cnblogs.com

数据分析与处理之二（Leveldb 实现原理） - Haippy - 博客园

Haippy 粉丝 - 794 关注 - 37 +加关注

16-20 minutes

郑重声明：本篇博客是自己学习 Leveldb 实现原理时参考了郎格科技系列博客整理的，原文地址：http://www.samecity.com/blog/Index.asp?SortID=12，只是为了加深印象，本文的配图是自己重新绘制的，大部分内容与原文相似，大家可以浏览原始页面 :-)，感兴趣的话可以一起讨论 Leveldb 的实现原理！

LevelDb日知录之一：LevelDb 101

说起LevelDb也许您不清楚，但是如果作为IT工程师，不知道下面两位大神级别的工程师，那您的领导估计会Hold不住了：Jeff Dean和Sanjay Ghemawat。这两位是Google公司重量级的工程师，为数甚少的Google Fellow之二。

Jeff Dean其人：http://research.google.com/people/jeff/index.html，Google大规模分布式平台Bigtable和MapReduce主要设计和实现者。

Sanjay Ghemawat其人：http://research.google.com/people

/sanjay/index.html，Google大规模分布式平台GFS，Bigtable和MapReduce主要设计和实现工程师。

LevelDb就是这两位大神级别的工程师发起的开源项目，简而言之，LevelDb是能够处理十亿级别规模Key-Value型数据持久性存储的C++ 程序库。正像上面介绍的，这二位是Bigtable的设计和实现者，如果了解Bigtable的话，应该知道在这个影响深远的分布式存储系统中有两个核心的部分：Master Server和Tablet Server。其中Master Server做一些管理数据的存储以及分布式调度工作，实际的分布式数据存储以及读写操作是由Tablet Server完成的，而LevelDb则可以理解为一个简化版的Tablet Server。

LevelDb有如下一些特点：

首先，LevelDb是一个持久化存储的KV系统，和Redis这种内存型的KV系统不同，LevelDb不会像Redis一样狂吃内存，而是将大部分数据存储到磁盘上。

其次，LevleDb在存储数据时，是根据记录的key值有序存储的，就是说相邻的key值在存储文件中是依次顺序存储的，而应用可以自定义key大小比较函数，LevleDb会按照用户定义的比较函数依序存储这些记录。

再次，像大多数KV系统一样，LevelDb的操作接口很简单，基本操作包括写记录，读记录以及删除记录。也支持针对多条操作的原子批量操作。

另外，LevelDb支持数据快照（snapshot）功能，使得读取操作不受写操作影响，可以在读操作过程中始终看到一致的数据。

除此外，LevelDb还支持数据压缩等操作，这对于减小存储空间以及增快IO效率都有直接的帮助。

LevelDb性能非常突出，官方网站报道其随机写性能达到40万

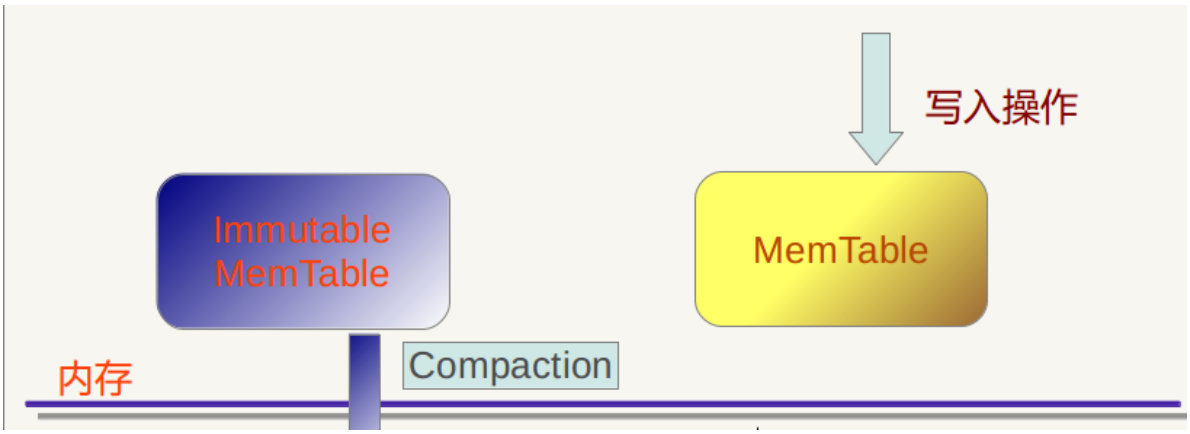
条记录每秒，而随机读性能达到6万条记录每秒。总体来说，LevelDb的写操作要大大快于读操作，而顺序读写操作则大大快于随机读写操作。至于为何是这样，看了我们后续推出的LevelDb日知录，估计您会了解其内在原因。

LevelDb日知录之二：整体架构

LevelDb本质上是一套存储系统以及在这套存储系统上提供的一些操作接口。为了便于理解整个系统及其处理流程，我们可以从两个不同的角度来看待LevleDb：静态角度和动态角度。从静态角度，可以假想整个系统正在运行过程中（不断插入删除读取数据），此时我们给LevelDb照相，从照片可以看到之前系统的数据在内存和磁盘中是如何分布的，处于什么状态等；从动态的角度，主要是了解系统是如何写入一条记录，读出一条记录，删除一条记录的，同时也包括除了这些接口操作外的内部操作比如compaction，系统运行时崩溃后如何恢复系统等等方面。

本节所讲的整体架构主要从静态角度来描述，之后接下来的几节内容会详述静态结构涉及到的文件或者内存数据结构，LevelDb日知录后半部分主要介绍动态视角下的LevelDb，就是说整个系统是怎么运转起来的。

LevelDb作为存储系统，数据记录的存储介质包括内存以及磁盘文件，如果像上面说的，当LevelDb运行了一段时间，此时我们给LevelDb进行透视拍照，那么您会看到如下一番景象：



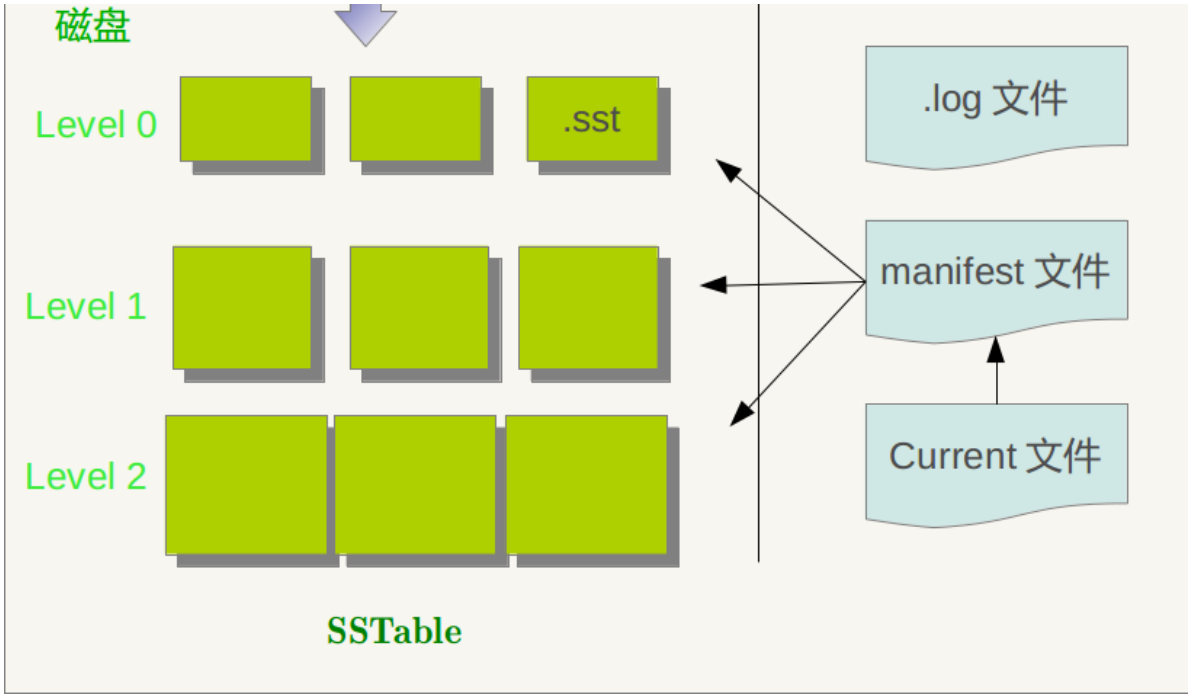


图1.1：LevelDb结构

从图中可以看出，构成LevelDb静态结构的包括六个主要部分：内存中的MemTable和Immutable MemTable以及磁盘上的几种主要文件：Current文件，Manifest文件，log文件以及SSTable文件。当然，LevelDb除了这六个主要部分还有一些辅助的文件，但是以上六个文件和数据结构是LevelDb的主体构成元素。

LevelDb的Log文件和Memtable与Bigtable论文中介绍的是一致的，当应用写入一条Key:Value记录的时候，LevelDb会先往log文件里写入，成功后将记录插进Memtable中，这样基本就算完成了写入操作，因为一次写入操作只涉及一次磁盘顺序写和一次内存写入，所以这是为何说LevelDb写入速度极快的主要原因。

Log文件在系统中的作用主要是用于系统崩溃恢复而不丢失数据，假如没有Log文件，因为写入的记录刚开始是保存在内存中的，此时如果系统崩溃，内存中的数据还没有来得及Dump到磁盘，所以会丢失数据（Redis就存在这个问题）。为了避免这种情况，LevelDb在写入内存前先将操作记录到Log文件中，然后再记入内存中，这样即使系统崩溃，也可以从Log文件中恢复内存中的

Memtable，不会造成数据的丢失。

当Memtable插入的数据占用内存到了一个界限后，需要将内存的记录导出到外存文件中，LevleDb会生成新的Log文件和Memtable，原先的Memtable就成为Immutable Memtable，顾名思义，就是说这个Memtable的内容是不可更改的，只能读不能写入或者删除。新到来的数据被记入新的Log文件和Memtable，LevelDb后台调度会将Immutable Memtable的数据导出到磁盘，形成一个新的SSTable文件。SSTable就是由内存中的数据不断导出并进行Compaction操作后形成的，而且SSTable的所有文件是一种层级结构，第一层为Level 0，第二层为Level 1，依次类推，层级逐渐增高，这也是为何称之为LevelDb的原因。

SSTable中的文件是Key有序的，就是说在文件中小key记录排在大Key记录之前，各个Level的SSTable都是如此，但是这里需要注意的一点是：Level 0的SSTable文件（后缀为.sst）和其它Level的文件相比有特殊性：这个层级内的.sst文件，两个文件可能存在key重叠，比如有两个level 0的sst文件，文件A和文件B，文件A的key范围是：{bar, car}，文件B的Key范围是{blue,samecity}，那么很可能两个文件都存在key=”blood”的记录。对于其它Level的SSTable文件来说，则不会出现同一层级内.sst文件的key重叠现象，就是说Level L中任意两个.sst文件，那么可以保证它们的key值是不会重叠的。这点需要特别注意，后面您会看到很多操作的差异都是由于这个原因造成的。

SSTable中的某个文件属于特定层级，而且其存储的记录是key有序的，那么必然有文件中的最小key和最大key，这是非常重要的信息，LevelDb应该记下这些信息。Manifest就是干这个的，它记载了SSTable各个文件的管理信息，比如属于哪个Level，文件名称叫啥，最小key和最大key各自是多少。下图是Manifest所存储内容的示意：

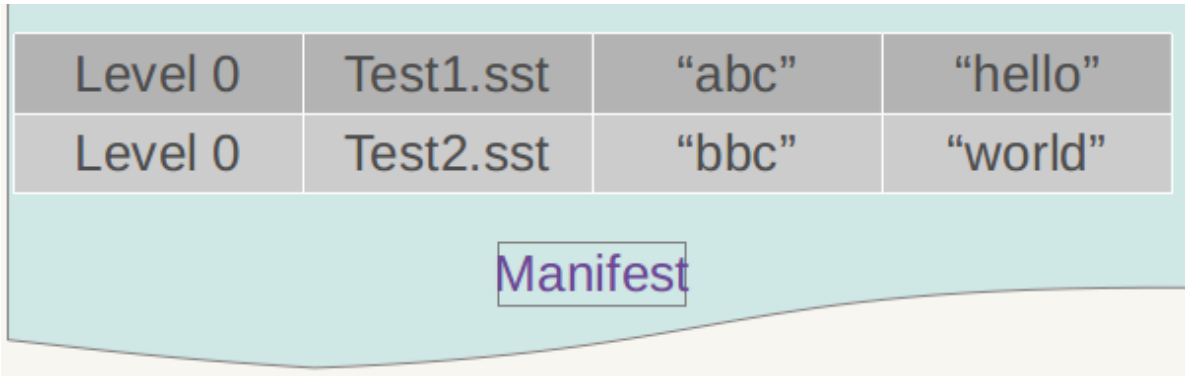


图2.1：Manifest存储示意图

图中只显示了两个文件（manifest会记载所有SSTable文件的这些信息），即Level 0的test.sst1和test.sst2文件，同时记载了这些文件各自对应的key范围，比如test.sstt1的key范围是“an”到 “banana”，而文件test.sst2的key范围是“baby”到“samecity”，可以看出两者的key范围是有重叠的。

Current文件是干什么的呢？这个文件的内容只有一个信息，就是记载当前的manifest文件名。因为在LevleDb的运行过程中，随着Compaction的进行，SSTable文件会发生变化，会有新的文件产生，老的文件被废弃，Manifest也会跟着反映这种变化，此时往往会新生成Manifest文件来记载这种变化，而Current则用来指出哪个Manifest文件才是我们关心的那个Manifest文件。

以上介绍的内容就构成了LevelDb的整体静态结构，在LevelDb日知录接下来的内容中，我们会首先介绍重要文件或者内存数据的具体数据布局与结构。

LevelDb日知录之三：log文件

上节内容讲到log文件在LevelDb中的主要作用是系统故障恢复时，能够保证不会丢失数据。因为在将记录写入内存的Memtable之前，会先写入Log文件，这样即使系统发生故障，Memtable中的数据没有来得及Dump到磁盘的SSTable文件，LevelDB也可以根据log文件恢复内存的Memtable数据结构内容，不会造成系统丢失数据，

在这点上LevelDb和Bigtable是一致的。

下面我们带大家看看log文件的具体物理和逻辑布局是怎样的，LevelDb对于一个log文件，会把它切割成以32K为单位的物理Block，每次读取的单位以一个Block作为基本读取单位，下图展示的log文件由3个Block构成，所以从物理布局来讲，一个log文件就是由连续的32K大小Block构成的。

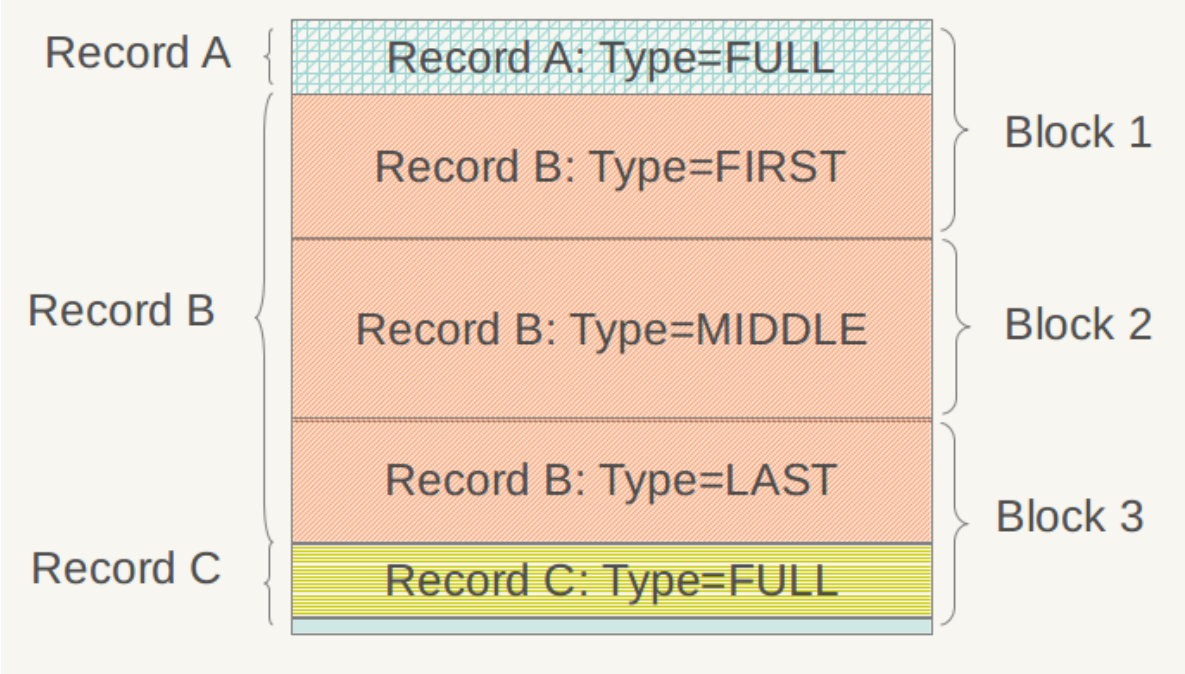


图3.1 log文件布局

在应用的视野里是看不到这些Block的，应用看到的是一系列的Key:Value对，在LevelDb内部，会将一个Key:Value对看做一条记录的数据，另外在这个数据前增加一个记录头，用来记载一些管理信息，以方便内部处理，图3.2显示了一个记录在LevelDb内部是如何表示的。

图3.2 记录结构

记录头包含三个字段，ChechSum是对“类型”和“数据”字段的校验码，为了避免处理不完整或者是被破坏的数据，当LevelDb读取记录数据时候会对数据进行校验，如果发现和存储的CheckSum相同，说明数据完整无破坏，可以继续后续流程。“记录长度”记载了

数据的大小，“数据”则是上面讲的Key:Value数值对，“类型”字段则指出了每条记录的逻辑结构和log文件物理分块结构之间的关系，具体而言，主要有以下四种类型：FULL/FIRST/MIDDLE/LAST。

如果记录类型是FULL，代表了当前记录内容完整地存储在一个物理Block里，没有被不同的物理Block切割开；如果记录被相邻的物理Block切割开，则类型会是其他三种类型中的一种。我们以图3.1所示的例子来具体说明。

假设目前存在三条记录，Record A，Record B和Record C，其中Record A大小为10K，Record B 大小为80K，Record C大小为12K，那么其在log文件中的逻辑布局会如图3.1所示。Record A是图中蓝色区域所示，因为大小为10K<32K，能够放在一个物理Block中，所以其类型为FULL；Record B 大小为80K，而Block 1因为放入了Record A，所以还剩下22K，不足以放下Record B，所以在Block 1的剩余部分放入Record B的开头一部分，类型标识为FIRST，代表了是一个记录的起始部分；Record B还有58K没有存储，这些只能依次放在后续的物理Block里面，因为Block 2大小只有32K，仍然放不下Record B的剩余部分，所以Block 2全部用来放Record B，且标识类型为MIDDLE，意思是这是Record B中间一段数据；Record B剩下的部分可以完全放在Block 3中，类型标识为LAST，代表了这是Record B的末尾数据；图中黄色的Record C因为大小为12K，Block 3剩下的空间足以全部放下它，所以其类型标识为FULL。

从这个小例子可以看出逻辑记录和物理Block之间的关系，LevelDb一次物理读取为一个Block，然后根据类型情况拼接出逻辑记录，供后续流程处理。

LevelDb日知录之四：SSTable文件

SSTable是Bigtable中至关重要的一块，对于LevelDb来说也是

如此，对LevelDb的SSTable实现细节的了解也有助于了解Bigtable中一些实现细节。

本节内容主要讲述SSTable的静态布局结构，我们曾在“LevelDb日知录之二：整体架构”中说过，SSTable文件形成了不同Level的层级结构，至于这个层级结构是如何形成的我们放在后面Compaction一节细说。本节主要介绍SSTable某个文件的物理布局和逻辑布局结构，这对了解LevelDb的运行过程很有帮助。

LevelDb不同层级有很多SSTable文件（以后缀.sst为特征），所有.sst文件内部布局都是一样的。上节介绍Log文件是物理分块的，SSTable也一样会将文件划分为固定大小的物理存储块，但是两者逻辑布局大不相同，根本原因是：Log文件中的记录是Key无序的，即先后记录的key大小没有明确大小关系，而.sst文件内部则是根据记录的Key由小到大排列的，从下面介绍的SSTable布局可以体会到Key有序是为何如此设计.sst文件结构的关键。

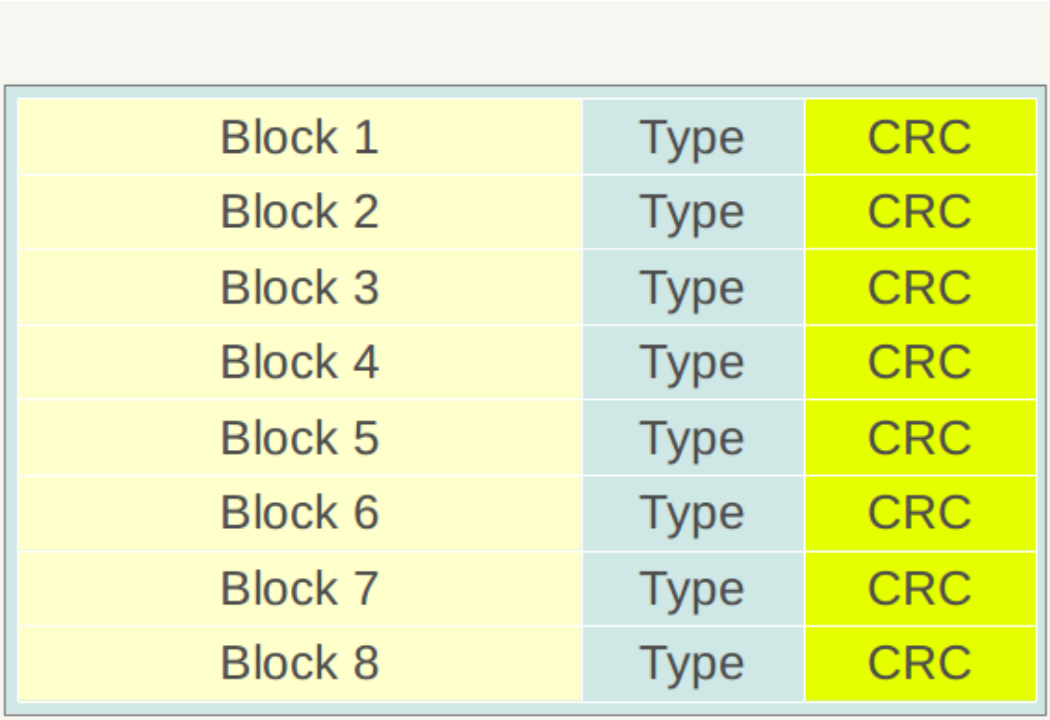


图4.1 .sst文件的分块结构

图4.1展示了一个.sst文件的物理划分结构，同Log文件一样，

也是划分为固定大小的存储块，每个Block分为三个部分，红色部分是数据存储区， 蓝色的Type区用于标识数据存储区是否采用了数据压缩算法（Snappy压缩或者无压缩两种），CRC部分则是数据校验码，用于判别数据是否在生成和传输中出错。

以上是.sst的物理布局，下面介绍.sst文件的逻辑布局，所谓逻辑布局，就是说尽管大家都是物理块，但是每一块存储什么内容，内部又有什么结构等。图4.2展示了.sst文件的内部逻辑解释。

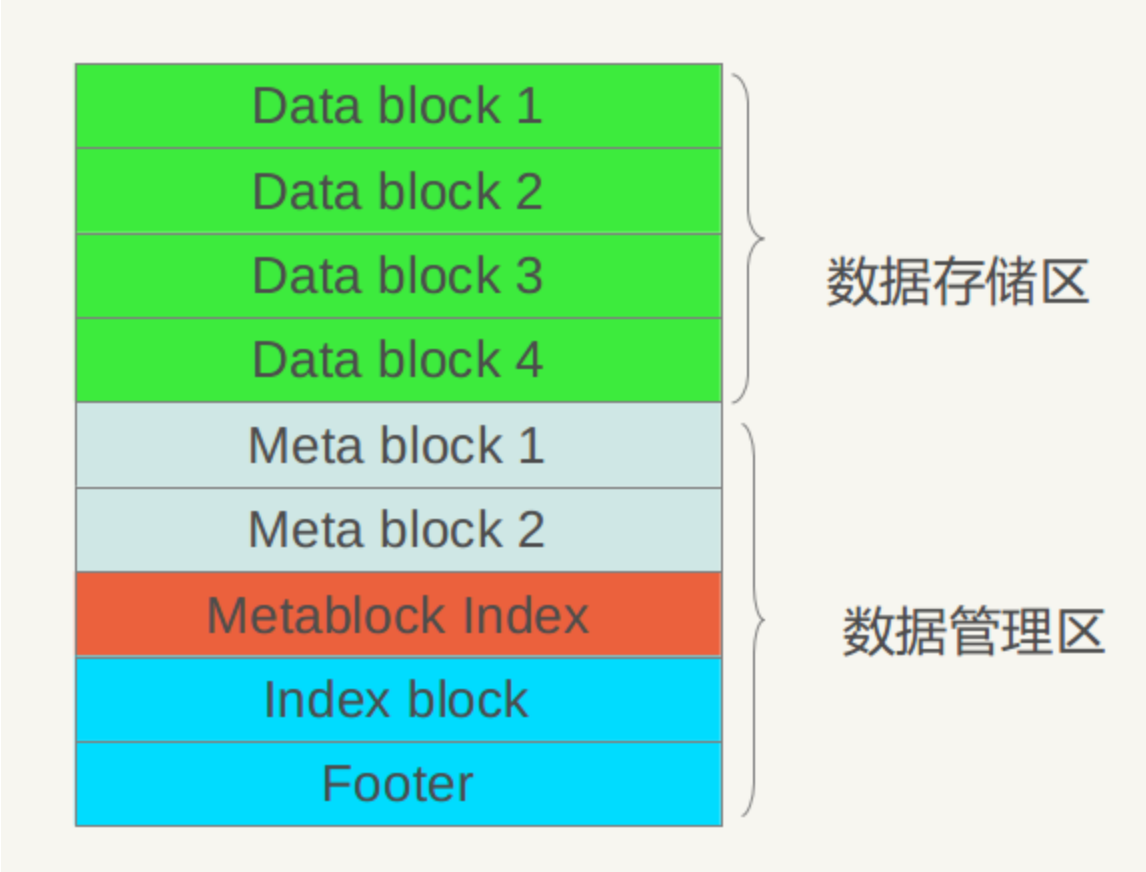


图4.2 逻辑布局

从图4.2可以看出，从大的方面，可以将.sst文件划分为数据存储区和数据管理区，数据存储区存放实际的Key:Value数据，数据管理区则提供一些索引指针等管理数据，目的是更快速便捷的查找相应的记录。两个区域都是在上述的分块基础上的，就是说文件的前面若干块实际存储KV数据，后面数据管理区存储管理数据。管理数据又分为四种不同类型：紫色的Meta Block，红色的MetaBlock 索引和蓝色的数据索引块以及一个文件尾部块。

LevelDb 1.2版对于Meta Block尚无实际使用，只是保留了一个接口，估计会在后续版本中加入内容，下面我们看看数据索引区和文件尾部Footer的内部结构。

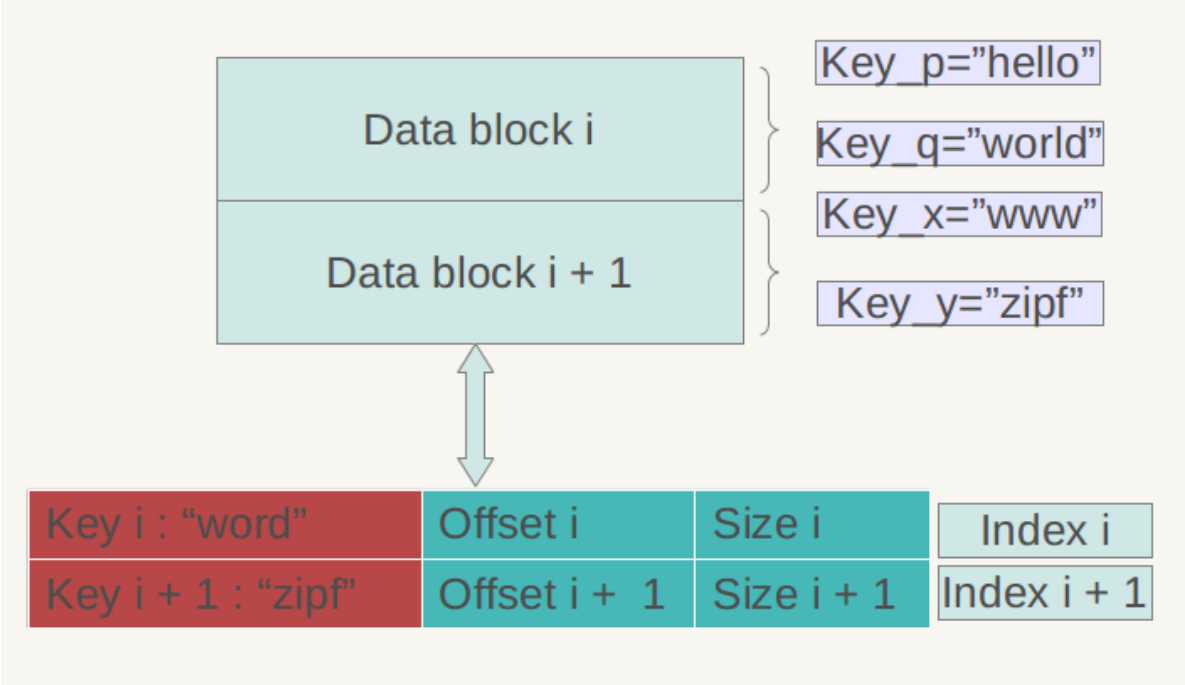


图4.3 数据索引

图4.3是数据索引的内部结构示意图。再次强调一下，Data Block内的KV记录是按照Key由小到大排列的，数据索引区的每条记录是对某个Data Block建立的索引信息，每条索引信息包含三个内容，以图4.3所示的数据块i的索引Index i来说：红色部分的第一个字段记载大于等于数据块i中最大的Key值的那个Key，第二个字段指出数据块i在.sst文件中的起始位置，第三个字段指出Data Block i的大小（有时候是有数据压缩的）。后面两个字段好理解，是用于定位数据块在文件中的位置的，第一个字段需要详细解释一下，在索引里保存的这个Key值未必一定是某条记录的Key,以图4.3的例子来说，假设数据块i 的最小Key="samecity"，最大Key="the best";数据块i+1的最小Key="the fox",最大Key="zoo",那么对于数据块i的索引Index i来说，其第一个字段记载大于等于数据块i的最大Key("the best")同时要小于数据块i+1的最小Key("the fox")，所以例子中Index i的第一个字段是："the c"，这个是满足要求的；而Index i+1的第一

个字段则是“zoo”，即数据块i+1的最大Key。

文件末尾Footer块的内部结构见图4.4，metaindex_handle指出了metaindex block的起始位置和大小；inex_handle指出了index Block的起始地址和大小；这两个字段可以理解为索引的索引，是为了正确读出索引值而设立的，后面跟着一个填充区和魔数。

图4.4 Footer

上面主要介绍的是数据管理区的内部结构，下面我们看看数据区的一个Block的数据部分内部是如何布局的（图4.1中的红色部分），图4.5是其内部布局示意图。

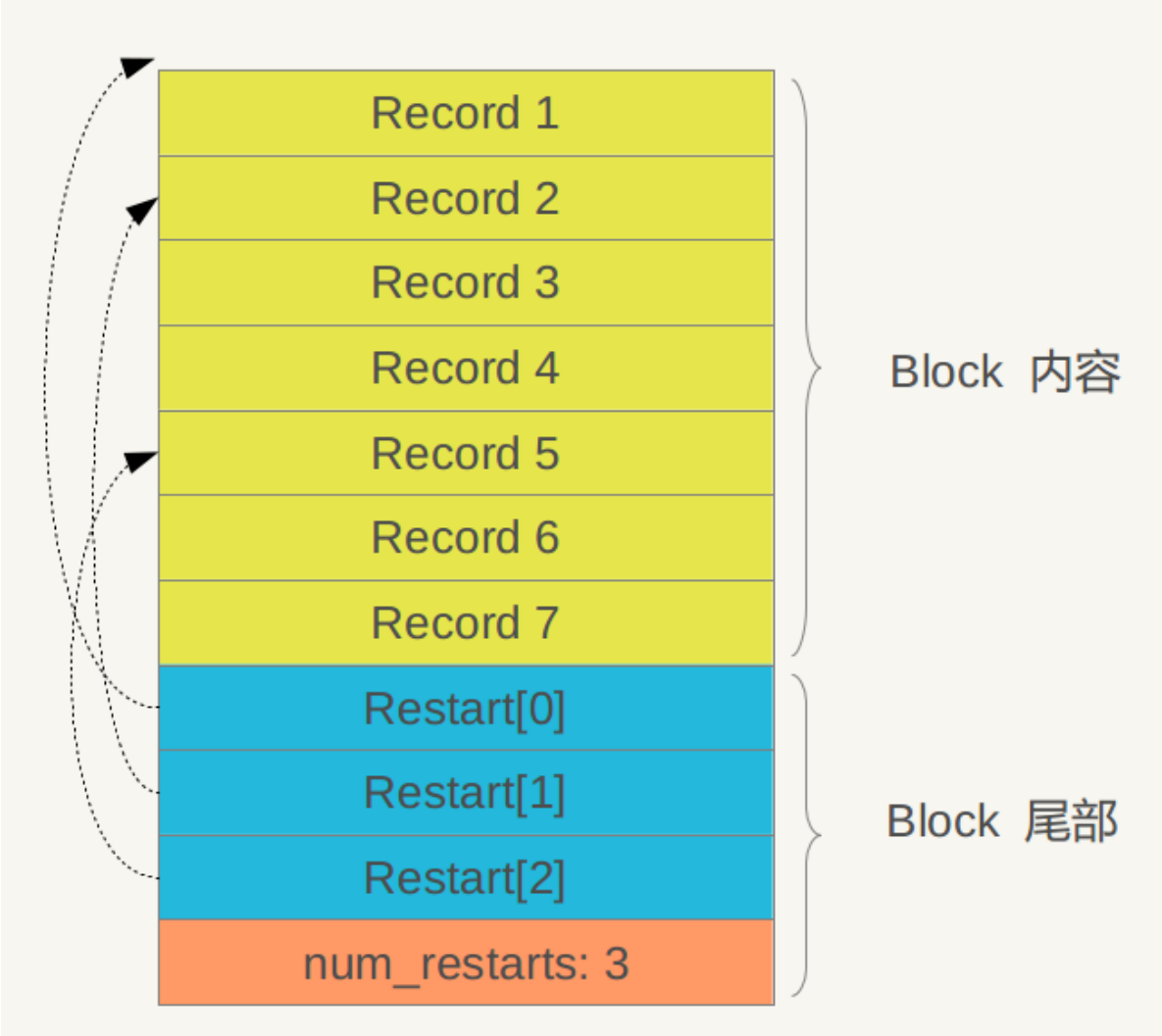


图4.5 数据Block内部结构

从图中可以看出，其内部也分为两个部分，前面是一个个KV记

录，其顺序是根据Key值由小到大排列的，在Block尾部则是一些“重启点”（Restart Point）,其实是一些指针，指出Block内容中的一些记录位置。

“重启点”是干什么的呢？我们一再强调，Block内容里的KV记录是按照Key大小有序的，这样的话，相邻的两条记录很可能Key部分存在重叠，比如key i=“the Car”，Key i+1=“the color”,那么两者存在重叠部分“the c”，为了减少Key的存储量，Key i+1可以只存储和上一条Key不同的部分“olor”，两者的共同部分从Key i中可以获得。记录的Key在Block内容部分就是这么存储的，主要目的是减少存储开销。“重启点”的意思是：在这条记录开始，不再采取只记载不同的Key部分，而是重新记录所有的Key值，假设Key i+1是一个重启点，那么Key里面会完整存储“the color”，而不是采用简略的“olor”方式。Block尾部就是指出哪些记录是这些重启点的。

图4.6 记录格式

在Block内容区，每个KV记录的内部结构是怎样的？图4.6给出了其详细结构，每个记录包含5个字段：key共享长度，比如上面的“olor”记录， 其key和上一条记录共享的Key部分长度是“the c”的长度，即5；key非共享长度，对于“olor”来说，是4；value长度指出Key:Value中Value的长度，在后面的Value内容字段中存储实际的Value值；而key非共享内容则实际存储“olor”这个Key字符串。

上面讲的这些就是.sst文件的全部内部奥秘。

LevelDb日知录之五：MemTable详解

LevelDb日知录前述小节大致讲述了磁盘文件相关的重要静态

结构，本小节讲述内存中的数据结构Memtable，Memtable在整个体系中的重要地位也不言而喻。总体而言，所有KV数据都是存储在Memtable，Immutable Memtable和SSTable中的，Immutable Memtable从结构上讲和Memtable是完全一样的，区别仅仅在于其是只读的，不允许写入操作，而Memtable则是允许写入和读取的。当Memtable写入的数据占用内存到达指定数量，则自动转换为Immutable Memtable，等待Dump到磁盘中，系统会自动生成新的Memtable供写操作写入新数据，理解了Memtable，那么Immutable Memtable自然不在话下。

LevelDb的MemTable提供了将KV数据写入，删除以及读取KV记录的操作接口，但是事实上Memtable并不存在真正的删除操作，删除某个Key的Value在Memtable内是作为插入一条记录实施的，但是会打上一个Key的删除标记，真正的删除操作是Lazy的，会在以后的Compaction过程中去掉这个KV。

需要注意的是，LevelDb的Memtable中KV对是根据Key大小有序存储的，在系统插入新的KV时，LevelDb要把这个KV插到合适的位置上以保持这种Key有序性。其实，LevelDb的Memtable类只是一个接口类，真正的操作是通过背后的SkipList来做的，包括插入操作和读取操作等，所以Memtable的核心数据结构是一个SkipList。

SkipList是由William Pugh发明。他在Communications of the ACM June 1990, 33(6) 668-676 发表了Skip lists: a probabilistic alternative to balanced trees，在该论文中详细解释了SkipList的数据结构和插入删除操作。

SkipList是平衡树的一种替代数据结构，但是和红黑树不相同的是，SkipList对于树的平衡的实现是基于一种随机化的算法的，这样也就是说SkipList的插入和删除的工作是比较简单的。

关于SkipList的详细介绍可以参考这篇文章：http://www.cnblogs.com/xuqiang/archive/2011/05/22

/2053516.html，讲述的很清楚，LevelDb的SkipList基本上是一个具体实现，并无特殊之处。

SkipList不仅是维护有序数据的一个简单实现，而且相比较平衡树来说，在插入数据的时候可以避免频繁的树节点调整操作，所以写入效率是很高的，LevelDb整体而言是个高写入系统，SkipList在其中应该也起到了很重要的作用。Redis为了加快插入操作，也使用了SkipList来作为内部实现数据结构。

LevelDb日知录之六 写入与删除记录

在之前的五节LevelDb日知录中，我们介绍了LevelDb的一些静态文件及其详细布局，从本节开始，我们看看LevelDb的一些动态操作，比如读写记录，Compaction，错误恢复等操作。

本节介绍levelDb的记录更新操作，即插入一条KV记录或者删除一条KV记录。levelDb的更新操作速度是非常快的，源于其内部机制决定了这种更新操作的简单性。

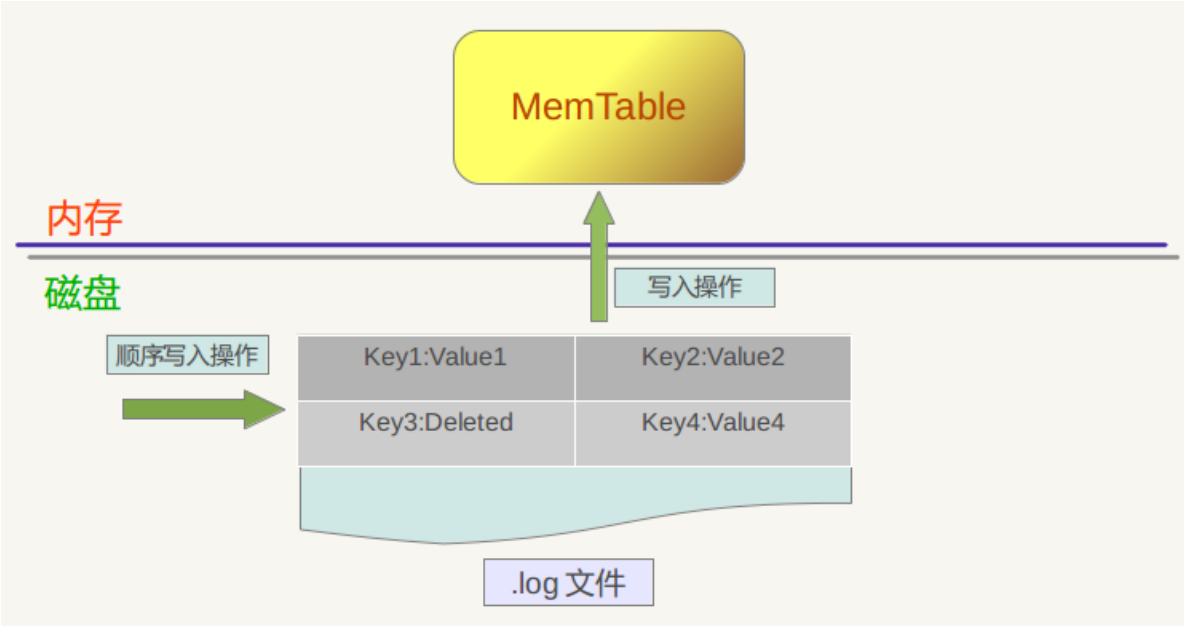


图6.1 LevelDb写入记录

图6.1是levelDb如何更新KV数据的示意图，从图中可以看出，对于一个插入操作Put(Key,Value)来说，完成插入操作包含两个具体

步骤：首先是将这条KV记录以顺序写的方式追加到之前介绍过的log文件末尾，因为尽管这是一个磁盘读写操作，但是文件的顺序追加写入效率是很高的，所以并不会导致写入速度的降低；第二个步骤是:如果写入log文件成功，那么将这条KV记录插入内存中的Memtable中，前面介绍过，Memtable只是一层封装，其内部其实是一个Key有序的SkipList列表，插入一条新记录的过程也很简单，即先查找合适的插入位置，然后修改相应的链接指针将新记录插入即可。完成这一步，写入记录就算完成了，所以一个插入记录操作涉及一次磁盘文件追加写和内存SkipList插入操作，这是为何levelDb写入速度如此高效的根本原因。

从上面的介绍过程中也可以看出：log文件内是key无序的，而Memtable中是key有序的。那么如果是删除一条KV记录呢？对于levelDb来说，并不存在立即删除的操作，而是与插入操作相同的，区别是，插入操作插入的是Key:Value 值，而删除操作插入的是“Key:删除标记”，并不真正去删除记录，而是后台Compaction的时候才去做真正的删除操作。

levelDb的写入操作就是如此简单。真正的麻烦在后面将要介绍的读取操作中。

LevelDb日知录之七：读取记录

LevelDb是针对大规模Key/Value数据的单机存储库，从应用的角度来看，LevelDb就是一个存储工具。而作为称职的存储工具，常见的调用接口无非是新增KV，删除KV，读取KV，更新Key对应的Value值这么几种操作。LevelDb的接口没有直接支持更新操作的接口，如果需要更新某个Key的Value,你可以选择直接生猛地插入新的KV，保持Key相同，这样系统内的key对应的value就会被更新；或者你可以先删除旧的KV， 之后再插入新的KV，这样比较委婉地完成KV的更新操作。

假设应用提交一个Key值，下面我们看看LevelDb是如何从存储的数据中读出其对应的Value值的。图7-1是LevelDb读取过程的整体示意图。

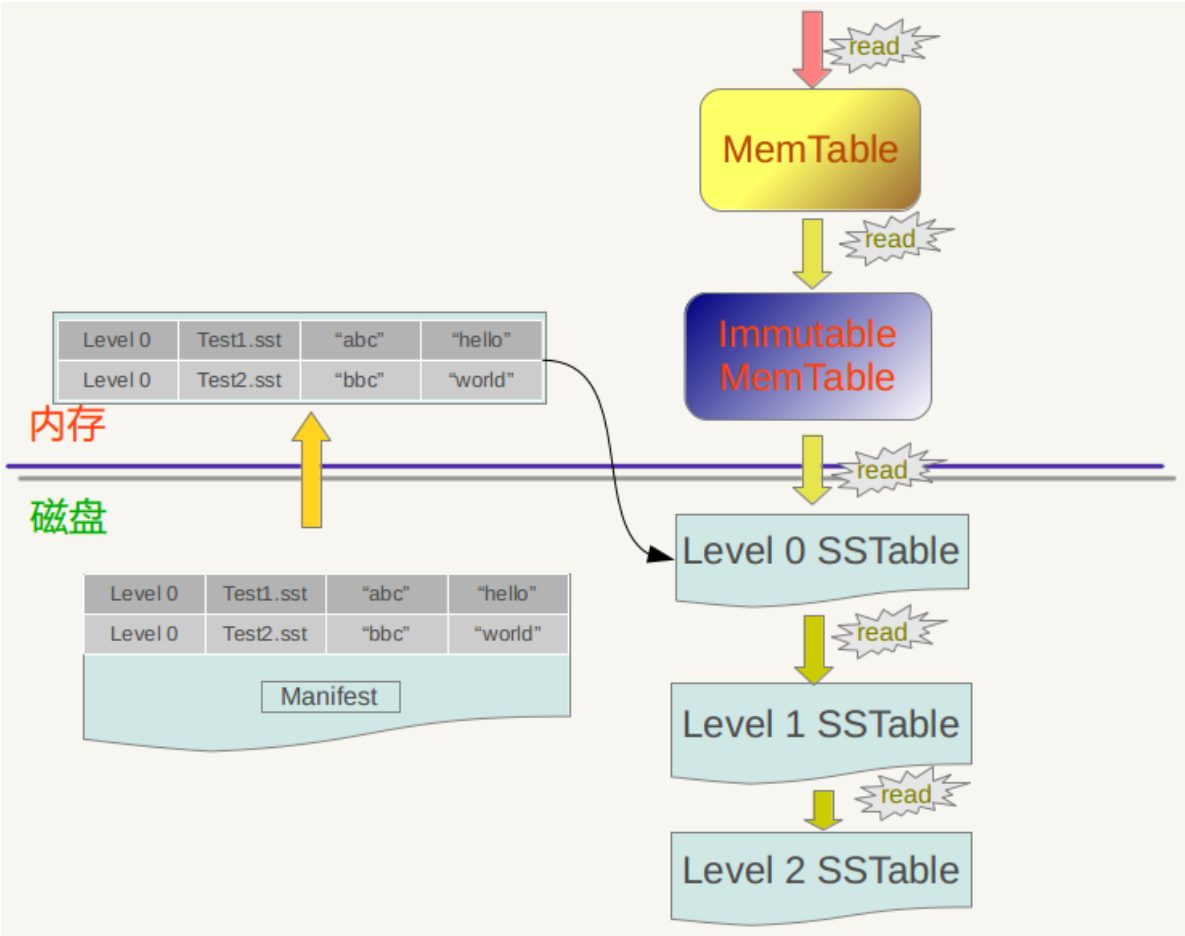


图7-1 LevelDb读取记录流程

LevelDb首先会去查看内存中的Memtable，如果Memtable中包含key及其对应的value，则返回value值即可；如果在Memtable没有读到key，则接下来到同样处于内存中的Immutable Memtable中去读取，类似地，如果读到就返回，若是没有读到,那么只能万般无奈下从磁盘中的大量SSTable文件中查找。因为SSTable数量较多，而且分成多个Level，所以在SSTable中读数据是相当蜿蜒曲折的一段旅程。总的读取原则是这样的：首先从属于level 0的文件中查找，如果找到则返回对应的value值，如果没有找到那么到level 1中的文件中去找，如此循环往复，直到在某层SSTable文件中找到这个key对应的value为止（或者查到最高level，查找失败，说明整个

系统中不存在这个Key)。

那么为什么是从Memtable到Immutable Memtable，再从Immutable Memtable到文件，而文件中为何是从低level到高level这么一个查询路径呢？道理何在？之所以选择这么个查询路径，是因为从信息的更新时间来说，很明显Memtable存储的是最新鲜的KV对；Immutable Memtable中存储的KV数据对的新鲜程度次之；而所有SSTable文件中的KV数据新鲜程度一定不如内存中的Memtable和Immutable Memtable的。对于SSTable文件来说，如果同时在level L和Level L+1找到同一个key，level L的信息一定比level L+1的要新。也就是说，上面列出的查找路径就是按照数据新鲜程度排列出来的，越新鲜的越先查找。

为啥要优先查找新鲜的数据呢？这个道理不言而喻，举个例子。比如我们先往levelDb里面插入一条数据{key="www.samecity.com" value="我们"},过了几天，samecity网站改名为：69同城，此时我们插入数据{key="www.samecity.com" value="69同城"}，同样的key,不同的value；逻辑上理解好像levelDb中只有一个存储记录，即第二个记录，但是在levelDb中很可能存在两条记录，即上面的两个记录都在levelDb中存储了，此时如果用户查询key="www.samecity.com",我们当然希望找到最新的更新记录，也就是第二个记录返回，这就是为何要优先查找新鲜数据的原因。

前文有讲：对于SSTable文件来说，如果同时在level L和Level L+1找到同一个key，level L的信息一定比level L+1的要新。这是一个结论，理论上需要一个证明过程，否则会招致如下的问题：为神马呢？从道理上讲呢，很明白：因为Level L+1的数据不是从石头缝里蹦出来的，也不是做梦梦到的，那它是从哪里来的？Level L+1的数据是从Level L 经过Compaction后得到的（如果您不知道什么是Compaction，那么.....也许以后会知道的），也就是说，您看到的现在的Level L+1层的SSTable数据是从原来的Level L中来的，现在

的Level L比原来的Level L数据要新鲜，所以可证，现在的Level L比现在的Level L+1的数据要新鲜。

SSTable文件很多，如何快速地找到key对应的value值？在LevelDb中，level 0一直都爱搞特殊化，在level 0和其它level中查找某个key的过程是不一样的。因为level 0下的不同文件可能key的范围有重叠，某个要查询的key有可能多个文件都包含，这样的话LevelDb的策略是先找出level 0中哪些文件包含这个key（manifest文件中记载了level和对应的文件及文件里key的范围信息，LevelDb在内存中保留这种映射表）， 之后按照文件的新鲜程度排序，新的文件排在前面，之后依次查找，读出key对应的value。而如果是非level 0的话，因为这个level的文件之间key是不重叠的，所以只从一个文件就可以找到key对应的value。

最后一个问题,如果给定一个要查询的key和某个key range包含这个key的SSTable文件，那么levelDb是如何进行具体查找过程的呢？levelDb一般会先在内存中的Cache中查找是否包含这个文件的缓存记录，如果包含，则从缓存中读取；如果不包含，则打开SSTable文件，同时将这个文件的索引部分加载到内存中并放入Cache中。 这样Cache里面就有了这个SSTable的缓存项，但是只有索引部分在内存中，之后levelDb根据索引可以定位到哪个内容Block会包含这条key，从文件中读出这个Block的内容，在根据记录一一比较，如果找到则返回结果，如果没有找到，那么说明这个level的SSTable文件并不包含这个key，所以到下一级别的SSTable中去查找。

从之前介绍的LevelDb的写操作和这里介绍的读操作可以看出，相对写操作，读操作处理起来要复杂很多，所以写的速度必然要远远高于读数据的速度，也就是说，LevelDb比较适合写操作多于读操作的应用场合。而如果应用是很多读操作类型的，那么顺序读取效率会比较高，因为这样大部分内容都会在缓存中找到，尽可能避免大量的随机读取操作。

LevelDb日知录之八：Compaction操作

前文有述，对于LevelDb来说，写入记录操作很简单，删除记录仅仅写入一个删除标记就算完事，但是读取记录比较复杂，需要在内存以及各个层级文件中依照新鲜程度依次查找，代价很高。为了加快读取速度，levelDb采取了compaction的方式来对已有的记录进行整理压缩，通过这种方式，来删除掉一些不再有效的KV数据，减小数据规模，减少文件数量等。

levelDb的compaction机制和过程与Bigtable所讲述的是基本一致的，Bigtable中讲到三种类型的compaction: minor ， major和full。所谓minor Compaction，就是把memtable中的数据导出到SSTable文件中；major compaction就是合并不同层级的SSTable文件，而full compaction就是将所有SSTable进行合并。

LevelDb包含其中两种，minor和major。

我们将为大家详细叙述其机理。

先来看看minor Compaction的过程。Minor compaction 的目的是当内存中的memtable大小到了一定值时，将内容保存到磁盘文件中，图8.1是其机理示意图。

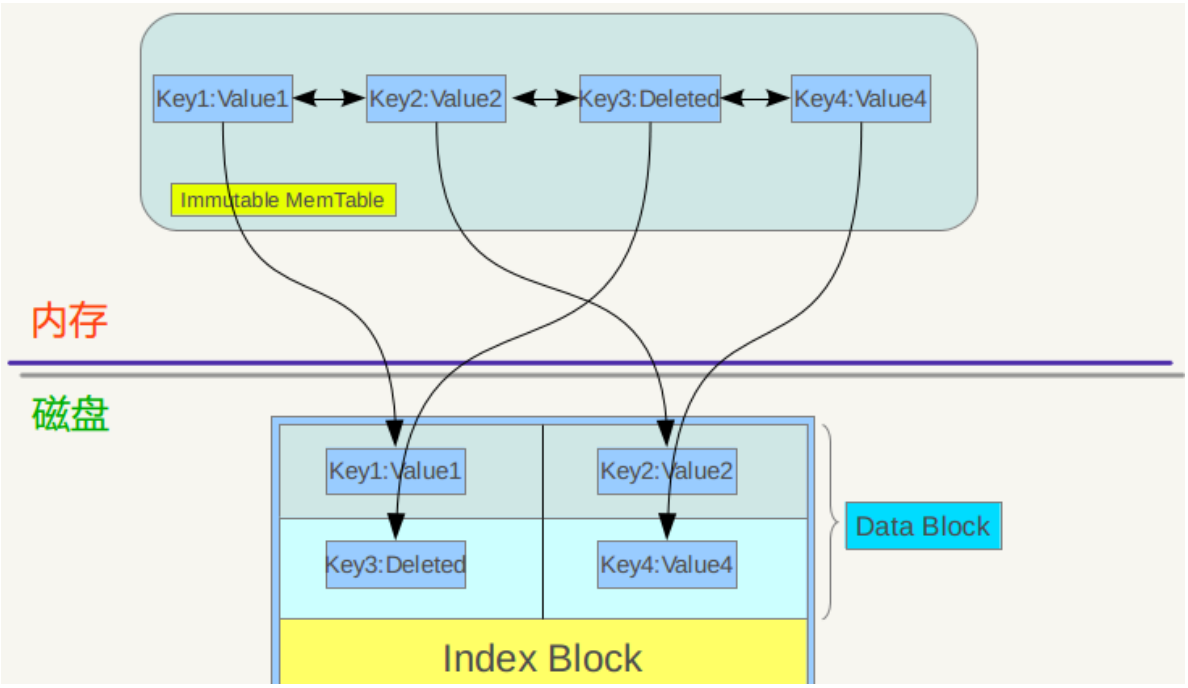


图8.1 minor compaction

从8.1可以看出，当memtable数量到了一定程度会转换为immutable memtable，此时不能往其中写入记录，只能从中读取KV内容。之前介绍过，immutable memtable其实是一个多层级队列SkipList，其中的记录是根据key有序排列的。所以这个minor compaction实现起来也很简单，就是按照immutable memtable中记录由小到大遍历，并依次写入一个level 0 的新建SSTable文件中，写完后建立文件的index 数据，这样就完成了一次minor compaction。从图中也可以看出，对于被删除的记录，在minor compaction过程中并不真正删除这个记录，原因也很简单，这里只知道要删掉key记录，但是这个KV数据在哪里?那需要复杂的查找，所以在minor compaction的时候并不做删除，只是将这个key作为一个记录写入文件中，至于真正的删除操作，在以后更高层级的compaction中会去做。

当某个level下的SSTable文件数目超过一定设置值后，levelDb会从这个level的SSTable中选择一个文件（level>0），将其和高一层级的level+1的SSTable文件合并，这就是major compaction。

我们知道在大于0的层级中，每个SSTable文件内的Key都是由小到大有序存储的，而且不同文件之间的key范围（文件内最小key和最大key之间）不会有任何重叠。Level 0的SSTable文件有些特殊，尽管每个文件也是根据Key由小到大排列，但是因为level 0的文件是通过minor compaction直接生成的，所以任意两个level 0下的两个sstable文件可能再key范围上有重叠。所以在做major compaction的时候，对于大于level 0的层级，选择其中一个文件就行，但是对于level 0来说，指定某个文件后，本level中很可能有其他SSTable文件的key范围和这个文件有重叠，这种情况下，要找出

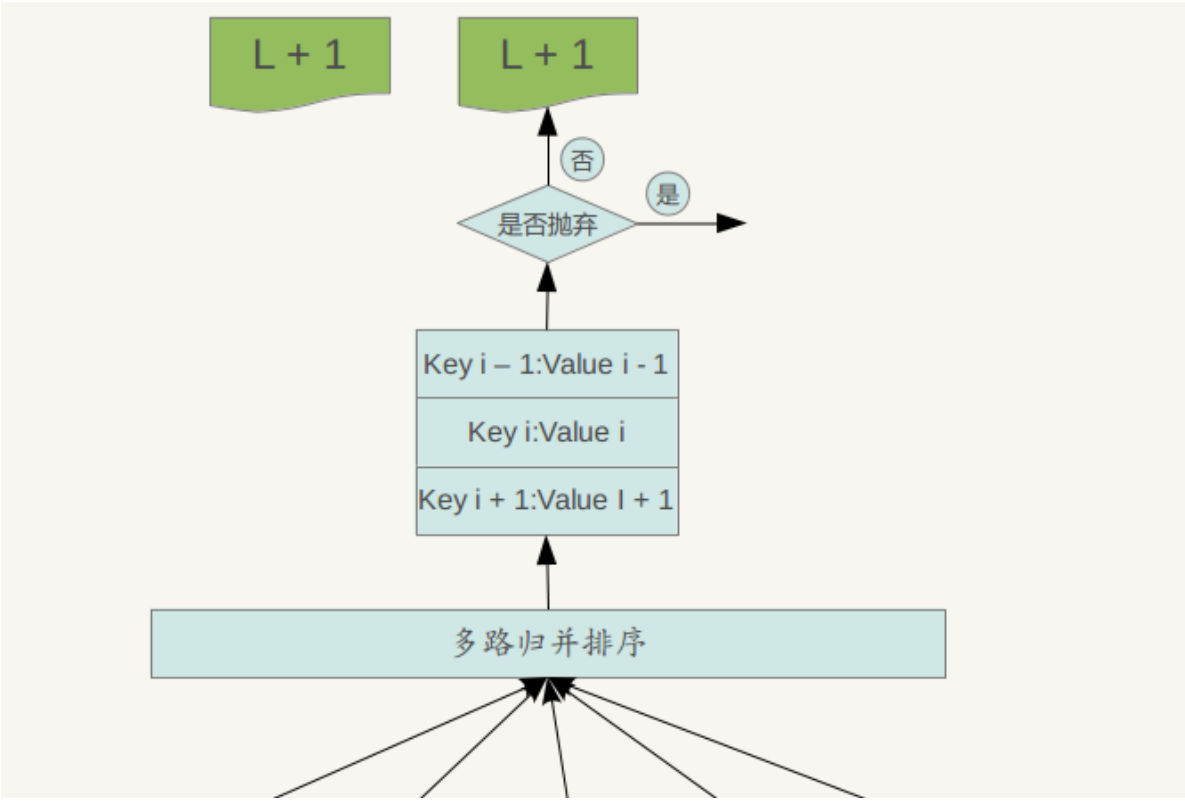
所有有重叠的文件和level 1的文件进行合并，即level 0在进行文件选择的时候，可能会有多个文件参与major compaction。

levelDb在选定某个level进行compaction后，还要选择是具体哪个文件要进行compaction，levelDb在这里有个小技巧， 就是说轮流来，比如这次是文件A进行compaction，那么下次就是在key range上紧挨着文件A的文件B进行compaction，这样每个文件都会有机会轮流和高层的level 文件进行合并。

如果选好了level L的文件A和level L+1层的文件进行合并，那么问题又来了，应该选择level L+1哪些文件进行合并？levelDb选择L+1层中和文件A在key range上有重叠的所有文件来和文件A进行合并。

也就是说，选定了level L的文件A,之后在level L+1中找到了所有需要合并的文件B,C,D…..等等。剩下的问题就是具体是如何进行major 合并的？就是说给定了一系列文件，每个文件内部是key有序的，如何对这些文件进行合并，使得新生成的文件仍然Key有序，同时抛掉哪些不再有价值的KV 数据。

图8.2说明了这一过程。



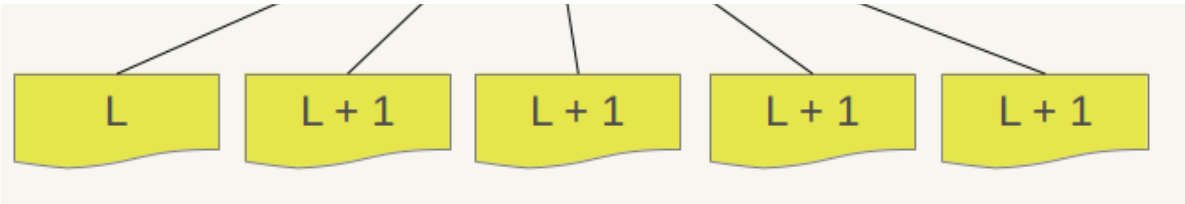


图8.2 SSTable Compaction

Major compaction的过程如下：对多个文件采用多路归并排序的方式，依次找出其中最小的Key记录，也就是对多个文件中的所有记录重新进行排序。之后采取一定的标准判断这个Key是否还需要保存，如果判断没有保存价值，那么直接抛掉，如果觉得还需要继续保存，那么就将其写入level L+1层中新生成的一个SSTable文件中。就这样对KV数据一一处理，形成了一系列新的L+1层数据文件，之前的L层文件和L+1层参与compaction 的文件数据此时已经没有意义了，所以全部删除。这样就完成了L层和L+1层文件记录的合并过程。

那么在major compaction过程中，判断一个KV记录是否抛弃的标准是什么呢？其中一个标准是:对于某个key来说，如果在小于L层中存在这个Key，那么这个KV在major compaction过程中可以抛掉。因为我们前面分析过，对于层级低于L的文件中如果存在同一Key的记录，那么说明对于Key来说，有更新鲜的Value存在，那么过去的Value就等于没有意义了，所以可以删除。

LevelDb日知录之九 levelDb中的Cache

书接前文，前面讲过对于levelDb来说，读取操作如果没有在内存的memtable中找到记录，要多次进行磁盘访问操作。假设最优情况，即第一次就在level 0中最新的文件中找到了这个key，那么也需要读取2次磁盘，一次是将SSTable的文件中的index部分读入内存，这样根据这个index可以确定key是在哪个block中存储；第二次是读入这个block的内容，然后在内存中查找key对应的value。

levelDb中引入了两个不同的Cache:Table Cache和Block

Cache。其中Block Cache是配置可选的，即在配置文件中指定是否打开这个功能。

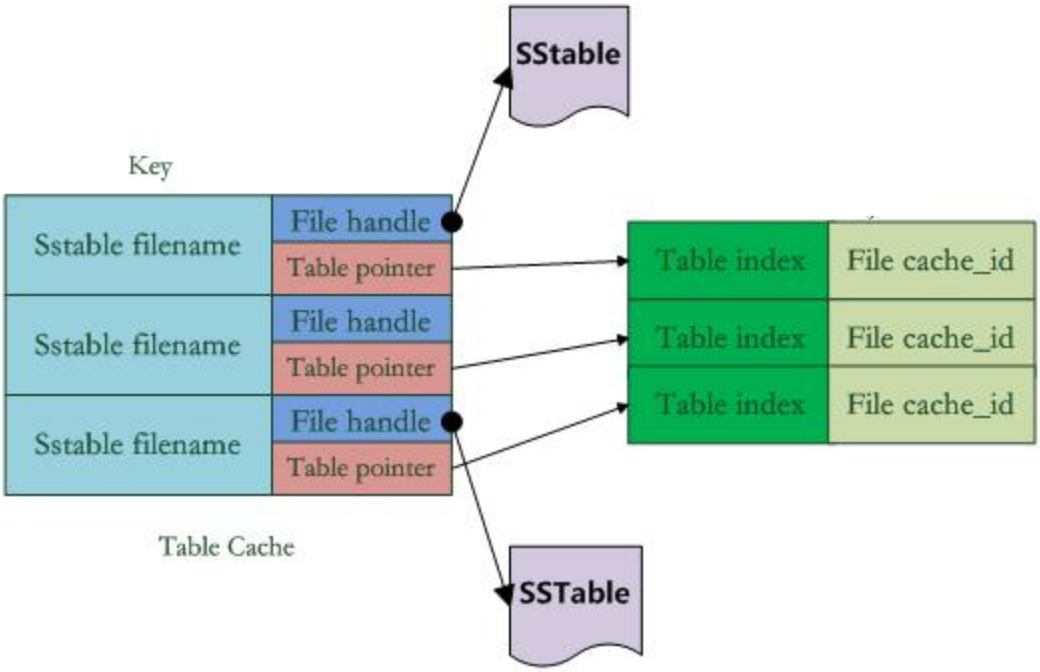


图9.1 table cache

图9.1是table cache的结构。在Cache中，key值是SSTable的文件名称，Value部分包含两部分，一个是指向磁盘打开的SSTable文件的文件指针，这是为了方便读取内容；另外一个是指向内存中这个SSTable文件对应的Table结构指针，table结构在内存中，保存了SSTable的index内容以及用来指示block cache用的cache_id ,当然除此外还有其它一些内容。

比如在get(key)读取操作中，如果levelDb确定了key在某个level下某个文件A的key range范围内，那么需要判断是不是文件A真的包含这个KV。此时，levelDb会首先查找Table Cache，看这个文件是否在缓存里，如果找到了，那么根据index部分就可以查找是哪个block包含这个key。如果没有在缓存中找到文件，那么打开SSTable文件，将其index部分读入内存，然后插入Cache里面，去index里面定位哪个block包含这个Key 。如果确定了文件哪个block包含这个key，那么需要读入block内容，这是第二次读取。

图9.2 block cache

Block Cache是为了加快这个过程的，图9.2是其结构示意图。其中的key是文件的cache_id加上这个block在文件中的起始位置block_offset。而value则是这个Block的内容。

如果levelDb发现这个block在block cache中，那么可以避免读取数据，直接在cache里的block内容里面查找key的value就行，如果没找到呢？那么读入block内容并把它插入block cache中。levelDb就是这样通过两个cache来加快读取速度的。从这里可以看出，如果读取的数据局部性比较好，也就是说要读的数据大部分在cache里面都能读到，那么读取效率应该还是很高的，而如果是对key进行顺序读取效率也应该不错，因为一次读入后可以多次被复用。但是如果是随机读取，您可以推断下其效率如何。

LevelDb日知录之十 Version、VersionEdit、VersionSet

Version 保存了当前磁盘以及内存中所有的文件信息，一般只有一个Version叫做"current" version（当前版本）。Leveldb还保存了一系列的历史版本，这些历史版本有什么作用呢?

当一个Iterator创建后，Iterator就引用到了current version(当前版本)，只要这个Iterator不被delete那么被Iterator引用的版本就会一直存活。这就意味着当你用完一个Iterator后，需要及时删除它。

当一次Compaction结束后（会生成新的文件，合并前的文件需要删除），Leveldb会创建一个新的版本作为当前版本，原先的当前版本就会变为历史版本。

VersionSet 是所有Version的集合，管理着所有存活的Version。

VersionEdit 表示Version之间的变化，相当于delta 增量，表示

有增加了多少文件，删除了文件。下图表示他们之间的关系。

Version0 +VersionEdit-->Version1

VersionEdit会保存到MANIFEST文件中，当做数据恢复时就会从MANIFEST文件中读出来重建数据。

leveldb的这种版本的控制，让我想到了双buffer切换，双buffer切换来自于图形学中，用于解决屏幕绘制时的闪屏问题，在服务器编程中也有用处。

比如我们的服务器上有一个字典库，每天我们需要更新这个字典库，我们可以新开一个buffer，将新的字典库加载到这个新buffer中，等到加载完毕，将字典的指针指向新的字典库。

leveldb的version管理和双buffer切换类似，但是如果原version被某个iterator引用，那么这个version会一直保持，直到没有被任何一个iterator引用，此时就可以删除这个version。

注：博文参考了郎格科技博客：http://www.samecity.com/blog/Index.asp?SortID=12