·RIP-59 Support DLedger Controller Snapshot

Status

- Current State: Draft
- Authors: tsunghanjacktsai, hzh0425, TheR1sing3un
- Shepherds: RongtongJin
- Mailing List discussion: dev@rocketmq.apache.org
- Pull Request:
- Released: no

Background & Motivation

What do we need to do

- Will we add a new module?

无

- Will we add new APIs?

不对任何客户端层面 API 进行新增或修改。
Controller状态机（DLedgerControllerStateMachine）会有新API接口。

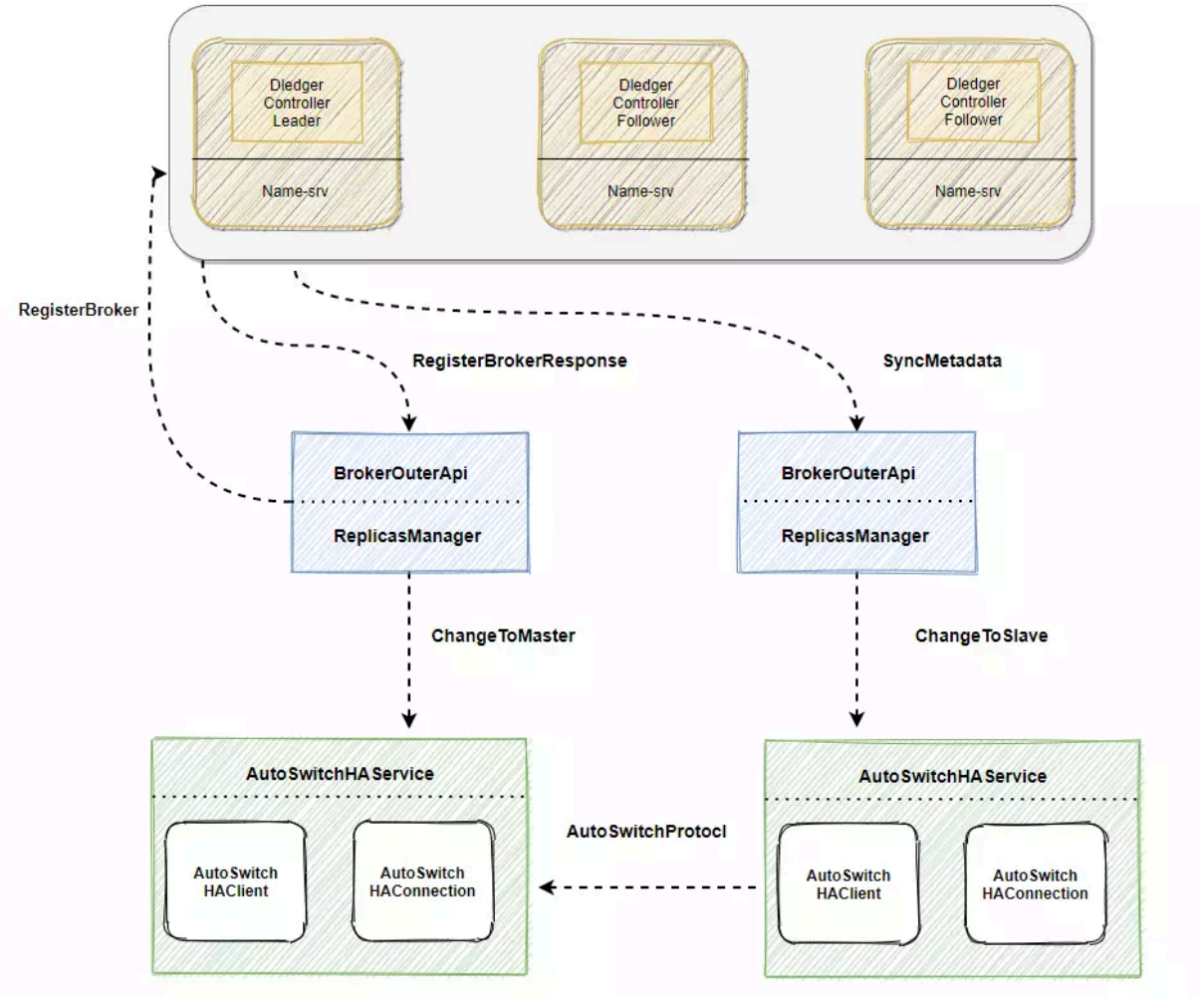
- Will we add new feature?

是

Why should we do that

- Are there any problems of our current project?

RocketMQ 5.0 版本后推出了 DLedger Controller 模式，其能够提供自动主从切换的功能。其原理如下：



DLedger Controller 是依赖于 DLedger (Raft) 构建的强一致元数据中心，其中保存着用于选主相关的元数据。

然而，因为 DLedger 先前没有提供 Snapshot 的能力，将会面临到以下几个问题：

1. 当 Controller 重启时，需要回放 DLedger 中所有的日志，从而恢复 Shutdown 之前的状态。重启所有的日志会导致重启会耗费大量的时间，特别是当日志非常多的时候。
2. 节点所需存储的日志将会随着时间不断扩张，很可能会造成存储空间不足。

- What can we benefit proposed changes?

通过状态机快照，DLedger Controller 节点重启后能够在较短的时间内恢复状态机进度，同时也能够通过将已被快照覆盖到的日志进行删除，有效节省磁盘空间。

Goals

- What problem is this proposal designed to solve?

解决DLedger Controller在节点重启后，状态机需重放本地日志来恢复进度和日志随着时间无限扩张会造成存储空间不足等问题。

Non-Goals

- What problem is this proposal NOT designed to solve?

无

Changes

整个RIP 变更会分为3个部分:

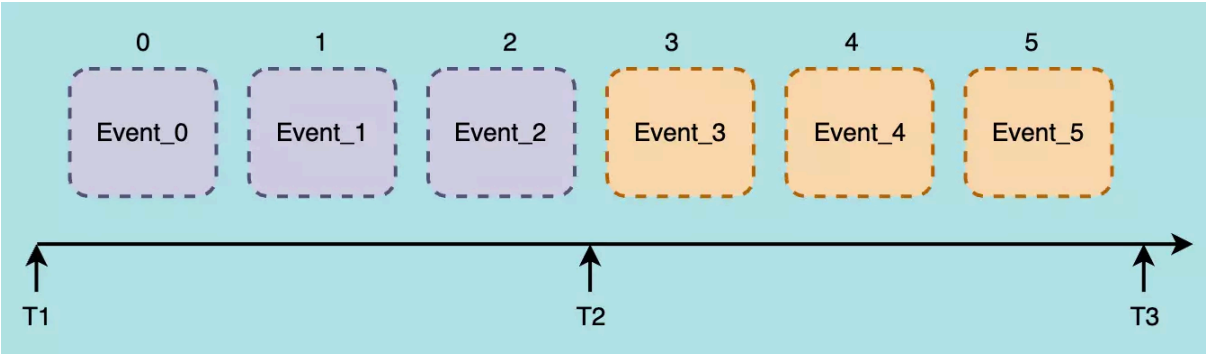
1. DLedger Snapshot实现
2. RocketMQ中 DLedger Controller Snapshot 实现
3. 针对DLedger状态机和Snapshot的Jepsen测试，验证线性一致性

DLedger Snapshot 实现

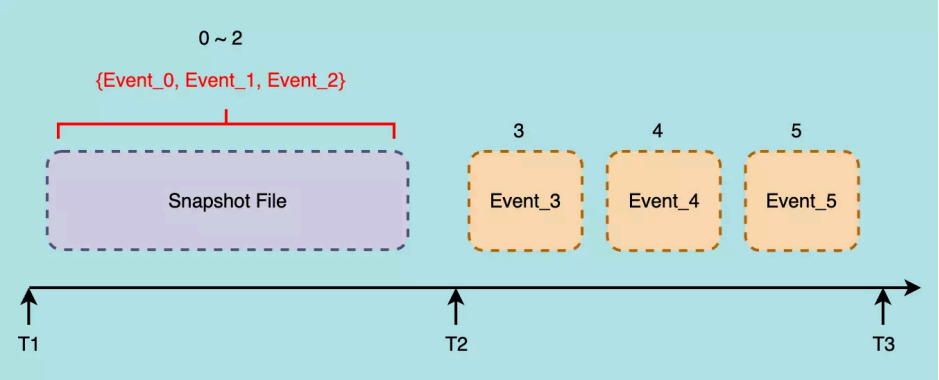
DLedger 将会引入快照（Snapshot）机制以结合 Raft 的数据同步场景。

解决节点重启后恢复状态机进度的时间成本问题

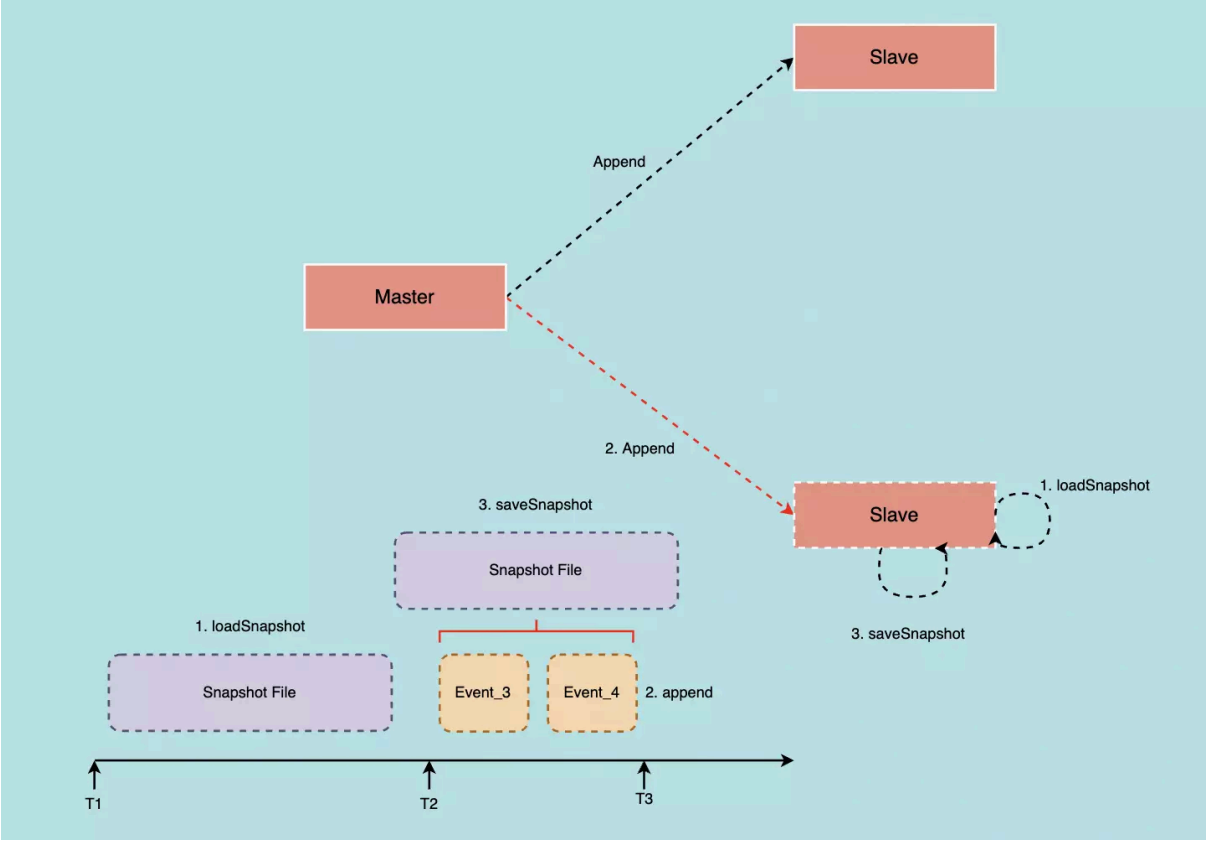
在未引入 Snapshot 机制前，节点重启后，状态机需重放所有日志以恢复进度。



引入 Snapshot 机制后，节点只需要加载最新的快照至状态机，然后以 Snapshot 数据的日志为起点进行日志重放，如下图所示，可以看到快照文件涵盖了三个指令的最终结果，重放的日志文件越少，启动速度也会越快，这将会提高重启恢复效率。



整体流程框架



由上图可知，在节点后进行重启后，会先开启快照加载流程，将先前的快照安装到状态机（如果有的话），然后才继续进行日志复制流程，接收后续的日志。

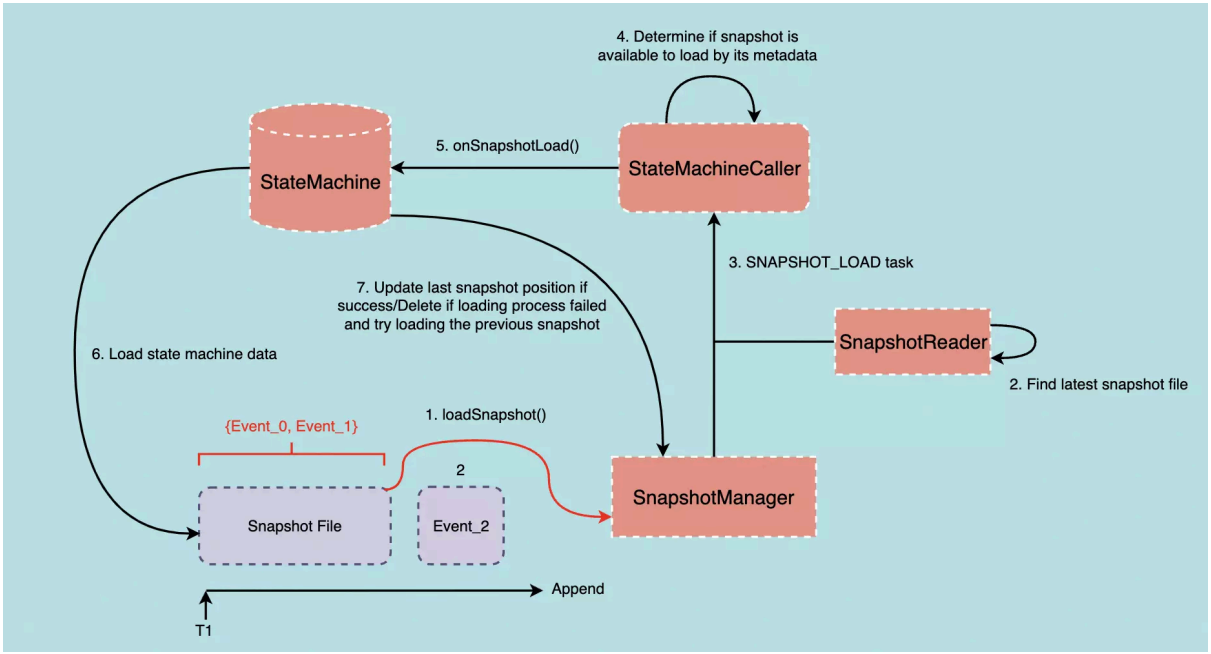
当日志增长到一定程度后，则会启动快照存储流程，合并日志数据集合并生成快照文件，同时会清理之前生成的快照文件并对之前的日志进行裁剪（根据保留快照数量maxReservedSnapshotNum参数设定）。

重要组件

以下为负责快照存储或加载的组件：

- **SnapshotManager**：负责管理快照的存储以及加载。
- **StateMachineCaller**：负责操控状态机执行快照的存储及加载。
- **StateMachine**：状态机。
- **SnapshotWriter**：状态机元数据（包含执行快照时，已被应用到状态机的最高日志索引 lastAppliedIndex & 领导人任期 lastAppliedTerm）存储器，负责将状态机元数据写入文件中。
- **SnapshotReader**：状态机元数据加载器，负责将加载文件中的状态机元数据。

快照加载流程



加载流程将会在状态机启动后当即执行，以将快照中的数据应用到状态机，快速恢复其进度。

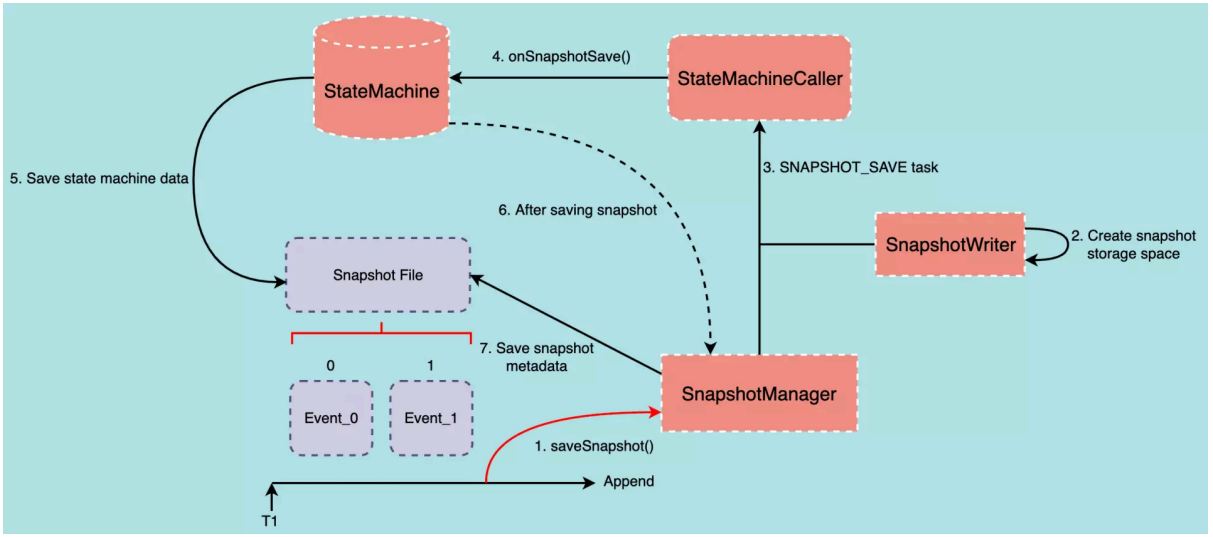
1. 执行入口为 SnapshotManager，SnapshotManager 会负责初始化状态机元数据加载器 SnapshotReader。
2. SnapshotReader 初始化过程中会取得最新的快照文件目录。
3. 开始进行快照加载，SnapshotManager 会注册一快照加载任务至 StateMachineCaller 中。
4. StateMachineCaller 执行到快照加载任务后，会加载状态机元数据并根据原数据信息对被加载快照的有效性进行判断，判断逻辑为：
    a. 比较 Leader 任期 term

i. 如果当前被应用到状态机的日志 Leader 任期大于快照所记录的任期，则代表该快照已失效，不继续加载流程。
ii. 如果当前被应用到状态机的日志 Leader 任期等于快照所记录的任期，则继续比较当前被应用到状态机的最高日志索引 index。
iii. 如果当前被应用到状态机的日志 Leader 任期小于快照所记录的任期，则代表该快照领先于状态机进度，可直接继续加载流程。

b. 比较日志索引 index
i. 如果当前被应用到状态机的最高日志索引大于快照所记录的日志索引，则代表该快照已失效，不继续加载流程。
ii. 如果当前被应用到状态机的最高日志索引小余等于快照所记录的日志索引，则代表该快照可被应用，继续加载流程。

5. 确定当前快照有效后，StateMachineCaller 便会操控状态机执行其维护数据的快照文件加载。
6. 状态机 StateMachine 会反JSON序列化快照文件中存储的状态机数据恢复进度。
7. 最后便根据快照加载结果执行钩子函数：
a. 如果加载成功，则根据加载的快照元数据，更新前次执行快照位置。
b. 如果加载到过期快照，则直接结束本次快照加载流程。
c. 如果加载失败，则尝试删除加载失败的快照，并加载之前的快照（快照保留个数预设为3个，可由 DLedgerConfig#maxSnapshotReservedNum 参数决定）。
i. 如果无法删除或没有可加载快照，则抛出异常并关闭 DLedgerServer。
ii. 如果有可加载的快照，则尝试加载当前快照的前一个生成的快照。

快照存储流程

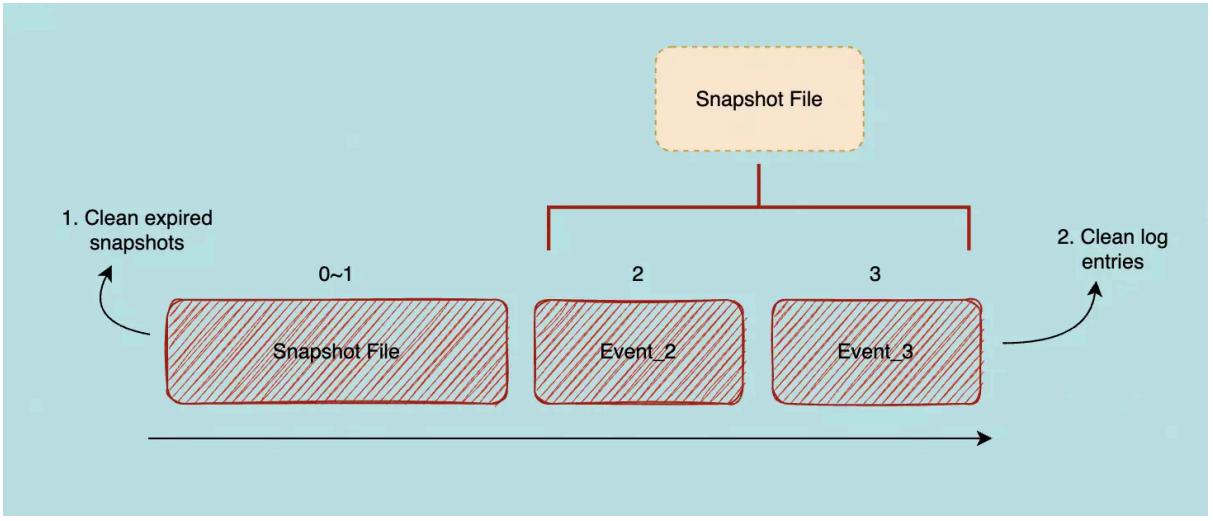


一般快照的触发机制分为两种：1. 每过一段时间定期进行快照 2. 判断日志应用到状态机的进度（根据已被应用到状态机的最高日志索引 lastAppliedIndex 判断）与上次执行快照的位点差

距是否达到阈值。本方案使用第二种方式触发快照，阈值预设为1000（每1000次提交生成一快照），可通过 DLedgerConfig#snapshotThreshold 参数更改。

1. 执行入口为 SnapshotManager，SnapshotManager 会进行一系列的判断，包括是否仍在存储其他快照和确定是否达到执行快照的阈值，确认可执行快照后，便进行状态机元数据存储器 SnapshotWriter 的创建。
2. SnapshotWriter 在创建过程中会清除脏数据（e.g. 前次创建失败的快照暂存）并创建快照的存储空间，快照还未生成完全前会先写入一暂存目录，完全生成后才迁移至正式目录，防止出现后续加载到不完整快照的情形。
3. 接下来 SnapshotManager 会注册一快照存储任务至 StateMachineCaller 中。
4. StateMachineCaller 执行到快照存储任务后，会负责构建状态机元数据（根据当前状态机的 lastAppliedIndex & lastAppliedTerm），并操控状态机执行针对其维护数据的快照存储。
5. 状态机 StateMachine 会将当前状态机所维护的数据（本场景为 ReplicasInfoManager 中的两张表）存储到快照文件中。
6. 状态机成功对其维护数据进行快照后，再执行钩子函数
7. 状态机元数据存储器 SnapshotWriter 将会通过JSON序列化状态机元数据并写入到文件中。

解决日志无限扩张可能造成磁盘空间不足问题



日志无限扩张极有可能造成磁盘空间不足，而引入 Snapshot 机制后，等于为当前状态机的最新状态做了份“镜像”，在这个时间点之前的日志以及快照就可以进行删除，这将可以有效地减少日志文件在磁盘的占用空间。

日志和过期快照的清理主要在快照成功存储后进行，主要逻辑如下：

1. 清理过期快照，根据预设的快照保留个数（当前保留3个快照），找到最早生成的快照文件并进行删除。
2. 清理过期日志，使用 DLedger 提供的位点重置功能（MmapFileList#resetOffset()），将快照执行位点前的日志文件都进行清理（保留当下未写满的日志文件）。

RocketMQ DLedger Controller Snapshot 实现

DLdeger 中实现 Snapshot 的功能之后，将会对 Controller 模块做出以下的改进：

- 设计一个 ControllerSnapshotFile 及其文件格式。
- 构建 SnapshotGenerator，用于创建和读取 SnapshotFile。
- 实现 ControllerStateMachine 中的 onSnapshotLoad 和 onSnapshotSave 两个 API。

MetadataManager

MetadataManager 代表 DLedgerController 中的一个元数据管理器，也可以认为是 DLedgerController 的内存状态机. DLedgerController 会将 Event Apply 到 MetadataManager 中。
每个目前在 DLedgerController 中只有一个 MetadataManager，也即 ReplicasInfoManager，但后续可能会扩展出其他的 MetadataManager，例如 TopicManager 等等，所以我们的 SnapshotFile 需要注意能够承载多个 MetadataManager 的元数据。
对于每个具体的 MetadataManager，其需要实现 SnapshotAbleMetadataManager interface，并定义属于其自己的 Snapshot 格式。

```
public interface SnapshotAbleMetadataManager {
    /**
     * Encode the metadata contained in this MetadataManager
     * @return encoded metadata
     */
    byte[] encodeMetadata();


    /**
     *
     * According to the param data, load metadata into the
MetadataManager
     * @param data encoded metadata
     */
    void loadMetadata(byte[] data);


    /**
```

```
     * Get the type of this MetadataManager
     * @return MetadataManagerType
     */
    MetadataManagerType getMetadataManagerType();
}
```

SnapshotFile 结构

SnapshotFile 由以下两个部分组成

- SnapshotFilHeader: 记录版本, CRC 等信息。
- Sections 数组: 由多个 Section 组成, 每个 Section 记载着一个 MetadataManager 的元数据。

具体的结构如下:

SnapshotFilHeader:

```
- Version: Char
- TotalSections: Int  (section 的数量)
- Magic: Int
- CRC: Int (CRC 校验位)
- Reversed: Long (保留位, 用于后续扩展)
```

Sections:

```
- SectionHeader:
    - SectionType: Char (指 MetadataManager 的类别, 目前只有
ReplicasInfoManager, 其 SectionType = 1)
    - Length: Int (SectionBody 的总长度)
- SectionBody: bytes
```

ReplicasInfoManager Snapshot 结构

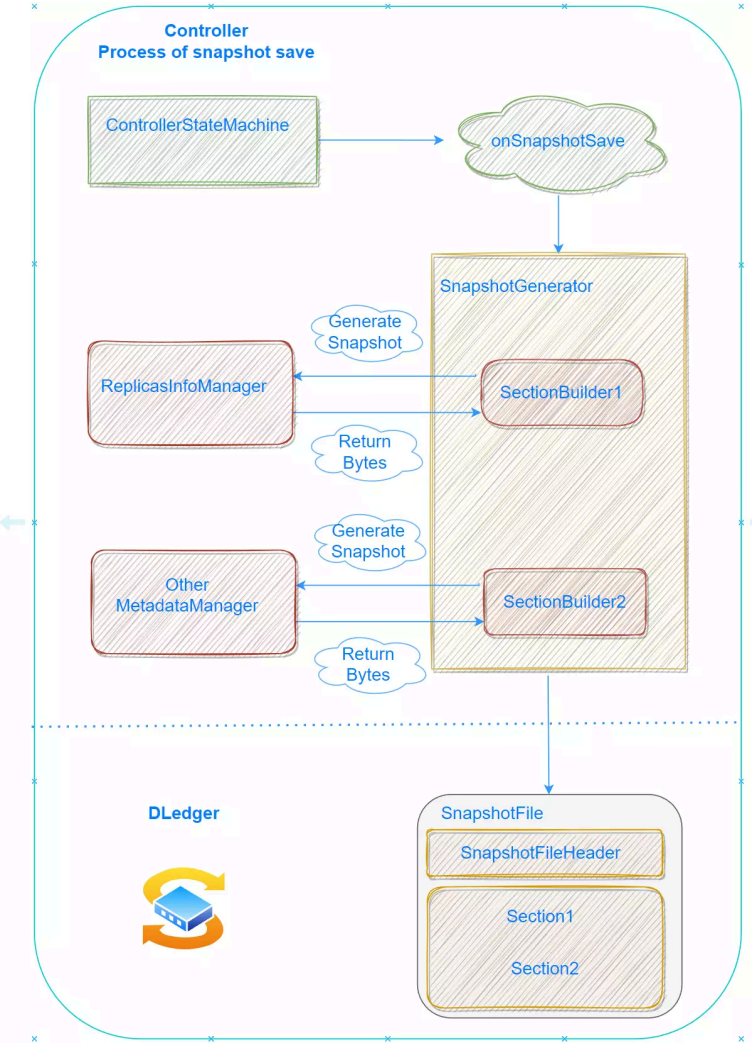
ReplicasInfoManager 中需要存储的元数据有:

```
private final Map<String/* brokerName */, BrokerInfo> replicaInfoTable;
private final Map<String/* brokerName */, SyncStateInfo>
syncStateSetInfoTable;
```

可以直接通过 JSON 序列化快照数据:

```
@Override
public byte[] encodeMetadata() {
    String json = RemotingSerializable.toJson(this, true);
    return json.getBytes(StandardCharsets.UTF_8);
}
@Override
public boolean loadMetadata(byte[] data) {
    String json = new String(data, StandardCharsets.UTF_8);
    ReplicasInfoManager obj = RemotingSerializable.fromJson(json,
ReplicasInfoManager.class);
    this.syncStateSetInfoTable.putAll(obj.syncStateSetInfoTable);
    this.replicaInfoTable.putAll(obj.replicaInfoTable);
    return true;
}
```

DLedgerController 完整快照流程



Snapshot Save process 如下:

- 当 DLedger 触发 Snapshot 后, 会调用 DLedgerControllerStateMachine 的 onSnapshotSave 接口。
- onSnapshotSave 构建 SnapshotGenerator。
- SnapshotGenerator 根据每个 MetadataManager, 创建对应的 SectionBuilder, 负责构建该 MetadataManager 的元数据快照。
- SnapshotGenerator 创建 SnapshotFile, 包括 Header 和 Sections。
- SnapshotGenerator 将 SnapshotFile 传入到 DLedger。
- Snapshot 结束。

Snapshot Load process 和 Save 流程相似。

Snapshot 模式 Jepsen 测试

现在DLedger已经实现了状态机、快照等功能，在之后也会实现成员变更、线性一致性等重要特性。如今的Jepsen已经无法满足这些新特性的测试，因此需要一个全新的Jepsen测试框架进行上述功能的验证。其目标分成以下几个部分：

- 快照模式下的状态机达到线性一致性
- 常见的网络分区、节点故障等情况下的高可用
- 详细的测试文档
- 集成CI

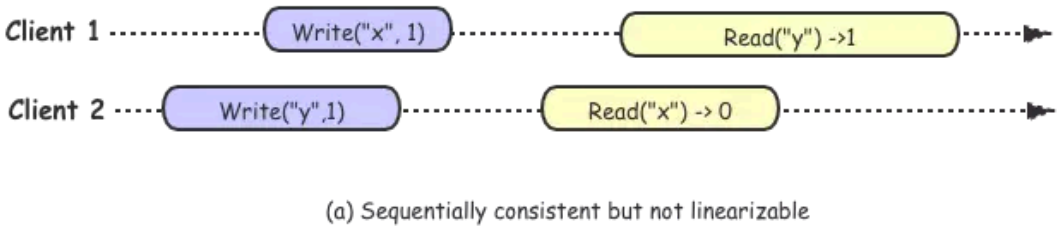
线性一致性

我们需要实现线性一致性的测试，那么可以使用Register模型进行验证，根据定义，一旦某一个读操作返回了新值，那么之后的所有读也都需要返回新值。
DLedger目前已经实现了一版Jepsen验证程序，但是目前的测试是测试RaftLog的最终写入一致性，和我们本次期待的测试目标不符，因此需要重新构建测试逻辑。

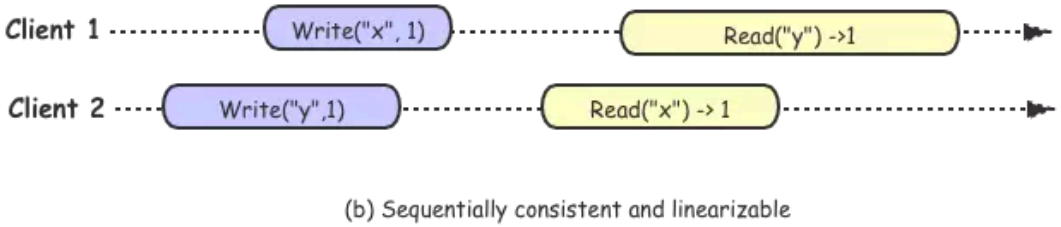
测试方案

RegisterStateMachine

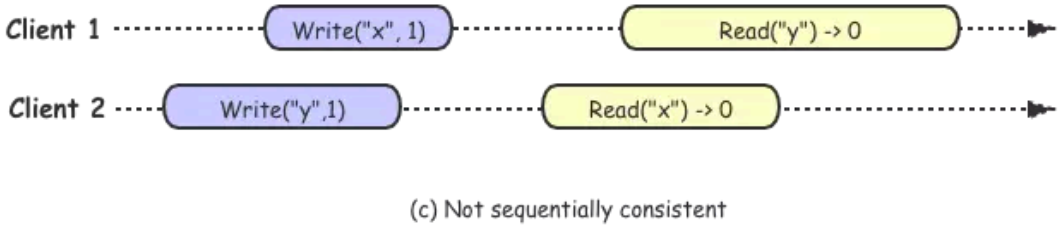
我们首先需要基于DLedger实现一个RegisterStateMachine，用于使用Register模型进行基本的Get和Write操作，将整个DLedger集群简化为一个寄存器状态机，我们只需要针对该寄存器进行读写，然后根据读写的操作日志，来验证该系统是否满足了线性一致性。



(a) Sequentially consistent but not linearizable



(b) Sequentially consistent and linearizable



(c) Not sequentially consistent

实际的代码实现则是简单的构建一个内存map(key->value)，其中key为String类型，value为Long类型，用于存储客户端的Write结果。

RaftLog Read or ReadIndex Read

由于目前DLedger只实现了针对RaftLog的线性写入和直接读出(直接访问目标index的存储位点)，因此我们还需要去实现对状态机数据的线性读取，也就是RaftLog Read 或者ReadIndex Read。该部分作为一个子模块在另外的文档中进行详细介绍。

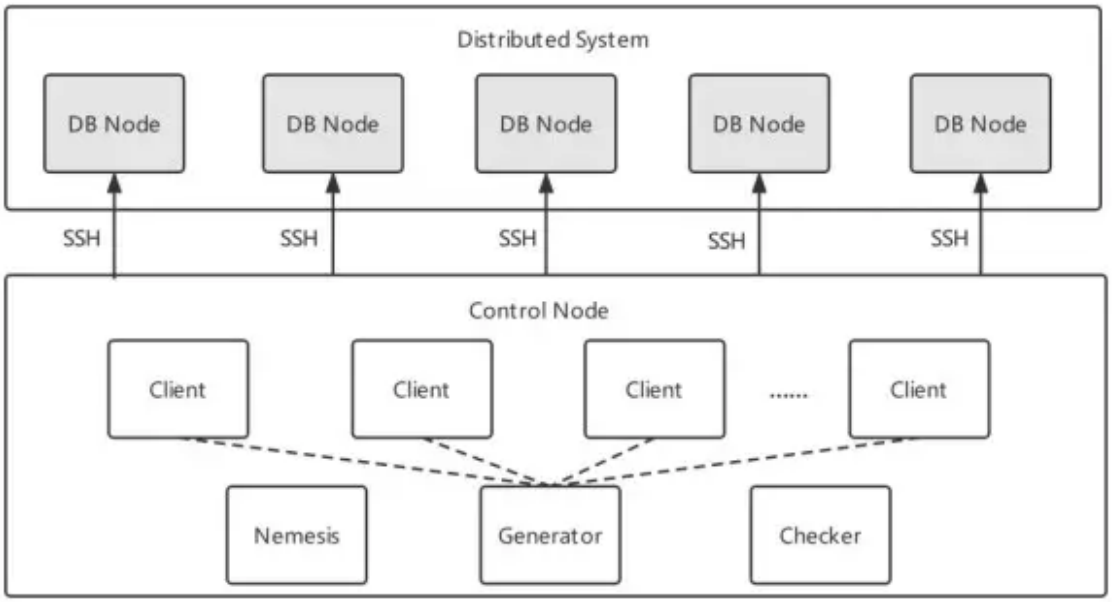
Nemesis

我们在测试过程中会持续的注入常见的故障，用于测试在故障环境下的高可用。

- partition-random-node: isolates a single node from the rest of the network.
- partition-random-halves: cuts the network into randomly chosen halves.
- kill-random-processes: kill random processes and restart them.
- crash-random-nodes: crash random nodes and restart them (kill processes and drop caches).
- hammer-time: pause random nodes with SIGSTOP/SIGCONT.

- bridge: a grudge which cuts the network in half, but preserves a node in the middle which has uninterrupted bidirectional connectivity to both components.
- partition-majorities-ring: every node can see a majority, but no node sees the *same* majority as any other. Randomly orders nodes into a ring.

测试流程



Start up

我们先启动5个DB node(DLedger)，这五个DLedger处于同一个Raft组中。
并且启动一个Control Node用于连接到DLedger所在的节点上进行初始化的操作。

Generator

然后启动Generator用于随机生成Client需要Invoke的操作序列。然后这些进程以用户访问系统的形式并发地执行操作。每个操作的结果会存在系统的日志中。

Nemesis

在client去invoke上述generator生成的操作序列时，会有各种故障进行注入。

Checker

在上述流程执行完中之后，会使用checker去分析上述流程中的测试历史(一系列invocation和response数据组成)。并且基于knossos库去使用WGL算法来验证上述测试是否满足了线性一致性。

Interface Design/Change

- Method signature/behavior changes

除Controller外，不涉及其他组件行为变更状态机接口如下（主要新增onSnapshotSave、onSnapshotLoad、onError接口）：

```
public interface StateMachine {
    /**
     * Update the user statemachine with a batch a tasks that can be
accessed
     * through |iterator|.
     *
     * @param iter iterator of committed entry
     */
    void onApply(final CommittedEntryIterator iter);
    /**
     * User defined snapshot generate function.
     *
     * @param writer snapshot writer
     */
    boolean onSnapshotSave(final SnapshotWriter writer);
    /**
     * User defined snapshot load function.
     *
     * @param reader snapshot reader
     * @return true on success
     */
    boolean onSnapshotLoad(final SnapshotReader reader);
    /**
     * Invoked once when the raft node was shut down.
     * Default do nothing
     */
    void onShutdown();
    /**
     * Once a critical error occurs, disallow any task enter the Dledger
node
     * until the error has been fixed and restart it.
     *
     * @param error DLedger error message
     */
    void onError(final DLedgerException error);
    /**
     * User must create DLedgerId by this method, it will generate the
DLedgerId with format like that: 'dLedgerGroupId#dLedgerSelfId'
```

```
     * @param dLedgerGroupId the group id of the DLedgerServer
     * @param dLedgerSelfId the self id of the DLedgerServer
     * @return generated unique DLedgerId
     */
    default String generateDLedgerId(String dLedgerGroupId, String
dLedgerSelfId) {
        return DLedgerUtils.generateDLedgerId(dLedgerGroupId,
dLedgerSelfId);
    }
    /**
     * User should return the DLedgerId which can be created by the
method 'StateMachine#generateDLedgerId'
     * @return DLedgerId
     */
    String getBindDLedgerId();
```

Controller组件参考 https://shimo.im/docs/WlArzngm9JhzY0A2#anchor-RM1W

- CLI command changes

无

- Log format or content changes

Controller新增以下参数：

- statemachineSnapshotThreshold：快照阈值，根据当前已应用至状态机的日志索引减去上次执行快照位置的高度差进行判断，初始值为1000。
- maxSnapshotReservedNum：最大快照保留个数，当快照个数超过该值会将最早过期快照进行清理，初始值为3。

Compatibility, Deprecation, and Migration Plan

- Are backward and forward compatibility taken into consideration?

不存在客户端的兼容性问题，该模式未对任何客户端层面 API 进行新增或修改。
Controller 不存在兼容性问题。如果使用 Controller 则默认启用 Snapshot 模式，日志应用至状态机达到阈值执行快照存储，节点重启则使用快照恢复进度。
除Controller外，不涉及其他组件行为变更，Controller本身无兼容问题。

- Are there deprecated APIs?

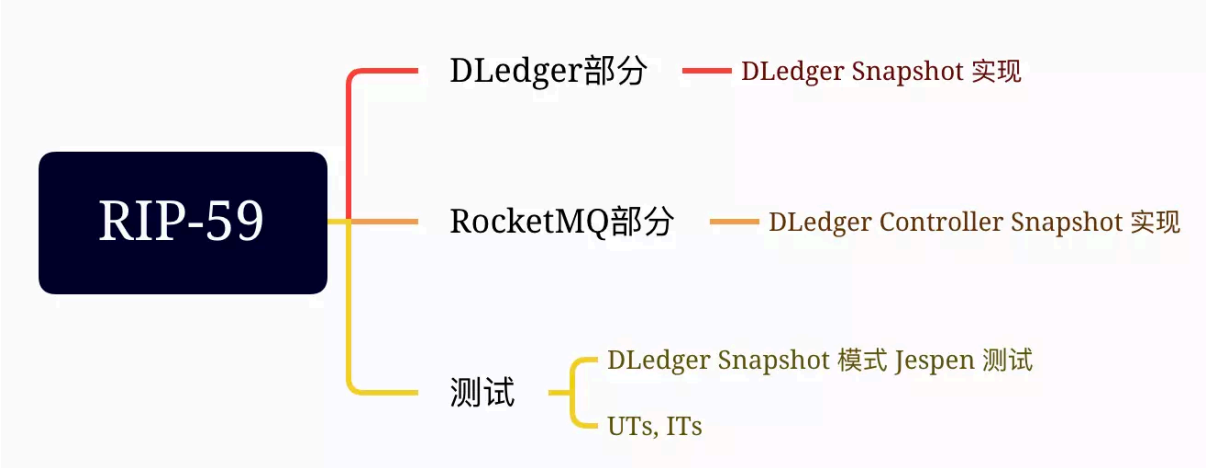
无

- How do we do migration?

除 Controller 外，不涉及其他组件，Controller 正常升级即可。Controller正常升级即可，无兼容性问题。

Implementation Outline

We will implement the proposed changes by 3 phase.



1. 实现 DLedger 自身的Snapshot能力@tsunghanjacktsai
2. RocketMQ中 DLedger Controller Snapshot 实现 @hzh0425
3. 针对DLedger状态机和Snapshot的Jepsen测试。@TheR1sing3un

Phase 2 和 Phase 3 同时进行。

Rejected Alternatives

无

Appendix

- [Issue #5585] Support DLedger Controller Snapshot
- 开发分支