Updateserver并发事务设计

[当前队列线程结构 1](#_Toc332120892)

[当前批处理流程 2](#_Toc332120893)

[并行事务处理概述 3](#_Toc332120894)

[隔离级别保证 3](#_Toc332120895)

[分发线程 3](#_Toc332120896)

[两阶段锁 3](#_Toc332120897)

[死锁处理 4](#_Toc332120898)

[Session和事务上下文 4](#_Toc332120899)

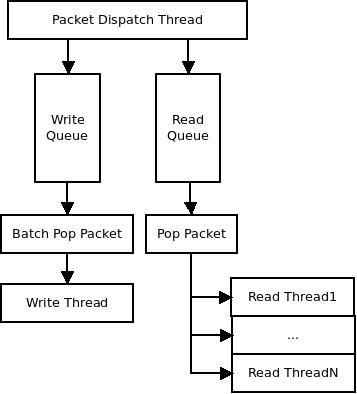
[预提交和批处理 4](#_Toc332120900)

Updateserver设计初期为了实现简单，使用单线程处理数据更新和内部管理命令(主要包括冻结、旁路加载、更新schema等)，分发线程根据packet code，将这类请求的packet对象加入写线程队列，由单一的写线程处理这个队列中的数据更新和内部管理命令。

在这种设计方案下，存在两个主要的限制：第一，性能瓶颈，就目前的测试来看，经过优化的单一写线程处理更新和插入行，每秒约6万行，不使用多核处理的话，性能很难有大的提升；第二，无法支持sql-session，单一写线程，没有并发的事务控制机制，无法支持sql所需要的大量session并行处理的需求。

本文的设计目标是在现有updateserver的基础上，以对现有结构尽量小的改动，和相对简单并发事务机制，实现对并发读写事务和session的支持。

## 当前队列线程结构



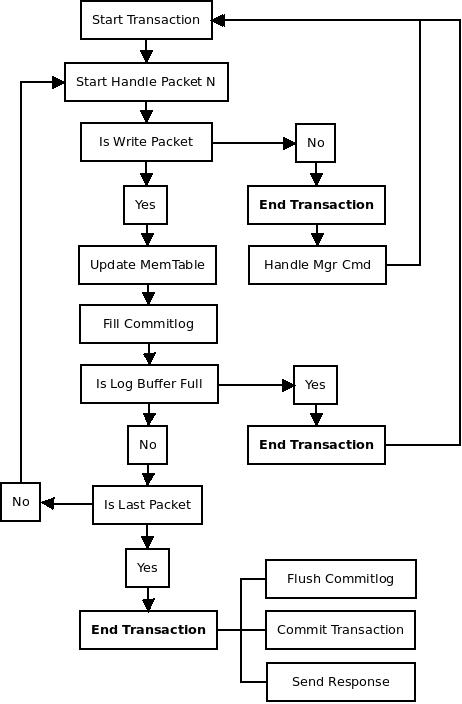
现有结构如上图所示，网络框架的收包线程即分发线程，框架收包完成后会调用ups实现的handlePacket方法，对于只读任务会被分发到读线程队列，对于会引起内部状态改变或要写commitlog的任务会被分发到写线程队列。

处理读任务的是一个线程池，每个线程每次从队列中取一个任务、执行、发送异步应答，然后再取下一个任务。处理写任务只有一个线程，它每次从队列中批量取最多1024个任务，执行后将commitlog写入缓冲区，在缓冲区满或批量取的任务做完后，将缓冲区同步到日志文件和备机，发送异步应答后，再取下一批任务。

## 当前批处理流程

现有的单线程批处理流程如下如所示，写线程一次性从队列批量取最多1024个任务，然后开启批处理事务，循环处理每个任务，对于普通数据更新请求，在当前批处理事务中执行，对于管理命令(如冻结命令)，需要保证当前没有正在处理的写任务，因此需要结束并提交当前批处理事务，然后再执行管理命令。

在执行批处理的过程中，每个写请求的commitlog写入缓冲区，但不立即写入磁盘，而是等批处理的所有写任务完成后，或缓冲区满的情况下，才flush到磁盘并同步到备机。在commitlog被同步到磁盘之后，与其相关的数据更新将被提交，此时其他线程可以读取到。



## 并行事务处理概述

没有特别说明的情况下以下设计专门针对serializable隔离级别下的读写混合事务。

#### 隔离级别保证

只读事务从快照读取，不需要考虑隔离级别；对于读写混合事务，全库提供两种隔离级别，read commited和serializable。由于基于快照技术实现，我们的read commited实际效果会相当于snapshot隔离级别，即完全满足read commited，而部分满足repeatable read（可能会lost update）。而serializable级别则通过经典的两阶段锁（行锁+表锁）方式保证。

在全库read commited级别的情况下，可以通过对单个事务指定隔离级别，来提高本次事务的隔离级别，有两种方式：一是在开启事务的时候指定隔离级别；二是使用select… for update语句指定查询对象的隔离级别。

#### 分发线程

收包线程将写任务和管理命令的packet放入写线程队列后，仍然使用一个线程（下称为TransDispatchThread）处理队列中的任务，但是这个线程对于普通更新请求，不在线程本地进行任何处理，而是直接分发给工作线程，由工作线程池来真正执行包的反序列化和更新事务。

而对于冻结和旁路导入等管理命令，则由TransDispatchThread等待所有已分发的事务提交后，再在本地执行这些管理命令。

#### 两阶段锁

只读事务都从历史快照读取，不需要加锁，而读写混合事务需要两个层次的锁，表锁和行锁，对于不带范围查询和修改（查询和修改明确指定rowkey）的情况，加表读锁后，根据读写需求对行加读锁或写锁；当涉及范围查询或修改的情况下，需要对表加写锁。

使用两阶段锁处理并发事务的同步，在每行的行头使用一个int64\_t来保存行的锁状态信息，使用其中的int32\_t来存储读写状态和引用计数，另外一个int32\_t用于保存进行死锁检测的必要信息（如锁owner的sessionID）。

两阶段锁需要保证的是，加锁过程和解锁过程是两个阶段的，也就是说需要保证在第一次解锁之后，不会再有加锁的操作，否则无法保证锁保护数据的一致性。为了获得较高的并发性，我们只有在真正需要对某行数据进行更新的时候才加写锁，对于查询操作只加读锁，因此也就有了读锁升级为写锁的需求，需要注意的是当读锁要升级为写锁的时候，不能先解读锁再加写锁，而是要保证原子性的升级为写锁。

一个简单的例子可以说明，初始为A=0，事务1执行A=A+1，事务2执行A=A+1。事务1先获取A的读锁，读取A的值为0，继续执行发现需要更新A的值，升级读锁为写锁，如果升级过程不原子，那么可能出现事务1刚解开读锁后，事务2加锁执行完成，等到事务1再获取写锁后，A的值已经被事务2更新为1，与之前读到的数据不一致。

使用”更新锁”这样的中间状态来避免读写升级为写锁时的死锁问题，因此对于保存读写状态的int32\_t，使用最低一位保存写（exclusive）锁状态标志，使用倒数第二位保存更新（upgrade）锁状态标志，其余的高30位保存读（share）锁引用计数。这样一来加读锁时只需要原子的加4后，再检查exclusive和upgrade状态标志即可，而避免了使用代价更高的CAS操作。读锁升级为写锁时，先用CAS原子的修改upgrade标记，修改成功后等待所有读锁释放，再修改exclusive标记。

为了满足两阶段锁的保证，在事务执行过程中，只加锁不解锁，而解锁需要等待事务提交（已同步日志并保证更新数据可见）之后再将事务已加的锁全部释放。

#### 死锁处理

为了一期实现简单，在遇到锁等待的情况下，使用随机等待重试的方式，在多次尝试获取锁都失败的情况下，回滚整个事务。后面的开发版本将实现基于事务优先级机制的死锁检测和处理算法。

一个简单的优化：由于两阶段锁的解锁要在事务提交之后，因此在尝试加锁失败的情况下，可以检查锁的own事务是否已经进入正在提交阶段，如果是的话表示当前不是死锁状态，可以继续等待。

#### Session和事务上下文

为了实现支持SQL多次交互的事务功能，为每个已开启的事务分配唯一的sessionID，每个sessionID与一个事务上下文结构（TransContext）关联，客户端通过sessionID在已开启的事务内进行读写操作、提交或回滚。因此事务经过预提交过程后，不一定会立即提交，而是可能反复多次执行预提交，知道客户端发送结束事务消息后。而每个事务也有最长生存周期，执行超过这个周期的事务将被强制回滚结束。

事务上下文中保存了事务执行过程中必要的状态信息、临时数据、锁等内容。其中比较重要的内容包括：

1. 行锁，事务执行过程中加的锁，要在结束时释放
2. 行更新数据，事务执行过程中每行的更新数据（已从memtable分配内存，并构造了内部结构），将暂存在事务上下文中，在事务提交时，将这些数据指针链接到每行的更新链表尾部
3. 事务状态标记，预提交或等待提交
4. 每行需要做checksum的数据buffer（数据在buffer中被连续存放）
5. 事务预提交执行结果，错误码和描述字符串
6. Session类型，立即提交或等待继续执行预提交

#### 预提交和批处理

事务的执行过程分为两大阶段：预提交和批处理。其中预提交阶段由线程池处理，主要包括事务读、写、加锁操作；批处理阶段由单线程完成，主要包括分配事务ID、同步操作日志、提交修改。以下是在serializable隔离级别下的设计说明。

###### 预提交

事务处理线程池将用户请求包反序列化后，开始处理事务，先对要操作的表加读锁，对于查询操作，需要对要查询的行（表）加读锁；对于更新操作，需要对要更新的行（表）加写锁（或读锁升级为写锁）。加锁成功后将行头结构指针保存在事务上下文中。

对于更新操作，构造行内数据结构并填充更新数据，然后将这个行内数据结构指针保存在事务上下文中（此时并不修改行头信息），多次更新的结构则依时间次序在事务上下文中保存。在执行事务的过程中，会同时build本次事务的操作日志和checksum所需数据，并保存到事务上下文中。

对于查询操作，先要读取memtable中行最新的数据，与本事务上下文中保存的这行的更新数据合并后作为查询结果。在read commited隔离级别下，事务中的查询操作不需要加锁，而是从事务开始时的快照点上读取memtable中的行数据，再与事务上下文中保存的这行的更新数据合并后作为查询结果。

###### 批处理

事务的预提交过程在线程池中执行完成后，将事务上下文发给单一的提交线程（下成为CommitThread）。CommitThread循环处理每个事务，首先分配一个全局唯一且自增的事务ID，然后将操作日志写入缓冲区。当日志缓冲区满，或5ms（可配置）内没有新提交事务，或最早等待提交的事务已经等待超过10ms（可配置）的情况下，将当前日志缓冲区同步到磁盘和备机。

然后对上述每个事务执行提交操作：将事务上下文中保存的每行的更新数据连同刚刚获取的事务ID，链接到每行的更新链表的尾部，计算checksum并修改行头结构一个事务提交完成后，将事务执行过程中加的锁释放掉，并修改当前全局的max\_commited\_trans\_id为当前事务ID。此时事务提交完成，向客户端返回应答。上述提交操作只涉及简单的指针访问和修改，没有复杂的计算，因此可以由CommitThread单线程执行。