TiKV 源码解析系列文章（十）Snapshot 的发送和接收 | PingCAP

> *TiKV 针对 Snapshot 收发场景做了特殊处理，解决了消息包过大会导致的一系列问题。*

背景知识

TiKV 使用 Raft 算法来提供高可用且具有强一致性的存储服务。在 Raft 中，Snapshot 指的是整个 State Machine 数据的一份快照，大体上有以下这几种情况需要用到 Snapshot：

1. 正常情况下 leader 与 follower/learner 之间是通过 append log 的方式进行同步的，出于空间和效率的考虑，leader 会定期清理过老的 log。假如 follower/learner 出现宕机或者网络隔离，恢复以后可能所缺的 log 已经在 leader 节点被清理掉了，此时只能通过 Snapshot 的方式进行同步。

2. Raft 加入新的节点的，由于新节点没同步过任何日志，只能通过接收 Snapshot 的方式来同步。实际上这也可以认为是 1 的一种特殊情形。

3. 出于备份 / 恢复等需求，应用层需要 dump 一份 State Machine 的完整数据。

TiKV 涉及到的是 1 和 2 这两种情况。在我们的实现中，网络发送给 Region follower/learner 所在的 TiKV。

理论上讲，我们完全可以把 Snapshot 当作普通的 `RaftMessage` 来发送，但这样做实践上会产生一些问题，主要是因为 Snapshot 消息的尺寸远大于其他 `RaftMessage`：

1. Snapshot 消息需要花费更长的时间来发送，如果共用网络连接容易导致网络拥塞，进而引起其他 Region 出现 Raft 选举超时等问题。
2. 构建待发送 Snapshot 消息需要消耗更多的内存。
3. 过大的消息可能导致超出 gRPC 的 Message Size 限制等问题。

基于上面的原因，TiKV 对 Snapshot 的发送和接收进行了特殊处理，为每个 Snapshot 创建单独的网络连接，并将 Snapshot 拆分成 1M 大小的多个 Chunk 进行传输。

源码解读

下面我们分别从 RPC 协议、发送 Snapshot、收取 Snapshot 三个方面来解读相关源代码。本文的所有内容都基于 v3.0.0-rc.2 版本。

Snapshot RPC call 的定义

与普通的 raft message 类似，Snapshot 消息也是使用 gRPC 远程调用的方式来传输的。在 pingcap/kvproto 项目中可以找到相关 RPC Call 的定义，具体在 tikvpb.proto 和 raft_serverpb.proto 文件中。

```
  bytes data = 2;
}
message Done {}
```

可以看出，Snapshot 被定义成 client streaming 调用，即对于每个 Call，客户端依次向服务器发送多个相同类型的请求，服务器接收并处理完所有请求后，向客户端返回处理结果。具体在这里，每个请求的类型是 `SnapshotChunk`，其中包含了 Snapshot 对应的 `RaftMessage`，或者携带一段 Snapshot 数据；回复消息是一个简单的空消息 `Done`，因为我们在这里实际不需要返回任何信息给客户端，只需要关闭对应的 stream。

Snapshot 的发送流程

shot 的发送过程的处理比较简单粗暴，直接在将要 `RaftMessage` 的地方截获 Snapshot 类型的消息，转过特殊的方式进行发送。相关代码可以在 r/transport.rs 中找到：

```
rite_data(&self, store_id: u64, addr: &str, msg: RaftMessage
 msg.get_message().has_snapshot() {
  return self.send_snapshot_sock(addr, msg);
}
if let Err(e) = self.raft_client.wl().send(store_id, addr, msg
  error!("send raft msg err"; "err" => ?e);
}
}
fn send_snapshot_sock(&self, addr: &str, msg: RaftMessage) {
...
if let Err(e) = self.snap_scheduler.schedule(SnapTask::Send {
  addr: addr.to_owned(),
  msg,
  cb,
}) {
  ...
}
}
```

从代码中可以看出，这里简单地把对应的 [redacted] 包 worker 去处理。值得注意的是，这里的 RaftMessage 只包含 Snapshot 的元信息，而不包括真正的快照数据。TiKV 中有一个单独的模块叫做 SnapManager ，用来专门处理数据快照的生成与转存，稍后我们将会看到从 SnapManager 模块读取 Snapshot 数据块并进行发送的相关代码。

我们不妨顺藤摸瓜来看看 snap-worker 是如何处理这个任务的，相关代码在 server/snap.rs ，精简掉非核心逻辑后的代码引用如下：

```
fn run(&mut self, task: Task) {
  match task {
      Task::Recv { stream, sink } => {
           ...
           let f = recv_snap(stream, sink, ...).then(move |resul
               ...
           });
           self.pool.spawn(f).forget();
      }
      Task::Send { addr, msg, cb } => {
          ...
          let f = future::result(send_snap(..., &addr, msg))
              .flatten()
              .then(move |res| {
                  ...
              });
          self.pool.spawn(f).forget();
      }
```


snap-worker 使用了 future 来完成收发 Snapshot 任务：通过调用 send_snap() 或 recv_snap() 生成一个 future 对象，并将其交给 FuturePool 异步执行。

现在我们暂且只关注 send_snap() 的 实现 ：

```
fn send_snap(
  ...
  addr: &str,
  msg: RaftMessage,
) -> Result<impl Future<Item = SendStat, Error = Error>> {
  ...
```

```
    let key = {
        ...
    let s = box_try!(mgr.get_snapshot_for_sending(&key));
    if !s.exists() {
        return Err(box_err!("missing snap file: {:?}", s.path()));
    }
    let total_size = s.total_size()?;
    let chunks = {
        let mut first_chunk = SnapshotChunk::new();
        first_chunk.set_message(msg);
        SnapChunk {
            first: Some(first_chunk),
            snap: s,
            remain_bytes: total_size as usize,
        }
    };
    let cb = ChannelBuilder::new(env);
    let channel = security_mgr.connect(cb, addr);
    let client = TikvClient::new(channel);
    let (sink, receiver) = client.snapshot()?;
    let send = chunks.forward(sink).map_err(Error::from);
    let send = send
        .and_then(|(s, _)| receiver.map_err(Error::from).map(|_| s
        .then(move |result| {
            ...
        });
    Ok(send)
}
```


发流程还是比较清晰的：先是用 Snapshot 元信息从 `nager` 取到待发送的快照数据，然后将 `RaftMessage` `ap` 一起封装进 `SnapChunk` 结构，最后创建全新的 连接及一个 Snapshot stream 并将 `SnapChunk` 写

这里引入 `SnapChunk` 是为了避免将整块 Snapshot 快照一次性加载进内存，它 impl 了 `futures::Stream` 这个 trait 来达成按需加载流式发送的效果。如果感兴趣可以参考它的 具体实现 ，本文就暂不展开了。

Snapshot 的收取流程

最后我们来简单看一下 Snapshot 的收取流程，其实也就是 gRPC Call 的 server 端对应的处理，整个流程的入口我们可以在 server/service/kv.rs 中找到：

```
  sink: ClientStreamingSink<Done>,
) {
  let task = SnapTask::Recv { stream, sink };
  if let Err(e) = self.snap_scheduler.schedule(task) {
      ...
  }
}
```

与发送过程类似，也是直接构建 `SnapTask::Recv` 任务并转发给 `snap-worker` 了，这里会调用上面出现过的 `recv_snap()` 函数， 具体实现 如下：

```
fn recv_snap<R: RaftStoreRouter + 'static>(
  stream: RequestStream<SnapshotChunk>,
  sink: ClientStreamingSink<Done>,
  ...
) -> impl Future<Item = (), Error = Error> {
  ...
  let f = stream.into_future().map_err(|(e, _)| e).and_then(
      move |(head, chunks)| -> Box<dyn Future<Item = (), Error =
          let context = match RecvSnapContext::new(head, &snap_m
              Ok(context) => context,
              Err(e) => return Box::new(future::err(e)),
          };
          ...
          let recv_chunks = chunks.fold(context, |mut context, m
              let data = chunk.take_data();
              ...
              if let Err(e) = context.file.as_mut().unwrap().wri
                  ...
              }
              Ok(context)
          });
          Box::new(
              recv_chunks
                  .and_then(move |context| context.finish(raft_r
                  .then(move |r| {
                      snap_mgr.deregister(&context_key, &SnapEnt
                      r
                  }),
          )
      },
  );
  f.then(move |res| match res {
      ...
  })
  .map_err(Error::from)
}
```

个 chunk 收取后都依次写入文件，最后调用 `context.finish()` 把之前保存的 `RaftMessage` 发送给 `raftstore` 完成整个接收过程。

总结

以上就是 TiKV 发送和接收 Snapshot 相关的代码解析了。这是 TiKV 代码库中较小的一个模块，它很好地解决了由于 Snapshot 消息特殊性所带来的一系列问题，充分应用了 `grpc-rs` 组件及 `futures` / `FuturePool` 模型，大家可以结合本系列文章的 《TiKV 源码解析系列文章（七）gRPC Server 的初始化和启动流程》 和 《TiKV 源码解析系列文章（八）grpc-rs 的封装与实现》 进一步拓展学习。

去查看更多 TiKV 源码解析系列文章

全文完