WiredTiger实现：数据页

在《WiredTiger实现：事务》一文中提到了WiredTiger（简称为WT）是为了适应现代CPU、内存和磁盘的特性而设计的存储引擎，它的特点就是充分利用CPU的速度和内存的容量来弥补磁盘访问速度不足。在介绍WT的数据组织前，先来看看传统数据库引擎的数据组织方式，一般存储引擎都是采用btree或者lsm tree来实现索引，但是索引的最小单位不是K/V记录对象，而是**数据页**，数据页的组织关系实现就是存储引擎的数据组织方式。传统数据库引擎大都是设计一个磁盘和内存完全一样的数据组织方式,这个结构是固定的空间大小（innodb的page是16KB）,访问它必须遵守严格的The FIX Rules规则：

 修改一个page需要获得该页的x-latch lock

访问一个page是需要获得该页的s-latch lock或者x-latch lock。

持有该page的latch直到修改或者访问该页的操作完成,latch unlock。

WT没有像传统的数据库引擎那样设计一套内存和磁盘page完全一致的数据组织方式，而是针对磁盘和CPU、内存三者之间特点设计了一套独特的数据组织方式，这种数据组织结构分为两部分：

in-memory page: 内存中的数据页(page)

disk extent: 基于磁盘文件的偏移量的范围存储

WT内存中的page是一个松散自由的数据结构，而磁盘上的extent只是一个变长的序列化后的数据块，这样做的目的（**设计目标**）有以下几点：

1.内存中的page松散结构可以不受磁盘存储方式的限制和The FIX Rules规则的影响，可以自由的构建page的无锁多核并发结构，充分发挥CPU多核的能力。

2.可以自由的在内存page和磁盘extent之间实现数据的压缩，提高磁盘的存储效率和减少I/O访问时间。

关于WT的压缩效率见下图

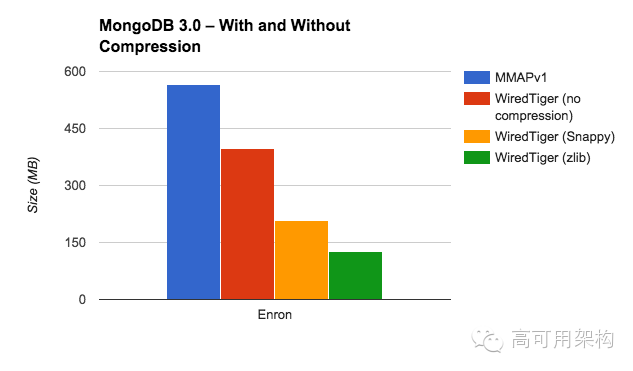


图1

page在WT引擎中起着承上启下的作用，上通索引、事务和LRU cache,下达文件、高速缓冲和磁盘I/O。要了解整个WT引擎的原理，首先要理解它的数据组织方式。**在本文中先通过一组测试样例来验证WT数据组织的设计目标，再分析WT数据组织的原理和实现**。

# WT测试样例

先来看看WT这种数据组织的相关测试，我们在一个普通的开发机器上对WT的原生API接口进行测试，测试环境如下：

CPU: i7-4710MQ CPU @ 2.50GHz，8核

内存：4G

硬盘：1TB SATA,5400转

WT数据库配置：

Cache size: 1GB, page max size:64KB, OS page cache:1GB

Key:一个从零开始自增长的整数

Value:一个长度介于100 ~ 200的随机字符串。

测试流程：

测试程序先新建一张表，用16条线程并发向表中插入指定数量（以百万为单位）的K/V对，在插入完成后做一次checkpoint让插入的数据写入磁盘，统计这个表在磁盘上的大小和这个过程的耗时。再用16条线程在表中随机查询20000个不同K/V并统计查询的耗时,通过耗时可以计算出insert和query的TPS。统计的这三个参数分别就是磁盘空间占用、写性能和读性能。我们分别进行不压缩extent的测试和进行ZIP压缩extent的测试。

## 磁盘空间统计对比

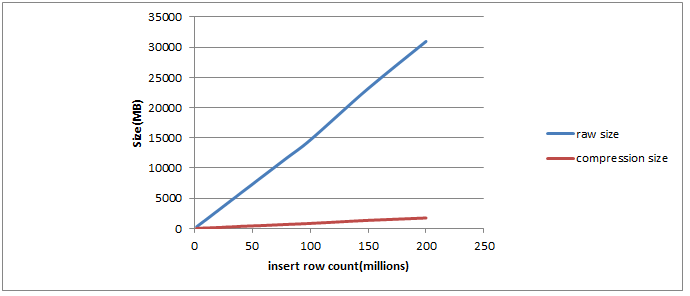


图2

从上图可以看出2亿条记录没有压缩的磁盘空间30GB，而压缩后的磁盘空间2GB左右。

## 写性能对比

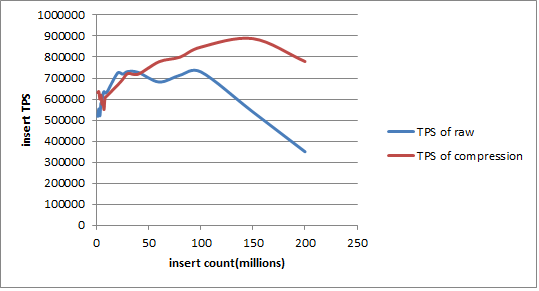


图3

从上图可以看出，写入的数据在1亿条（14GB）压缩(518K TPS)和没压缩(631k TPS)的写性能相当，但随着数据量增大，大量的数据在内存和磁盘间swap，选择extent压缩的写性能(790K TPS)要好于没压缩的写性能（351K TPS）。

## 读性能对比

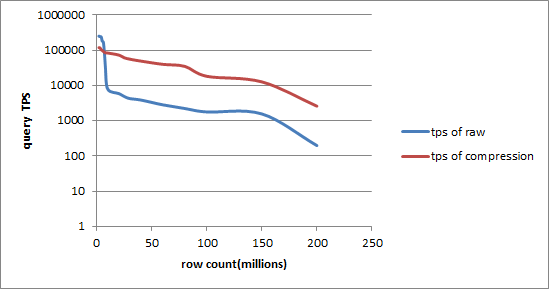


图4

从上图可以得出在表空间文件上的数据大小没有超过WT的OS\_page\_cache限制时，**无压缩的读性能**要好于**有压缩的读性能**，这是因为extent缓冲在操作系统高速缓冲区中，当发生访问时从高速缓冲区直接获取无压缩extent转换成内存中的page，只要一次内存中的结构重建即可，而压缩的extent需要进行解压缩后再重建，所以没有压缩的读性能更好。 如果文件数据大小超出OS\_page\_cache限制后，extent数据要从磁盘上读取，而有压缩的extent占用的空间比较小，从磁盘上读取的I/O访问时间是小于无压缩读取的时间的，这时有压缩的读性能要好于无压缩的读性能。（后面单独用一个篇幅来分析WT的磁盘I/O相关的实现）

除了压缩优化了数据的读写，WT内存中的无锁page结构也使得读写操作具有更好的并发性，才使得WT在非常普通机器上有如此好的表现。

# 内存中的数据页

通过 上面的介绍已经对WiredTiger的数据组织基本的了解，WT数据组织方式就是in-memory page加block-extent。先来对它内存部分的in-memory page（内存数据页，简称为page）来做分析。WT引擎中的page分以下几类：

row internal page: 行存储b-tree的索引页

row leaf page: 行存储b-tree的数据页

column internal page: 列存储b-tree的索引页

column fix leaf page: 列存储b-tree的定长数据页

column var leaf page: 列存储b-tree的变长数据页

因为mongoDB主要是使用行存储，所以在这里主要分析row leaf page这个结构和原理，结构图如下:



图5

上图中主要有以下几个单元：

wt\_page: 内存中的page结构对象，page的访问入口。

wt\_modify: page的修改状态信息，主要包含脏页标示、当前更新事务ID和page的insert lock等。

row\_array: 磁盘上原有row的位置索引数组，主要用于页内检索。

row\_insert\_array: 一个增加kv(insert k/v)的跳表对象数组

row\_update\_array:一个在row基础上做更新(update k/v)的value mvcc list对象素组。

page\_disk: 从磁盘上读取到的extent数据缓冲区，包含一个page\_header和一个数据存储缓冲区(disk data)

page disk data：存储在磁盘上的page k/v cell行集合数据

罗列了这么多结构，他们之间是通过什么方式来关联的呢？我们通过一个实例来说明，假如一个page存储了一个[0, 100]的key范围，磁盘上原来存储的行key=2, 10 ,20, 30 , 50, 80, 90,他们的值分别是value = 102, 110, 120, 130, 150, 180, 190。在page数据从磁盘读到内存后，分别对key=20的value进行了两次修改，两次修改的值是分别402,502。对key = 20 ,50的value做了一次修改，修改后的value = 122, 155，后有分配insert了新的key = 3,5, 41, 99，value = 203,205,241,299。那么在内存中的page就是如下图组织数据的:



图6

row\_array的长度是根据page从磁盘中读取出来的行数确定的，每个数组单元(wt\_row)存储的是这个kv row在page\_disk\_data缓冲区偏移的位置和编码方式（这个位置和编码方式在WT上定义成一个wt\_cell对象，在后面的K/V cell章节来分析），通过这个信息偏移位置信息就可以访问到这一样在disk\_data缓冲区中的K/V内容值。

每一个wt\_row对象在row\_update\_array数组中对应一个mvcc list对象，mvcc\_list与wt\_row是一一对应的，mvcc list当中存储对wt\_row修改的值，修改的值包括值更新和值删除,是一个无锁单向链表。

相邻的两wt\_row之间可能不是连续的，他们之间可以插入新的单元，例如row1(key = 2)和row2(key=10)可以插入3 和5，这两个row之间需要有一个排序的数据结构（WT用skiplist数据结构）来存储插入的K/V，就需要一个skiplist对象数组page\_insert\_array与row array对应。这里需要说明的是图6当中红色框当中的skiplist8，它是用于存储row1(key=2)范围之前的insert数据，图6中如果有key =1的数据insert,那么这个数据会新增到skiplist8当中。那么图6中row与insert skiplist的对应关系就是：

**row1之前的范围对应insert是skiplist8**

row1和row2之间对应的insert是skiplist1

row2和row3之间对应的insert是skiplist3

…

**row7之后的范围对应的insert是skiplist7**

## wt\_row结构

从上面对应page的整体分析来看在WT的page中，row对象是整个row leaf page的关键结构，row其实就是K/V位置的描述值（kv\_pos），它的定义:

wt\_row{

uint64 kv\_pos; //这个值是在page读入内存时根据KV存储在page\_disk的位置确定的

}

这个kv\_pos的对应的数据有对应的K/V存储位置信息，kv\_pos表示这个K/V的值有三种方式，结构如下图：



图7

上图中的wt\_row对应的空间上都有2个bit的flag，这个flag的值：

CELL\_FLAG: 0x01,表示这个row用cell对象来k/v标示存储位置的,因为key和value的值有可能很大，一个page存储不下，这个时候引入cell只是存储这些超长值对应的overflow page的索引值（extent adress）。

K\_FLAG: 0x02,表示这个row只标示了k/v中key的存储位置，但是value比较大，value是CELL\_FLAG方式标示位置存储的，因为value的CELL是紧跟在key存储的位置后面，找到key就能读取value的cell并找到value。

KV\_FLAG: 0x03,表示这个row同时标示了k/v中key和value的存储位置。

## cell结构

cell是一个值key或者value信息被序列化后的数据块，cell在磁盘上和在内存中内容是一致的，它是根据值（key或者value）内容、长度、值类型序列化构建的。在内存中cell是存储在page\_disk缓冲区中，在磁盘上是存储在extent body上，在读取cell的时候需要根据cell数据内容进行发序列化得到一个cell\_unpack内存结构对象，让后再根据这个cell\_unpack对象中的内容来读取这个值（key或者value）。以下是它们之间的结构关系：



图8

那么在row对象中的CELL\_FLAG row、K\_FLAG row和KV\_FLAG row对应的值是怎么产生的呢？其实是在page的数据从磁盘读到内存中时，先会对整个page\_disk按照cell为单位转化成cell\_unpack,并根据cell\_unpack中的信息构造row的这三种格式，这样做的目的是**让能生成K\_FLAG /KV\_FLAG的row在每次被访问的时候不需要去做这个过程的转化，加快访问速度。**

## 内存中的修改

内存中的row对象主要是为了帮助page数据从磁盘上载入到内存中后建立查询索引，而page数据被载入内存后除了查询读以外，还会对其进行修改行为（增删改），对于修改行为WT并没有在row内存结构上进行操作，而是设计两个结构，一个是针对insert操作的insert\_skiplist，一个针对删改操作的mvcc list。关于这两个对象结构与row之间的关系在图6中有过描述。在这里重点来分析它们的内部构造和运行原理。

### 跳表(skiplist)

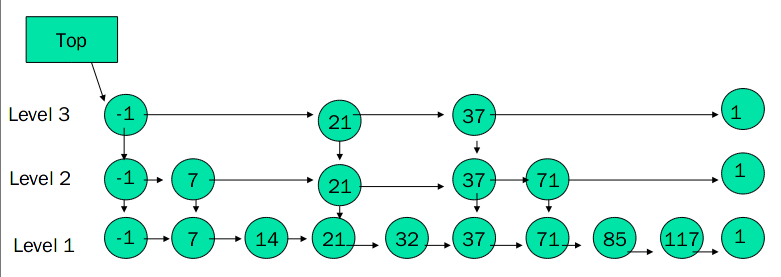
从前面的介绍知道page在内存中存储新增的k/v时采用的是skiplist数据结构，在WT中不仅仅这个地方使用了skiplist，在其他需要快速查询和增删的地方基本上都使用了skiplist，了解skiplist的原理有利于理解WT的实现。skiplist其实是个多层链表，层级越高越稀疏，最底层是个普通的链表。

图9

skiplist原理戳这里<https://en.wikipedia.org/wiki/Skip_list>。

skiplist实现戳这里<https://github.com/yuanrongxi/wb-skiplist>。

### 新增的K/V结构

WT中insert\_skiplist实现时是结合了它本身的k/v内存结构来实现的，WT基于skiplist定义了一个wt\_insert的k/v跳表单元结构，定义如下:

wt\_insert{

key\_offset: //存储key的缓冲区偏移

key\_size: //key的长度

value: //存储值的mvcc list的头单元,一个wt\_update结构对象

next[]: //skiplist的各层的下一个单元指针

key\_data[]: //存储key的缓冲区

}

insert\_skiplist的结构示意图：



图10

WT为什么选用skiplist来作为新增记录的数据结构，主要是几个方面的考虑：

1. skiplist实现起来简单，而且可以根据使用场景因地制宜的和K/V在内存做比较结合。
2. skiplist复杂度为O(log n)，可以在内存中管理大量k/v的增删改查操作，而且skiplist在内存中按照key大小排序的，可以进行范围查找。
3. skiplist在增删操作时可以不排斥读操作，**也就是说一个线程在读skiplist时不需要检查skiplist的write\_lock,相当于无锁读,这实现增加了skiplist的读并发**。

### 修改的value

WT引擎在insert一个k/v时，key值是存储在wt\_insert中，那么它的value存储在什么地方？在wt\_insert结构中有一个wt\_update类型的value说明，这个结构其实就是来存储内存中各个修改版本的value值的链表对象，也就是提到的MVCC list链表。在图6中也有提到row更新的时候，会向row\_update\_array对应的mvcc list当中加入更新的值单元(wt\_update)，这个结构的定义如下:

wt\_update{

txnid: //产生修改的事务ID

next ptr: //链表的下一个wt\_update单元指针

size: //value值长度

value[]: //存储value的缓冲区

}

**size = 0，表示这个是一个删除k/v的修改。**

****

图11（mvcc list）

整个链表在平常情况下只会进行append操作，而且每次append都是在链表头的位置，这样做的目的是为了整个链表的无锁读写操作。这里涉及无锁读好理解，只要做到无锁append就可以做到无锁读。mvcc list无锁append采用的是CPU的CAS操作来完成，大致的步骤如下：

1. 先将需要append的单元(new\_upd)的next指针指向list header对应的单元(header\_upd)，并记录list\_header指针。
2. 使用原子操作CAS\_SWAP将list header设置成为new\_upd,CAS\_SET设置期间如果没有其他线程先以它完成设置，那么本次append就完成。如果有其他线程先以它设置，那么本次设置失败，进入第3步。
3. 用memory barrier读取list\_header的指针，重复第1步。

**第2步的判断是否有其他线程先以自己设置list\_header的依据就是CAS\_SWAP时list\_header的值不是自己读取到值。关于这个过程更多的细节可以去了解GCC编译器的\_\_sync\_val\_compare\_and\_swap函数功能和实现**

## overflow page

WT支持是支持超大的K/V,key和 value的值最大可以到4GB(**其实不到4G，大概是4GB-1KB,因为除了数据外还需要存储page头信息**)。WT通过定义一种叫做overflow page来存储超出leaf page最大存储范围的超大k/v。超大的k/v在insert到leaf page还是存储在insert\_skiplist当中，只有当这个leaf page进行存盘的时候，WT会对超出page允许的最大空间的k/v值用单独的overflow page来存储，overflow page在磁盘文件中有自己单独的extent。那么什么时候在内存中会出现overflow page呢？在用于overflow page的leaf page从磁盘上读入内存中时会构建对应的overflow page内存对象。overflow page本身的结构很简单，就是一个page\_header和一个page\_disk缓冲区。leaf page与overflow page之间通过row cell信息来关联，cell里面存有这个overflow page的extent address信息。为了对overflow page的快速访问，WT定义了一个的skiplist（**extent address与overflow page内存对象映射关系**）来缓存内存中的overflow page内存对象，对overflow page的读流程如下：

1. 先cell\_unpack对应的cell，获取到overflow page的extent addr
2. 使用extent address在overflow page缓存skiplist查找overflow page是否已经读入内存，已经读入内存，返回对应的overflow page读取者,如果没有进入第3步.
3. 根据extent address 信息从磁盘文件中将overflow page的信息读入内存，并构建一个overflow page加入到skiplist当中。最后返回读入的overflow page对象给读取者。

这里提到的extent address参考下面的**extent addresss结构**章节

## page的页内检索

WT实现松散的内存page结构为的就是能快速检索和修改，也使得数据在内存中的组织方式更加自由。不管是读还是修改，需要依赖page的页内检索，在读取或者修改某个k/v值前需要根据对应的key在page内部做一次检索来定位k/v的位置，而整个页内检索的核心参考轴是通row\_array这个数组做二分查找来定位的。在这里还是以**图6**来进行说明，假设需要在图6中查找key=41的值，步骤如下：

1. 先通过二分法在row\_array定位到存储key=41的对象row4
2. 定位到row4后先匹配row key 与检索的key是否匹配，如果匹配，在row4对应的mvcc list（upd4）中读取可以访问的值。如果不匹配，在其对应的insert\_skiplist 进行查找
3. 用key=41在跳表skiplist4进行查找，定位到value = 241，返回。

因为都是row\_array/insert\_array/update\_array数组一一对应的查找，而且这些数组的在发生修改时也不会发生改变，所以不需要对其进行锁保护,insert\_skiplist和mvcc list都是支持修改时无锁读取的（这个在分析这两个结构时已经说明过），所以说整个检索过程是**无锁的**。

如果是增删改(insert/delete/update)操作，也是先用检索过程找到对应修改的位置，再进行对应修改。如果是insert，或获取wt\_modify中的page\_lock来串行化insert操作，如果是对值进行update/delete，只是在mvcc list无锁增加一个修改后的值即可（**这个过程在上面已经分析过**）。

# disk extent结构

page在磁盘上文件上对应的结构叫做extent,其实它就是磁盘文件上的一块区域。在WT引擎中，每一个索引对应一个文件，文件中按照page写入的大小和当前文件被使用的空间来确定写入的位置和写入的长度，写入的位置（offset）和写入的长度（size）被命名成extent，并且将extent的位置信息(extent address)记录到一个索引空间中。

extent由三部分构成，他们分别是

page header: 记录当前数据页的状态信息

block header: extent存储的头信息，主要存有数据的chucksum和长度等。

extent data: 存储的数据，是一个cell数据集合.



图12

page header在page的内存对象中，对应的是page\_disk的头信息部分wt\_page\_header,他们的内容是完全一致的。page header包含当前page的记录实例数（entries）、page类型（row leaf page/internal page等），在page数据载入内存时需要用这些数据来构建page内存对象。

Block header中的checksum是extent data的数据checksum，也是extent address中的checksum。用于extent读入内存时做合法性校验。

## extent address

extent address用于索引extent的信息，它作为一个数据条目存在一个特殊的索引extent中（**关于这个特殊的索引extent在后续的磁盘I/O篇来详细分析**）。这里主要分析下它的内部构造和定义，extent address中有三个值：

offset: extent在btree文件中的偏移位置

size: extent的长度

chucksum:extent的checksum, 用page header/block header/extent data整体计算得到的checksum，用于判断extent的合法性。

**这三个值是序列化后作为extent address条目存储的。**

## extent data

extent data是真正存储page数据的地方，它是page中所有k/v cell的集合。Page数据从内存存入磁盘时，会将每个k/v pair用cell\_pack函数转化成一个key cell和一个value cell存入序列化缓冲区中。这个缓冲区的数据写入到extent中就是extent data，存储结构如下图：



图13

# Page的磁盘读写

btree索引管理的最小单元是page,那么从磁盘到内存的读操作和从内存到磁盘的写操作都是以page为单位来读写，在WT引擎中从磁盘读取一个页到内存的操作叫做in-memory,从内存page写入磁盘的操作叫做reconcile。in-memory过程就是将extent从磁盘文件中读取出来转换成内存page ,而reconcile操作就是将内存中的page转换成extent写入到磁盘上，reconcile过程会造成btree page的分裂。读写序列图如下：



图14

## 读过程（in-memory）

page的读过程是磁盘到内存的过程，步骤如下：

1. 根据btree索引上的extent address信息，从btree对应的文件中读取这个extent到内存缓冲区中。
2. 通过读取的extent数据生成checksum,并与extent address中的checksum校验合法性。
3. 根据btree配置的meta信息判断是否开启压缩，如果没有压缩直接到第5步，如果有压缩进行第4步。
4. 根据配置的压缩算法信息获取WT支持的compressor对象，并对extent data做解压缩。
5. 通过wt\_page\_header信息构建内存中的page对象。
6. 通过wt\_page\_header的entries数对整个page\_disk\_data遍历，按照cell的信息逐行构建row\_array、insert\_array和update\_array.

**如果读入的page包含olverflow page**,**overflow page并不会在这个过程中读取到内存中，而是在访问它的时候读取到内存中的，这个过程只会读取overflow page对应的extent address作为row对象的内容。在overflow page一节分析过overflow page的读取过程。**

## 写过程（reconcile）

写过程比较复杂，page的写过程如果page的内存空间过大会对page做split操作，对于超过page容忍的大K/V会生成overflow page。整个写过程步骤如下：

1. 按照row\_array为轴，扫描整个row\_array/upate\_array和insert\_array,将k/v内存中的值生成cell对象，并将其存入一个缓冲区（rec buffer）中。
2. 判断这个缓冲区是否超出了配置的page\_max\_size,如果超过了，进行split操作。
3. 对单个key或者value超过page\_max\_size,生成一个overflow page,并将overflow page对应的extent address生成cell。
4. 重复1 ~ 3步直到所有的k/v cell都写入到rec buffer中。
5. 假如btree配置了数据压缩项，在WT引擎中查找压缩对象，并用这个压缩对象对rec buffer进行数据压缩得到data buffer，如果没有配置压缩跳过这一步(data buffe = rec buffer)。
6. 根据page对象中的wt\_page\_header的信息将它对应的信息写入到data buffer头位置。
7. 根据btree文件的偏移和空间状态产生一个extent，计算data buffer的checksum.并将data buffer填充到extent data当中。
8. 根据extent的信息填充block\_header,将整个externt写入到btree的文件当中并返回extent address。

## page压缩

通过page的读写过程分析知道这两个过程如果配置了压缩，就需要调用压缩解压缩操作。WT实现压缩和解压缩是通过一个外部自定义的插件对象来实现的，下面是这个对象的接口定义

\_\_wt\_compressor{

compress\_func(); //压缩接口函数

pre\_size\_func(); //预计算压缩后数据长度接口函数

decompress\_func(); //解压缩接口函数

terminate\_func(); //销毁压缩对象,有点像析构函数

};

WT提供LZO/ZIP/snappy这几个压缩算法，也支持自定义压缩算法，只要按照上面的对象接口实现即可。要让WT支持压缩算法，需要在WT启动时通过wiredtiger\_open加载压缩算法模块，例子如下：

wiredtiger\_open(db\_path, NULL, “extensions=[/usr/local/lib/libwiredtiger\_zlib.so]”,&connection);

然后在WT引擎创建表时可以配置压缩启用压缩配置即可，例如：

session->create(session, “mytable”, “block\_compressor=zlib”);

WT的插件式压缩非常灵活和方便，MongoDB默认支持ZIP和snappy压缩，在MongoDB创建collection时是可以进行选择压缩算法。

# 后记

WT引擎采用内存和磁盘上不同的结构来实现page的数据组织，目标还是让内存中的page结构更加方便在CPU多核下的增删查改的并发操作,精简磁盘上的extent结构，让磁盘上的表空间管理不受内存结构的影响。基于extent(偏移 + 数据长度)的方式也让WT引擎的数据文件结构更简便，可以轻松实现数据压缩。

然而这种内存和磁盘上结构不一致的设计也有不好的地方，数据从磁盘到内存或者从内存到磁盘需要多次拷贝，中间还需要额外的内存作为这两种结构的临时缓冲区，在物理内存不足的情况下会让swap问题雪上加霜,性能会急剧下降，这个在测试样例里面有体现。所以要让WT引擎发挥好的性能，尽让配备更大物理内存给它使用。

MongoDB 3.2版本已经将WiredTiger作为默认引擎，我们在使用MongoDB时一般不会对WiredTiger做配置，这可能会有些业务场景发挥不出WT的优势。MongoDB在创建collection时可以对WiredTiger的表做配置，格式如下:

db.createCollection("<collectionName>", {storageEngine: {

wiredtiger: {configString:"<option>=<setting>,<option>=<setting>"}}});

不同的业务场景是可以配置进行不同的配置。例如：

如果是读多写少的表在创建时我们可以尽量将page size设置的比较小 ，比如16KB,如果表数据量不太大(<2G）,甚至可以不开启压缩。那么createCollection的configString可以这样设置：

"internal\_page\_max=16KB,leaf\_page\_max=16KB,leaf\_value\_max=8KB,os\_cache\_max=1GB"

如果这个读多写少的表数据量比较大，可以为其设置一个压缩算法，例如:

"block\_compressor=zlib, internal\_page\_max=16KB,leaf\_page\_max=16KB,leaf\_value\_max=8KB

如果是写多读少的表，可以将leaf\_page\_max设置到1MB,并开启压缩算法，也可以为其制定操作系统层面page cache大小的os\_cache\_max值，让它不会占用太多的page cache内存，防止影响读操作。

这些MongoDB的配置项都是和WT引擎数据组织相关的配置项，在了解WT 的数据组织方式细节情况下，可以根据具体的业务场景调整collection的表配置属性。

以上是通过分析和测试WiredTiger源码得到的一些认识，有些细节可能会有差错，但大体是这样的工作原理。对WT源码感兴趣的同学戳这里<https://github.com/yuanrongxi/wiredtiger>，我花了些时间对WiredTiger-2.5.3的源码做了分析和注释。

后续工作是对WiredTiger的索引（btree/LSM tree）和磁盘 I/O相关的模块做深入的分析和测试，会不定时的把分析的心得分享出来。