserBase *pCertainUser = clsCertainWrapper::GetInstance()->GetCertainUser();
pCertainUser->TickHandleCallBack();

```

```

    return 0;
}

// 协程函数: 处理 PLog Req
void *clsPLogWorker::PLogRoutine(void *arg) {
    PLogRoutine_t *pPLogRoutine = (PLogRoutine_t *)arg;
    co_enable_hook_sys();

    clsCertainUserBase *pCertainUser = clsCertainWrapper::GetInstance()->GetCertainUser();
    pCertainUser->SetRoutineID(pPLogRoutine->iRoutineID);

    while (1) {
        clsPLogWorker *pPLogWorker = (clsPLogWorker *)pPLogRoutine->pSelf;

        if (!pPLogRoutine->bHasJob) {
            AssertEqual(pPLogRoutine->pData, NULL);
            pPLogWorker->m_poCoWorkList->push(pPLogRoutine);
            co_yield_ct();
            continue;
        }

        AssertNotEqual(pPLogRoutine->pData, NULL);
        clsCmdBase *poCmd = (clsCmdBase *)pPLogRoutine->pData;

        if (poCmd->GetCmdID() == kPaxosCmd) {
            // PaxosCmd 执行 DoWithPaxosCmd
            pPLogWorker->DoWithPaxosCmd(dynamic_cast<clsPaxosCmd *>(poCmd));
        } else {
            AssertEqual(poCmd->GetCmdID(), kRecoverCmd);
            // RecoverCmd 执行 DoWithRecoverCmd

```

```

        pPLogWorker->DoWithRecoverCmd(dynamic_cast<clsRecoverCmd *>(poCmd));
    }

    pPLogRoutine->bHasJob = false;
    pPLogRoutine->pData = NULL;
}

return NULL;
}

// 处理 Paxos Cmd
// 一种是读取 PLogDB, 而后通过 IO 回包
// 一种是存储 PaxosCmd 中的 MaxPLogEntry 和 Record 到 PLogDB, 而后进入回复阶段
// 注意有 CheckHasMore 的逻辑, 会检查当前 Entry 之后还有没有更新的记录, 后续博文会提高这部分逻辑
int clsPLogWorker::DoWithPLogRequest(clsPaxosCmd *poPaxosCmd) {
    int iRet;
    uint64_t iEntityID = poPaxosCmd->GetEntityID();
    uint64_t iEntry = poPaxosCmd->GetEntry();

    if (poPaxosCmd->IsPLogReturn()) {
        // 如果设定了 PLogReturn (MaxContChosenEntry >= iEntry)
        EntryRecord_t tSrcRecord;
        // 从 PLog Engine 中读取 Record
        iRet = m_poPLogEngine->GetRecord(iEntityID, iEntry, tSrcRecord);
        if (iRet == 0) {
            CertainLogInfo("record: %s bChose %d", EntryRecordToString(tSrcRecord).c_str(),
                           tSrcRecord.bChosen);
            if (tSrcRecord.bChosen) {
                // 如果被 Chosen 了, 那么直接将 Record 存到 Cmd 里
                poPaxosCmd->SetSrcRecord(tSrcRecord);
            }
        }
    }
}

```



```

    } else {
        // 如果没有被 Chosen, 直接将返回值设定为未找到 (未 Chosen, 皆为变数)
        poPaxosCmd->SetResult(eRetCodeNotFound);
    }
} else if (iRet == eRetCodeNotFound) {
    poPaxosCmd->SetResult(eRetCodeNotFound);
} else {
    CertainLogFatal("BUG cmd: %s ret %d", poPaxosCmd->GetTextCmd().c_str(), iRet);
    return -1;
}

// 通过 IO 发送回复
m_poIOWorkerRouter->GoAndDeleteIfFailed(poPaxosCmd);
} else if (poPaxosCmd->IsCheckHasMore() || m_poConf->GetUsePLogWriteWorker() == 0) {
    EntryRecord_t tRecord = poPaxosCmd->GetSrcRecord();

    uint64_t iMaxPLogEntry = poPaxosCmd->GetMaxPLogEntry();
    if (m_poConf->GetEnableMaxPLogEntry() == 0) {
        iMaxPLogEntry = INVALID_ENTRY;
    }

    // 将 Cmd 中的 MaxPLogEntry 和 EntryRecord 写入 PLogDB
    iRet = m_poPLogEngine->PutRecord(iEntityID, iEntry, iMaxPLogEntry, tRecord);
    if (iRet != 0) {
        CertainLogFatal("E(%lu, %lu) PutRecord ret %d", iEntityID, iEntry, iRet);
        poPaxosCmd->SetPLogError(true);
    }

    if (poPaxosCmd->IsCheckHasMore()) {
        // 如果设定了要 CheckHasMore

```

```

    bool bHasMore = false;
    vector<pair<uint64_t, string> > vecRecord;
    TIMERUS_START(iRangeLoadUseTimeUS);
    // 从 PLogDB 中读取未 Commit 的 Record 列表信息
    iRet = m_poPLogEngine->LoadUncommittedEntrys(iEntityID, iEntry, iEntry, vecRecord, b
    TIMERUS_STOP(iRangeLoadUseTimeUS);
    s_poLoadUncommittedEntrysTimeStat->Update(iRangeLoadUseTimeUS);
    OSS::ReportPLogRangeLoadTimeMS(iRet, iRangeLoadUseTimeUS / 1000);
    if (iRangeLoadUseTimeUS > 100000) {
        CertainLogError("E(%lu, %lu) more %u iRangeLoadUseTimeUS %lu", iEntityID, iEntry,
                        iRangeLoadUseTimeUS);
    }

    if (iRet != 0) {
        CertainLogFatal("E(%lu, %lu) LoadUncommittedEntrys ret %d", iEntityID, iEntry, iRe
        poPaxosCmd->SetPLogError(true);
    } else {
        // PLogDB 中还有更多的 Record
        poPaxosCmd->SetHasMore(bHasMore);
    }
}

// 进入回复队列
clsPLogWorker::EnterPLogRspQueue(poPaxosCmd);
} else {
    SendToWriteWorker(poPaxosCmd);
}

return 0;
}

```

```

// 将回复塞到队列里
int clsPLogWorker::EnterPLogRspQueue(clsCmdBase *poCmd) {
    uint64_t iEntityID = poCmd->GetEntityID();
    clsAsyncQueueMng *poQueueMng = clsAsyncQueueMng::GetInstance();
    clsConfigure *poConf = clsCertainWrapper::GetInstance()->GetConf();

    clsPLogRspQueue *poQueue =
        poQueueMng->GetPLogRspQueue(Hash(iEntityID) % poConf->GetEntityWorkerNum());

    while (1) {
        int iRet = poQueue->PushByMultiThread(poCmd);
        if (iRet == 0) {
            break;
        }

        CertainLogError("PushByMultiThread ret %d cmd: %s", iRet, poCmd->GetTextCmd().c_str())

        poll(NULL, 0, 1);
    }

    uint64_t iUseTimeUS = GetCurrTimeUS() - poCmd->GetTimestampUS();
    s_poPLogCmdOuterTimeStat->Update(iUseTimeUS);

    return 0;
}

```

代码中的 m_poPLogEngine 实际上是一个 clsPLogBase 对象，用以实现真正的 **PLogDB** 持久化。PaxosStore 中定义了其基类，剩余了部分虚函数需要用户实现：

```

struct PLogEntityMeta_t {
    uint64_t iMaxPLogEntry;
};

class clsPLogBase {
public:
    static void PrintUseTimeStat();
    static void InitUseTimeStat();

    int GetRecord(uint64_t iEntityID, uint64_t iEntry, EntryRecord_t &tSrcRecord);
    int PutRecord(uint64_t iEntityID, uint64_t iEntry, uint64_t iMaxPLogEntry, EntryRecord_t &tSrcRecord);

public:
    virtual ~clsPLogBase() {}

    virtual int Put(uint64_t iEntityID, uint64_t iEntry, const string &strRecord) = 0;

    virtual int Get(uint64_t iEntityID, uint64_t iEntry, string &strRecord) = 0;

    virtual int PutValue(uint64_t iEntityID, uint64_t iEntry, uint64_t iValueID,
                        const string &strValue) = 0;

    virtual int GetValue(uint64_t iEntityID, uint64_t iEntry, uint64_t iValueID,
                        string &strValue) = 0;

    virtual int PutWithPLogEntityMeta(uint64_t iEntityID, uint64_t iEntry,
                                    const PLogEntityMeta_t &tMeta, const string &strRecord) = 0;

    virtual int GetPLogEntityMeta(uint64_t iEntityID, PLogEntityMeta_t &tMeta) = 0;

```

```

virtual int LoadUncommittedEntrys(uint64_t iEntityID, uint64_t iMaxCommittedEntry,
                                   uint64_t iMaxLoadingEntry,
                                   vector<pair<uint64_t, string> > &vecRecord, bool &bHas
};

// 从 PLogDB 中恢复 EntryRecord 信息, 包括 Value
int clsPLogBase::GetRecord(uint64_t iEntityID, uint64_t iEntry, EntryRecord_t &tSrcRecord
int iRet;
string strTemp;

TIMERUS_START(iGetUseTimeUS);
// 在 PLogDB 中读取 Entry 序列化的 Record 数据
iRet = Get(iEntityID, iEntry, strTemp);
TIMERUS_STOP(iGetUseTimeUS);
s_poGetTimeStat->Update(iGetUseTimeUS);
OSS::ReportPLogGetTimeMS(0, iGetUseTimeUS / 1000);

if (iGetUseTimeUS > 100000) {
    CertainLogError("E(%lu, %lu) iGetUseTimeUS %lu", iEntityID, iEntry, iGetUseTimeUS);
}

if (iRet != 0) {
    if (iRet == eRetCodeNotFound) {
        InitEntryRecord(&tSrcRecord);
        return eRetCodeNotFound;
    }

    CertainLogFatal("BUG probably E(%lu, %lu) Get ret %d", iEntityID, iEntry, iRet);

    return -1;
}

```

```

}

// 反序列化
iRet = StringToEntryRecord(strTemp, tSrcRecord);
if (iRet != 0) {
    CertainLogFatal("E(%lu, %lu) StringToEntryRecord ret %d", iEntityID, iEntry, iRet);
    return -2;
}

iRet = CheckEntryRecordMayWithVIDOnly(tSrcRecord);
if (iRet != 0) {
    CertainLogFatal("E(%lu, %lu) CheckEntryRecordMayWithVIDOnly ret %d", iEntityID, iEntry, iRet);
    return -3;
}

uint64_t iValueID = tSrcRecord.tValue.iValueID;
if (iValueID > 0) {
    if (!tSrcRecord.tValue.bHasValue) {
        TIMERUS_START(iGetValueUseTimeUS);
        // 从 PLogDB 中通过 ValueID 读取 Value
        iRet = GetValue(iEntityID, iEntry, iValueID, strTemp);
        TIMERUS_STOP(iGetValueUseTimeUS);
        s_poGetTimeStat->Update(iGetValueUseTimeUS);
        OSS::ReportPLogGetValueTimeMS(iRet, iGetValueUseTimeUS / 1000);

        if (iGetValueUseTimeUS > 100000) {
            CertainLogError("E(%lu, %lu) iGetValueUseTimeUS %lu", iEntityID, iEntry,
                            iGetValueUseTimeUS);
        }
    }
}

```

```

    if (iRet == eRetCodeNotFound) {
        return eRetCodeNotFound;
    }

    if (iRet != 0) {
        CertainLogFatal("E(%lu, %lu) GetValue ret %d", iEntityID, iEntry, iRet);
        return -4;
    }

    tSrcRecord.tValue.strValue = strTemp;
    tSrcRecord.tValue.bHasValue = true;
}

iRet = CheckEntryRecord(tSrcRecord);
if (iRet != 0) {
    CertainLogFatal("E(%lu, %lu) CheckEntryRecord ret %d", iEntityID, iEntry, iRet);
    return -5;
}

return 0;
}

// 将 EntryRecord 写入 PLogDB
// 如果已经 Chosen, 则可以丢掉长 Value
// 若当前已知的 MaxPLogEntry 还未初始化, 或 Entry <= MaxPLogEntry
// 直接存储 <(Entity, Entry), Record>
// 否则从 PLogDB 中读取存储的 MaxPLogEntry, 如果 Entry <= MaxPLogEntry 仍然直接存
// 否则还需要更新并存储 MaxPLogEntry
int clsPLogBase::PutRecord(uint64_t iEntityID, uint64_t iEntry, uint64_t iMaxPLogEntry,

```

```

        EntryRecord_t tRecord) {

    int iRet;
    string strRecord;

    CertainLogDebug("E(%lu, %lu) iMaxPLogEntry %lu record: %s", iEntityID, iEntry, iMaxPLog
        EntryRecordToString(tRecord).c_str());

    iRet = CheckEntryRecord(tRecord);
    if (iRet != 0) {
        CertainLogFatal("E(%lu, %lu) CheckEntryRecord ret %d", iEntityID, iEntry, iRet);
        return -1;
    }

    clsConfigure *poConf = clsCertainWrapper::GetInstance()->GetConf();
    if (!tRecord.bChosen && tRecord.tValue.strValue.size() > poConf->GetMaxEmbedValueSize())
        // Record 还没有被 Chosen, 并且 Value 的长度超过直接嵌入的限制
        if (tRecord.iStoredValueID != tRecord.tValue.iValueID && tRecord.tValue.iValueID > 0)
            // 当前已经存储的 ValueID 与当前 ValueID 不一致, 说明要更新
            TIMERUS_START(iPutValueUseTimeUS);
            // 单独写 Value
            iRet = PutValue(iEntityID, iEntry, tRecord.tValue.iValueID, tRecord.tValue.strValue
                CertainLogDebug("E(%lu, %lu) iValueID %lu size %lu ret %d", iEntityID, iEntry,
                    tRecord.tValue.iValueID, tRecord.tValue.strValue.size(), iRet);
                TIMERUS_STOP(iPutValueUseTimeUS);
                s_poPutTimeStat->Update(iPutValueUseTimeUS);
                OSS::ReportPLogPutTimeMS(iRet, iPutValueUseTimeUS / 1000);

            if (iPutValueUseTimeUS > 100000) {
                CertainLogError("E(%lu, %lu) iPutValueUseTimeUS %lu", iEntityID, iEntry,
                    iPutValueUseTimeUS);
            }
        }
    }
}

```



```

    }

    if (iRet != 0) {
        CertainLogFatal("E(%lu, %lu) PutValue ret %d", iEntityID, iEntry, iRet);
        return -2;
    }
    // 更新 StoredValueID
    tRecord.iStoredValueID = tRecord.tValue.iValueID;
}

// 如果没有超过限制, 直接序列化存储即可
if (tRecord.tValue.strValue.size() <= poConf->GetMaxEmbedValueSize()) {
    tRecord.iStoredValueID = 0;
}

// 序列化
iRet = EntryRecordToString(tRecord, strRecord);
if (iRet != 0) {
    CertainLogFatal("E(%lu, %lu) EntryRecordToString ret %d", iEntityID, iEntry, iRet);
    return -3;
}

if (iMaxPLogEntry == INVALID_ENTRY || iEntry <= iMaxPLogEntry) {
    // 当 MaxPLogEntry 未初始化或 iEntry <= iMaxPLogEntry 时, 直接写入 Record
    TIMERUS_START(iPutUseTimeUS);
    iRet = Put(iEntityID, iEntry, strRecord);
    TIMERUS_STOP(iPutUseTimeUS);
    s_poPutTimeStat->Update(iPutUseTimeUS);
    OSS::ReportPLogPutTimeMS(iRet, iPutUseTimeUS / 1000);
}

```

```
} else {  
    // 否则先给 PLog 上锁  
    clsAutoPLogEntityLock oPLogEntityLock(iEntityID);  
  
    PLogEntityMeta_t tMeta = {0}; // 注意这里初始化为 0  
    TIMERUS_START(iGetPLogMetaUseTimeUS);  
    // 获取 PLog 中存储的 Entity Meta: MaxPLogEntry  
    iRet = GetPLogEntityMeta(iEntityID, tMeta);  
    TIMERUS_STOP(iGetPLogMetaUseTimeUS);  
    OSS::ReportPLogGetMetaKeyTimeMS(iRet, iGetPLogMetaUseTimeUS / 1000);  
  
    if (iRet != 0 && iRet != Certain::eRetCodeNotFound) {  
        CertainLogFatal("E(%lu, %lu) GetPLogEntityMeta ret %d", iEntityID, iEntry, iRet);  
        return -5;  
    }  
  
    CertainLogInfo("E(%lu, %lu) iMaxPLogEntry %lu tMeta.iMaxPLogEntry %lu", iEntityID, iEntry,  
        iMaxPLogEntry, tMeta.iMaxPLogEntry);  
  
    TIMERUS_START(iPutUseTimeUS);  
    if (tMeta.iMaxPLogEntry < iEntry) {  
        // 更新 PLogMeta 同时 Put Record  
        tMeta.iMaxPLogEntry = iEntry;  
        iRet = PutWithPLogEntityMeta(iEntityID, iEntry, tMeta, strRecord);  
    } else {  
        // iEntry >= tMeta.iMaxPLogEntry, store directly  
        // to unlock  
        iRet = Put(iEntityID, iEntry, strRecord);  
    }  
    TIMERUS_STOP(iPutUseTimeUS);
```



```

virtual int Put(uint64_t iEntityID, uint64_t iEntry, const std::string &strRecord);

virtual int Get(uint64_t iEntityID, uint64_t iEntry, std::string &strRecord);

virtual int PutWithPLogEntityMeta(uint64_t iEntityID, uint64_t iEntry,
                                   const Certain::PLogEntityMeta_t &tMeta,
                                   const std::string &strRecord);

virtual int GetPLogEntityMeta(uint64_t iEntityID, Certain::PLogEntityMeta_t &tMeta);

virtual int LoadUncommittedEntrys(uint64_t iEntityID, uint64_t iMaxCommittedEntry,
                                    uint64_t iMaxLoadingEntry,
                                    std::vector<std::pair<uint64_t, std::string> > &vecRec
                                    bool &bHasMore);
};

int clsPLogImpl::PutValue(uint64_t iEntityID, uint64_t iEntry, uint64_t iValueID,
                          const std::string &strValue) {
    clsAutoDisableHook oAuto;
    std::string strKey;
    EncodePLogValueKey(strKey, iEntityID, iEntry, iValueID);

    dbtype::WriteOptions tOpt;
    dbtype::WriteBatch oWB;
    oWB.Put(strKey, strValue);

    dbtype::Status s = m_poLevelDB->Write(tOpt, &oWB);
    if (!s.ok()) {
        return Certain::eRetCodePLogPutErr;
    }
}

```

```

    }
    return 0;
}

int clsPLogImpl::GetValue(uint64_t iEntityID, uint64_t iEntry, uint64_t iValueID,
                          std::string &strValue) {
    clsAutoDisableHook oAuto;
    std::string strKey;
    EncodePLogValueKey(strKey, iEntityID, iEntry, iValueID);

    dbtype::ReadOptions tOpt;
    dbtype::Status s = m_poLevelDB->Get(tOpt, strKey, &strValue);
    if (!s.ok()) {
        if (s.IsNotFound()) {
            return Certain::eRetCodeNotFound;
        }
        return Certain::eRetCodePLogGetErr;
    }
    return 0;
}

int clsPLogImpl::Put(uint64_t iEntityID, uint64_t iEntry, const std::string &strRecord) {
    clsAutoDisableHook oAuto;
    std::string strKey;
    EncodePLogKey(strKey, iEntityID, iEntry);

    dbtype::WriteOptions tOpt;
    dbtype::WriteBatch oWB;
    oWB.Put(strKey, strRecord);

```

```

dbtype::Status s = m_poLevelDB->Write(tOpt, &oWB);
if (!s.ok()) {
    return Certain::eRetCodePLogPutErr;
}
return 0;
}

int clsPLogImpl::Get(uint64_t iEntityID, uint64_t iEntry, std::string &strRecord) {
    clsAutoDisableHook oAuto;
    std::string strKey;
    EncodePLogKey(strKey, iEntityID, iEntry);

    dbtype::ReadOptions tOpt;
    dbtype::Status s = m_poLevelDB->Get(tOpt, strKey, &strRecord);
    if (!s.ok()) {
        if (s.IsNotFound()) {
            return Certain::eRetCodeNotFound;
        }
        return Certain::eRetCodePLogGetErr;
    }
    return 0;
}

int clsPLogImpl::PutWithPLogEntityMeta(uint64_t iEntityID, uint64_t iEntry,
                                        const Certain::PLogEntityMeta_t &tMeta,
                                        const std::string &strRecord) {

    clsAutoDisableHook oAuto;
    std::string strKey;
    EncodePLogKey(strKey, iEntityID, iEntry);

```

```

std::string strMetaKey;
EncodePLogMetaKey(strMetaKey, iEntityID);

CertainPB::PLogEntityMeta tPLogEntityMeta;
tPLogEntityMeta.set_max_plog_entry(tMeta.iMaxPLogEntry);
std::string strMetaValue;
assert(tPLogEntityMeta.SerializeToString(&strMetaValue));

dbtype::WriteOptions tOpt;
dbtype::WriteBatch oWB;
oWB.Put(strKey, strRecord);
oWB.Put(strMetaKey, strMetaValue);

dbtype::Status s = m_poLevelDB->Write(tOpt, &oWB);
if (!s.ok()) {
    return Certain::eRetCodePLogPutErr;
}
return 0;
}

int clsPLogImpl::GetPLogEntityMeta(uint64_t iEntityID, Certain::PLogEntityMeta_t &tMeta)
{
    clsAutoDisableHook oAuto;
    std::string strKey;
    EncodePLogMetaKey(strKey, iEntityID);

    std::string strMetaValue;
    dbtype::ReadOptions tOpt;
    dbtype::Status s = m_poLevelDB->Get(tOpt, strKey, &strMetaValue);
    if (!s.ok()) {
        if (s.IsNotFound()) {

```

```

        return Certain::eRetCodeNotFound;
    }
    return Certain::eRetCodePLogGetErr;
}

CertainPB::PLogEntityMeta tPLogEntityMeta;
if (!tPLogEntityMeta.ParseFromString(strMetaValue)) {
    return Certain::eRetCodeParseProtoErr;
}

tMeta.iMaxPLogEntry = tPLogEntityMeta.max_plog_entry();

return 0;
}

int clsPLogImpl::LoadUncommittedEntrys(uint64_t iEntityID, uint64_t iMaxCommittedEntry,
                                         uint64_t iMaxLoadingEntry,
                                         std::vector<std::pair<uint64_t, std::string> > &vec
                                         bool &bHasMore) {

    clsAutoDisableHook oAuto;
    bHasMore = false;
    vecRecord.clear();

    std::string strStartKey;
    EncodePLogKey(strStartKey, iEntityID, iMaxCommittedEntry + 1);

    dbtype::ReadOptions tOpt;
    std::unique_ptr<dbtype::Iterator> iter(m_poLevelDB->NewIterator(tOpt));

    for (iter->Seek(strStartKey); iter->Valid(); iter->Next()) {

```



```
const dbtype::Slice &strKey = iter->key();
uint64_t iCurrEntityID = 0;
uint64_t iEntry = 0;
uint64_t iValueID = 0;

if (!DecodePLogKey(strKey, iCurrEntityID, iEntry) &&
    !DecodePLogValueKey(strKey, iCurrEntityID, iEntry, iValueID)) {
    break;
}

if (iCurrEntityID > iEntityID) {
    break;
}

if (iValueID != 0) {
    continue;
}

if (iMaxLoadingEntry < iEntry) {
    bHasMore = true;
    break;
}

std::string strValue = iter->value().ToString();
vecRecord.push_back(std::make_pair(iEntry, strValue));
}

return 0;
}
```

实现很简单。**PLogDB** 中存储了键值对 $\langle (\text{EntityID}, \text{EntryID}), \text{EntryRecord} \rangle$, 以及 Entity 的 PLog Meta 信息 `MaxPLogEntry` 。

3. 恢复流程

PLogDB 中存储的信息可以用来恢复本机的实际 DB 数据, 也可以用来恢复集群其他机器的数据。以后者为例, 当本机发现其他机器的 Entity 信息较旧时, 会触发 PLog 数据的读取:

```
// 处理请求的 PaxosCmd
int clsEntityWorker::DoWithPaxosCmd(clsPaxosCmd *poPaxosCmd) {
    ...
    // 对方的 EntryID 较小
    if (ptEntityInfo->iMaxContChosenEntry >= iEntry) {
        // 且还没有 Chosen 时, 那么可以获悉对方的 Entity 信息陈旧
        if (poPaxosCmd->GetSrcRecord().bChosen) {
            return 0;
        }

        // 构造一个 PaxosCmd
        clsPaxosCmd *po =
            new clsPaxosCmd(iLocalAcceptorID, iEntityID, iEntry, NULL, &poPaxosCmd->GetSrcRec

        // 目标是对方机器
        po->SetDestAcceptorID(iAcceptorID);
        po->SetPLogReturn(true);
        po->SetMaxChosenEntry(uint64_t(ptEntityInfo->iMaxChosenEntry));
        po->SetUUID(poPaxosCmd->GetUUID());
    }
}
```

```
// 塞入请求队列
iRet = clsPLogWorker::EnterPLogReqQueue(po);
if (iRet != 0) {
    delete po, po = NULL;
    CertainLogError("EnterPLogReqQueue ret %d", iRet);
    return -1;
}

return 0;
}
```

PLogWorker 线程中会依次执行 CoEpollTick -> PLogRoutine -> DoWithPaxosCmd -> DoWithPLogRequest -> PLogDB::GetRecord -> IO::GoAndDeletelfFailed , 将会从 PLogDB 中读取对应的 EntryRecord 并将其发送给对方机器。

4. 总结

Paxos 过程中, 节点需要持久化自己做出的承诺, 否则将有可能在重启后违背自己的承诺。同时存储自身状态也有利于自身和其他节点做数据恢复。下一篇将继续分析 PaxosStore 协议过程中的一些细节。

0 comments

Write

Preview

Aa

Except where otherwise noted, content on this site is licensed under a CC BY-SA 4.0 license.
Copyright©2017 SF-Zhou, All Rights Reserved. Powered by GitHub Pages and GitHub Actions.