/*文章来自朗格科技 */

LevelDb 日知录之一：初识 LevelDb

LevelDb 日知录之一：初识 LevelDb

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，看了朗格科技后续推出的 LevelDb 日知录，估计您会了解其内在原因。

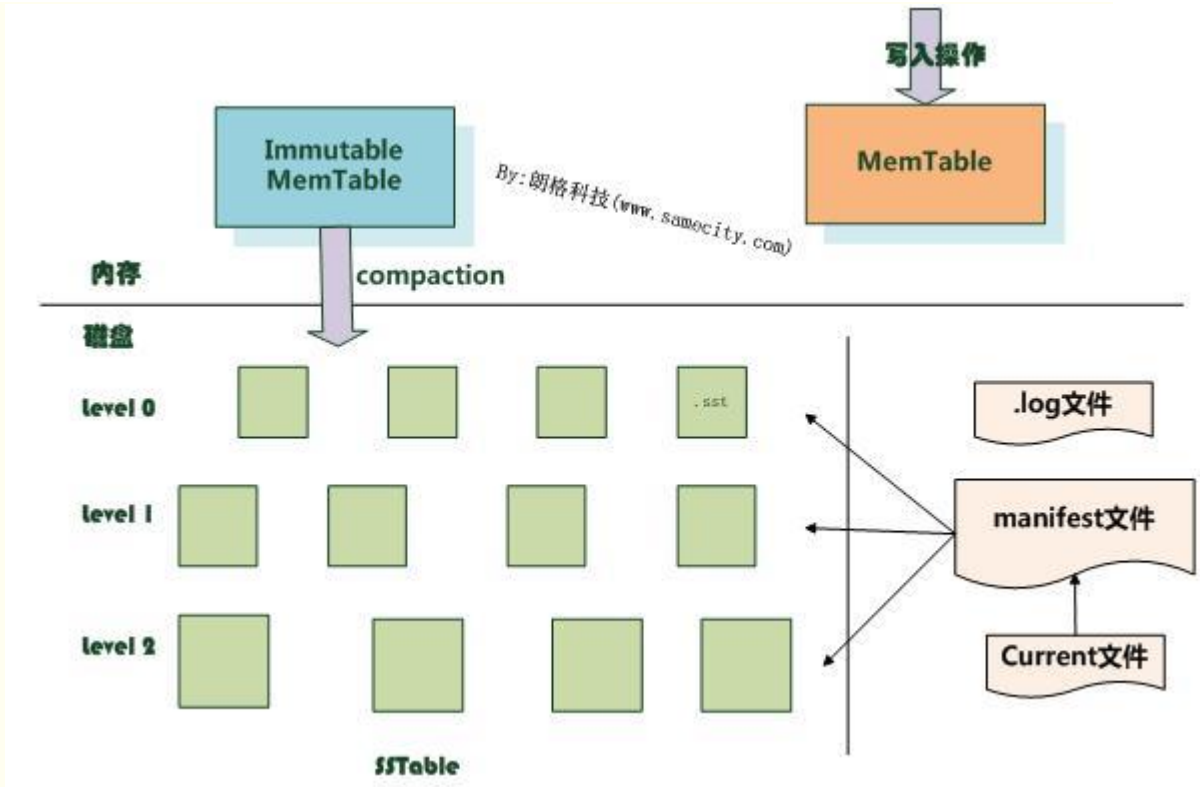
LevelDb 日知录之二：整体架构

LevelDb 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：



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

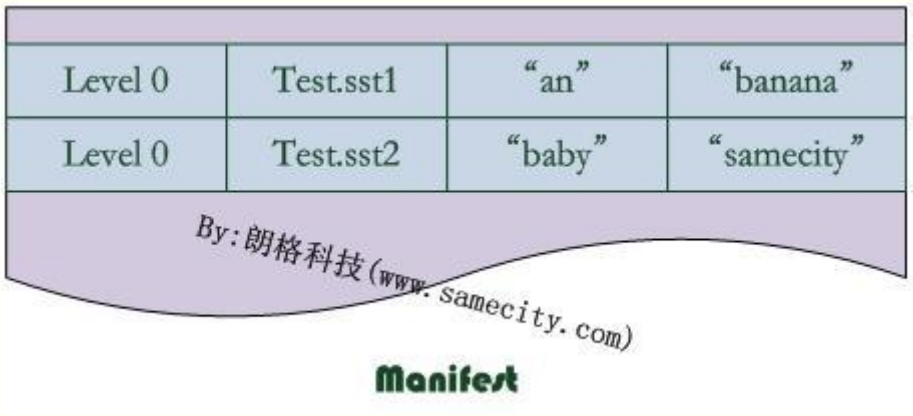
系统崩溃，内存中的数据还没有来得及 Dump 到磁盘，所以会丢失数据（Redis 就存在这个问题）。为了避免这种情况，LevelDb 在写入内存前先将操作记录到 Log 文件中，然后再记入内存中，这样即使系统崩溃，也可以从 Log 文件中恢复内存中的 Memtable，不会造成数据的丢失。

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，朗格科技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

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。

下面朗格科技带大家看看 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。

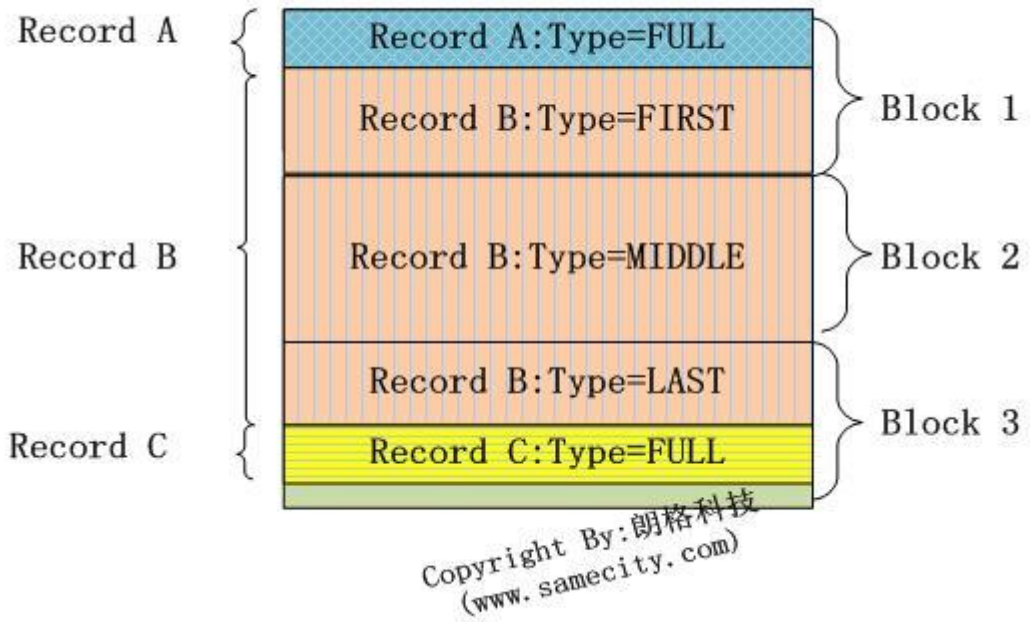


图 3.1 log 文件布局

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

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，朗格科技曾在“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。

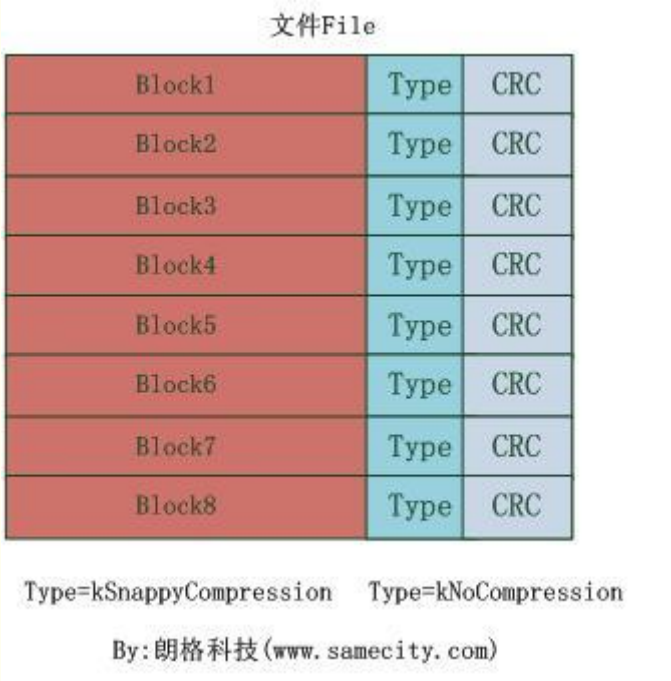


图 4.1 .sst 文件的分块结构

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。

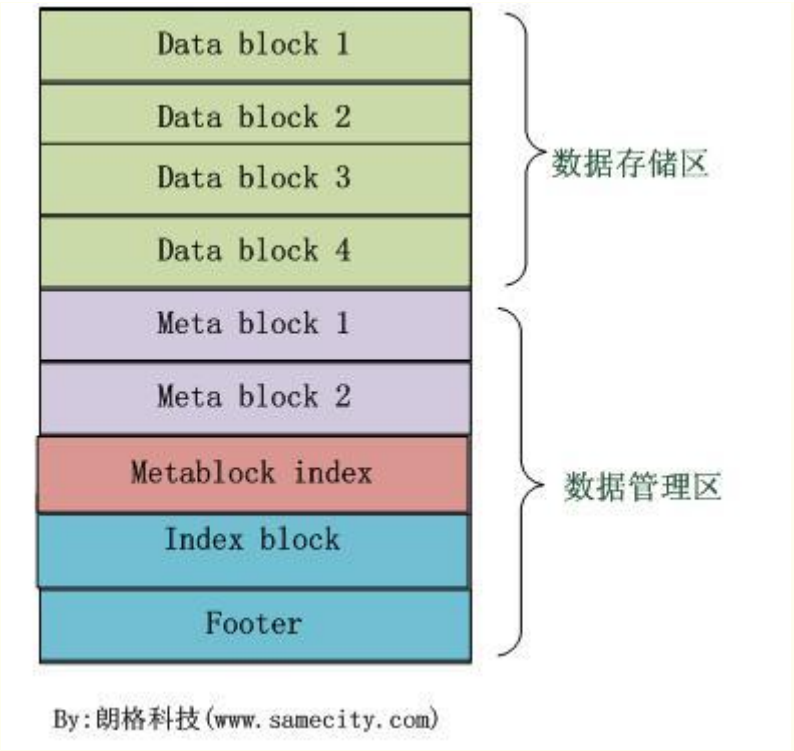


图 4.2 逻辑布局

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。

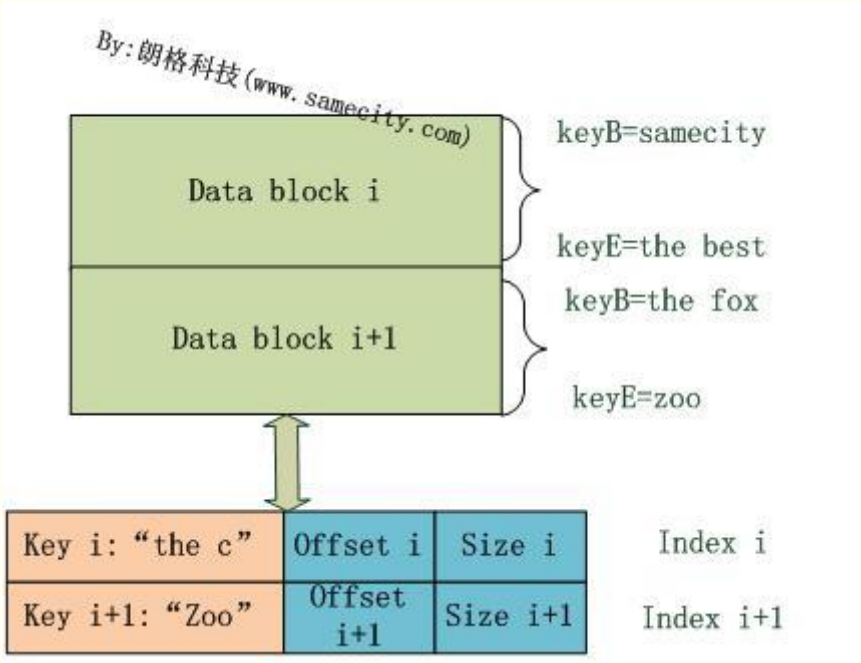


图 4.3 数据索引

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。

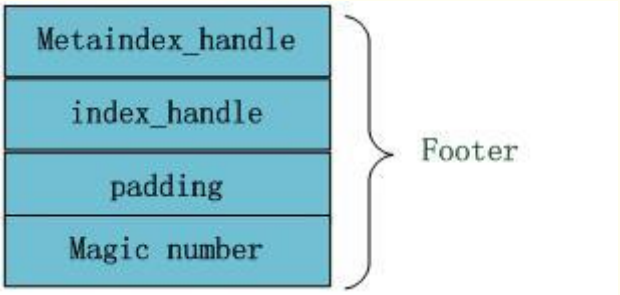


图 4.4 Footer

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。

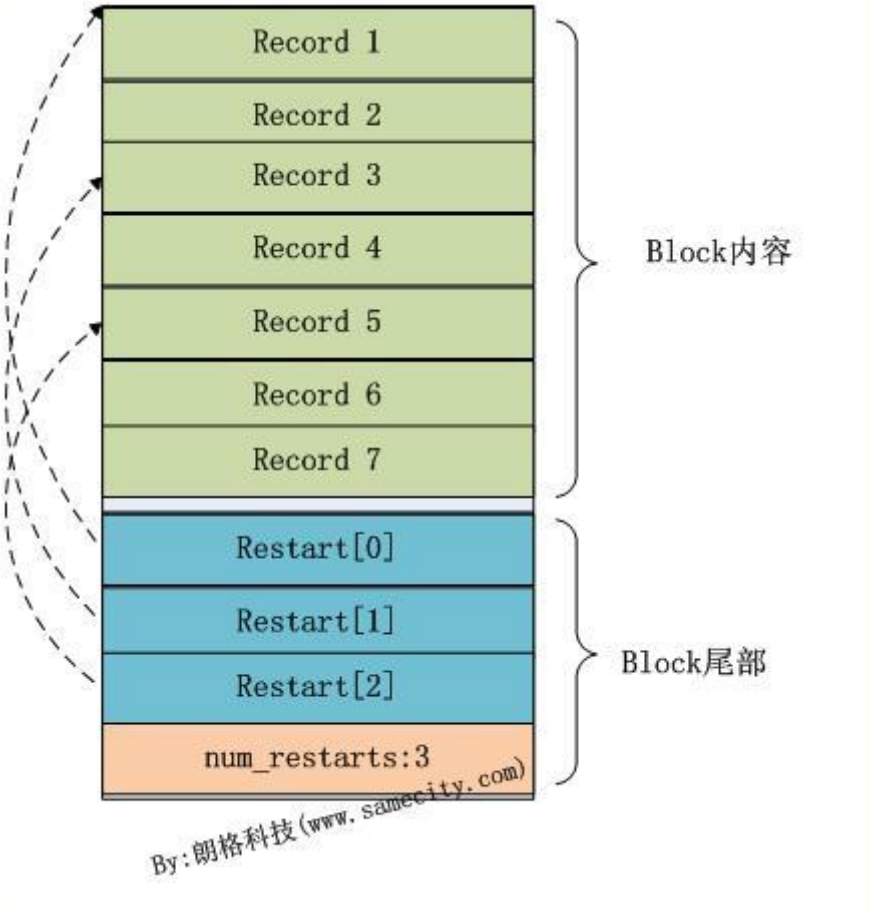


图 4.5 数据 Block 内部结构

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

LevelDb 日知录之五 MemTable

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

LevelDb 日知录之六 写入与删除记录

在之前的五节 LevelDb 日知录中，朗格科技介绍了 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。

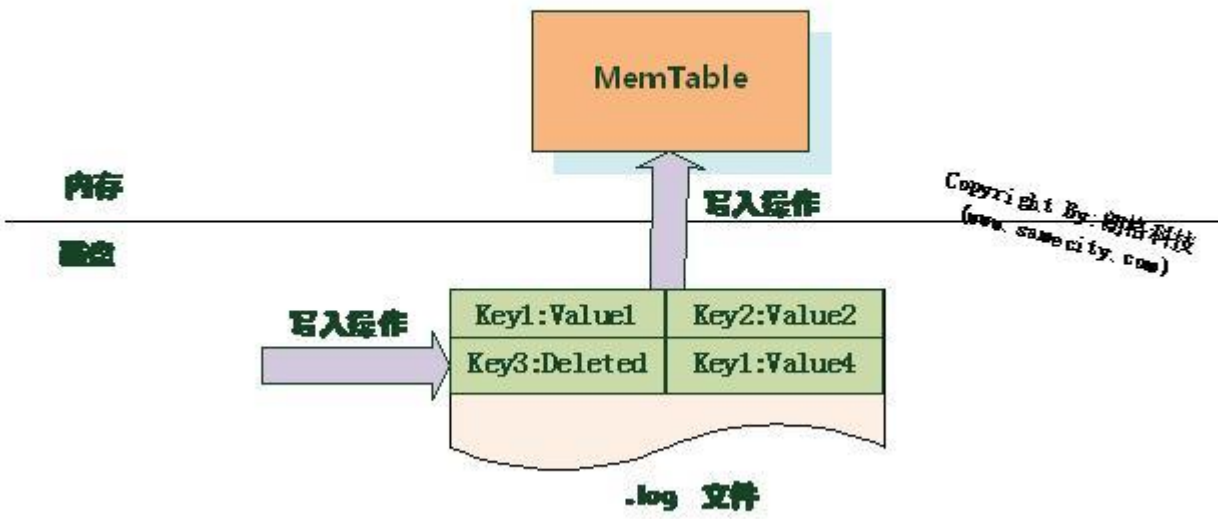


图 6.1 LevelDb 写入记录

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：如何根据 Key 读取记录?

LevelDb 日知录之七如何根据 Key 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。

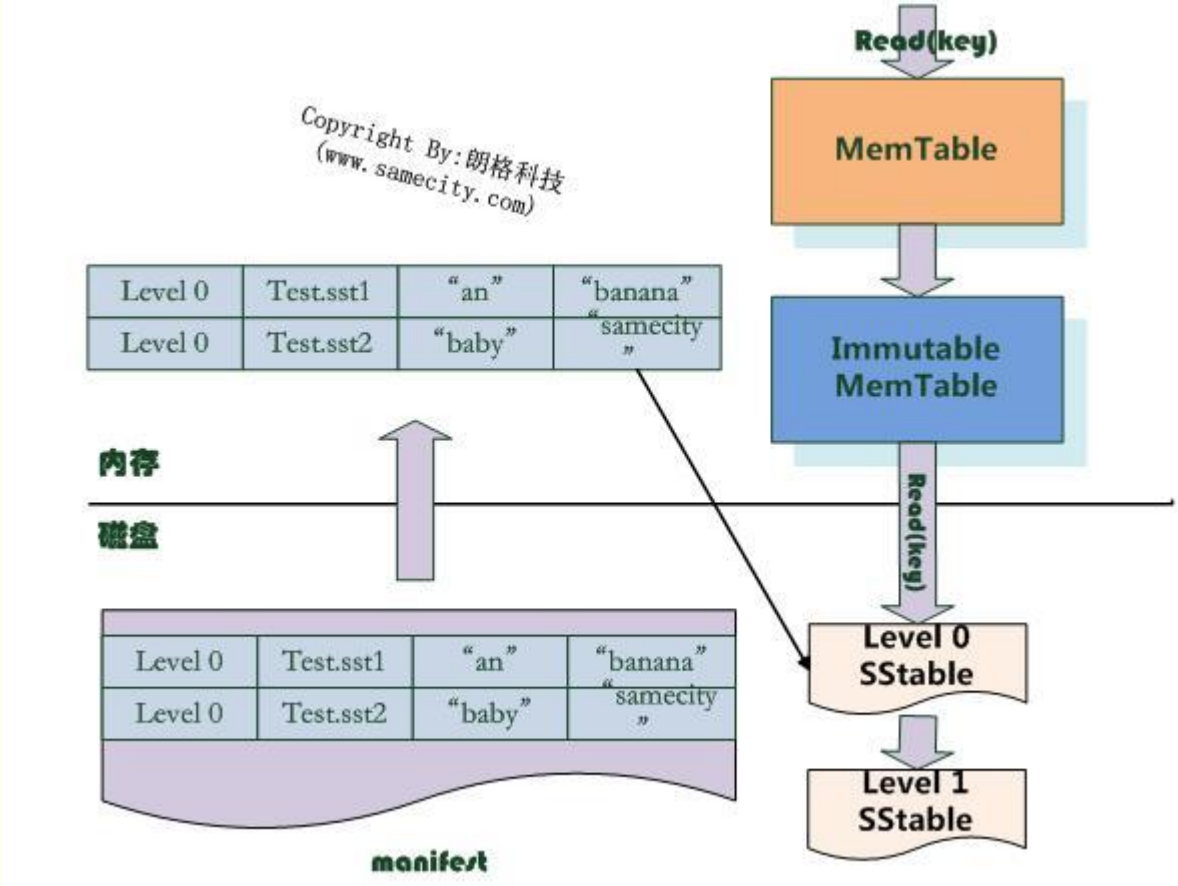


图 7-1　LevelDb 读取记录流程

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="朗格科技"},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。

证毕。

如果您没看明白上面的证明过程，那么请记得往您的 IQ 卡内充值。

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

LevelDb 日知录之八 Compaction

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了，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。

朗格科技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。

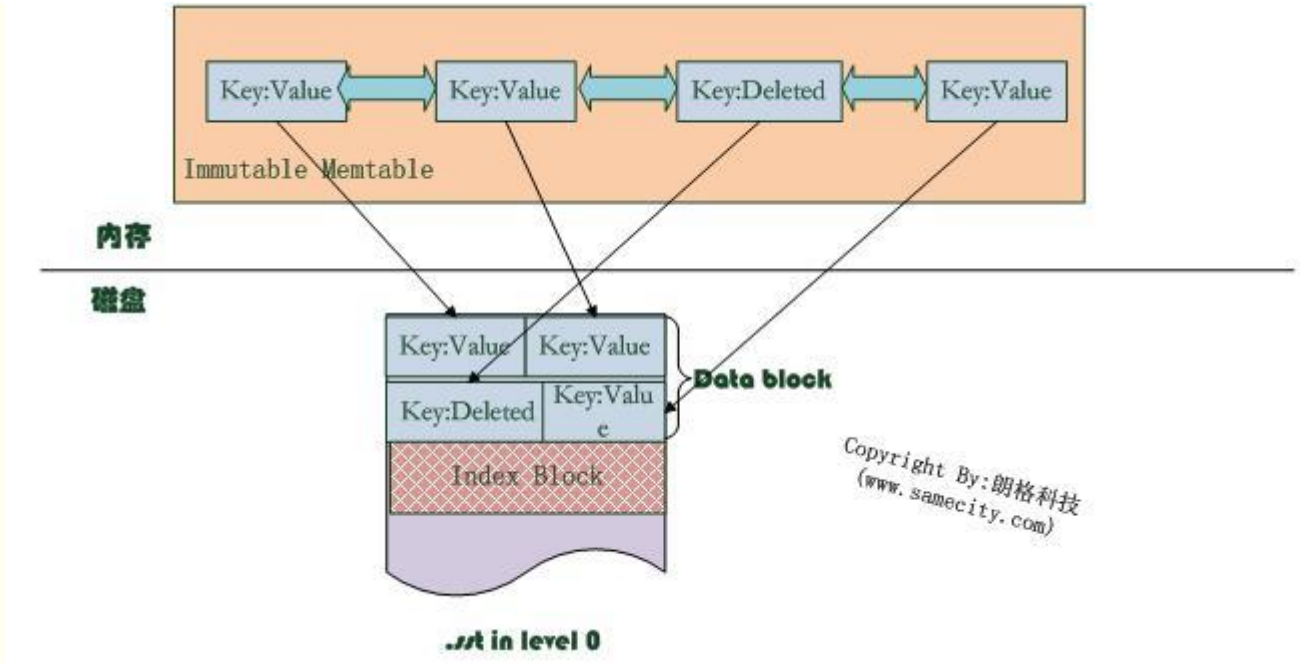


图 8.1 minor compaction

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。

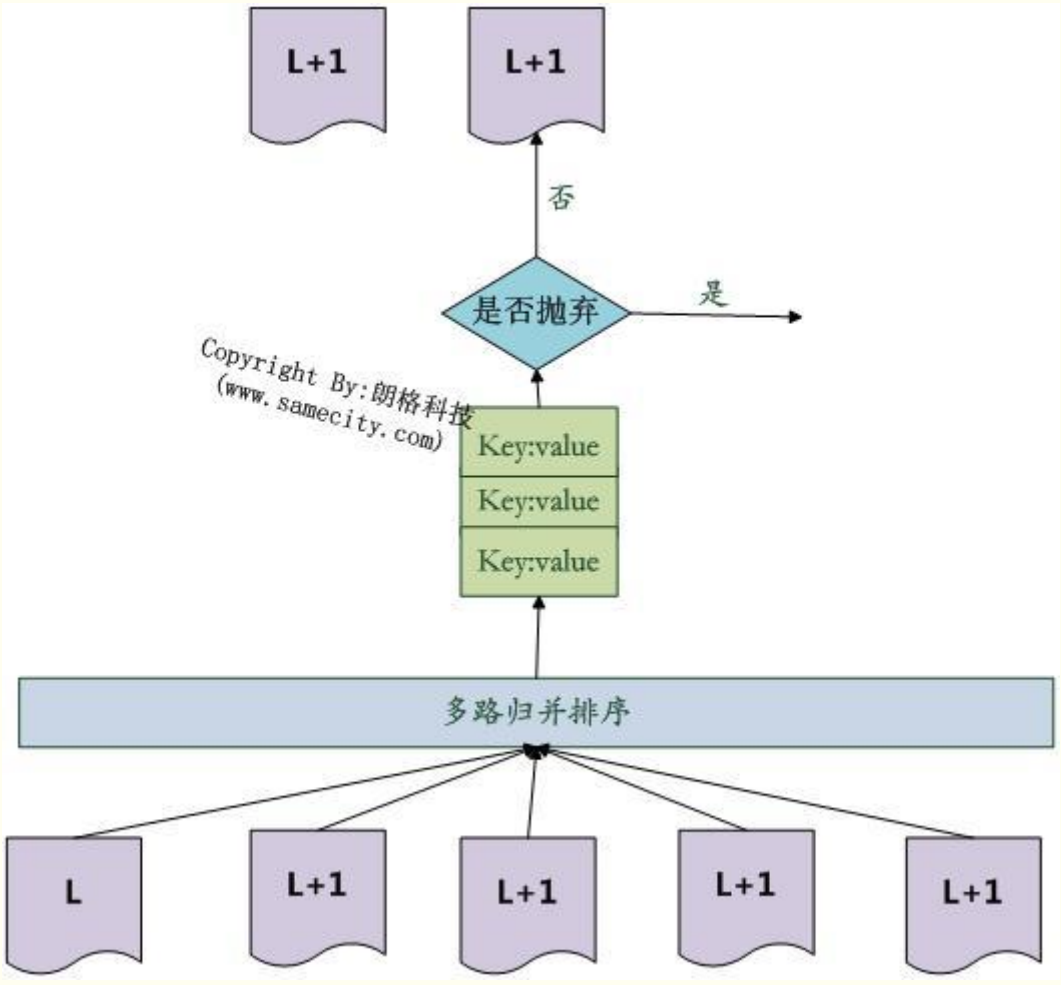


图 8.2 SSTable Compaction

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。

是的，compacition 的过程就是如此简单。

LevelDb 日知录之九：levelDb 中的 Cache

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。

哎！

为什么？

怎么会这样？

这样效率有点低。

levelDb 想到这点了。

它引入了 Cache 来部分解决读取数据效率低下的问题。

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。

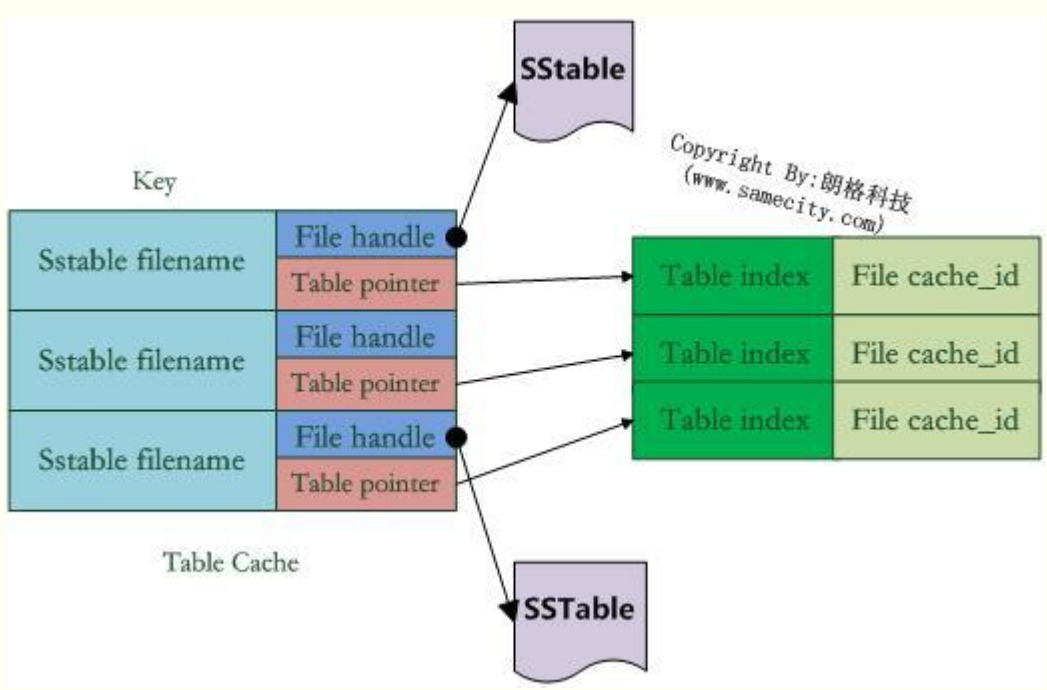


图 9.1 table cache

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。

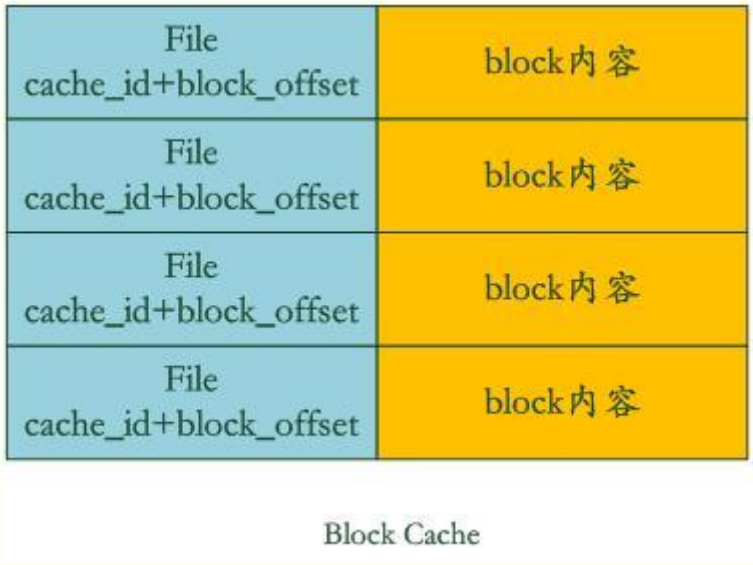


图 9.2 block cache

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。