1 概述

本文档主要介绍 OSS 写缓存相关的硬盘结构，内存索引结构。同时对这些结构的更新逻辑也做了简要描述。附录中有对一些概念的补充描述。

2 硬盘结构

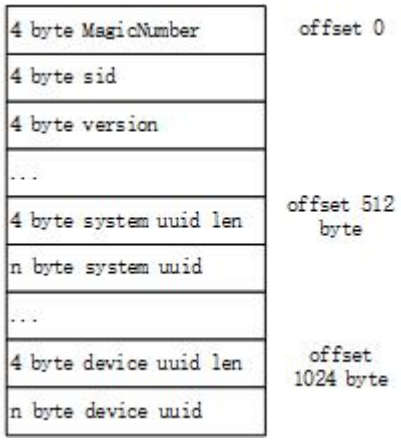
2.1 总览

SSD 磁盘布局：

各个部分描述如下：

- 1M 预留的一部分空间，上面会有一部分管理系统记录的磁盘信息
- Magic block 魔术区，用来确定磁盘是否已经被 OSS 使用
- Index Log 更新 Index Records 时记录的日志
- Index Records LogUnit 元信息
- LogUnit 实际记录数据的连续区域

2.2 Magic block



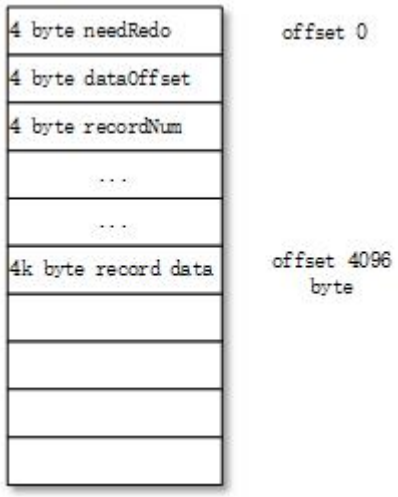
Magic block 区预留了 512K 的空间，但是实际并没有使用到这么多。各个部分描述如下：

- Magic Number 魔数，当前值为 0x20160310ULL
- Sid OSS 服务的 Sid，启动参数中指定
- Version 数据结构版本，当前值为 1
- System UUID 存储系统 UUID，启动参数中指定

- Device UUID 当前版本使用系统 UUID 填充

系统启动之初会读取 Magic Block 信息，如果 SSD 头部记录的 Magic Number, Sid, System UUID, Device UUID 与启动 OSS 指定的信息一致，那么认为 SSD 是之前就已经添加给 OSS 的磁盘，后续执行 Recover 操作。如果上述任意一个信息不一致，则认为 SSD 磁盘是第一次添加给 OSS，后续执行 Format 步骤。

2.3 Index Log



从 2.1 中可以看到 Index Log 区域总共预分配了 512K 的空间。但是实际只是用了很少的一部分。各个部分描述如下：

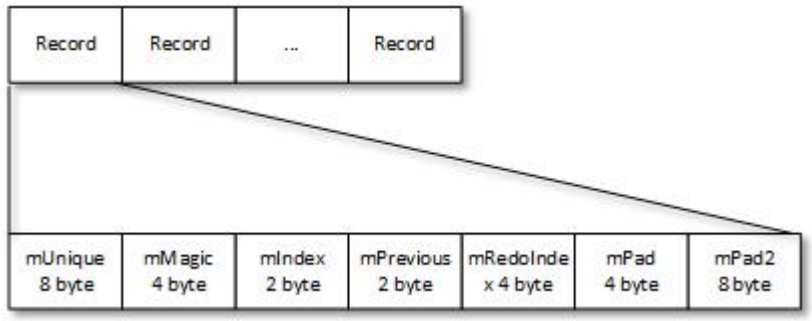
- needRedo 重启 OSS 是是否需要重新将 record data 区数据写入 Index Records 的标识。为 1 表示需要重写，为 0 表示不需要
- dataOffset record data 相对与 Index Records 区的偏移。
- recordNum IndexRecord 区记录了多少个 Index Record。该值与当前 SSD 中的 LogUnit 总数一致
- record data 更新 Index Record 的数据

这里需要注意的是一个 Index Record 虽然只占有 32 byte，但是写入到 record data 的时候为了保持 4K 对齐，写入了一个 page。后面更新 Index Records 区也是 4K 大小来更新的。

Index Log 使用一种 In-Place Update 的方式，使得对 Index Record 的更新不会出错。工作逻辑如下：

1) 将要更新的 Index Record 所在的 page 数据写入到 record data 中
2) 将 needRedo 置为 1
3) 将 record data 中的数据写入到 Index Records 区中
4) 将 needRedo 置为 0

2.4Index Records



Index Records 区总共预分配了 2M 的空间，由连续的 Index Record 构成，Index Record 用来唯一标识一个 LogUnit。每个 Record 占用 32 byte（由此可以推算，总共可以容纳 2M/32Byte = 64K 个 record，或者说 64K 个 LogUnit）。

Index Record 个部分描述：

- mUnique　　Storage 层工作线程组标识
- mMagic　　当前使用随机数填充
- mIndex　　record 在 Index Records 区的索引。同时也标识 LogUnit 的索引
- mPrevious　　记录 Unit 链表中上一个 LogUnit 的 Record，如果为 Unit_invalid_index 则表明当前 LogUnit 为 Idle
- mRedoIndex　当前未使用
- mPad　　当前未使用
- mPad2　　当前未使用

2.4.1 内存维护

1) RecordManager

初始化的时候 RecordManager 将 IndexRecord 区所有的数据读入到内存中。

2) UnitAllocator

初始化时，UnitAllocator 遍历 RecordManager 独处的 IndexRecord 数据。进行如下工作：

- 根据 mPrevious 判断是否未空闲 Record。如果是加入到 mIdle 中，否则加入到 mUsing 中
- 根据 mUnique 将 Record 分配到不同的线程组中
- 根据 mPrevious 回溯所有 Record，将属于同一个线程组的 Record 放到一个列表中

2.4.2 更新时机

1) 分配 LogUnit

分配 LogUnit 的时候需要首先分配 IndexRecord。从 Idle 的集合中(DeviceHeader::UnitAllocator 中维护)找到上一次分配的 Index Record 之后的一个空闲的

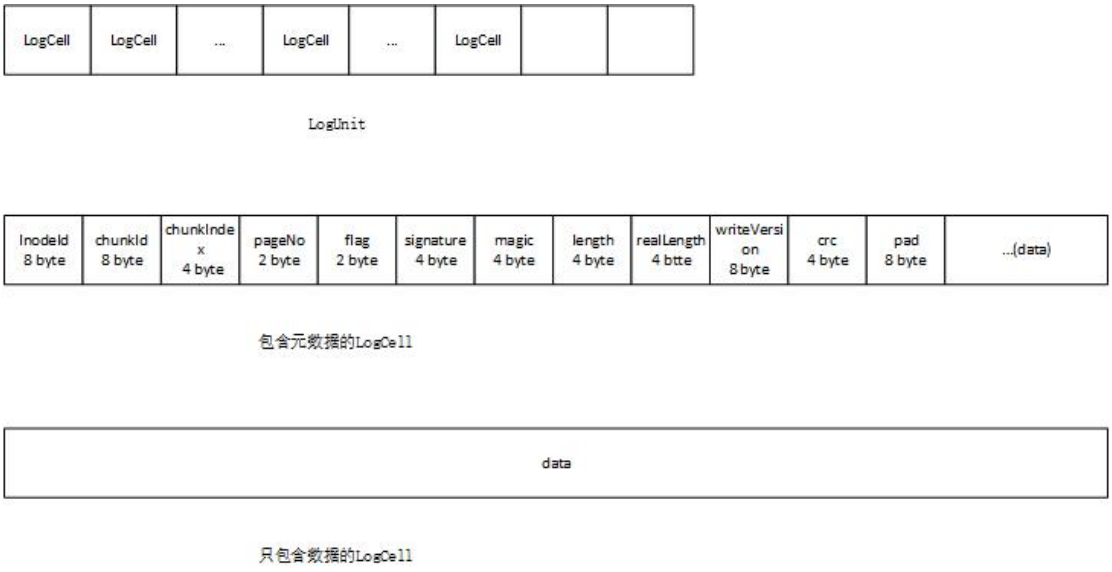
Record。设置其 mUnique,， mPrevious， mMagic 后写入到硬盘 Index Records 区。

2) 释放 LogUnit

释放某个 LogUnit 的时，要将对应的 Index Record 的 mPrevious 置为 Unit_invalid_index，写入磁盘头部。同时，如果 LogUnit 后面还有 LogUnit，那么也需要将后面 LogUnit 对应的 Index Record 的 mPrevious 做修改。

2.5 LogUnit

2.5.1 总览



磁盘上一个 LogUnit 占用 32M 空间，OSS 将这 32M 按照 4K 大小划分为了 8K 个 LogCell。

2.5.2 LogSegment

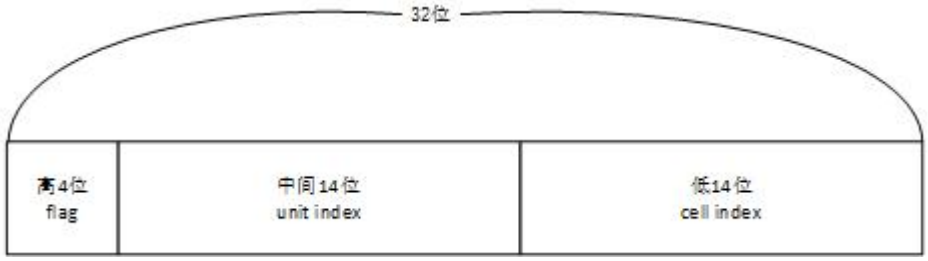
OSS 对 SSD 盘进行写操作的时，逻辑上按照 LogSegment 来进行写入，一个 LogSegment 包含 1 个或多个 LogCell。按照请求分类，LogSegment 主要由如下几种：

- LCF_normal 执行写操作后生成的 LogSegment
- LCF_truncate 执行 truncate 请求后生成的 LogSegment
- LCF_drop_block 执行 Replicate write 请求时生成的 LogSegment
- LCF_delete 执行 delete 请求时生成的 LogSegment

其中 LCF_drop_block 和 LCF_delete 的 LogSegment 只包含由 1 个带元数据的 LogCell。而 LCF_normal 和 LCF_truncate 的 LogSegment 既包含 1 个元数据的 LogCell 也包含带有数据的 LogCell。

2.5.3 LogIndex

每个 LogSegment 对应着一个 LogIndex，用来标识改 LogSegment 的位置，后面内存索引中会大量用到这个变量。其组成如下：



LogIndex 由 32 位整形数标识。其中高 4 位留作 flag 使用，目前使用了最高位表示是否压缩。中间 14 位 Unit Index 表示所属 LogUnit 的索引，14 位可表示 16K 个 LogUnit。低 14 位可表示 16K 个 LogCell。

2.5.4 LogCell

物理上看，LogCell 就是 SSD 上连续的 4K 的一个区域。我们在使用的时候，总体上分为三类：

1) 只包含元数据的 Cell
2) 包含元数据，也包含部分压缩数据的 Cell
3) 只包含数据的 Cell

包含元数据的 LogCell 各个字段描述：

- InodeId 操作的文件 InodeId
- ChunkId 全局的 Chunk 文件 Id
- ChunkIdex 操作的 Chunk 在文件内部的索引
- pageNo 本次 LogSegment 包含数据的起始 page
- flag 上文中提到的 LogSegment 的标志
- signature 当前未使用
- magic LogCell 所属 LogUnit 对应的 Index Record 记录的 magic
- length LogSegment 上数据长度（压缩成功表示压缩后的长度，压缩失败表示未压缩长度）
- realLength LogSegment 上数据的实际长度（未压缩长度）
- writeVersion Chunk 的 WriteVersion
- crc CRC 校验码
- pad 未使用

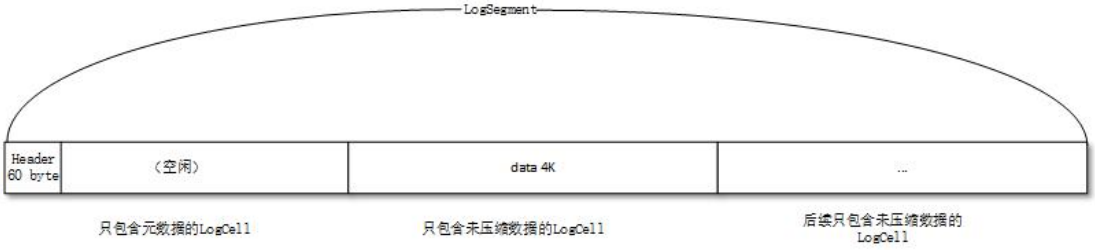
2.5.5 数据压缩

Write，Truncate 操作包含了数据部分，OSS 处理的时候会尝试对数据部分进行压缩（当

前压缩工具使用 snappy，且只针对数据部分进行压缩）。假设数据部分实际长度未 realLength，第一个 LogCell 的元数据部分大小为 LogCell_header_size(60 byte)，数据部分压缩后的大小为 compressedLength；当 LogCell_header_size + compressedLength < realLength 时表示压缩成功。

- 压缩成功时 LogSegment 数据分布
  数据从元数据 LogCell 的 data 部分开始填充

- 压缩失败时 LogSegment 数据分布
  数据从元数据 LogCell 后的 LogCell 开始填充。元数据 LogCell 的 data 部分空闲。



2.6v2.1 相对与 v2.0 的变动

磁盘布局上唯一变动的部分就是 LogUnit 的大小从 64M 变成了 32M。此举是为了减少 LogUnit 中的索引数量，加快 Redo 搜索。

3 索引结构

3.1 Index Tree

3.1.1 结构

Index Tree 可以理解伟一个基于 B*树的的 map。这个 map 的 key 由类 SearchKey 表示；map 的 value 由 DataRecord 表示。

SearchKey 的组成为(InodeId, ChunkIndex, PageNunber)组成的一个三元组。表达的意思也即某个文件，某个 Chunk 的某个 Page 号。

DataRecord 由两个部分组成 SearchKey, LogIndex。SearchKey 即 DataRecord 对应的 key。LogIndex 表示某一个 LogSegment 的索引。DataRecord 中记录 SearchKey 是为了在直接查找 DataRecord 的时候更加方便（比如 LogThreadContext::DropMatchLogUnit）。

因此很清楚，Index Tree 记录的是某个 Chunk 的 Page 在哪个 LogSegment 上，显然，对

于读操作，Index Tree 可以迅速的定位请求的数据在 SSD 的哪个位置。

3.1.2 索引操作

3.1.2.1 索引重建

Index Tree 结构在 SSD 启动的 Recover 阶段重新构建。

以 Recover 索引为 logIndex 的写操作 LogSegment 为例，读出每个 LogSegment 的 StartPage，并通过 LogSegment 的 realLength 字段计算出当前 LogSegment 共包含的 page 数 pageNum。然后将 IndexTree 中，[StartPgae, StartPage + pateNum - 1]这个范围内的 page 记录指向 logIndex。

3.1.2.2 索引更新

任然以写操作为例，对于写操作，上层传递的请求中，总是包含如下信息：InodeId，ChunkIndex，offset，size。SSD 中的写操作总是按 page（4K）对齐的，任意一个写操作的起始，结束偏移总会分别落在某个 beginPage, endPage 上。

写操作将会生成一个 LogSegment，将写入的数据放入这个 LogSegment 中。同时将写入的[beginPage, endPage]信息，在 Index Tree 中更新。

3.1.2.3 索引删除

1. 重新分配 LogUnit

SSD 写缓存的 Redo 机制，将 LogSegment 中记录的数据写入下层的 HDD 磁盘。一旦某个 LogUnit 中的所有 LogSegment 全部被 Redo 完成，该 LogUnit 就存在被重新分配使用的问题。重新分配 LogUnit 使用的时候，需要将 IndexTree 中，所有 value 值（LogIndex）在该 LogUnit 内的项删除。（LogThreadContext::DropMatchLogUnit）。

2. Truncate 操作删除

Truncate 的时候，会将指定起始 page 到 Chunk 末尾的所有 page 记录从 IndexTree 中删除。

3. Replicate Write 操作删除

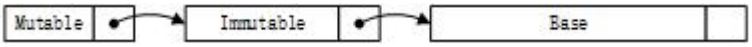
Replicat Write 的时候，将 write 操作 beginPage,endPage 范围内的记录从 IndexTree 中删除。

3.2 HashIndexTree

为了加速索引查找，v2.1 里面增加了 HanshIndexTree，将 v2.0 中的大 IndexTree 分片成 INDEX_hash_shard_size（当前值为 16）个相对较小的 IndexTree。将（InodeId, ChunkIndex）做 Hash 得到的 hashValue 对 INDEX_hash_shard_size 取余得到对应的 IndexTree。

v2.1 中，又分别引入了 Mutable，Immutable，Base Index Tree 的概念。分别表示正在修

改的 IndexTree，不变的 Index Tree，基本的 Index Tree（或者说正在 Redo 的）。他们的关系如下：



Mutable 会向 Immutalbe 转换，Immutalbe 最终会合并到 Base 中。

3.2.1 MutableHashIndex

表示正在修改的 Index Tree，比如写操作更新的 Index Tree 信息只会操作 MutableIndexTree。

3.2.2 ImmutablehashIndex

表示不变的 Index Tree，ImmutableIndexTree 是从 MutableIndexTree 转变而来的。再写操作重新分配 LogUnit 的时候，有判断，如果 ImmutableIndexTree 已经被合并到了 BaseIndex 中。那么就将 MutableIndexTree 转变成 ImmutableIndexTree，并重新生成 MutablesIndexTree。

3.2.3 BaseIndex

Redo 某个 LogUnit 的时候，只会在 BaseIndex 中进行查找。

3.3 ReadRangeTable

3.3.1 结构

ReadRangeTable 可以理解为基于 B*树存储的 map 结构。这个 map 的 key 为类 ReadRangeKey，value 为 ReadRangeRecord。

ReadRangeKey 记录的是 LogIndex。

ReadRangeRecord 记录的是 PageNums。

因此很清楚，ReadRangeTable 记录的就是某个 LogSegment 在 SSD 中占据了多少个 page。这对于读出某个 LogSegment 的内容是很有帮助的。依靠 Index Tree，OSS 可以知道请求的 page 在哪个 LogSegment 上，接下来就会尝试从 SSD 中读出数据。但由于 LogSegment 的数据可能是被压缩的，因此总是需要将这个 LogSegment 全部读出来。ReadRangeTable 的作用就是在尝试读取的时候，告诉 OSS 这个 LogSegment 有多长，需要读 SSD 上的多少个 page。

3.3.2 索引操作

3.3.2.1 索引重建

与 Index Tree 的重建同步，在 Recover 阶段，读出某个 LogSegment 的内容后，将其占用

的 pageNums 数更新倒 ReadRangeTable 中。

3.3.2.2 索引更新

OSS 进行写操作时，重新生成一个 LogSegment，其写入的 page 长度为 PageNums。将该 LogSegment 的<LogIndex, PageNums>更新到 ReadRangeTable 中。

3.3.2.3 索引删除

与 IndexTree 中删除类似。在重新分配 LogUnit 时，LogUnit 内原本记录的所有 LogSegment 信息都需要从 ReadRangeTable 中移除。

3.4 HanshReadRangeTable

与 HashIndexTree 类似，v2.1 中将之前大的 ReadRangeTable 分片成 INDEX_hash_shard_size（当前值为 16）个相对较小的 ReadRangeTable。同时也引入了 Mutable，Immutable，Base 三类 HanshReadRangeTable。分别表示，正在修改的，不变的，基本的（可以 Redo）的 HanshReadRangeTable。

3.4.1 MutableHashReadRangeTable

正在进行写操作的 LogUnit 产出的 LogSegment 的长度信息都会记录在 MutableHashReadRangeTable 上，随 MutableHashIndexTree 一起做合并操作。

3.4.2 ImmutableHashReadRangeTable

由 Mutable 的 HanshReadRangeTable 转换而来，与 ImmutableHashIndexTree 一起合并到 BaseHashReadRangeTable 中。

3.4.3 BaseHashReadRageTable

Redo 某个 LogUnit 的时候，只会在 BaseIndex 中进行查找。

3.5 ReverseIndexTree

3.5.1 结构

从类名上可以猜测到 ReverIndexTree 是对 IndexTree 的反转，确实 ReverIndexTree 记录的

是：某个 LogSegment 中有效 page 的集合（有效 page 请参见附录中的描述）。采用 std::map 来记录信息。map 的 key 为 LogIndex，value 为 MetaIndex::Segment。

类 Segment 中记录有(IndoeId, ChunkIndex, [pages])，也即某个文件的某个 Chunk 的某些 page 集合。

3.5.2 构建

ReverIndexTree 是专门为 Redo 创建的。开始对某个 LogUnit 进行 Redo 前的 PrepreForRedo 阶段，将 IndexTree 中属于该 LogUnit 的所有记录项查找出来保存在 mRecordSet 中，并基于 mRecordSet 构建出 ReverseIndexTree。

Redo 过程从 ReverseIndexTree 中取出 LogIndex 尝试进行合并，写入到后端 HDD 中。

3.6 RedoRangeTable

RedoRangeTable 类型上与 ReadRangeTable 一致，只是 RedoRangeTable 中的内容记录的是正在 Redo 的 LogUnit 中所有 LogSegment 的信息。

3.7 ShardIndexFilter

3.7.1 结构

由 N 个 ShardIndexRecord 结构组成，N 是当前 SSD 支持的最大 LogUnit 数量。ShardIndexRecord 类的主要构成为 std::bitset<32>，每一位用来表示 HashIndexTree 中的子树。

ShardIndexFilter 作用是，记录 LogUnit 在 HashIndexTree 的子树中是否由索引存在。

3.7.2 索引操作

3.7.2.1 索引重建

与 Index Tree 类似，在 Recover 阶段，Recover LogSegment 的时候会将 ShardIndexFilter 中的指定 bit 置为有效。

3.7.2.2 索引更新

与 IndexTree 类似，写相关操作时，会对操作的 LogUnit 所在的 bit 位置为有效。

3.7.2.3 索引删除

与 IndexTree 的删除类似，在 LogThreadContext::DropMatchLogUnit 中，将某个 LogUnit 在 ShardIndexFilter 中所有相关的 bit 位全部置为无效。

3.7.3 作用

以 PrepareForRedo 为例，首先判断 LogUnit 在 HashIndexTree 的分片中是否由索引存在。如果存在索引，则就在分片的 IndexTree 进行查找。

ShardIndexFilter 就是用来判断 LogUnit 在分片中是否存在索引的。减少了一部分索引树的遍历。

3.8ShardRangeFilter

3.8.1 结构

由 N * M 个 ShardRangeRecord 结构组成，N 表示 LogUnit 数量，M 表示 IndexTree 分片的数量（当前为 INDEX_hash_shard_size 16）。ShardRangeRecord 中记录两个 SearchKey 结构，表示最大和最小的 page。

从上面描述中可以知道，ShardRangeFilter 记录的是某个 LogUnit 在某个 IndexTree 分片上的最大最小的 page。

3.8.2 索引操作

ShardRangeFilter 的更新时机与 ShardIndexFilter 相同。

3.8.3 作用

以 PrepareForRedo 为例，在 v2.0 版本的 SSD 中，要获取某个 LogUnit 在 IndexTree 中的所有索引，需要通过 unitIndex 相等进行查找。有了 ShardRangeFilter 后，就可以使用 MinSearchKey, MaxSerachKey 进行查找，减少了查找范围，加快了搜索速度。

3.9v2.1 相对与 v2.0 的变动

1. 增加 HanshIndexTree，HashReadRangeTable 将索引分片
2. 引入 Mutable，Immutable，Base 概念将索引分类
3. 增加 ShardIndexFilter，ShardRangeFilter

4 ByPass SSD

实际测试中遇到的了 SSD 写满的问题，在 v2.0 的版本下，SSD 写满，后续的写操作如果没有 LogUnit 可用，上层的写操作返回 EAGAIN。相当于写操作停滞了。

v2.1 里面，当出现这种情况是会将写入模式切换。数据部分不再写入到 SSD 中，而是直接写入后端的 HDD 中。同时在 SSD 中写入一个 LCF_drop_block 类型的 LogSegment，表示写入的数据在 SSD 中已经失效了，需要到后端读取，同时将索引树中相关项删除掉。

5 附录

5.1 LogSegment 中的有效 page

假设一个写操作在某个 LogSegment 中写入了连续的 N 个 page，并在 IndexTree 中做了更新。紧接着，在后续的写操作中，重写了其中的 M 个 page，这样上一个 LogSegment 中的 M 个 page 数据就失效了，因此有效的 page 数为 N-M 个。

5.2 引入 Mutable,Immutable,Base Index 的好处

v2.0 版本的 SSD 中，所有的所有都存在 IndexTree 中。这样 Read,Write,Redo 在操作索引是加锁粒度很大，互相互斥。

引入了三层概念后，写操作只会操作 Mutable。Redo 操作之后操作 Base 和 Immutable。Read 操作首先在 Mutable 中查找，其次在 Immutable 中查找，最后在 Base 中查找。总的来说加锁的粒度会变小。

5.3 SearchKey 结构

由 InodeId, ChunkIndex, PageNumber 组成的三元组。IndexTree 的 key 即用 SearchKey 表示。