重构数据库实现

jask

2024-08-26

基本数据结构

BufferPoolManager

重新设计了 BufferPoolManager 与 Replacer 之间的关系，Replacer 作为一个模块只被 BufferPoolManager 使用，因此没有必要单独提取出来实现，如果后续需要用 Clock 算法来做页面置换，应当到时候再重构。

新加入的 Raft 算法

raft_server

std::atomic catching_up_ 作用：
1. 标识追赶状态：当 catching_up_ 被设置为 true 时，表示该副本的日志已经落后于 Leader，并且正在通过从 Leader 获取日志条目的方式进行追赶。在此期间，副本不处理正常的 append_entries 请求，因为它需要优先同步落后的日志。

2. 暂停正常日志追加：在追赶状态下，该副本不会接收正常的 append_entries 请求。这意味着 Leader 不会向该副本发送新的日志条目，直到该副本成功追赶到当前集群的最新日志状态。这是为了避免该副本在未完全同步之前处理新的日志条目，导致日志的不一致性。

逻辑：
1. 在 Raft 协议中，副本可能由于网络延迟或短暂的宕机而导致日志落后于 Leader。当副本重新加入集群时，它需要从 Leader 获取并应用所有缺失的日志条目，以便与集群保持一致。

2. 通过设置 catching_up_ 变量，系统可以将该副本标记为正在追赶的状态，确保在追赶过程中不处理新的日志条目，从而保证日志的完整性和一致性。

3. 当副本完成日志追赶后，catching_up_ 将被设置为 false，副本才会恢复正常的日志追加操作。

std::atomic out_of_log_range_ 作用：
1. 标识日志是否落后：当 out_of_log_range_ 被设置为 true 时，意味着该副本的日志已经落后于 Leader 发来的日志范围。这可能是由于网络分区或其他原因导致的长时间未收到来自 Leader 的心跳或日志追加消息。

2. 阻止重新发起选举：一旦 out_of_log_range_ 被设置为 true，该副本将不再发起新的 Leader 选举。这是为了防止一个日志已经落后于集群的副本尝试发起选举，从而影响集群的一致性。

逻辑：
1.Raft 协议要求集群中的大多数副本都要有最新的日志条目。如果一个副本的日志落后于集群的其他成员，并且尝试发起选举，它可能会导致新的 Leader 没有包含集群中最新的日志条目，这会违反 Raft 的一致性保证。

2. 通过设置 out_of_log_range_ 变量并阻止该副本发起选举，能够确保只有拥有最新日志条目的节点才有资格成为 Leader，从而保证集群的一致性。

bool config_changing_ 作用：
1. 标识未提交的配置变更：当 config_changing_ 被设置为 true 时，表示当前有一个尚未提交的配置变更。这意味着集群正在处理一项配置更改，但该变更尚未在日志中完全提交并生效。

2. 拒绝新的配置变更：如果 config_changing_ 为 true，则在这种状态下，集群将拒绝任何新的配置变更请求。这是为了避免多个配置变更同时进行，导致配置的状态不一致或难以追踪。

3. 保护机制：变量声明注释中提到该变量受 lock_ 保护。这意味着在读取或修改 config_changing_ 时，必须持有相关的锁，以确保对配置状态的修改是线程安全的。

逻辑：
在 Raft 协议中，配置变更（例如添加或删除服务器）是通过在日志中记录配置条目来实现的。为了保证一致性，配置变更需要像普通的日志条目一样经过提交过程才能生效。

如果在未完成的配置变更中发起新的配置变更请求，可能会导致集群中的节点对当前配置状态产生不同的理解。因此，使用 config_changing_ 变量来确保只有在当前配置变更提交之后，才允许处理新的配置变更请求。

在配置变更被成功提交并应用到集群之后，config_changing_ 会被设置为 false，表明集群可以继续处理其他配置变更。

void yield_leadership(bool immediate_yield=false,int successor_id=-1) 功能：
领导权的放弃：该函数会让当前领导者服务器自动放弃其领导者角色，成为一个 follower。这可以是手动触发的操作，通常用于系统维护或负载均衡等场景。

立即放弃领导权 (immediate_yield)：如果 immediate_yield 参数被设置为 true，则当前服务器会立即放弃领导权。在这种情况下，后续的领导者选举将完全是随机的，所有的节点都有可能成为新的领导者，包括当前放弃领导权的服务器本身。

延迟放弃领导权：如果 immediate_yield 为 false，那么当前服务器会先暂停写操作，等待继任者（通常为其他服务器）将最新的日志追赶同步完毕后再辞去领导者的角色。这种情况下，下一个领导者会相对更可预测，避免选举过程中出现不必要的竞争。

指定继任者 (successor_id)：用户可以通过 successor_id 参数来指定哪个服务器应当成为新的领导者。如果不指定（默认为 -1），则系统会自动选择优先级最高的服务器作为继任者。

应用场景：
负载均衡：当某个领导者节点的负载过高或由于其他原因需要释放压力时，可以通过这个函数让其主动放弃领导权，以便其他节点接管。

系统维护：在某些维护场景中，可能需要让领导者节点暂时停止服务，此时可以通过这个函数安全地移交领导权，并避免服务中断。

有序的领导权交接：通过延迟放弃领导权以及指定继任者，可以实现平滑的领导权过渡，避免集群内的不必要竞争和不确定性。

timer_helper leadership_transfer_timer_ 用于管理领导权转移过程中的定时器。当服务器成为领导者时，这个定时器将会被重置。它的作用主要是跟踪领导者的状态变化，确保在领导权转移过程中，某些时间相关的操作能按预期进行。

作用：

定时器管理，防止长时间的领导权转移，触发超时处理。

std::string get_aux(int32 srv_id) const 用于从服务器配置中获取与特定服务器（通过其 srv_id 标识）相关联的辅助上下文（auxiliary context）。这个辅助上下文通常是一个与服务器配置相关的附加信息，它以字符串的形式存储。

功能：

辅助上下文的获取：该函数的主要作用是根据传入的 srv_id 参数，检索并返回与该服务器相关的辅助上下文信息。这些信息可能包括服务器的附加配置信息、标识符、状态信息等。

只读函数：由于函数签名中包含 const 修饰符，表示该函数不会修改类的成员变量。这意味着该函数是一个只读操作，只负责从已有配置中检索信息，而不修改配置信息本身。

灵活性：辅助上下文的设计使得每个服务器都可以存储特定的附加信息，而这些信息可能不会直接包含在服务器的核心配置中。通过 get_aux 函数，可以方便地获取这些附加信息，为服务器的管理和操作提供更多的灵活性。

可能用途：

服务器管理：在分布式系统中，可能会为每个服务器配置一些特定的附加信息，这些信息可以通过 get_aux 函数获取，用于系统监控、调试或者配置管理。

动态配置：当某些配置项需要动态调整或扩展，而又不适合放入核心配置中时，可以利用辅助上下文的机制，将这些信息存储并通过 get_aux 函数访问。

状态记录：辅助上下文还可能用于存储特定服务器的状态信息，比如自定义的标识符、运行时的元数据等，便于在运行期间进行查询和处理。

static int64_t get_stat_gauge(const std::string &name); 于根据给定的统计名称 name 获取对应的计量值（gauge value）

性能监控：系统在运行过程中，可以通过 get_stat_gauge 获取当前某个性能指标的值，并用于分析系统的运行状态。

资源管理：在资源管理中，监控某个资源的使用量来决定是否进行扩展或收缩操作。

void peer::send_req(ptr self, ptr &req, rpc_handler &handler) 在分布式系统中，通过远程过程调用 (RPC) 发送请求消息 (req_msg) 给其他节点 (peer)，并处理请求结果。

myself： 这是一个指向当前 peer 对象的智能指针 (ptr)，通常在异步调用或回调中使用，防止对象在操作完成之前被销毁。

req： 这是一个指向请求消息 (req_msg) 的引用，表示要发送的消息内容。

handler： 这是一个 RPC 处理器 (handler)，用于处理 RPC 结果的回调函数。

bool peer::recreate_rpc(ptr& config, context& ctx) 重新创建 RPC 客户端以与服务器端进行通信，确保 peer 可以通过 RPC 与其他节点进行交互。

重连逻辑：通过指数回退机制控制重连的频率，以避免频繁的重连操作。

```
bool peer::recreate_rpc(ptr<srv_config> &config, context &ctx) {
  //检查 peer 是否被废弃
  if (abandoned_) {
    p_tr("peer %d is abandoned", config->get_id());
    return false;
    //false 表明已经废弃不能重建连接
  }
  //获取 rpc 工厂
  ptr<rpc_client_factory> factory = nullptr;
  {
    //从 context 对象中获取 RPC 客户端工厂 (rpc_client_factory)。为了线程安全，使用了 std::lock_guard 锁定上下文的锁 (ctx_lo
    //如果工厂为空，说明无法创建 RPC 客户端，直接返回 false。
    std::lock_guard lock{ctx.ctx_lock_};
    factory = ctx.rpc_cli_factory_;
  }
  if (!factory) {
    p_tr("client factory is empty");
    return false;
  }
  std::lock_guard l_{rpc_protector_};
  //检查是否禁用了重连退避机制
  bool backoff_timer_disabled =
      debugging_options::get_instance().disable_reconn_backoff_.load(std::memory_order::relaxed);
  if (backoff_timer_disabled) {
    p_tr("reconnection back-off timer is disabled");
  }
  //重连逻辑与指数退避
  //如果退避机制被禁用或者退避计时器超时，则执行重连逻辑。
//使用指数退避机制控制重连频率，每次重连尝试后将重连等待时间增加一倍，直到达到心跳间隔 (hb_interval_)。
//如果新计算的等待时间为 0，则将其设置为 1 毫秒。
  if (backoff_timer_disabled || reconn_backoff_.timeout()) {
    reconn_backoff_.reset();
    size_t new_duration_ms = reconn_backoff_.get_duration_us() / 1000;
    new_duration_ms = std::min(heartbeat_interval_, (int32)new_duration_ms * 2);
    if (!new_duration_ms)
      new_duration_ms = 1;
    reconn_backoff_.set_duration_ms(new_duration_ms);
    //创建新的 RPC 客户端
    //使用工厂对象创建新的 RPC 客户端 (rpc_)。
    //记录日志信息，显示新创建的 RPC 客户端指针和对应的 peer ID。
    rpc_ = factory->create_client(config->get_endpoint());
    p_tr("%p reconnect peer %zu", rpc_.get(), config_->get_id());

    // WARNING:
    //   A reconnection attempt should be treated as an activity,
    //   hence reset timer.
    reset_active_timer();

    set_free();
    set_manual_free();
    return true;
  } else {
    p_tr("skip reconnect this time");
  }
  return false;
}
```

核心逻辑：该函数负责重新创建 RPC 客户端连接，并通过指数退避机制控制重连频率，以避免频繁的重连尝试。

线程安全：函数内部使用了多个互斥锁来确保线程安全，包括上下文锁（ctx_lock_）和 RPC 锁（rpc_protector_）。

活动状态管理：函数会根据重连操作重置活动计时器，并设置资源释放标志来管理资源的生命周期。

ptr raft_server::handle_out_of_log_msg(req_msg& req,ptr msg,ptr resp) 用于处理 Raft 协议中日志超出范围的情况，更新节点的状态并重启选举计时器，以避免错误的选举请求。通过日志和回调机制通知系统并采取适当措施。

ptr raft_server::handle_leadership_takeover(req_msg& req,ptr msg,ptr resp) 用于处理领导权接管的请求。如果当前节点还不是领导者，它将发起一次强制选举，并重启选举计时器，以便尽快选出新的领导者。

void raft_server::commit(ulong target_idx)

更新 quick_commit_index_：
    如果 target_idx 大于当前的 quick_commit_index_，则函数会将 quick_commit_index_ 更新为新的目标索引。
    同时，将 lagging_sm_target_index_ 设置为这个值，这可能表示用于延迟状态机更新的目标索引。
    使用调试日志（p_db）记录已触发提交到该索引的操作。

领导者逻辑：
    如果当前节点是领导者（Leader），则需要通知其余节点提交日志条目，直到 quick_commit_index_。
    函数遍历集群中的所有节点（peers_），并尝试向每个节点发送追加日志请求（request_append_entries）。
        如果请求发送成功，日志提交会立即执行。
        如果节点繁忙（请求未成功），则设置该节点的挂起提交标志（set_pending_commit），以便稍后处理。

本地提交逻辑：
    通过日志记录（p_tr），函数记录当前日志索引、目标提交索引以及状态机提交索引等详细信息，用于调试。

提交请求：
    如果本地日志索引大于状态机提交索引，并且 quick_commit_index_ 也大于状态机提交索引，表示有新的日志需要提交给
    检查是否存在全局线程池（nuraft_global_mgr）。如果存在，请求全局线程池处理提交操作。
        否则，使用本地线程通过条件变量（commit_cv_）唤醒提交线程进行处理。

数据更新逻辑：
    如果当前节点是跟随者（Follower），并且在服务器重新启动后，跟随者的日志条目与领导者一样新，那么提交线程可能没
    如果 leader_commit_index_ 和本地的 sm_commit_index_ 索引差距足够小（根据参数判断），则将 data_fresh_ 标志设

总结：

这个函数处理了 Raft 中日志提交的关键操作，确保日志条目正确提交并应用于状态机，特别是在领导者节点时需要通知集群中的其他节点进行同步。

db_state_machine 这是处理 raft 中，处理提交的日志条目，并根据这些日志条目更新集群的状态。利用了继承 state_machine 来处理，简化了实现细节。

函数介绍

```
ptr<buffer> pre_commit(const ulong log_idx, buffer& data);
```

作用：

1 在日志条目被正式提交之前，pre_commit 函数会被调用。

2. 它可以用于验证或处理即将提交的日志数据，在这里只是简单地将日志数据提取并打印出来。

对于当前项目的实现：

使用 buffer_serializer 提取 data 中的字符串，然后打印日志条目索引和数据内容。

该函数返回 nullptr，表示不进行进一步处理。

```
ptr<buffer> commit(const ulong log_idx, buffer& data);
```

作用：

1. 当日志条目被正式提交到状态机时，Raft 会调用 commit 函数。

2. 在实际的应用中，这个函数通常会执行与日志内容相关的状态更新操作。

实际中，提取日志数据并打印日志索引和数据内容。更新最后提交的日志索引 last_committed_idx_。返回 nullptr 表示没有额外的处理。

```
void commit_config(const ulong log_idx, ptr<cluster_config>& new_conf);
```

作用：

1.Raft 集群配置变更时调用此函数。

2. 一般用于处理集群成员变更等操作。

这里没有对配置变更做任何处理，仅仅更新了最后提交的日志索引 last_committed_idx_。

```
void rollback(const ulong log_idx, buffer& data);
```

当需要回滚未提交的日志条目时（例如由于某些失败导致的回退），会调用 rollback 函数。

```
int read_logical_snp_obj(snapshot& s, void*& user_snp_ctx, ulong obj_id, ptr<buffer>& data_out, bool& is_last_ob
```

作用：

用于从快照中读取数据，通常在构建新的快照时调用。

实现：

返回一个包含整数 0 的缓冲区，并指示这是最后一个对象 (is_last_obj = true)。
这里是一个简单的占位实现，实际系统中需要根据快照内容返回相关的数据。

```
void save_logical_snp_obj(snapshot& s, ulong& obj_id, buffer& data, bool is_first_obj, bool is_last_obj);
```

作用：

当系统从快照恢复状态时，Raft 会调用该函数保存快照对象。

实现：

打印出快照的日志索引、任期和对象 ID。
更新对象 ID，以请求下一个对象。

```
bool apply_snapshot(snapshot& s);
```

作用：

当 Raft 从快照中恢复状态时，会调用此函数。
这个函数的目的是应用快照的内容，恢复系统状态。

实现：

打印快照的最后日志索引和任期。
使用快照中的数据更新 last_snapshot_ 成员变量，确保系统中的状态与快照内容保持一致。

```
void free_user_snp_ctx(void*& user_snp_ctx);
```

作用：

释放快照操作中的上下文数据，通常用于清理快照过程中分配的资源。

实现：

这里是空实现，因为当前示例没有使用任何自定义的上下文数据。

```
ptr<snapshot> last_snapshot();
```

作用：
    返回最新的快照，通常用于 Raft 系统在需要持久化或者同步快照时调用。
实现：
    通过互斥锁保护快照的读取操作，返回当前保存的最新快照 last_snapshot_。

```
void create_snapshot(snapshot& s, async_result<bool>::handler_type& when_done);
```

作用：
    当需要生成快照时，Raft 系统会调用此函数，保存当前状态的快照。
实现：
    将快照数据序列化并保存到 last_snapshot_ 成员变量中。
    调用 when_done 回调函数，通知 Raft 系统快照生成成功。

void append_log(const std::string& cmd, const std::vector& tokens) 用于处理追加日志的情况。

tokens 从第二个开始被连接为一个字符串，序列化到缓冲区中，然后利用 boost::asio::steady_timer 进行计时，然后调用 raft_instance 中的 append_entries 追加，最后需要判断 ret_get_accepted() 是否正确。

处理日志追加请求的不同模式：

阻塞模式（nuraft::CALL_TYPE == raft_params::blocking）：
    如果 Raft 操作被配置为阻塞模式（等待操作完成），则直接调用 handle_result 函数处理日志追加的结果。这里将计时器
异步模式（nuraft::CALL_TYPE == raft_params::async_handler）：
    如果 Raft 操作是异步模式，则在操作完成时通过回调函数处理结果。ret->when_ready 接受一个绑定的 handle_result 函
其他情况：
    如果 CALL_TYPE 不属于上述两种情况，直接触发断言失败，因为这是不符合预期的。