**Chunkserver预读优化**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **编号** | **文档版本** | **修订章节** | **修订原因** | **修订日期** | **修订人** |
| **1** | 1.0 |  | 创建 | 2011.2.20 | 华庭 |
| **2** |  |  |  |  |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |

Cs的第一个版本已经完成，但性能上却没有经过充分的调优，为了增大cs的吞吐量，降低响应时间，下面对cs值得优化的地方做了一些分析，并给出了简要的优化方案。

主要从五个方面对cs进行优化：

# 内存拷贝

由于oceanbase的特点决定了要做到“零拷贝”是很困难的，oceanbase系统的中涉及到序列化，反序列化，压缩和解压缩，这种类型的操作一定会把数据从一个buffer中“拷贝”到另外一个buffer中，这种内存“拷贝”暂时没有好办法优化，但除了这种类型的“拷贝”外，cs中存在真正的内存拷贝（不修改内容的拷贝），将同样的数据从一个buffer拷贝到另外一个buffer，这种拷贝是可以优化的。暂时发现chunkserver中有一处内存拷贝可以优化：

Cs处理request流程如下：

Tbnet收到client发来的request -> cs将request拷贝到packet的inner\_buffer中 -> 处理线程将packet的inner\_buffer反序列化到scan\_parm或者get\_param buffer -> 调用cs\_scan或cs\_get得到数据obscanner -> 将obscanner序列化到处理线程的out\_buffer中 -> 发送响应，将out\_buffer拷贝到packet的inner\_buffer中，并加上record header和tbnet header -> 将packet的inner\_buffer拷贝到tbnet的buffer中 –> tbnet通过单线程组合一些packet后，发送响应给client。

从上面的流程描述中可以看到，将处理线程的out\_buffer拷贝到packet的inner\_buffer是可以优化的，可以直接将packet的inner\_buffer反序列化后，该buffer作为处理线程的out\_buffer，并且packet的inner\_buffer已经预留tbnet header（16字节）和record header（32字节）空间，将packet的inner\_buffer作为out\_buffer交给处理线程使用，这样可以省去一次内存拷贝。

由于内存拷贝暂时不是cs的瓶颈，若海即将对obpacket的内存分配做优化，所以cs在减少内存拷贝方面的优化不是很必要。暂时不考虑对tbnet的内存管理优化。

# 上下文切换

上下文切换是cs的另外一个性能问题，但暂时还没有好的解决办法，tbnet使用一个单独的线程发送和接收数据包，cs创建多个工作线程处理从tbnet收到的数据包，发送数据包时只需要将数据拷贝到tbnet的buffer就完事，当cs总线程数与机器的cpu数量差不多时，线程的上下文切换是不严重的，如果cs的总线程太多，就会导致线程上下文切换增加，特别是tbnet的线程如果在多个cpu上切来切去，是比较影响性能的。

暂时不考虑对tbnet做大修改，但可以绑定tbnet的线程到一个固定的cpu上，配置cs的总线程数不要超过系统cpu个数太多。比如在每日合并时，cs对增加10线程用于合并，每日合并是相当消耗资源的，但系统只有16核，merge server还需要消耗一些资源，所以这个时候应该限制cs中活跃的工作线程的数量，将合并的10个线程绑定在10个不同的cpu上，tbnet会消耗一个cpu，所以还剩下5个cpu可以用（这也是cs在每日合并时吞吐量只有平时1/3的一个原因），使用信号量的方式，限制系统中活跃的线程数量不能超过16，尽量减少上下文切换。

上下文切换的另外一个原因就是用户态和内核态的切换，优化这种上下文切换需要减少系统调用，采用内存预分配的方式减少cs分配内存时调用mmap的次数，使用预读的方式尽量读取大块连续block数据，减少系统调用的次数。下面几节会对内存预分配和预读做详细介绍。

# 内存管理

Cs的内存管理有很多地方值得优化，由于我们的内存池做得不够好，所以这些优化需要cs根据内存的使用情况自己来处理。现在想到的主要有以下的几个方面：

## 内存分配优化

由于cs中的分配内存时调用glibc的分配函数，并禁用了mmap阈值动态调整机制，会导致大于128K的内存分配使用用系统调用mmap向操作系统申请内存，释放时调用munmap，cs中有许多地方在不停的分配释放大块内存（1M或是2M），而这些分配的内存完全可以重用，这样就导致了很多不必要的系统调用，增大了上下文切换次数。更糟糕的是，用mmap分配的内存而不重用内存，增加了缺页异常的次数，在发生缺页异常，操作系统为缺页的分配物理页时，会强制将物理页清0，相当于做了很多次memset，这是很消耗cpu的操作。

所以在cs中需要尽量重用分配的大内存块，能预分配的尽量预分配，减少系统缺页异常，在处理每个request中如果能做到不向操作系统申请内存，直接从cs管理的内存中分配内存，这种方式是最高效的。这里举两个cs内存优化的例子：

1. 从cs处理request流程可以看到，packet每次request中会先分配2M内存存储client发来的param，处理完param后会释放掉这块2M内存，然后在发生响应前，又会分配2M内存存储结果数据，然后这2M内存被tbnet释放。也就是说，每个request需要调用mmap 2M内存两次，munmap 2M内存两次，代价比较高，我们可以用thread mempool来优化，为每个线程预分配10M内存，packet分配的内存都从thread mempool中获取，释放时还回到thread mempool。郁白的实现的thread mempool类可以完成类似的优化，cs也可以用上。
2. Block cache的内存分配时调用memalign函数分配内存，其实也是调用mmap向操作系统申请的，由于没有管理block cache分配的内存，在需要加载新的block又没有空闲的内存块时，会先分配1M的内存块，然后加入该block，在加入block时发现block cache的内存超过阈值，于是从block cache淘汰掉一个内存块（1M），需要free该内存块，也就是调用了munmap系统调用，这个过程也是在反复的调用mmap和munmap系统调用分配释放内存，效率不高。这里可以优化为预分配所有的block cache内存块，当block cache空闲内存块用光时，先淘汰一块已使用的内存块，再把新的block数据加入这个淘汰的内存块，这样就避免了不必要的内存分配释放。

## 预分配优化

我们都知道由于使用静态分配而对程序的功能加上人为限制是一种糟糕的设计。但是还是有许多其它很不错的预分配方案。通常认为，通过系统一次性分配内存要比分开几次分配要好，即使这样做在程序中浪费了某些内存。如果能够确定在程序中会有几项内存使用，在程序启动时预分配就是一个合理的选择。即使不能确定，在开始时为请求句柄预分配可能需要的所有内存也比在每次需要一点的时候才分配要好。通过系统一次性连续分配多项内存还能极大减少错误处理代码。在内存比较紧张时，预分配可能不是一个好的选择，但是除非面对最极端的系统环境，否则预分配都是一个稳赚不赔的选择。

Cs中存在的几项内存使用都是确定的，在cs启动时预分配是一个合理的优化，暂时想到的对cs的预分配优化主要有下面几处：

1. Cs启动时会读取所有sstable文件的trailer到内存中，并建立tablet的索引，其实在这一个过程中可以做更多事情，可以将每个sstable的文件描述符加入file info的cache中。
2. Cs启动时加载所有sstable文件的block index cache，预分配block index cache的内存。
3. Cs启动时预分配好所有的block cache内存。
4. 每个工作线程在处理request的内存是一定的，全部从thread mempool中分配，在cs启动时预分配每个工作线程的mempool，每个工作线程需要的内存主要由下面几部分组成：packet的buffer，存放get\_param或scan\_param的buffer，存放obscanner的buffer，存放解压数据的buffer，所有这些buffer都是按2M进行分配。所以我们可以在cs启动时为工作线程分配好这些buffer，在处理request时不再分配内存，减少操作系统的缺页异常，加快cs处理request的速度。

cs内存管理优化的终极解决方案是设计出适合ob系统的内存池，而不需要每个服务器程序都为工作线程分配私用内存，让工作线程自己去管理内存。所以上面的一些优化是暂时的，当内存池设计好后，cs的内存分配都可以使用内存池。

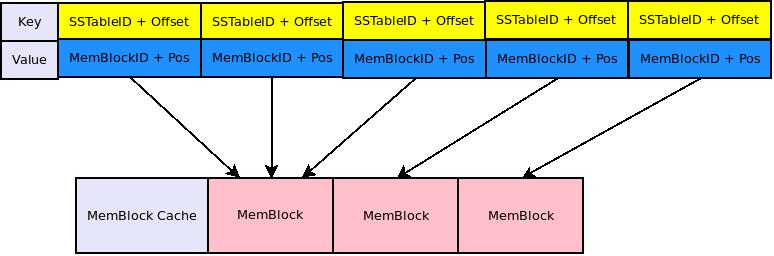
# 锁

Cs中主要有下面几个地方需要锁：

1. 全局的packet队列，每个工作线程都会从该队列中取packet进行处理，每次取一个，packet队列锁在使用ssd可能会有影响，优化的方法之一就是一次从队列中读取多个packet进行处理。暂时不对这个锁做优化，基于磁盘的cs在这个锁上的竞争不是很激烈。如果真要优化，可以使用lock-free技术实现单向链表来作为packet队列。
2. 内存池的全局锁，由于cs预分配了几乎所有需要的内存，在运行时分配的内存很少，所有对内存池的依赖很少，内存池的全局锁影响不大。
3. Tablet image现在是只读的，不需要锁，但如果以后去掉了每日合并，tablet image会动态修改，这时需要tablet image进行加锁，具体的实现还没有确定。
4. Block cache中的锁，现在每次从block cache中读取数据，都需要先锁住整个block cache，这个锁的粒度比较大，在cs使用ssd时，对性能有比较大的影响，郁白已经有了优化block cache的解决方案，下面附上郁白的优化方案。

Cache的性能优化主要针对多线程大并发情况的锁冲突，在现有的设计中使用lru管理cache资源，因此读取操作都要在加锁后查询hashmap并对LRUList进行修改。在性能优化的设计为了避免全局锁而去掉了LRUList，而为每个cache项目保存一个访问计数和最后访问时间，并在hashmap中使用以桶为粒度的锁保护。

Cache采用两层设计，上层为sstable+offset为key，内存块ID+块内偏移为value的hashmap索引层；下层为内存块cache，每个内存块有一个自增的ID，多个block的数据存放在一个内存块中，以内存块作为淘汰的最小单位。如下图所示：



索引层的hashmap使用桶粒度的锁保护，下层内存块cache在初始化的时候申请一个长度为N(最大cache内存块的数量)的定长数组，数据元素用来保存每个已分配内存块的访问计数和最近访问时间。这样在读取cache的时候避免了全局的锁，而只需要使用原子操作更新访问计数和最近访问时间即可，在每次cache需要淘汰的时候，无需加锁遍历上述这个数组选出访问频度最小的M个内存块(使用堆为O(N)\*logM的复杂度)，一次性淘汰这M个内存块。

索引层的定期整理，由于对内存块的淘汰并不影响上层索引，而只会在访问时发现索引指向的某个内块已被淘汰，因此在多次淘汰后索引层会存在很多已经被淘汰的block索引，为了不影响淘汰的效率，使用定时线程检查在淘汰的内存块容量超过256M(可配置)的情况下扫描一次索引层(不修改访问计数和最近访问时间)，删除那些内存块已经被淘汰的block索引。

更进一步的优化：Hash桶的多线程读写也可以使用无锁优化，读取操作不需要加锁，更新操作通过锁来实现互斥，对hash冲突链上的node使用一个allocator进行复用，使用引用计数+桶ID+更新计数的方式实现无锁保护，即每个桶有唯一的标识保存在node中；每次更新桶内容时累加更新计数并把这个计数值保存到本次更新涉及的node中；每次读取操作都修改引用计数。

# IO

Cs的IO优化主要包括网络IO和磁盘IO，由于网络使用tbnet，tbnet使用单线程进行数据收发，对基于磁盘的cs是足够了，对于使用ssd的cs还需要进一步测试，根据测试结果来决定怎么优化使用tbnet。所以这里暂时不考虑对tbnet的优化。对磁盘IO的优化，主要使用aio预读方式。

## 异步io和预读

### 概述

改造现有接口， blockcache提供一个advise接口，用户在get\_block之前预先调用advise接口通知blockcache所有需要读取的block数据，blockcache根据数据范围和cache信息发起一次或多次异步io，由回调函数将异步回调返回的数据放入线程缓存供get\_block接口来获取数据。

主要包括两个功能模块，AIOBufferMgr和AIOEventMgr，其中AIOBufferMgr每个线程一个实例，用来管理异步IO所使用的内存、调度异步IO操作；AIOEventMgr是全局的异步IO处理器，封装了libaio的接口，提供回调函数通知IO事件完成。

### AIOEventMgr设计

AIOEventMgr是对libaio的封装，不同于经典的使用独立线程轮询异步IO完成事件的模式，我们放弃独立的轮询线程，而是在工作线程中获取异步IO的完成事件，通过状态标记，条件变量等线程同步手段实现多个工作线程之间的协同工作。

首先简单介绍libaio的使用方式，我们主要使用到libaio的io\_setup io\_submit io\_getevents io\_destroy这四个api，其中：

io\_setup 构建一个异步io上下文的句柄，用法类似于epoll中的epoll\_create

io\_destroy 销毁一个异步io上下文的句柄，类似于epoll中的close

io\_submit 提交一个异步io操作到指定的异步io上下文句柄中，传入参数为一个struct iocb的结构，其中包括了要读写的文件句柄，offset，size，buffer和一个void\*类型的指针

io\_getevents 阻塞等待指定数量的异步io完成事件，传入最长的阻塞时间，最少获取的事件数量，最多获取的事件数量，用法类似于epoll\_wait

最简单的伪代码流程如下：

io\_context = **io\_setup**()

fd1 = open()

fd2 = open()

iocb1 = prepare\_read\_param(fd1)

iocb2 = prepare\_read\_param(fd2)

**io\_submit**(iocb1, io\_context)

**io\_submit**(iocb2, io\_context)

events = **io\_getevents**(timeout, min\_num, max\_num)

foreach events

check\_read\_result(events[i].iocb)

close(fd1)

close(fd2)

**io\_destroy**(io\_context)

#### 工作线程交替侦测方案

不使用一个单独的线程来侦测AIO读取状态，而是由工作线程交替侦测AIO状态，通知其它工作线程。这种方案省去了一个线程，但相对复杂一些。整个cs进程共享一个io\_context，当至少一个工作线程在等待AIO数据时，需要保证有且仅有一个工作线程阻塞在io\_getevents上，其它工作线程要么阻塞在等待iocb完成的condition上，要么在处理实际的工作，要么空闲处于睡眠状态。如果没有工作线程在等待AIO读取数据，也就没有线程调用io\_getevents。

这种设计的优点很明显：

1. 不使用额外的线程，系统相对简单。
2. 没有线程等待AIO读取时，不会调用io\_getevents，由于io\_getevents会周期性被唤醒，所以这样设计减少了不必要的上下文切换。

这种设计的缺点也比较明显：在还有至少一个工作线程在等待AIO数据的情况下，需要保证有且仅有一个工作线程阻塞在io\_getevents上，要保证这个状态成立，需要用到锁的机制，实现相对复杂。

在我们的设计中每个线程固定使用双buffer来提交异步io，操作流程如下：

1. 用户调用aio\_submit提交异步io任务，每个异步io请求与一个iocb结构相关连
2. 用户调用aio\_wait阻塞等待所关联的iocb是否已经异步读取完成，如果已经完成则直接返回，否则执行下面几个步骤
3. 先尝试获取一个锁(mutex\_trylock)，如果获取锁成功则调用io\_getevents等待事件，如果获取锁不成功则阻塞等待iocb上的状态变化，直到超时或被唤醒
4. 在调用io\_getevents返回之后，依次修改返回的每个iocb的状态并发信号唤醒正在等待的线程
5. 在完成第4步操作后，判断异步完成的iocb中是否含有本线程正在等待的iocb，如果没有则再次调用io\_getevents；如果有则遍历全局维护的信息中找到一个正在阻塞等待iocb返回的线程，唤醒它
6. 被第5步操作唤醒的线程，首先检查iocb状态，判断异步io是否已经返回，如果异步io已经返回，则遍历全局维护的信息中找到一个正在阻塞等待iocb返回的线程，唤醒它；如果异步还没有返回，则执行第3步操作

为了适应双buffer的应用场景，我们为每个线程维护两个iocb结构，保存在上面第5步提到的全局数组中，根据上面的操作流程，需要为iocb设置如下四种状态，来表示不同的处理阶段：

FREE 表示目前没有异步任务与iocb关联，可以随时用来发起异步io

RETURN 表示异步io已经返回，数据已经准备好或有错误发生

WAITING 表示已经提交异步io，正在等待返回

CANCEL 表示已经提交异步io，但是已经不care它的返回结果了

各个状态之间的转移描述如下：

FREE → (io\_submit提交异步io) → WAITING

WAITING → (io\_getevents获取事件) → RETURN

WAITING → (aio\_wait超时，并不再等待) → CANCEL

CANCEL → (io\_getevents获取事件) → FREE

RETURN → (aio\_wait成功，释放buffer) → FREE

AIOEventMgr对外接口设计如下：

**class** AIOEventBufferInterface  
{  
**public**:  
    // 用于发起异步io时获取buffer  
**virtual void**\*get\_buffer(int32\_t& fd, int32\_t& size, int64\_t& offset) = 0;  
    // 在异步io返回后调用 ret\_code表示执行的返回值 ret\_size表示读取数据的长度  
**virtual void** aio\_finished(**const int** ret\_code, **const int32\_t** ret\_size) = 0;  
};

**class** AIOEventMgr  
{  
  **public**:  
    **int** aio\_submit(**const int** fd, **const int64\_t** offset, **const int32\_t**size,                   AIOEventBufferInterface \*buffer, AIOHandle &handle);  
    **int** aio\_wait(**const** AIOHandle &handle, **const int64\_t** timeout\_us);  
};

其中aio\_submit用于提交异步io，并对外返回一个用来wait的handle，默认情况下每个线程同时可以提交两个异步io，在已经提交两个并且没有wait成功的情况下，再次调用aio\_submit将会返回错误。

aio\_wait用于阻塞等待一个异步io的返回，可能返回成功，超时或错误；返回超时的情况下可以继续等待；返回成功或其他错误，表示异步任务已经结束。

上层的双buffer管理器需要实现AIOEventBufferInterface中的接口，在aio\_submit的时候传入这个接口类的指针，AIOEventMgr在处理异步io的提交和返回时会调用它的虚函数以通知执行过程中状态的改变。

aio\_wait操作流程伪代码如下：

**int** aio\_wait(**const** AIOHandle &handle, **const int64\_t** timeout\_us) {  
**if**(***check\_aio\_result***(hanlde)) { // 判断iocb状态是否为异步已返回 不阻塞  
    ret = succ;  
**return** ret;  
  }  
**while**(true) {  
**if**(mutex\_trylock()) {  
      io\_getevents(timeout\_us);  
      ***handle\_events***();            // 处理异步返回事件 唤醒正在等待的线程  
      mutex\_unlock();  
**if**(***check\_aio\_result***(handle)) {  
        ret = succ;  
        ***wakeup\_other\_thread***();    // 唤醒一个阻塞在wait\_aio\_result的线程  
**break**;  
      }**else**{  
**continue**;  
      }  
    }**else**{  
**if**(***wait\_aio\_result***(handle, timeout\_us)) {  // 阻塞等待异步返回  
**if**(***check\_aio\_result***(handle)) {  
          ret = succ;  
          ***wakeup\_other\_thread***();    // 唤醒一个阻塞在wait\_aio\_result的线程  
**break**;  
        }**else**{  
**continue**;  
        }  
      }**else**{  
        ret = timeout;  
**break**;  
      }  
    }  
  }  
**return** ret;

}

#### 独立侦测线程方案

整个cs进程共享一个io\_context，使用一个独立的线程来侦测AIO的读取状态，该线程循环调用io\_getevents函数检查AIO读取状态，如果AIO读取完成，通知相应的iocb。这种设计优点就是比较简单，是经典的设计方案。缺点是需要一个独立的线程来侦测AIO读取状态，由于io\_getevents会周期性地被唤醒，这样设计增加了不必要的上下文切换。

#### 工作线程自侦测方案

每个工作线程都创建一个io\_context，每个工作线程调用io\_getevents侦测属于自己的io\_context，每个工作线程之间相互独立，互不影响，实现最为简单，也不需要额外的线程来侦测AIO读取状态，在工作线程不需要读取数据时，也不会有不必要的上下文切换。由于cs中的工作线程最多也就50个，不会很多，操作系统能够高效的处理多个io\_context。

这种方案最简单也最容易实现，并且具有前两种设计方案的优点，推荐采用这种方案。

### AIOBufferMgr设计

AIOBufferMgr基于线程同步处理请求的模型设计，每个完整同步流程中都包括advise，get\_block两步调用，在一次scan请求中，advise只能调用一次，线程处理流程结束后advise和get\_block接口不可重入。

AIOBufferMgr管理异步IO所使用的双buffer、调度异步IO操作，属于线程私有实例。AIOBufferMgr使用定长两个定长的buffer，每个buffer 1M，每次需要发起异步io的时候从这两个buffer中寻找个一个可用的，一个buffer上可能会存有多个block的数据，当这些block的数据都被读取完毕之后就可以在下次发起异步io时复用。

上层应用读取数据的流程：

1. 调用advise接口传入一个需要预读的block的集合。
2. 调用get\_block按顺序读取block，直到读到所需的所有的block数据。

#### 接口设计

class AIOEventBufferInterface

{

public:

// 用于发起异步io时获取buffer,读取的文件描述符，文件偏移地址和长度

virtual void \***get\_buffer**(int32\_t& fd, int32\_t& size, int64\_t& offset) = 0;

// 在异步io返回后调用 ret\_code表示执行的返回值 ret\_size表示读取数据的长度

virtual void **aio\_finished**(const int ret\_code, const int32\_t ret\_size) = 0;

};

class AIOBuffer ：public AIOEventBufferInterface

{

public:

enum AIOBufferState {

WAIT,

READY,

FREE

};

public:

int **reset**();

int **get\_state**();

int **set\_state**(enum AIOBufferState state);

int **set\_read\_info**(int32\_t fd, struct BlockInfo\* block, int32\_t block\_count);

int **get\_block**(const uint64\_t sstable\_id,

const int32\_t offset,

const int32\_t size,

BufferHandle &buffer\_handle);

private:

enum AIOBufferState state\_;

int32\_t fd\_;

char\* buffer\_;

int32\_t buf\_size\_;

int32\_t buf\_read\_;

int32\_t buf\_to\_read\_;

int32\_t buf\_pos\_;

struct BlockInfo\* block\_;

int32\_t block\_count\_;

int32\_t cur\_block\_idx\_;

AIOEventMgr event\_mgr\_;

};

AIOBuffer提供了两个接口给AIOEventMgr调用，当AIOEventMgr调用io\_submit前，需要组装iocb的结构体，get\_buffer函数能返回AIOEventMgr组装iocb结构体所需的变量，如文件描述符，文件偏移地址，数据长度，buffer。当AIO读取完成时，AIOEventMgr调用aio\_finished函数，这个函数主要作用就是将AIOBuffer的状态由WAIT修改为READY。

reset ()函数用于将AIOBuffer的内部状态置位，便于重新使用。

get\_state() 用于取AIOBuffer的状态。

set\_state ()用于设置AIOBuffer的状态。

set\_read\_info()用于设置AIOBuffer需要读取的文件描述符，block集合，被AIOBufferMgr调用。

get\_block()用于从AIOBuffer中读取一个指定的block，被AIOBufferMgr调用。

class AIOBufferMgr

{

enum DoubleAIOBufferState

{

FREE\_FREE,

READY\_READY,

WAIT\_WAIT,

WAIT\_READY,

WAIT\_FREE,

READY\_FREE

};

struct BlockInfo

{

uint64\_t sstable\_id\_;

int64\_t offset\_;

int32\_t size\_;

int32\_t set2cache\_:1;

int32\_t cached\_:1;

};

class BufferHandle

{

public:

**BufferHandle**();

~**BufferHandle**();

char \***get\_buffer**();

};

public:

int **advise**(const BlockInfoList &block\_list, BlockCache &block\_cache);

int **get\_block**(const uint64\_t sstable\_id,

const int32\_t offset,

const int32\_t size,

BufferHandle &buffer\_handle);

private:

int32\_t fd\_;

struct BlockInfo\* block\_collection\_; *// block collection to read*

int64\_t block\_count\_;

struct BlockInfo\* cur\_read\_block\_; *// current block collection to read*

int64\_t cur\_read\_block\_count\_;

struct BlockInfo\* preread\_block\_; *// preread block collection to read*

int64\_t preread\_block\_count\_;

struct BlockInfo\* prev\_read\_block\_; *// previous read block*

AIOBuffer buffer\_[2]; *//two aio buffer*

int32\_t cur\_buf\_idx\_;

};

AIOBufferMgr用于管理两个AIOBuffer和读取的block集合，每个线程都有一个AIOBufferMgr私有实例，并使用AIOBufferMgr提供的get\_block()函数读取block数据。Advise()函数用于在每次scan操作中，传入需要读取的block集合，这些block必须在文件中是连续的。

#### 数据读取策略

在get\_block使用双buffer读取数据时，会根据advise传入的block数量和总数据量不同采用不同的读取策略，但每个buffer一定是读取整数个block，不允许block跨buffer的情况发生。根据读取的数据量分为如下几种情况：

1. 数据量小于512K

使用一个buffer一次性读取

1. 数据量大于512K，小于2M

分为两个buffer读取，每个buffer读取大约一半的数据，比如，需读取600K数据，共20个block，每个buffer读取10个block，大约每个buffer读取300K。

1. 数据量大于2M

使用两个buffer交替读取，每个buffer大约读取1M的数据，当上层应用处理其中一个buffer的数据时，AIO预读到另外一个buffer，实现处理数据与读取数据的并行。

#### Advise处理流程

Advise的处理流程如下：

1. 检查传入一个或是多个连续的block集合，对不连续的block或不属于同一个文件的block读取直接报错返回。
2. 将block集合以定长数组形式存储在AIOBufferMgr实例的block\_collection\_中，这个定长block info数组最多可以容纳上万个block info，保证能够容纳整个sstable文件的所有block info，便于应用层一次传入整个sstable文件的block集合。
3. 遍历block集合，查找每个block是否在block cache中，如果在cache中，将block info的cached的bit置位，并修改该block所在内存块的访问时间，保证该block暂时不被淘汰出cache。
4. 把第一个不在cache中的block偏移地址作为第一次读取起始偏移地址和需要读取的连续block个数，读取一定以block为单位。计算好预读block的起始偏移地址和block个数。

每个线程的AIOBufferMgr实例将需要批量读取的block集合一次性通知AIOBuffer读取，然后调用get\_block按顺序取出所需的block，get\_block读取block一定是按顺序的，不能乱序。

#### 双buffer状态处理

AIOBufferMgr默认为双buffer，因为ob读取一般为顺序读取，如果前一个buffer都没有返回的情况的下读取下一个buffer，意义不大，并且很有可能会导致读出的数据无效，应用层也可能由于出错，导致预读的数据无效，所以，在前一个buffer完成读取的时候，才发起下一个buffer的预读，同时AIOBufferMgr需要处理预读数据无效的情况。单个buffer有三种状态：

1. WAIT —— 等待AIO读取数据到buffer中，读取的数据不一定有效。
2. READY ­—— buffer中的数据AIO读取完成，但数据不一定有效。
3. FREE —— buffer空闲，可以读取新的数据。

单个buffer状态转移如下图所示：

从上面的状态转移图可以看出，对FREE状态的buffer调用aio\_submit后，buffer进入WAIT状态，当AIO将buffer所需的数据读入后，修改buffer的状态为READY状态，get\_block读取到buffer中的最后一个block时，将buffer状态设置为FREE状态，从而该buffer在下次get\_block时可以重用；如果当前线程读取的文件与处于READY状态的buffer所读取的文件相同时，意味着以前的某个get\_block超时，或是上层应用出错，应用层没有通知停止正在读取的数据的buffer，所有buffer状态仍然为WAIT，当AIO读取完成时，丢弃无效的数据，修改buffer的状态为READY状态。READY状态转变成FREE状态有两种可能情况：1）取到了buffer中最后一个block；2）buffer中的数据是无效的。

判断处于READY状态的buffer中的数据是否为无效数据的方法：1）buffer所属的文件不是当前线程正在读取的文件。2）buffer中的block不是期望的block。如果两个条件有一个不满足，则该buffer的数据是无效的。这个判断在双buffer状态的转换过程中经常使用。

导致处于READY状态的buffer中数据无效的原因有两个：1）get\_block成功后，上层应用出错退出了，不会通知AIO停止读取，也不会修改buffer的状态。2）get\_block超时了，AIO仍然继续读取数据，buffer的状态保持为WAIT。AIO读取完成时会把处于WAIT状态的buffer修改为READY状态，由于上面提到的两个原因，导致了处于READY状态的数据无效，下文提到buffer数据无效，指的就是这两种情况。

由于单个buffer具有3种状态，那么两个buffer会有6种状态组合，分别为FREE + FREE、READY + READY、WAIT + WAIT、WAIT + READY、WAIT + FREE和READY + FREE。双buffer的状态转移图如下图所示：

Get\_block实现了上图所示的状态机，当调用get\_block时，根据双buffer状态按照状态机进行切换，直到获取到一个所需的block，或者超时，或者出错返回。正常的读取流程如下：

1. 开始时，双buffer处于状态1，使用其中一个buffer读取数据，转入状态5。
2. AIO完成读取返回第一个buffer数据，转入状态6。
3. 从处于READY状态的buffer按顺序取出block返回给应用层，如果另一个buffer空闲，并且还有数据需要读取，发起预读，转入状态4。
4. 如果预读比上层处理慢，第一个READY的buffer中的block都已经全部取走，第二个buffer的数据都还没有准备好，转入状态5等待。
5. 如果预读比上层处理快，两个buffer数据都准备好了，转入状态2。
6. 如果处于状态2，并且其中一个buffer的block已经全部取走，转入状态6。
7. 根据预读快慢和所读数据量多少，会在状态2,4,5,6之间反复切换，直到读取了所有的数据，最后只剩下一个READY的buffer，处于状态6，如果最后的一个buffer的数据也完全取走了，转入状态1，等待下次读取开始。

但在实际的读取过程中，每次读取不一定是从状态1开始，到状态1结束，上图中6中状态都有可能是起始状态，也有可能是结束状态，状态机需要处理所有的状态转移情况，所以，下面给出双buffer处于不同状态情况下的不同处理方式：

1. **FREE + FREE**

**三种情况：**

* 1. 双buffer的初始化状态。
  2. 由状态2异常情况转变而来，两个buffer中的数据都无效，丢弃，修改buffer状态为FREE。
  3. 由状态6转变而来，所有的block数据都读取完了，或者处于READY状态的buffer中的数据无效，丢弃，将处于READY状态的buffer设置为FREE。

**正常情况处理：**选择第一个buffer调用aio\_submit发起AIO读取数据，设置buffer状态为WAIT，由于buffer中没有可用的数据，只能调用aio\_wait等待AIO读取完成。如果没有超时，转入状态6处理。

**异常情况处理：**如果超时，返回错误，退出状态机循环。

1. **READY + READY**

**三种情况：**

* 1. 状态4的正常情况转变而来，预读速度很快，两个buffer都读取完成了；
  2. 保持状态2，两个buffer数据都有效，block还没有被get\_block全部取走。
  3. 由状态3的异常情况转变而来，两个处于WAIT状态的buffer都读取完成了，一般情况下，这两个buffer读取的文件是是不一样的，至少有一个buffer的数据无效。

**正常情况处理：**如果其中一个buffer包含有效数据，get\_block直接从该buffer中获取block。如果返回的block是buffer中最后一个block，将该buffer状态设置为FREE，否则不改变buffer状态。如果另一个buffer的数据也有效，则返回block并退出状态机循环；如果另一个buffer数据无效，将该buffer状态设置为FREE，由于已经取到block，**就不要转入状态1处理了**， 如果还有数据需要预读，调用aio\_submit使用该buffer发起预读，设置buffer状态为WAIT，否则返回block并退出状态机循环。

**异常情况处理：**如果两个buffer数据都无效，将两个buffer的状态都设置为FREE，由于还没有读取到block数据，需要转入状态1处理。

1. **WAIT + WAIT**

由状态5的异常情况转变而来，一个buffer正在读取无效数据还没有完成，只有一个buffer可用，可用的buffer读取数据同样没有完成。并且读取超时了，双buffer都还处于WAIT状态。WAIT + WAIT比较特殊，状态机内部不会转到这个状态执行处理代码，在get\_block时，这种状态只会是双buffer的一种初始状态，这种情况可能AIO读取hang住了，系统出问题了，需要状态机做特殊处理。

**异常情况处理：**

1）调用aio\_wait等待任何一个buffer读取完成，超时时间设置为get\_block的超时时间，如果aio\_wait等待超时，如果另外一个buffer仍然处于WAIT状态，双buffer仍然处于状态3，没有buffer可用，没办法处理了，为了防止死循环，只能退出状态机循环，并停止让当前线程工作。如果系统中多个线程都处于这种状态，可能操作系统或是磁盘读取出问题了，只能kill cs了。

2）如果没有超时，有一个buffer转变为READY状态，则转入状态4处理。

3）如果么有超时，两个buffer都转变为READY状态，则转入状态2处理。

1. **WAIT + READY**

**三种情况：**

1. 由状态3转变而来，处于WAIT状态的buffer在读取无效数据没有返回，只有一个buffer可用时，get\_block使用该可用buffer发起AIO读取数据，但get\_block超时了，两个buffer都处于WAIT状态，上层应用再次调用get\_block读取block，这时其中一个buffer读取完成了。
2. 由状态4正常情况转变而来，正常的预读，get\_block从处于READY状态的buffer读取block，处于WAIT状态的buffer正在预读接下来的数据。
3. 由状态6正常情况转变而来，前一个buffer的数据都返回给应用层了，前一个buffer空闲了，后一个buffer（READY状态）的数据也是有效的，从有效的buffer中读取block，用处于FREE状态的buffer进行预读，于是buffer状态从FREE转变为WAIT。
4. 由状态5的异常情况间接转变而来，处于WAIT状态的buffer正在读取无效的数据，使用另外一个FREE的buffer读取数据，并调用aio\_wait等待完成，如果没有超时，buffer处于READY状态，并且另外一个buffer仍然处于WAIT状态。其实这是一个间接过程转换的，状态5转变为状态3，再由状态3转变为状态4。实际上是不存在从状态5转变到状态4的直接路径。参见状态5的异常情况处理分析。

**正常情况处理：**如果处于READY状态的buffer有效，无论处于WAIT状态的buffer正在读取的数据是否有效，从READY的buffer中获取block，正常情况下，处于WAIT状态的buffer正在读取接下来会用到的block数据。如果返回的block是处于READY状态的buffer中最后一个block，将buffer状态设置成FREE，退出状态机循环。在从READY状态的buffer读取block时，即使处于WAIT状态的buffer转变为READY状态，也不影响正常操作，只会影响到下次get\_block时双buffer的初始状态。

**异常情况处理：**如果处于READY状态的buffer数据无效，设置该buffer为FREE状态；判断处于WAIT状态的buffer正在读取的数据是否为需要的数据，如果是，调用aio\_wait等待AIO读取完成，如果aio\_wait等待没有超时，则转入状态6处理，如果aio\_wait等待超时，直接返回错误，退出状态机循环。否则，转入状态5处理。

1. **WAIT + FREE**

**四种情况：**

* + 1. 状态1转变而来，get\_block读取第一个block时，并且可能以前的某次get\_block超时了或者应用层出错退出了，处于WAIT的buffer读取的数据无效。
    2. 由状态4异常情况转变而来，参见状态4的异常情况处理分析。
    3. 由状态5异常情况转变而来，处于WAIT状态的buffer正在读取的数据为有效数据，调用aio\_wait等待buffer读取完成，但超时了，buffer状态保持不变。
    4. 由状态6正常情况转变而来。

**正常情况处理：**如果处于WAIT状态的buffer读取的数据正是所需要的（应用层处理数据比从磁盘读取数据快，下一个buffer还没有准备好），调用aio\_wait等待AIO读取完成。Aio\_wait没有超时，转入状态6处理；如果aio\_wait超时，buffer状态保持不变，返回错误，退出状态机循环。

**异常情况处理：**如果处于WAIT状态的buffer读取的数据是无效的，处于WAIT状态的buffer不能做任何处理，使用FREE的buffer调用aio\_submit读取当前需要的block数据，设置buffer状态为WAIT，并调用aio\_wait等待AIO读取完成。这种情况一般发生在一个AIO读取请求长时间没有返回导致只有一个buffer可用，由于没有预读剩下的数据，所以需要发起新的AIO读取数据，相当于没有预读，使用单个buffer读取数据。如果aio\_wait超时了，返回错误，退出状态机循环。如果aio\_wait没有超时，处于WAIT状态的buffer会转变成READY状态，如果另一个buffer仍然为WAIT状态，转入状态4处理；如果另一个buffer转变为READY状态，转入状态2处理。（这里已经处理了WAIT + WAIT正常情况的状态转变，所以不用再转到状态3执行，从而保证了状态3只会是get\_block的初始状态，便于错误处理）。

1. **READY + FREE**

**九种情况**：

* 1. 所需要的读取的数据小于512K，用一个buffer一次就可以读完，并且这个buffer已经完成读取了，这是从状态5转变到状态6的一种特殊情况。
  2. 所有的数据都读完了，就剩下这个buffer里的数据还没有被get\_block取走，这是状态6保持不变的一种特殊情况。
  3. 由状态2的正常情况转变而来，预读的速度比应用层处理速度快，上一个buffer返回自己包含的最后一个block时，下一个buffer都准备好了，这种情况在状态2已经返回了block，状态机循环已经退出，不会再执行到状态6的处理代码。
  4. 由状态2的异常情况间接转变而来，状态2的异常情况导致转变成状态1，状态1再转变成状态5，最后由状态5转变到状态6。
  5. 由状态4的异常情况转变而来，处于READY状态的buffer数据无效，设置该buffer为FREE状态；处于WAIT状态的buffer数据有效，在超时时间内，buffer由WAIT状态转变为READY状态。
  6. 由状态1的正常情况间接转变而来，状态1转变成状态5，由状态5转变到状态6。
  7. 由状态4正常情况转变而来，在读取READY状态buffer的最后一个block时，设置该buffer为FREE状态，处于WAIT状态的buffer转变为READY状态。
  8. 由状态5正常情况转变而来，参见状态5正常情况处理分析。
  9. 由状态6正常情况转变而来；

**正常情况处理：**如果处于READY状态的buffer数据有效，从处于READY状态的buffer中获取block，如果返回的block是处于READY状态的buffer中最后一个block，将buffer状态设置成FREE。如果还有剩余数据需要读取，使用FREE的buffer调用aio\_submit发起AIO读取剩余数据，设置buffer的状态为WAIT，不用等待AIO完成。返回从READY状态的buffer取到的block数据，退出状态机循环。

**异常情况处理：**如果处于READY状态的buffer数据无效，如果应用层没有按顺序读取block，将buffer状态设置为FREE，报错返回，退出状态机循环。否则，将buffer的状态设置为FREE，转入状态1处理。

#### Get\_block处理流程

Get\_block处理流程如下：

1. 检查是否为顺序读取，即检查当前读取的block是否在前一个读取的block后并紧邻，如果不是顺序读取，返回错误，否则，转入2执行。
2. 判断所需block从block cache读取或是从AIOBuffer中读取，如果从block cache中读取，读取并返回，否则转入3执行。
3. 获取双AIOBuffer的状态，启动5.1.3.5节描述的状态机。如果状态机返回超时，直接返回超时错误，否则，取到了所需的block，转入4执行。
4. 如果需要将该block拷贝到blockcache中，并且该block不在block cache中，则拷贝该block到block cache中。
5. 更新AIOBufferMgr内部状态，返回读取到的block。

#### 多column group和多文件支持

当需要同时读取多个文件的多个column group时，同一个线程中有多个ObAIOBufferMgr实例，每个实例中有两个AIObuffer，每个线程的ObAIOBufferMgr以数组形式管理，可以通过查询（table\_id，sstable\_id，column\_group\_id）对应的ObAIOBufferMgr实例来过的所需的ObAIOBufferMgr。

通过这种水平扩展来实现对多文件和多column group的支持，当同时读取的文件和column group很多时，需要较多的内存，这个问题暂时还不是很严重，留待以后优化。

#### 其它

1. 对预读数据已部分进入block cache的优化，对超过512K(可配置)的连续数据都在block cache中数据，不发起AIO，而是读取一次block cache以保持数据热度，保证get\_block的时候能后尽量从block cache获取到。
2. 在读取block cache失败(数据已经被淘汰)的情况，需要在get\_block的逻辑中执行同步的io读取(用AIO模拟同步)，这里处理有点困难，要是block cache能保证打过标记的block一定不会被淘汰，get\_block处理就简单多了。
3. 处理异步io返回超时或一直不返回的情况，状态机中的表现就是AIOBufferMgr实例中的两个AIOBuffer的状态都是WAIT，如果再次等待超时，没有可用的buffer，该线程就不能提供服务，直到至少有一个AIOBuffer可用为止。如果系统中多个线程都发生了两个AIOBuffer的状态长时间处于WAIT，意味着系统或是读取磁盘数据出问题了，没有好的解决办法，直接kill cs。
4. Get\_block函数会将需要加入block cache的block拷贝到block cache中，尽量保证相邻的block拷贝到同一个内存块中，这样可以保证淘汰了一个内存块，意味着多个相邻的block都被淘汰出去，下次如果再次读取这一组block，可以通过一次IO读取。这样相邻的block块组存在block cache中，读取时可以避免由于相邻的block中有某些block不在cache中，从而导致重新读入整个block组。每个block组可能是512K左右，可以配置。
5. 由于AIOBufferMgr是线程的私有实例，WAIT->READY状态转变可能在另外一个线程里执行，其它两种AIOBuffer状态的转变（FREE->WAIT,READY->FREE），都是在本线程中完成，仔细分析状态转变的情况，在本线程处理get\_block时，对处于WAIT状态的buffer要么不做任何处理，要么调用aio\_wait等待，即使另外一个线程修改buffer的WAIT状态为READY状态也没有问题，所有不需要锁保护。

# 其它

这部分主要考虑对特别消耗cpu的操作进行优化，比如序列化和反序列化，而对crc校验算法和压缩算法都已经做过优化，所以在这里比较有针对性的优化int类型的序列化和反序列化。正在尝试去掉一些不必要的检查，优化C代码，用汇编实现等多种方法，希望能大幅度提升序列化和发序列化的效率。