解析MongoDB存储引擎WiredTiger：事务实现

袁荣喜

2016-05-30

WiredTiger从被MongoDB收购到成为MongoDB的默认存储引擎的一年半得到了迅猛的发展，也逐步被外部熟知。WiredTiger（以下简称WT）是一个优秀的单机数据库存储引擎，它拥有诸多的特性，既支持BTree索引，也支持LSM Tree索引，支持行存储和列存储，实现ACID级别事务、支持大到4G的记录等。WT的产生不是因为这些特性，而是和计算机发展的现状息息相关。

现代计算机近20年来CPU的计算能力和内存容量飞速发展，但磁盘的访问速度并没有得到相应的提高，WT就是在这样的一个情况下研发出来，它设计了充分利用CPU并行计算的内存模型的无锁并行框架，使得WT引擎在多核CPU上的表现优于其他存储引擎。针对磁盘存储特性，WT实现了一套基于BLOCK/Extent的友好的磁盘访问算法，使得WT在数据压缩和磁盘I/O访问上优势明显。实现了基于snapshot技术的ACID事务，snapshot技术大大简化了WT的事务模型，摒弃了传统的事务锁隔离又同时能保证事务的ACID。WT根据现代内存容量特性实现了一种基于Hazard Pointer 的LRU cache模型，充分利用了内存容量的同时又能拥有很高的事务读写并发。

在本文中，我们主要针对WT引擎的事务来展开分析，来看看它的事务是如何实现的。说到数据库事务，必然先要对事务这个概念和ACID简单的介绍。

## 事务与ACID

什么是事务？事务就是通过一系列操作来完成一件事情，在进行这些操作的过程中，要么这些操作完全执行，要么这些操作全不执行，不存在中间状态，事务分为事务执行阶段和事务提交阶段。一般说到事务，就会想到它的特性—ACID，那么什么是ACID呢？我们先用一个现实中的例子来说明：AB两同学账号都有1000块钱，A通过银行转账向B转了100，这个事务分为两个操作,即从A同学账号扣除100，向B同学账号增加100.

原子性(Atomicity):

组成事务的系列操作是一个整体，要么全执行，要么不执行。通过上面例子就是从A同学扣除钱和向B同学增加100是一起发生的，不可能出现扣除了A的钱，但没增加B的钱的情况。

一致性(Consistency):

在事务开始之前和事务结束以后，数据库的完整性和状态没有被破坏。这个怎么理解呢？就是A

B两人在转账钱的总和是2000，转账后两人的总和也必须是2000。不会因为这次转账事务破坏这个状态。

隔离性(Isolation):

多个事务在并发执行时，事务执行的中间状态是其他事务不可访问的。A转出100但事务没有确认提交，这时候银行人员对其账号查询时，看到的应该还是1000，不是900。

持久性(Durability):

事务一旦提交生效，其结果将永久保存，不受任何故障影响。A转账一但完成，那么A就是900，B就是1100这个结果将永远保存在银行的数据库中，直到他们下次交易事务的发生。

## WT事务构造

知道了基本的事务概念和ACID后，来看看WT引擎是怎么来实现事务和ACID的。要了解实现先要知道它的事务的构造和使用相关的技术，WT在实现事务的时使用主要是使用了三个技术：snapshot(事务快照)、MVCC (多版本并发控制)和redo log(重做日志)，为了实现这三个技术，它还定义了一个基于这三个技术的事务对象和全局事务管理器。事务对象描述如下

wt\_transaction{

transaction\_id: 本次事务的全局唯一的ID，用于标示事务修改数据的版本号

snapshot\_object: 当前事务开始或者操作时刻其他正在执行且并未提交的事务集合,用于事务隔离

operation\_array: 本次事务中已执行的操作列表,用于事务回滚。

redo\_log\_buf: 操作日志缓冲区。用于事务提交后的持久化

State: 事务当前状态

}

### WT的多版本并发控制

WT中的MVCC是基于key/value中value值的链表，这个链表单元中存储有当先版本操作的事务ID和操作修改后的值。描述如下：

wt\_mvcc{

transaction\_id: 本次修改事务的ID

value: 本次修改后的值

}

WT中的数据修改都是在这个链表中进行append操作，每次对值做修改都是append到链表头上，每次读取值的时候读是从链表头根据值对应的修改事务transaction\_id和本次读事务的snapshot来判断是否可读，如果不可读，向链表尾方向移动，直到找到读事务能都的数据版本。样例如下：



图1

上图中，事务T0发生的时刻最早，T5发生的时刻最晚。T1/T2/T4是对记录做了修改。那么在mvcc list当中就会增加3个版本的数据，分别是11/12/14。如果事务都是基于snapshot级别的隔离，T0只能看到T0之前提交的值10，读事务T3访问记录时它能看到的值是11，T5读事务在访问记录时，由于T4未提交，它也只能看到11这个版本的值。这就是WT 的MVCC基本原理。

### WT事务snapshot

上面多次提及事务的snapshot，那到底什么是事务的snapshot呢？其实就是事务开始或者进行操作之前对整个WT引擎内部正在执行或者将要执行的事务进行一次截屏，保存当时整个引擎所有事务的状态，确定哪些事务是对自己见的，哪些事务都自己是不可见。说白了就是一些列事务ID区间。WT引擎整个事务并发区间示意图如下：



图2

WT引擎中的snapshot\_oject是有一个最小执行事务snap\_min、一个最大事务snap max和一个处于[snap\_min, snap\_max]区间之中所有正在执行的写事务序列组成。如果上图在T6时刻对系统中的事务做一次snapshot，那么产生的snapshot\_object = {

snap\_min=T1,

snap\_max=T5,

snap\_array={T1, T4, T5},

};

那么T6能访问的事务修改有两个区间：所有小于T1事务的修改[0, T1)和[snap\_min, snap\_max]区间已经提交的事务T2的修改。换句话说，**凡是出现在snap\_array中或者事务ID大于snap\_max的事务的修改对事务T6是不可见的**。如果T1在建立snapshot之后提交了，T6也是不能访问到T1的修改。这个就是snapshot方式隔离的基本原理。

### 全局事务管理器

通过上面的snapshot的描述，我们可以知道要创建整个系统事务的快照截屏，就需要一个全局的事务管理来进行事务截屏时的参考，在WT引擎中是如何定义这个全局事务管理器的呢？在CPU多核多线程下，它是如何来管理事务并发的呢？下面先来分析它的定义：

wt\_txn\_global{

current\_id: 全局写事务ID产生种子,一直递增

oldest\_id: 系统中最早产生且还在执行的写事务ID

transaction\_array: 系统事务对象数组，保存系统中所有的事务对象

scan\_count: 正在扫描transaction\_array数组的线程事务数，用于建立snapshot过程的无锁并发

}

transaction\_array保存的是图2正在执行事务的区间的事务对象序列。在建立snapshot时，会对整个transaction\_array做扫描，确定snap\_min/snap\_max/snap\_array这三个参数和更新oldest\_id**,在扫描的过程中，凡是transaction\_id不等于WT\_TNX\_NONE都认为是在执行中且有修改操作的事务，直接加入到snap\_array当中。**整个过程是一个无锁操作过程,这个过程如下：



图3

创建snapshot截屏的过程在WT引擎内部是非常频繁，尤其是在大量自动提交型的短事务执行的情况下，由创建snapshot动作引起的CPU竞争是非常大的开销，所以这里WT并没有使用spin lock ,而是采用了上图的一个无锁并发设计，这种设计遵循了我们开始说的并发设计原则。

### 事务ID

从WT引擎创建事务snapshot的过程中现在可以确定，snapshot的对象是有写操作的事务，纯读事务是不会被snapshot的，因为snapshot的目的是隔离mvcc list中的记录，通过MVCC中value的事务ID与读事务的snapshot进行版本读取，与读事务本身的ID是没有关系。在WT引擎中，开启事务时，引擎会将一个WT\_TNX\_NONE( = 0)的事务ID设置给开启的事务，**当它第一次对事务进行写时，会在数据修改前通过全局事务管理器中的current\_id来分配一个全局唯一的事务ID**。这个过程也是通过CPU的CAS\_ADD原子操作完成的无锁过程。

## WT的事务过程

一般事务是两个阶段：事务执行和事务提交。在事务执行前，我们需要先创建事务对象并开启它，然后才开始执行，如果执行遇到冲突和或者执行失败，我们需要回滚事务(rollback)。如果执行都正常完成，最后只需要提交(commit)它即可。从上面的描述可以知道事务过程有：创建开启、执行、提交和回滚。那么从这几个过程中来分析WT是怎么实现这几个过程的。

### 事务开启

WT事务开启过程中，首先会为事务创建一个事务对象并把这个对象加入到全局事务管理器当中，然后通过事务配置信息确定事务的隔离级别和redo log的刷盘方式并将事务状态设为执行状态，最后判断如果隔离级别是ISOLATION\_SNAPSHOT(snapshot级的隔离)，在本次事务执行前创建一个系统并发事务的snapshot截屏。至于为什么要在事务执行前创建一个snapshot，在后面WT事务隔离章节详细介绍。

### 事务执行

事务在执行阶段，如果是读操作，不做任何记录，因为读操作不需要回滚和提交。如果是写操作，WT会对每个写操作做详细的记录。在上面介绍的事务对象(wt\_transaction)中有两个成员，一个是操作operation\_array，一个是redo\_log\_buf。这两个成员是来记录修改操作的详细信息，在operation\_array的数组单元中，包含了一个指向MVCC list对应修改版本值的指针。那么详细的更新操作流程如下：

1. 创建一个mvcc list中的值单元对象(update)
2. 根据事务对象的transaction id和事务状态判断是否为本次事务创建了写的事务ID，如果没有，为本次事务分配一个事务ID，并将事务状态设成HAS\_TXN\_ID状态。
3. 将本次事务的ID设置到update单元中作为mvcc版本号。
4. 创建一个operation对象，并将这个对象的值指针指向update,并将这个operation加入到本次事务对象的operation\_array
5. 将update单元加入到mvcc list的链表头上。
6. 写入一条redo log到本次事务对象的redo\_log\_buf当中。

示意图如下：



图4

### 事务提交

WT引擎对事务的提交过程比较简单，先将要提交的事务对象中的redo\_log\_buf中的数据写入到redo log file(重做日志文件)中，并将redo log file持久化到磁盘上。清除提交事务对象的snapshot object,再将提交的事务对象中的transaction\_id设置为WT\_TNX\_NONE，保证其他事务在创建系统事务snapshot时本次事务的状态是已提交的状态。

### 事务回滚

WT引擎对事务的回滚过程也比较简单，先遍历整个operation\_array，对每个数组单元对应update的事务id设置以为一个WT\_TXN\_ABORTED（= uint64\_max），标示mvcc 对应的修改单元值被回滚，在其他读事务进行mvcc读操作的时候，跳过这个放弃的值即可。整个过程是一个无锁操作，高效、简洁。

## WT的事务隔离

传统的数据库事务隔离分为:Read-Uncommited(未提交读)、Read-Commited(提交读)、Repeatable-Read(可重复读)和Serializable(串行化)，WT引擎并没有按照传统的事务隔离实现这四个等级，而是基于snapshot的特点实现了自己的Read-Uncommited、Read-Commited和一种叫做snapshot-Isolation(快照隔离)的事务隔离方式。在WT中不管是选用的是那种事务隔离方式，它都是基于系统中执行事务的快照截屏来实现的。那来看看WT是怎么实现上面三种方式的。



图5

#### Read-uncommited

Read-Uncommited(未提交读)隔离方式的事务在读取数据时总是读取到系统中最新的修改，哪怕是这个修改事务还没有提交一样读取，这其实就是一种脏读。WT引擎在实现这个隔方式时，就是将事务对象中的snap\_object.snap\_array置为空即可，那么在读取MVCC list中的版本值时，总是读取到MVCC list链表头上的第一个版本数据。举例说明，在图5中，如果T0/T3/T5的事务隔离级别设置成Read-uncommited的话，那么T1/T3/T5在T5时刻之后读取系统的值时，读取到的都是14。一般数据库不会设置成这种隔离方式，它违反了事务的ACID特性。可能在一些注重性能且对脏读不敏感的场景会采用，例如网页cache。

#### Read-Commited

Read-Commited(提交读)隔离方式的事务在读取数据时总是读取到系统中最新提交的数据修改，这个修改事务一定是提交状态。这种隔离级别可能在一个长事务多次读取一个值的时候前后读到的值可能不一样，这就是经常提到的“幻象读”。**在WT引擎实现read-commited隔离方式就是事务在执行每个操作前都对系统中的事务做一次截屏，然后在这个截屏上做读写**。还是来看图5，T5事务在T4事务提交之前它进行读取前做事务snapshot={

snap\_min=T2,

snap\_max=T4,

snap\_array={T2,T4},

}; 在读取MVCC list时，12和14修个对应的事务T2/T4都出现在snap\_array中，只能再向前读取11，11是T1的修改，而且T1 没有出现在snap\_array，说明T1已经提交，那么就返回11这个值给T5。

之后事务T2提交，T5在它提交之后再次读取这个值，会再做一次snapshot={

snap\_min=T4,

snap\_max=T4,

snap\_array={T4},

}，这时在读取MVCC list中的版本时，就会读取到最新的提交修改12。

#### Snapshot- Isolation

Snapshot-Isolation（快照隔离）隔离方式是读事务开始时看到的最后提交的值版本修改，这个值在整个读事务执行过程只会看到这个版本，不管这个值在这个读事务执行过程被其他事务修改了几次，这种隔离方式不会出现“幻象读”。WT在实现这个隔离方式很简单，在事务开始时对系统中正在执行的事务做一个snapshot,这个snapshot一直沿用到事务提交或者回滚。还是来看图5，T5事务在开始时，对系统中的执行的写事务做snapshot={

snap\_min=T2,

snap\_max=T4,

snap\_array={T2,T4}

}，那么在他读取值时读取到的是11。即使是T2完成了提交，但T5的snapshot执行过程不会更新，T5读取到的依然是11。

这种隔离方式的写比较特殊，就是如果有对事务看不见的数据修改，那么本事务尝试修改这个数据时会失败回滚，这样做的目的是防止忽略不可见的数据修改。

通过上面对三种事务隔离方式的分析，WT并没有使用传统的事务独占锁和共享访问锁来保证事务隔离，而是通过对系统中写事务的snapshot截屏来实现。这样做的目的是在保证事务隔离的情况下又能提高系统事务并发的能力。

## WT事务日志

通过上面的分析可以知道WT在事务的修改都是在内存中完成的，事务提交时也不会将修改的MVCC list当中的数据刷入磁盘，那么WT是怎么保证事务提交的结果永久保存呢？WT引擎在保证事务的持久可靠问题上是通过redo log（重做操作日志）的方式来实现的，在本文的事务执行和事务提交阶段都有提到写操作日志。WT的操作日志是一种基于K/V操作的逻辑日志，它的日志不是基于btree page的物理日志。说的通俗点就是将修改数据的动作记录下来，例如：插入一个key = 10,value= 20的动作记录在成:

{

Operation = insert,(动作)

Key = 10,

Value = 20

};

将动作记录的数据以append追加的方式写入到wt\_transaction对象中redo\_log\_buf中，等到事务提交时将这个redo\_log\_buf中的数据已同步写入的方式写入到WT的重做日志的磁盘文件中。如果数据库程序发生异常或者崩溃，可以通过上一个checkpoint(检查点)位置重演磁盘上这个磁盘文件来恢复已经提交的事务来保证事务的持久性。根据上面的描述，有几个问题需要搞清楚：

1. 操作日志格式怎么设计?
2. 在事务并发提交时，各个事务的日志是怎么写入磁盘的？
3. 日志是怎么重演的？它和checkpoint的关系是怎样的？

在分析这三个问题前先来看WT是怎么管理重做日志文件的，在WT引擎中定义一个叫做LSN序号结构，操作日志对象是通过LSN来确定存储的位置的，LSN就是Log Sequence Number(日志序列号)，它在WT的定义是文件序号加文件偏移，wt\_lsn{

file: 文件序号，指定是在哪个日志文件中

offset: 文件内偏移位置，指定日志对象文件内的存储文开始位置

}。WT就是通过这个LSN来管理重做日志文件的。

### 日志格式

WT引擎的操作日志对象（以下简称为logrec）对应的是提交的事务，事务的每个操作被记录成一个logop对象，一个logrec包含多个logop，logrec是一个通过精密序列化事务操作动作和参数得到的一个二进制buffer，这个buffer的数据是通过事务和操作类型来确定其格式的。

WT中的日志分为4类：分别是建立checkpoint的操作日志(LOGREC\_CHECKPOINT)、普通事务操作日志(LOGREC\_COMMIT)、btree page同步刷盘的操作日志(LOGREC\_FILE\_SYNC)和提供给引擎外部使用的日志(LOGREC\_MESSAGE)。这里介绍和执行事务密切先关的LOGREC\_COMMIT，这类日志里面由根据K/V的操作方式分为：LOG\_PUT(增加或者修改K/V操作)、LOG\_REMOVE(单KEY删除操作)和范围删除日志,这几种操作都会记录操作时的key，根据操作方式填写不同的其他参数，例如：update更新操作，就需要将value填上。除此之外，日志对象还会携带btree的索引文件ID、提交事务的ID等，整个logrec和logop的关系结构图如下：



图6

对于上图中的logrec header中的为什么会出现两个长度字段：logrec磁盘上的空间长度和在内存中的长度，因为logrec在刷入磁盘之前会进行空间压缩，那么磁盘上的长度和内存中的长度就不一样了。压缩是根据系统配置可选的。

### WAL与日志写并发

WT引擎在采用WAL（Write-Ahead Log）方式写入日志，WAL通俗点说就是说在事务所有修改提交前需要将其对应的操作日志写入磁盘文件。在事务执行的介绍小节中我们介绍是在什么时候写日志的，这里我们来分析事务日志是怎么写入到磁盘上的，整个写入过程大致分为下面几个阶段：

1. 事务在执行第一个写操作时，先会在事务对象（wt\_transaction）中的redo\_log\_buf的缓冲区上创建一个logrec对象，并将logrec中的事务类型设置成LOGREC\_COMMIT。
2. 然后在事务执行的每个写操作前生成一个logop对象，并加入到事务对应的logrec中。
3. 在事务提交时，把logrec对应的内容整体写入到一个全局log对象的slot buffer中并等待写完成信号。
4. Slot buffer会根据并发情况合并同时发生的提交事务的logrec，然后将合并的日志内容同步刷入磁盘（sync file），最后告诉这个slot buffer对应所有的事务提交刷盘完成。
5. 提交事务的日志完成，事务的执行结果也完成了持久化。

整个过程的示意图如下：



图7

WT为了减少日志刷盘造成写IO，对日志罗刷盘操作做了大量的优化，实现一种类似MySQL组提交的刷盘方式。这种刷盘方式会将同时发生提交的事务日志合并到一个slot buffer中，先完成合并的事务线程会同步等待一个完成刷盘信号，最后完成日志数据合并的事务线程将slot buffer中的所有日志数据sync到磁盘上并通知在这个slot buffer中等待其他事务线程刷盘完成。并发事务的logrec合并到slot buffer中的过程是一个完全无锁的过程，这减少了必要的CPU竞争和操作系统上下文切换。

为了这个无锁设计WT在全局的log管理中定义了一个acitve\_ready\_slot和一个slot\_pool数组结构，大致如下定义：

wt\_log{

. . .

active\_slot: 准备就绪且可以作为合并logrec的slot buffer对象

slot\_pool: 系统所有slot buffer对象数组，包括：正在合并的、准备合并和闲置的slot buffer。

}

slot buffer对象是一个动态二进制数组，可以根据需要进行扩大。定义如下:

wt\_log\_slot{

. . .

state: 当前slot的状态,ready/done/written/free这几个状态

buf: 缓存合并logrec的临时缓冲区

group\_size: 需要提交的数据长度

slot\_start\_offset: 合并的logrec存入log file中的偏移位置

. . .

}

通过一个例子来说明这个无锁过程，假如在系统中slot\_pool中的slot个数为16，设置的slot buffer大小为4KB,当前log管理器中的active\_slot的slot\_start\_offset=0,有4个事务（T1、T2、T3、T4）同时发生提交，他们对应的日志对象分别是logrec1、logrec2、logrec3和logrec4。

Logrec1 size = 1KB, logrec2 szie =2KB, logrec3 size =2KB, logrec4 size =5KB。他们合并和写入的过程如下：

1. T1事务在提交时，先会从全局的log对象中的active\_slot发起一次JION操作，JION过程就是向active\_slot申请自己的合并位置和空间，logrec1\_size + slot\_start\_offset < slot\_size并且slot处于ready状态，那T1事务的合并位置就是active\_slot[0, 1KB],slot\_group\_size = 1KB
2. 这是T2同时发生提交也要合并logrec,也重复第1部JION操作，那么它申请到的位置就是active\_slot [1KB, 3KB], slot\_group\_size = 3KB。
3. 在T1事务JION完成后，它会判断自己是第一个JION这个active\_slot的事务，判断条件就是返回的写入位置slot\_offset=0。如果是第一个它立即会将active\_slot的状态从ready状态置为done状态，并未后续的事务从slot\_pool中获取一个空闲的active\_slot\_new来顶替自己合并数据的工作。
4. 与此同时T2事务JION完成之后，它也是进行这个过程的判断，T2发现自己不是第一个，那么它将会等待T1将active\_slot置为done.
5. T1和T2都获取到了自己在active\_slot中的写入位置，active\_slot的状态置为done时，T1和T2分别将自己的logrec写入到对应buffer位置。加入在这里T1比T2先将数据写入完成。那么T1就会等待一个slot\_buffer完全刷入磁盘的信号，而T2写入完成后会将slot\_buffer中的数据写入log文件，并对log文件做sync刷入磁盘的操作，最高发送信号告诉T1同步刷盘完成，T1和T2各自返回，事务提交过程的日志刷盘操作完成。

那这里有几种其他的情况，假如在第2步运行的完成后，T3也进行JION操作，这个时候

slot\_size(4KB) < slot\_group\_size（3KB）+ logrec\_size(2KB).那么T3不JION当时的active\_slot，而是自旋等待active\_slot\_new顶替active\_slot后再JION到active\_slot\_new。

如果在第2步时，T4也提交，因为logrec4(5KB) > slot\_size(4KB),那么T4就不会进行JION操作，而是直接将自己的logrec数据写入log文件，并做sync刷盘返回。在返回前因为发现有logrec4大小的日志数据无法合并，全局log对象会试图将slot buffer的大小放大两倍，这样做的目的是尽量让下面的事务提交日志能进行slot合并写。

**WT引擎之所以引入slot日志合并写的原因就是为了减少磁盘的I/O访问，通过无锁的操作，减少全局日志缓冲区的竞争**。

### 事务恢复

从上面关于事务日志和MVCC list相关描述我们知道，事务的redo log主要是防止内存中已经提交的事务修改丢失，但如果所有的修改都存在内存中，随着时间和写入的数据越来越多，内存就会不够用，这个时候就需要将内存中的修改数据写入到磁盘上，一般在WT中是将整个BTREE上的page做一次checkpoint并写入磁盘。WT中的checkpoint是一个append方式管理的，也就是说WT会保存多个checkpoint版本。不管从哪个版本的checkpoint开始度可以通过重演redo log来恢复内存中已提交的事务修改。整个重演过程就是就是简单的对logrec中各个操作的执行。这里值得提一下的是因为WT保存多个版本的checkpoint,那么它会将checkpoint做为一种元数据写入到元数据表中，元数据表也会有自己的checkpoint和redo log，但是保存元数据表的checkpoint是保存在WiredTiger.wt文件中，系统重演普通表的提交事务之前，先会重演元数据事务提交修改。后面单独用一个篇幅来说明btree、checkpoint和元数据表的关系和实现。

WT的redo log是通过配置开启或者关闭的，MongoDB并没有使用WT的redo log来保证事务修改不丢，而是采用了WT的checkpoint和MongoDB复制集的功能结合来保证数据的完成性的。大致的细节是如果某个mongoDB实例宕机了，重启后通过MongoDB的复制协议将自己最新checkpoint后面的修改从其他的MongoDB实例复制过来。

## 后记

虽然WT实现了多操作事务模型，然而MongoDB并没有提供事务，这或许和MongoDB本身的架构和产品定位有关系。但是MongoDB利用了WT的短事务的隔离性实现了文档级行锁，对MongoDB来说这是大大的进步。

可以说WT在事务的实现上另辟蹊径，整个事务系统的实现没有用繁杂的事务锁，而是使用snapshot和MVCC这两个技术轻松的而实现了事务的ACID，这种实现也大大提高了事务执行的并发性。除此之外，WT在各个事务模块的实现多采用无锁并发，充分利用CPU的多核能力来减少资源竞争和I/O操作，可以说WT在实现上是有很大创新的。通过对WiredTiger的源码分析和测试，也让我获益良多，不仅仅了解了数据库存储引擎的最新技术，也对CPU和内存相关的并发编程有了新的理解，很多的设计模式和并发程序架构可以直接借鉴到现实中的项目和产品中。

后续的工作是继续对Wiredtiger做更深入的分析、研究和测试，并把这些工作的心得体会分享出来，让更多的工程师和开发者了解这个优秀的存储引擎。