I. SQL 全链路实现

概览

在这一章节，我们将实现从客户端输入 SQL 到数据写入分布式的 KV 数据库中的全链路。

详细的资料可以参考文章 TiDB 源码阅读系列文章（三）SQL 的一生，此处仅做简要说明。

通用调用链路

- 1. `server/conn.go`，当客户端连接到 TinySQL/TiDB 时，会开启一个 goroutine，会启动一个 `clientConn.Run` 函数，这个函数会不停的循环从客户端读取请求数据并执行。
- 2. `server/conn.go`，不同种类的请求会在 `clientConn.dispatch` 进行分类，我们主要关注的 SQL 请求会在这里被解析为 SQL 字符串，然后交给 `clientConn.handleQuery` 函数执行。
- 3. `session/session.go`，SQL 的执行会调用到 `TiDBContext.Execute` 函数进而调用 `session.Execute` 和 `session.execute`，`session.execute` 函数会负责一条 SQL 执行的生命周期，包括语法分析、优化、执行等阶段。
    - 3.1. `session/session.go`，首先调用 `session.ParseSQL` 将 SQL 字符串转化为一颗或一些语法树，然后逐个执行。
    - 3.2. `executor/compiler.go`，`Compiler.Compile` 将一颗语法树进行优化，依次生成逻辑执行计划和物理执行计划。
    - 3.3. `session/session.go`，通过 `session.executeStatement` 在 `runStmt` 函数中调用执行器的 `Exec` 函数。
    - 3.4. `session/tidb.go`，在执行完 Exec 函数后，如果没有出现错误，则调用 `session.StmtCommit` 方法将这一条语句 Commit 到整个事务所属的 membuffer 当中去。
- 4. `executor/adapter.go`，我们将 `ExecStmt.Exec` 函数的执行作为一个阶段，展开描述。
    - 4.1. `executor/adapter.go`，`ExecStmt.Exec` 会调用 `ExecStmt.buildExecutor`，通过物理执行计划，构建执行器。
    - 4.2. `executor/adapter.go`，`Executor` 是一个层叠的结构，在调用顶层

的 `Executor.Open` 方法后，会传递到其中的子 `Executor` 当中，这一操作会递归地将所有的 `Executor` 都初始化。

    - 4.3. `executor/adapter.go`，在 `ExecStmt.handleNoDelay` 中，如果这个 `Executor` 不会返回结果，那么它会在 `ExecStmt.handleNoDelayExecutor` 函数内部立即执行。
        - 4.3.1. `executor/adapter.go`，在 `ExecStmt.handleNoDelayExecutor` 通过 `Next` 函数递归执行 `Executor`，这里会使用 `newFirstChunk` 函数来生成存储结果的 `Chunk`， `Chunk` 是一种使用 Apache Arrow 表达的数据格式。
    - 4.4. `executor/adapter.go`，如果这个 `Executor` 会返回结果，那么执行器会被层层返回到第 2 步的 `clientConn.handleQuery` 中，随后在 `clientConn.writeResultset` 中调用执行 `clientConn.writeChunks` 执行，这么做的原因是为了流式的将执行的结果返回给客户端，而不是将所有结果存放在 DBMS 的内存中。在 `clientConn.writeChunks` 中，会调用 `ResultSet.Next` 函数来执行，每次调用会返回一条数据，直到返回的数据为空，说明执行完成。
- 5. `executor/simple.go`，在 4.3.1 阶段中，存在几种特殊的执行器，执行入口在 `SimpleExec.Next` 里，这里主要列举和事务相关的 Begin/Commit/Rollback。
    - 5.1. `executor/simple.go`， `SimpleExec.executeBegin` 会通过 `session/session.go` 中的 `session.NewTxn` 函数（被定义在 `sessionctx.Context` 接口中）来创建一个新的事务，如果此时这个 session 中有尚未提交的事务， `NewTxn` 会先提交事务后开启一个新事务。在开启新事务后，会通过 `session.Txn` 函数（也被定义在 `sessionctx.Context` 接口中）等待这个事务获取到 `startTS`。此外，begin 时会将环境变量中的 `mysql.ServerStatusInTrans` 设置为 `true`。
    - 5.2. `executor/simple.go`， `SimpleExec.executeCommit` 会将 5.1 中的 `mysql.ServerStatusInTrans` 变量设置为 `false`。
        - 5.2.1. `session/tidb.go` 中的 finishStmt 会在第 4 结束时被调用，5.2 中将 `mysql.ServerStatusInTrans` 变量设置为 `false` 导致 `sessVars.InTxn()` 的返回值为 `false`，此时会调用 `session.CommitTxn` 提交事务。
    - 5.3 `executor/simple.go`， `SimpleExec.executeRollback` 也会将 `mysql.ServerStatusInTrans` 设置为 `false`，但是会在 `executeRollback` 函数内部就对事物进行 Rollback。和 5.1 一样，会通

过 `session.Txn` 函数来获取当前事务，但是不会等待事务激活（注意输入的参数）。如果获取到了事务，则会调用这个事务的 `Rollback` 方法进行清理。

以上是 SQL 执行的关键链路，但是这个调用链路中的每一步都有关键函数被移除了，你需要根据调用链路的描述进行填充。