LevelDB SSTable 格式详解

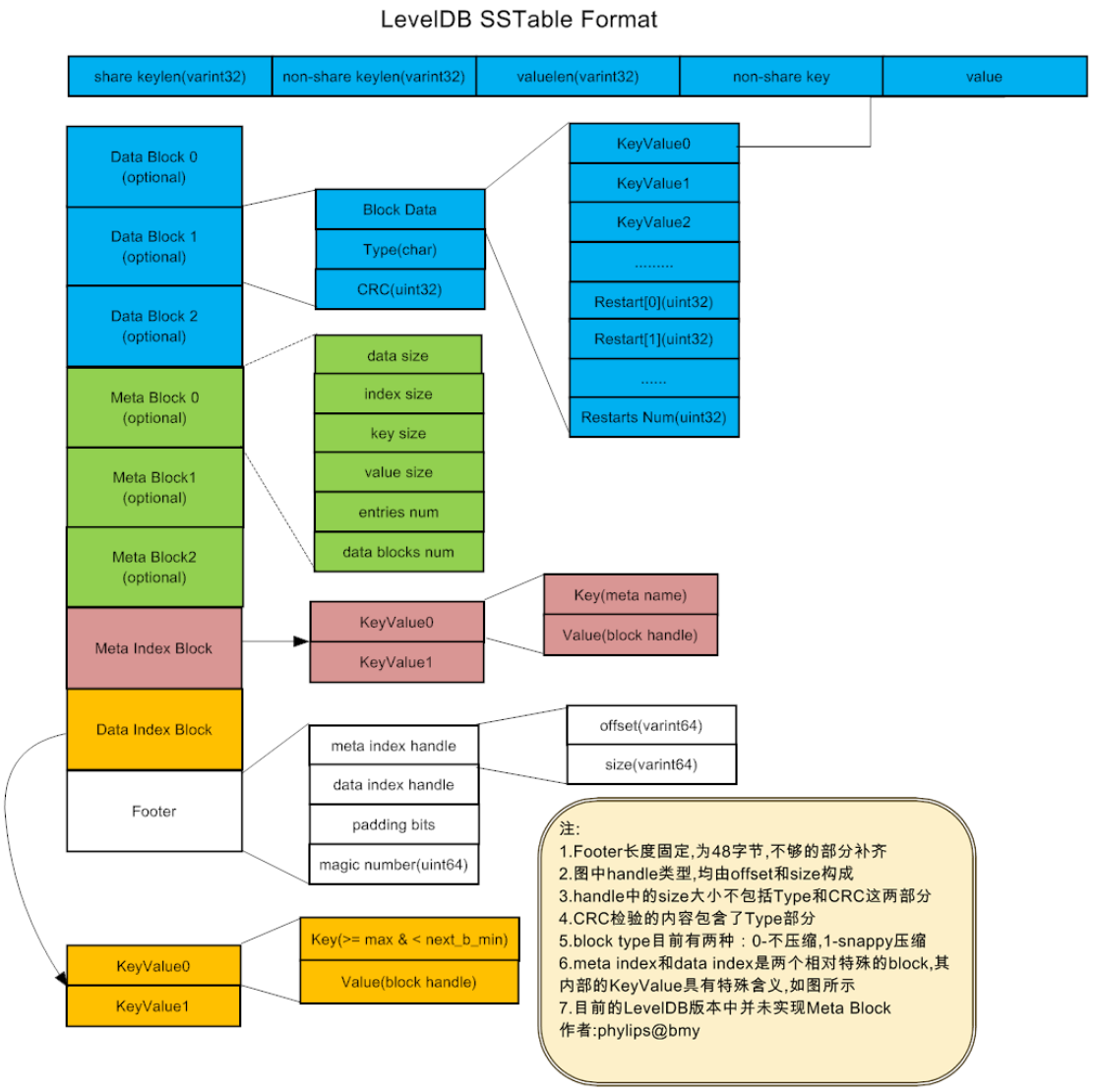
作者：phylips@bmy
2012-01-16

1. SSTable 文件格式

SSTable 是 Sorted String Table 的简称，也就是 Bigtable 底层的数据存储格式。SSTable 文件是用来存储一系列有序的 KeyValue 对的，Key 和 Value 都是字节串，KeyValue 对根据固定比较规则有序地写入到文件中，文件内部分成一系列的 Blocks(Block 不会太大，常见的是 64KB 大小)，同时具有必要的索引信息。这样就既可以顺序地读取内部 KeyValue 记录，也能够根据某个 Key 值进行快速定位。

Google 开源的 LevelDB 对应了 Bigtable 中的 tablet server，LevelDB 的代码中自然也包含了 SSTable 这一重要结构。下面会对 SSTable 的格式进行详细地解说，同时还会就一些影响性能的关键点进行分析，并将它与开源类 Bigtable 系统 HBase 中的 HFile 进行一个对比。

1.1. 格式说明



单个 SSTable 文件的格式如上图所示，文件由五大部分组成：Data Blocks，Meta Blocks，MetaIndexBlock，DataIndexBlock，Footer。除 Footer 部分外，其余都是一些 block 组成的结构，每个 block 则是由多个 KeyValue 组成的数据块。文件包含一些内部指针。每个这样的指针被称为一个 BlockHandle，包含如下信息：

offset: varint64
size: varint64

如图所示，Footer 中会有一个 meta index handle 用来指向 Meta Index Block，还有一个 data index handle 用来指向 Data Index Block，然后这两个 Index Block，实际上是一系列 Data Blocks 和 Meta Blocks 的索引，其内部的 KeyValue 值就包含了指向文件中的一系列 Meta Block 和 Data Block 的 handle。

(1)文件内的 key/value 对序列有序排列，然后划分到一系列的 data blocks 里。这些 blocks 一个接一个的分布在文件的开头。每个 data block 会根据 block_builder.cc 里的代码进行格式化，然后进行可选地压缩。
(2)在数据 blocks 之后存储的一些 meta blocks，目前支持的 meta block 类型会在下面进行描述。未来也可能添加更多的 meta block 类型。每个 meta block 也会根据 block_builder.cc 里的代码进行格式化，然后进行可选地压缩。
(3) A "metaindex" block。会为每个 meta block 保存一条记录，记录的 key 值就是 meta block 的名称，value 值就是指向该 meta block 的一个 BlockHandle。
(4) An "index" block。会为每个 data block 保存一条记录,key 值是>=对应的 data block 里最后那个 key 值，同时在后面的那个 data block 第一个 key 值之前的那个 key 值，value 值就是指向该 meta block 的一个 BlockHandle。
(5) 文件的最后是一个定长的 footer，包含了 metaindex 和 index 这两个 block 的 BlockHandle，以及一个 magic number。

```
metaindex_handle:    char[p];     // Block handle for metaindex
index_handle:        char[q];     // Block handle for index
padding:             char[40-p-q]; // 0 bytes to make fixed length
magic:               fixed64;     // == 0xdb4775248b80fb57
```

所以 footer 的总大小为 40+8=48。

另需注意如下几点:
1.图中 handle 类型,也就是上面提到的 BlockHandle，由 offset 和 size 组成
2.handle 中的 size 大小不包括 Type 和 CRC 这两部分
3.CRC 检验的内容包含了 Type 部分
4.block type 目前有两种：0-不压缩,1-snappy 压缩
5.meta index 和 data index 是两个相对特殊的 block,其内部的 KeyValue 具有特殊含义，如图所示。目前的 LevelDB 版本中并未实现 Meta Block。

需要着重解释的是：上图中除 DataBlock 外，其他类型的 Block：MetaBlock，MetaIndexBlock，DataIndexBlock 也都同属 Block 这一相同结构，都具有 BlockData，Type，CRC 这三部分，为简化起见，并没有画出这一级，只是画出了存在于这些 Block 中的 KeyValue 内容。同时对于 BlockData 来说，也是具有内部结构的，如上图中 DataBlock 部分画出的那样，BlockData 又由如下几部分组成：KeyValue 数据，Restart 数组，RestartsNum。其中 KeyValue 部分结构如图顶部所示，Restart

数组和 RestartsNum 都是与前缀压缩相关的结构，Restart 数组记录了重启点的偏移位置，RestartsNum 则是重启点的个数，也即 Restart 数组的元素个数。它们实际上担当了 BlockData 内部数据索引的角色，具体细节见下面的关于前缀压缩机制的分析。

1.2. 基本机制

1.2.1. 数据压缩

SSTable 中的压缩是以 Block 为单位进行的。目前只支持一种压缩方式：Snappy，用户也可以选择不进行压缩。该压缩算法本身的实现并不在 LevelDB 内，用户如果使用的话需要首先安装 Snappy，这是 Google 开源的一个压缩库。

当然，即使没有安装 Snappy，LevelDB 也是可以工作的。因为为了保证可移植性，LevelDB 中对该压缩算法的调用也做了一层封装，如下：port/port_posix.h

```
inline bool Snappy_Compress(const char* input, size_t length,
                                    ::std::string* output) {
#ifdef SNAPPY
   output->resize(snappy::MaxCompressedLength(length));
   size_t outlen;
   snappy::RawCompress(input, length, &(*output)[0], &outlen);
   output->resize(outlen);
   return true;
#endif

   return false;
}
```

这样如果没有 Snappy 库的话，也不会编译失败，即使用户在 option 中指定了采用压缩，压缩也不会生效，内部也只是采用非压缩格式。当然如果用户已经安装了 Snappy，那么 LevelDB 的 Makefile 就能检测出来并定义好相关的宏。

1.2.2. Varint 编码

1.2.2.1. 基本原理

从上面图中可以看到，很多字段都是 varint 类型的。varint 是对整数类型进行了变长编码。比如 int32 原本只有 4 字节，而编码后最短只需 1 个字节，最长需 5 个字节。在 key，value 长度都很小的情况下，采用 varint 编码的方式所带来的结构信息的空间节省会非常明显。

在 varint 编码中，编码后的每个字节的最高位用来表示后面的那个字节是否属于

当前的数，如果最高位为 1 表明，下一个字节也是当前数值的组成部分。这样对于一个 varint 来说除最后一个字节的最高位为 0 外，其他字节的高位都是 1。

比如对于整数 1，只需要一个字节存储：00000001。更复杂的比如 400，它的二进制格式是 00000001 10010000，按照 7 位一组就是：0000011 0010000 假设存储是按照小端格式进行的，那么首先取出低 7 位(0010000)+1 组成，10010000，然后取出下一个 7 位 0000011，因为已经没有下一个非零值，所以编码后的结果就是：10010000 00000011。解码的过程刚好与之相反，去掉最高位后得出 0010000 0000011，反转后得到最终结果：0000011 0010000。

1.2.2.2. 代码分析

varint 编解码代码在 util/coding.cc 里。代码很清晰，此处不再详细解释。

1.2.3. CRC 校验

1.2.3.1. 基本原理

CRC校验的基本思想是利用线性编码理论，在发送端根据要传送一个n比特的帧或报文，发送器生成一个r比特的序列，称为帧检验序列（FCS）。这样所形成的帧将由（n+r）比特组成。这个帧刚好能被某个预先确定的数整除。接收器用相同的数去除外来的帧，如果无余数，则认为无差错。二进制码多项式的加减运算为模 2 加减运算，即两个码多项式想加减时，对应项系数进行模 2 加减。所谓模 2 加减就是各位做不带进位、借位的加减。这种加减运算实际上就是逻辑上的异或运算，即加法和减法等价。更具体的介绍可参考：循环冗余校验码CRC原理详解。LevelDB中采用的是CRC-32C，其对应的生成多项式如下图所示：

名称	多项式	简记	应用
CRC-4	x^4+x+1	0x13	ITU G.704
CRC-16	$x^{16}+x^{15}+x^2+1$	0x8005	IBM SDLC
CRC-CCITT	$x^{16}+x^{12}+x^5+1$	0x1201	ISO HDLC, ITU X.25, SDLC, V.34/V.41/V.42, PPP-FCS
CRC-32	$x^{32}+x^{26}+x^{23}+x^{22}+x^{16}+x^{12}+x^{10}+x^8+x^7+x^5+x^4+x^2+x+1$	0x104C11DB7	ZIP, RAR, IEEE 802 LAN/FDDI, IEEE 1394, PPP-FCS
CRC-32C	$x^{32}+x^{28}+x^{27}+x^{26}+x^{25}+x^{23}+x^{22}+x^{20}+x^{19}+x^{18}+x^{14}+x^{13}+x^{11}+x^{10}+x^9+x^8+x^6+1$	0x11EDC6F41	SCTP

表1-1 标准CRC多项式

1.2.3.2. 代码分析

CRC 相关的代码在 util/crc32c.cc，代码并不多，关键是理解其中的原理。计算的核心在于如何利用以前的计算结果，对新来的数据实现增量计算，而不是全部从头计算。这里采用了查表法来加速 crc32 值的计算，同时 LevelDB 内部的这个实

现还加入了其它的一些优化：比如支持以 4 字节为单位进行的计算，内存访问 4 字节对齐，循环展开，对于 crc32 值进行逆序存储。

首先看 crc32 的值如何以一个字节为单位进行扩展，即如下一个问题：假设目前已经计算出一系列字节串 M 的 crc32 值 CRC，然后又来了一个新的字节 B，现在需要计算字节串 M 加上字节 B 后形成的新字节串 M’的 crc32 值 CRC’。

证明：设使用的生成多项式为 G，M 除以 G 的商为 Q，M’ 除以 G 的商为 Q’。X8 代表 100000000，*X8 对于模 2 运算来说就意味着左移 8 位，同理 *X32 意味着左移 32 位。另 CRC3, CRC2, CRC1, CRC0 代表了组成 CRC 从高位到低位的四个字节。根据 crc 值的定义，可得如下等式(下面的运算均是模 2 运算)：

M*X32=GQ+CRC (1)

M’*X32=GQ’+CRC’ (2) 又 M’=M*X8+B，代入(2)得

M*X8*X32+B*X32=GQ’+CRC’ (3) (1)+(3)*X8 得

M*X8*X32+B*X32+M*X32*X8=GQ*X8+CRC*X8+GQ’+CRC’ (4)

B*X32=G(Q*X8+Q’)+CRC*X8+CRC’ (5) 将 CRC*X8 移到等式左侧

B*X32+ CRC*X8=G(Q*X8+Q’)+CRC’ (6) 整理成()*X32=GQ+CRC 的形式

B*X32+(CRC3*X24+CRC2*X16+CRC1*X8+CRC0)*X8=G(Q*X8+Q’)+CRC’

(B+CRC3)*X32= G(Q*X8+Q’)+CRC’+(CRC2*X16+CRC1*X8+CRC0)*X8

如上已经转换为()*X32=GQ+CRC 的形式了，也就是 B+CRC3 的 crc32 值就是 CRC’+(CRC2*X16+CRC1*X8+CRC0)*X8，这样如果我们知道了 B+CRC3 的 crc32 值，那么再异或上(CRC2*X16+CRC1*X8+CRC0)*X8，就是 CRC’的值了。结合代码来看：

```
int c = (l & 0xff) ^ *p++;
l = table0_[c] ^ (l >> 8);
```

看上面两行代码，第一行就是在计算 B+CRC3，但是(l & 0xff)取的是最低位，这就意味着 l 是逆序存放的，也就是说它的低位实际上存放的是 crc32 值的高位。然后，第二行，首先通过查表找到 B+CRC3(这两个都只有一个字节，因此异或后也只有一个字节，我们可以事先计算出单个字节的所对应的所有 crc32 值)。然后再异或(l>>8)，根据我们的推论 l 是逆序存放的，所以左移 8 位得到的就是低 3 个字节，同时也意味着是*X8。这也就是以一个字节单位扩展的算法原理。

再看如何进行以 4 字节为单位的扩展。只需令上面的问题中 M’= M*X32+W 即可，W 代表一个 4 字节值。证明过程如下：

M*X32=GQ+CRC (1)

M’*X32=GQ’+CRC’ (2)又 M’=M*X32+W，代入(2)得

M*X32*X32+W*X32=GQ’+CRC’ (3) (1)+(3)*X32 得

M*X32*X32+B*X32+M*X32*X32=GQ*X32+CRC*X32+GQ’+CRC’ (4)

W*X32=G(Q*X32+Q’)+CRC*X32+CRC’ (5) 将 CRC*X32 移到等式左侧

W*X32+ CRC*X32=G(Q*X32+Q’)+CRC’ (6) 整理成()*X32=GQ+CRC 的形式

(W+CRC) *X32=G(Q*X32+Q’)+CRC’

如上已经转换为()*X32=GQ+CRC 的形式了，也就是说 W+CRC 的 crc32 值就等于 M’的 crc32 值。但是不能直接采用查表法，因为 W+CRC 已经是个 4 字节值。但是通过进一步的转化可以使用查表法。

M*X32=GQ+CRC，设 M=(A+B+C+D)代入
(A+B+C+D) *X32=GQ+CRC　(7)
A*X32=GQa+CRCa (8)
B*X32=GQb+CRCb (9)
C*X32=GQc+CRCc (10)
D*X32=GQd+CRCd (11)
(8)+(9)+(10)+(11)得(A+B+C+D)*X32=G*(Qa+Qb+Qc+Qd)+CRCa+CRCb+CRCc+CRCd
所以只要我们根据 M=(M3*X24+M2*X16+M1*X8+M0)，将 M 分为 4 部分就可以继续使用查表法了。结合代码来看：

```
uint32_t c = l ^ LE_LOAD32(p);                    \
p += 4;                                                \
l = table3_[c & 0xff] ^                           \
    table2_[(c >> 8) & 0xff] ^                    \
    table1_[(c >> 16) & 0xff] ^                   \
    table0_[c >> 24];
```

l ^ LE_LOAD32(p);实际上就是在计算 W+CRC，然后利用查表法分别查出 c 的四个组成部分对应的 CRC 值，然后将它们异或就是 CRC 值。

CRC 的计算原理大体如上，但是要理解这里的代码，还要能回答如下几个问题：
Q：为什么在计算 crc32 值的前后，都异或了 0xffffffffu？这里的异或与后面的 Mask 有何区别？
A：这相当于对 crc32 值进行了一次掩码，这样最终外部调用得到的 crc32 值就不是标准的 crc32 值。我们可以这样理解这样一个处理，整个计算过程是这样的：[异或 0xffffffffu；计算 CRC 值；异或 0xffffffffu]…[异或 0xffffffffu；计算 CRC 值；异或 0xffffffffu]，如此循环往复。如果调整一下组合顺序，[异或 0xffffffffu；计算 CRC 值；][异或 0xffffffffu 异或 0xffffffffu；]…[计算 CRC 值；异或 0xffffffffu]，可以看到中间的[异或 0xffffffffu 异或 0xffffffffu]是可以抵消的，所以无论多少次调用，实际上都是这个过程[异或 0xffffffffu；计算 CRC 值；异或 0xffffffffu]，这可以理解成是以 0xffffffffu 作为初始 CRC 值，然后计算当前数据的 CRC 值，然后再将结果异或 0xffffffffu 作为返回值。这样在 Extend 函数内部看到仍然是 CRC 值，而外部看到的则是经掩码后的值，当然也可以选择其他值进行异或。与后面的 Mask 不同，这里注意为了不暴露实际的 CRC 值，后面的 Mask 是解决嵌套 CRC 计算的问题。

Q：代码中的 4 个由 CRC 值构成的表格分别代表了哪些序列的 CRC 值？这些表格是如何生成的？
A：生成 crc32c 查找表的具体代码可参考附录 crc32c_defs.h 和 gen_crc32ctable.c。首先我们可以看到使用的生成多项式是：
x^32+x^28+x^27+x^26+x^25+x^23+x^22+x^20+x^19+x^18+x^14+x^13+x^11+x^10+x^9+ * x^8+x^6+x^0
0x82F63B78，即 10000010111101100011101101111000。可以看出，它是逆序的，即左侧是低位，右侧是高位，同时并没有包含 x^32 这一位。下面分析下 gen_crc32ctable.c 使用的计算方法：

```
static void crc32c_init(void)
{
```

```
    unsigned i, j;
    uint32_t crc = 1;
    crc32c_table[0][0] = 0;
    //计算出 crc32c_table[0][0…255]的所有值
    for (i = LE_TABLE_SIZE >> 1; i; i >>= 1) {
        // crc=table[0][i]，而 i 则是 2 的幂，也就是所有只有一个非 0 位的数
        crc = (crc >> 1) ^ ((crc & 1) ? CRC32C_POLY_LE : 0);
        // crc32c_table[0][i + j]= crc32c_table[0][i]^ crc32c_table[0][j]
        //为使上式成立，需要满足一个条件：i+j=i^j。可以看到是 i 均为 2 的幂
        //j 则是 i 的偶数倍，所以满足 i+j=i^j。
        for (j = 0; j < LE_TABLE_SIZE; j += 2 * i)
            crc32c_table[0][i + j] = crc ^ crc32c_table[0][j];
    }
    //计算出 crc32c_table[1…7][0…255]的所有值，crc32c_table[j][i]的计算基于
    crc32c_table[0][i]的值，而二者的关系则是：crc32c_table[0][i]是 i 对应的 CRC
    值，而 crc32c_table[j][i]则是 i 左移 j*8 位后的值对应的 CRC 值，注意 i 和 CRC
    都是逆序存放的
    for (i = 0; i < LE_TABLE_SIZE; i++) {
        crc = crc32c_table[0][i];
        for (j = 1; j < 8; j++) {
            crc = crc32c_table[0][crc & 0xff] ^ (crc >> 8);
            crc32c_table[j][i] = crc;
        }
    }
}
```

以 crc32c_table[0][127]为例，它的结果是 0x82f63b78，也就是生成多项式的值。也就是说它实际上是 1 所对应的逆序 CRC 值(127 也是逆序存放的，所以实际上是 1)。所以在 Extend 函数中，只需要通过 LE_LOAD32 完成字节为单位的逆序即可，字节内部的 bit 级的逆序，crc32c_table 本身已经完成了这个功能。比如输入数据是 1000 0000 0000 0000，那么经过 LE_LOAD32 实际上会被变成 0000 0000 0000 1000，而在查表时查的将是 table3_[127]，所以计算出来的实际上是 1000 0000 0000 0000 的逆序 CRC 值。

Q：crc32c.h 中的 Mask 函数是用来做什么的？

A：根据注释：是因为当字串内部包含嵌套的 CRC 值时再次计算 CRC 值会有问题，因此对于计算出的 crc32 值需要进行 mask。但是到底会有什么问题呢？假设我们现有数据 X0…Xi CRC0 Xi+1…Xn CRC1，其中 CRC0 是 X0…Xi 的 CRC 值，CRC1 则是它之前所有数据的 CRC 值，当然也将 CRC0 也包含在内了。实际上，如果 CRC0 不 mask 会有一个明显的问题，根据 CRC 的定义，X0…Xi CRC0 实际上刚好整除生成多项式，那么单纯的 Xi+1…Xn 这部分的 CRC 值实际上也是 CRC1，这样校验值所能验证的范围就变成了 Xi+1…Xn 了，最简单地比如即使我们将 X0…Xi CRC0 部分全部用 0 取代，CRC 校验也能成功，而且对于任何取值的字串这样做都是行之有效的。举个更具体的例子，假设我们应用程序层对数据有一个 CRC 校验，然后(数据)+(CRC 校验值)+(其他数据)会通过网络传输，为了确保传输无误，会在最外

层对所有数据进行 CRC 计算。但是现在对于(数据)+(CRC 校验值)这部分，就具有很高的风险，比如我们随便用(另一个数据)+(对应的 CRC)值替掉真实的数据，虽然发生了变化，但是最上层 CRC 校验结果还是正确的。Mask 之后，就排除了这种影响，比如如果不知道(其他数据)，就很难伪造出通过校验的数据。

1.2.4. 前缀压缩

1.2.4.1. 基本原理

Key 的存储采用了前缀压缩机制，前缀压缩的概念很简单，就是对于 key 的相同前缀，尽量只存储一次以节省空间。但是对于 SSTable 来说它并不是对整个 block 的所有 key 进行一次性地前缀压缩，而是设置了很多区段，处于同一区段的 key 进行一次前缀压缩，每个区段的起点就是一个重启点。之所以设置很多区段是为了更好的支持随机读取，这样就可以在 block 内部对重启点进行二分，然后再在单个区段内遍历即可找到对应 key 值的记录，所以分段压缩就承担了 block 内部二级索引的功能。如果整体进行前缀压缩，那么就没法在 block 内进行二分，只能遍历，因为如果要恢复第 i+1 条记录需要知道第 i 条记录的 key，如果要恢复第 i 条记录需要知道第 i-1 条记录的 key，如此递归下去，意味着都要从第一条开始，才能将 key 恢复出来。这样 block 内部的查找就必须是顺序的了。

前缀压缩机制导致每条记录需要记住它对应的 key 的共享长度和非共享长度。所谓 key 的共享长度，是指当前这条记录的 key 与上一条记录的 key 公共前缀的长度，非共享长度则是去掉相同部分后的不同部分的长度。这样当前这条记录只需要存储不同的那部分 key 的值即可。

同时分段压缩，又导致在 block 尾部需要用一个数组记录这些重启点的 offset，同时 block 的最后 4 个字节则被固定用来保存重启点的个数，同时每个重启点都是一个 4 字节的整数，这样知道了 block 起始地址及长度后，很容易就能计算出 restarts_了。加上这些机制之后，导致解析过程变得比较复杂，但是解析速度还是很快的。另外每条记录的 key 共享长度，key 非共享长度，value 长度本身进行了 varint32 变长编码，这又是一个节省空间的优化。

1.2.4.2. 代码分析

涉及前缀压缩的读取逻辑在 table/block.cc，写入逻辑在 table/block_builder.cc 中。

对于读取过程来说，在 Block::Iter::ParseNextKey()中，需要理解其对 RestartPoint 的处理，首先需要明确 restart_index_所代表的意义，它是当前的 Entry 所属的 RestartPoint，而不是代表未来的那个 Entry 所属的 RestartPoint。它是用最后的 while 循环来保证的，GetRestartPoint (restart_index_ + 1) < current_。另外整个过程中并没有调用 key.clear()，那么如何保证可以在重启点将 key 归为重启点的 key 呢？实际上在连续遍历过程中，key_.resize(shared);就担当了 clear()的功能。让重

启点的 shared 长度=0，non_shared=key 实际长度。再通过
key_.resize(shared);
key_.append(p, non_shared);
就可以让重启点的 key 值重新更新为该点的 key 而不再跟上一个重启点共享 key 前缀。

对于写入来说，每次 Add(key,value)都要检查下当前区间的记录数是已经达到限制，如果达到限制，就将该 KeyValue 放到下一个区间，同时将当前 buffer 大小作为它所在区间在 restarts_数组的值，可以看到此时 buffer 还未添加当前的这个 KeyValue 的数据，也就是说数组 restarts_记录的是对应区间的起始地址。

1.2.5. 索引优化

1.2.5.1. 基本原理

我们知道在 Meta Index Block 和 Data Index Block 这两个 Block 中存放了指向 Data Block 和 Meta Block 的索引。对于 Data Index Block 来说，它的每个 KeyValue 值均指向一个一个 Data Block，通过这个 KeyValue 值就可以寻址到对应的 Data Block。按照常理来说，我们可以采用它指向的那个 Data Block 中的最大或最小的那个 key 值作为索引的 key 值，这样当给定某个 key 时就可以先查看索引，来定位它应该位于哪个 Data Block。但是 LevelDB 对此进行了优化，并不要求采用 Data Block 出现的 key 值作为索引中的 key，它只需要该 key>=对应的那个 DataBlock 的最大的 key，同时<下一个 block 的最小的 key 即可。

也就是说这就提供了一种选择空间，可以为索引选择长度更小的 key。举例说明，假设当前 Data Block 的最大 key 是”apple on the tree”，下一个 DataBlock 的最小 key 是”fly in the sky”，那么索引中就可以选择”b”作为该 Data Block 的 key，而不一定非要将”apple on the tree”存储在索引中，”b”与”apple on the tree”相比显然占的空间少多了。对于最后一个 DataBlock 来说，相对比较特殊，因为没有下一个 Data Block 了，因此只要找到满足>=该 Block 的最大 key 的 key 就可以了。

关于如何确定满足这样条件的 key，LevelDB 已经将这部分逻辑做了很好的封装，用户通过自己的 Comparator 实现，就可以定制这些逻辑。

1.2.5.2. 代码分析

实际实现是通过 Comparator 的两个函数：
void FindShortestSeparator(std::string* start, const Slice& limit) const;
void FindShortSuccessor(std::string* key) const;
来找到满足条件的 key 的。其中第一个函数用于找到位于[start,limit)之间的最段字串，第二个函数则用于找到>=*key 的后继者。

SSTable 在生成索引的 KeyValue 值时会调用这两个函数，来决定 key。用户可以选择定义自己的 Comparator 实现。

在 util/comparator.cc 中的 BytewiseComparatorImpl 实现中，FindShortestSeparator 会查找 start 和 limit 之间第一个不相同的字节，如果*start[diff_index]+1 仍然小于 limit[diff_index]，那么就将公共前缀加上*start[diff_index]+1 作为结果返回即可。FindShortSuccessor 的实现更简单，只需要找到不等于 0xFF 的第一个字节然后将其加 1，返回该字节及其前面的部分即可。在 table/table_builder.cc 中会调用这两个函数，用于索引生成。

1.3. 几个问题

Q：看 SSTable 的格式图可知，Data Index Block 和 Meta Index Block 都是 block，另外我们也知道对于 SSTable 文件来说，有一个参数即 block_size(该参数用来限制单个 block 大小的上限)，如果 block_size 很小，那么是否会导致出现多个 Data Index Block 和 Meta Index Block，而 Footer 中只能有两个 handle，那么到底该怎么处理呢？
A：查看下 table/table_builder.cc 的代码可知，对于普通的 block，在 Add(key,value) 的过程中，会判断 estimated_block_size >= r->options.block_size，如果满足该条件，就将其 Flush 出去。但是对于 Data Index Block 和 Meta Index Block 这两个 block，它们的写出是在 TableBuilder::Finish()中完成的，对于 Meta Index Block，目前的版本中只提供了一个空的 Block，对于 Data Index Block，则对应着 r->index_block 的内容。也就是说 block_size 这个参数只是用来限制普通 block 的大小的，而对于 Data Index Block 和 Meta Index Block，无论多大都会被写成一个 block。

Q：由于同一个 KeyValue 记录不会跨越两个不同的 Block，因此 block 大小就无法保证严格等于 block_size。那么 block 大小通常是大于规定的 block_size 还是小于 block_size 呢？
A：结果 是大于，即在组装 block 时，判断条件是：如果当前 block 的大小>=block_size，就结束该 block，同时第一个不满足这个条件的那个 KeyValue 也会被写入到该 block，也就是说是采用的检查策略是写入后检查而不是写入前。可以换个角度考虑，如果是保证实际 block 大小<= block_size，那么极端情况下，比如我们的单个 value 就很大，已经超过了 block_size，那么对于这种情况，SSTable 就没法进行存储了。所以通常，实际的 Block 大小都是要略微大于 block_size 的。

Q：在开启压缩的情况下，block 大小是指压缩前的还是压缩后的呢？
A：这首先取决于 block 大小的使用场景，比如是用来与 block_size 进行比较以限制 block 大小呢，还是 Block handle 中的 size 字段。Block handle 中的 size 记录的就是 block 压缩后的实际大小。而在与 block_size 进行比较以限制 block 大小时，则用的是压缩前的大小，原因是这样的：与 block_size 的比较是在每次 Add 一个 KeyValue 时就需要进行的，但是如果要知道压缩后的实际大小，那么就需要实际地进行一次压缩才能知晓，显然不可能每次添加一个 KeyValue，就尝试着压缩一下，这样 CPU 开销就太大了，因此直接使用压缩前的大小进行比较是一个更合理的选择。所以采用压缩时，假设压缩率是 1/3，那么实际存储的 block 大小要

比 block_size 小得多，大概是 block_size 的 1/3。

Q：如果 CRC 校验失败如何处理？
A：首先 CRC 检验部分的代码位于 table/format.cc 中的 ReadBlock 函数，如果校验失败，该函数会返回一个状态：Status::Corruption("block checksum mismatch")，同时*block=NULL。而读取时，该函数又会被 table/table.cc 中的 Table::BlockReader 上层函数调用，如果 ReadBlock 失败，可以看到它会利用返回的状态产生一个 NewErrorIterator(s)，而该 block iterator 又会被 TwoLevelIterator 在上层调用，因此还要看 TwoLevelIterator::SkipEmptyDataBlocksBackward()，通过代码可知如果 data_iter_.iter() == NULL || !data_iter_.Valid()，则该 data_iter_将会被跳过。而 NewErrorIterator 实际上会返回一个 EmptyIterator，EmptyIterator 就属于这种情况，因此由这一系列分析可知在 SSTable 层面如果发现 CRC 校验失败该 block 会被跳过，但是与此同时它会通过 SaveError()将该错误保存。更上层的调用者可以通过调用 status()判断是否有错误发生，从而根据自己的需要选择处理方式。

Q：CRC 校验中 ReadOptions::verify_checksums 和 Options::paranoid_checks 的关联？
A：LevelDB 会为它存储在文件系统中的所有数据生成相应的 checksums。通常有两种方式来控制对 checksums 进行何种程度的验证：
ReadOptions::verify_checksums 设为 true，会强制对从文件系统中读出的所有数据都进行 checksum 验证。默认情况下，不进行验证。
Options::paranoid_checks 可以在打开数据库之前将其设为 true，这会使得数据库底层实现只要检测到内部数据错误时就会产生一个 error。Error 产生的时机取决于数据库的哪个部分出了问题，比如可能在数据库打开时或者是在后面执行某个数据库操作时。默认情况下，paranoid checking 是关闭的，这样即使数据库的某些永久性存储出了问题，它仍然也是可以使用的。 如果数据库被破坏了(可能是因为打开了 paranoid checking 而导致它无法打开)，leveldb::RepairDB 函数可以用来尽量地对数据进行恢复。

2. 一个实际的 SSTable 文件

为加深理解，我们来具体分析一个实际的 SSTable 文件，对于一个 KeyValue 对个数为 0 的 SSTable 文件来说，其长度并不等于 0，也会包含一些必要的元信息。下面就以这种最简单的情况来开始分析。

2.1. 数据组成

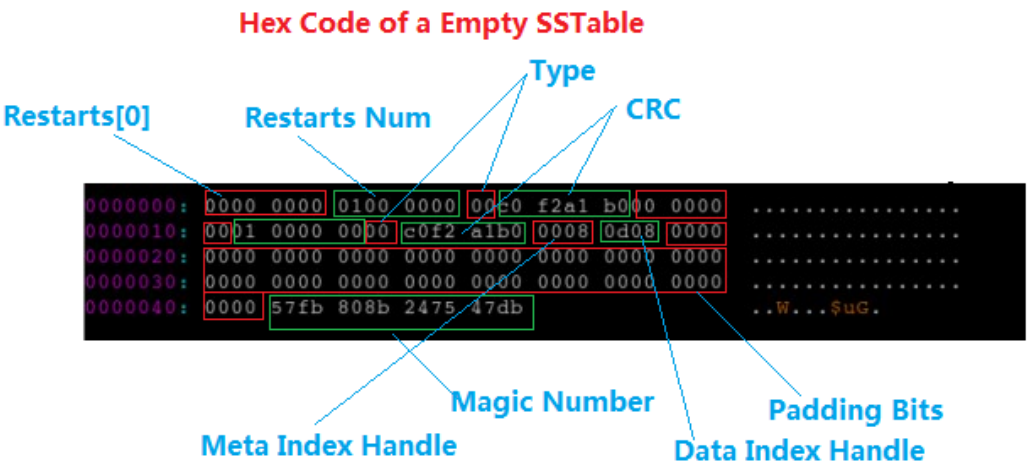
一个内容为空(KeyValue 对的个数为 0)的 SSTable 文件的总大小为 74 个字节。其内部只有 MetaIndexBlock，DataIndexBlock，Footer 三个部分。其中 MetaIndexBlock 和 DataIndexBlock，大小均为 13 个字节，Footer 部分大小固定为 48 字节，因此总大小就是 74 字节，各部分更具体的大小及其取值如下图所示。

Empty SSTable Format

	Name	Value	Size
MetaIndexBlock	restarts[0]	0	4
	restarts num	1	4
	Type	0	1
	CRC		4
DataIndexBlock	restarts[0]	0	4
	restarts num	1	4
	Type	0	1
	CRC		4
rooter	MetaIndexHandle	(0,8)	2
	DataIndexHandle	(13,8)	2
	padding bits	0	36
	magic number	0xdb4775248b80fb57	8

2.2. 二进制内容

更进一步地，我们从二进制数据的级别上来看：



观察上面两个图，由于 DataBlock 和 MetaBlock 都为空，所以这两个部分都未出

现。同时 MetaIndexBlock 和 DataIndexBlock 没有 KeyValue 数据，只有一些元数据：restart num 为 1 且 restarts[0]=0，说明无 KeyValue；type=0，说明是未压缩数据。因为这两个 Block 数据一致，所以 CRC 值都是 0xc0f2a1b0。在 rooter 部分，MetaIndexHandle 是[0,8]，意味着 MetaIndexBlock 在文件内的偏移指针为 0，8 代表大小，是指除去 Type 和 CRC 部分后的大小。DataIndexHandle 是[13,8]，意味着 DataIndexBlock 在文件内的偏移指针为 13(MetaIndexBlock 大小为 4+4+1+4=13)。

可以看到 MetaIndexHandle 和 DataIndexHandle 采用 varint 编码后，只占用了 2 个字节。关于 CRC 值，可以看到它所校验的数据是 0x0000 0000 0100 0000 00。

3. 单个文件读写过程

相关代码逻辑主要位于 table 目录下。该模块主要是 SSTable 的实现。包括 SSTable 整体及内部 block 的格式定义，SSTable 文件及内部 block 的构建及读取。

3.1. 读文件

3.1.1. 基本过程

一个 SSTable 文件会被抽象成一个 Table 对象，文件内部的一个数据块会被抽象成一个 Block 对象。

1 首先调用 Table::Open 函数，在该函数内会将 Footer 部分加载到内存，并将其中的 magic number，MetaIndexHandle 和 DataIndexHandle 解析出来，如果 magic number 不对，或解析失败会返回错误信息，之后会根据 DataIndexHandle 将 DataIndexBlock 这个 Block 加载到内存，这样完成了读取的准备工作。这里共涉及到两次随机读取。

2 用户要能够对 Table 对象进行访问，需要调用 Table::NewIterator 得到一个迭代器，然通过该迭代器提供的接口进行访问。该迭代器实际上是一个 TwoLevelIterator，共有两层，最外层用于迭代 Blocks，具体来说是迭代 DataIndexBlock 得到所有 Blocks 的地址，内层用于迭代 Block 内部的 KeyValue。迭代器的接口支持如下几种操作：顺序扫描，逆序扫描，随机读取。

3 首先来看 Seek 操作，共有三种类型：Seek(target)，SeekToFirst()，SeekToLast()。

3.1 这三种操作都具有类似的工作模式：首先外层迭代器会 Seek 到对应的 DataBlock；然后根据当前的 Block Index Handle，读取对应的 Block，并设置好内层迭代器；内层迭代器 Seek 到对应 KeyValue；SkipEmptyDataBlocks。内部还有些优化，比如如果上一次读取的 Block 就是当前所需要的，则不需再重新读取。因此这里可能会涉及到一次随机读取。

3.2 下面具体来看外层迭代器和内层迭代器的具体工作原理。外层迭代器和内存迭代器都是在一个 Block 上的迭代，都具有相同类型 Block::Iter，区别在于外层迭代器是在 DataIndexBlock 上，而内存迭代器则是在

DataBlock 上。对于一个 Block 来说，由于内部是有重启点的，首先会移到正确的重启点处，然后移到正确的 KeyValue 处：Seek(target)会通过二分找到对应的重启点，然后移到>=target 的第一个 KeyValue 处；SeekToFirst()会移到第一个重启点，并解析出第一个 KeyValue；SeekToLast()会移到最后一个重启点，并跳过所有的 KeyValue。

3.3 SkipEmptyDataBlocks 的原因在于：外层迭代器在查找对应的 Block 时，可能会有些问题，比如该 Block 的 CRC 校验失败，就需要跳过该 Block，或者当前的 Block 已经 Seek 到末尾，没有 KeyValue 记录了。

4 然后来看 Next()和 Prev()操作。与 Seek 类似，这两个操作也主要依赖于 Block::Iter 的实现，同时也需要 SkipEmptyDataBlocks。对于 Next()来说实现起来很简单，只需解析出下一个 KeyValue 即可。而 Prev()则要复杂一些，但只需要找到在当前位置处之前的那个重启点，然后再找到当前位置的前一个 KeyValue 记录即可。

3.1.2. 代码分析

相关代码主要集中在：table/table.cc table/format.cc table/block.cc table/iterator.cc table/two_level_iterator.cc

table.cc 负责完成 sstable 格式的文件的读取。首先在 Open 函数中，通过 RandomAccessFile seek 到文件末尾，取出 footer 信息，然后进行解码，然后根据 footer 中的 index_handle，读出 index_block 块的内容，该块包含了所有数据块的信息，在 index_block 中的每条记录，key 值是满足“大于等于对应的 data block 的最大 key，而小于该 block 的下一个 block 的最小 key 值”的某个 key，该 key 值不一定是某个出现在 block 中的 key 值。value 值则是该 block 对应的 encoded BlockHandle。这样一个 BlockHandle 实际上就对应着一个具体的 block。

也就是说通过 Open 函数，已经将 index_block 读入了内存。之后用户就可以通过 Table::NewIterator 返回一个 TwoLevelIterator 来迭代所有的 block 下的 KeyValue 值。

而对于一个 TwoLevelIterator，它实际上会首先创建一个在 index_block 上的 iterator，然后我们知道 index_block 中的一个 KeyValue 实际上又对应着一个 data block，这样实际上就是一个两层循环，最外层对 index_block 的 KeyValue 进行迭代，内层根据这个 KeyValue 找到对应的那个 block，再完成该 block 的读取。

通过如下接口可以读取 table 内的某个 block，它会返回在该 block 上的一个 Iterator。index_value 实际上就是一个 BlockHandle 经过编码后的值。

```
Iterator* Table::BlockReader(void* arg,
                             const ReadOptions& options,
                             const Slice& index_value)
```

由于系统会进行 block 级别的缓存，因此该函数会先查看 cache 中是否存在该 block。不存在才需要重新去文件读取。每个 Table 对象具有一个独立的 cache_id 与之对应。

BlockReader 内部会调用 format.cc 中的 ReadBlock 函数完成 block 文件读取。它首先会根据 block handle，通过 RandomAccessFile 读取出 block 的内容，如果开启了 verify_checksums，会先进行一个 CRC 校验，然后会查看该 block 中的 Type 值，如果是进行了压缩，还需要进行解压。

最后就会根据原始的纯 KeyValue 部分的数据，生成一个 Block 对象。由该 Block 对象(在 block.cc 中)负责单个 block 数据的解析和读取，它主要提供了两个接口：
Block(const char* data, size_t size);
用户可以根据指定的内存中的一块数据，构建 block 对象。然后通过函数：
Iterator* NewIterator(const Comparator* comparator);
提供在该 block 上的迭代器，可以通过该迭代器对 block 内部的 key value 对进行访问。具体的迭代器实现由其内部类 Iter 负责。在这里，Comparator 主要用于 seek 操作，对于遍历操作来说则是不需要的。

具体的 table 格式相关的定义主要在 format.h 和 format.cc 中。BlockHandle，Footer 及某些常量的定义被放到了 format.h。而具体的编解码实现，及 block 读取后进行的 crc 校验及解压过程则放到了 format.cc 中。

3.2. 写文件

3.2.1. 基本过程

文件写入就是一个不断 append 数据的过程。用户创建 TableBuilder 对象后，不断调用 Add(key,value)接口，最后调用 Finish()结束。具体过程根据前面的 SSTable 格式图就可以了解个大概。

3.2.2. 代码分析

相关代码主要集中在：table/ block_builder.cc table/ table_builder.cc

block_builder.cc 负责根据前缀压缩及重启点等机制进行 block 数据的写入，主要提供了如下接口：
BlockBuilder(const Options* options);
Add(const Slice& key, const Slice& value);
Slice Finish();
size_t CurrentSizeEstimate() const;
bool empty() const;

用户需要保证此次 Add 调用的 key>上次调用的 key，其内部有一个变量 last_key_ 用于记录上次 Add 的 key。Add 调用中会通过 assert 进行检查。

用户可以通过 Options 来设定 block_restart_interval 大小，来控制隔多少条记录创建一个重启点。

与 block_builder.cc 类似，table_builder.cc 负责将数据按照定义格式生成后写入文件，也有一个 Add 接口。void Add(const Slice& key, const Slice& value);

同时用户也要保证写入的记录是排好序的，即此次 Add 调用的 key>上次调用的 key，其内部也有一个变量 last_key_用于记录上次 Add 的 key。Add 调用中会通过 assert 进行检查。同时 Add 调用会检查当前 block 大小是否已经>=options.block_size，如果满足该条件，就会触发 Flush()操作，Flush 操作会调用 WriteBlock 操作，同时会调用 WritableFile 的 Flush 操作。也由此可见，实际的 block 大小是有可能大于 options.block_size 大小的。

WriteBlock 操作负责进行 block 的压缩，crc 校验值的生成，crc 校验根据 block 的原始数据及 type 字段生成。所以最终一个 block 将会由三部分组成：
block_data: uint8[n]
type: uint8
crc: uint32
在生成一个 block 时，需要做的一件事情就是将该 block 对应的 block handle 放到 index_block 中，由于 block handle 的 key 的确定需要看到下一个 block 的 key，因此 block handle 的添加是会被延迟到下一个 block 的第一个 key 到来时，这是通过变量 r->pending_index_entry 来控制的，该变量会在 Flush()中被置为 true，然后在 Flush 之后的最近一次 add 操作中，会将上一个 block 的 block handle 加入到 index_block 中，然后将它重新被置为 false。

在此处也进行了类似于 block 中 key 使用前缀压缩机制来节省空间的方法。在生成 block 对应的 block handle 时，它会查看该 block 的最后一个 key 及下一个 block 的第一个 key。比如如果当前 block 的最后一个 key 是"the quick brown fox" 而下一个 block 的第一个 key 是 "the who"，它不会直接选择"the quick brown fox"作为该 block 对应的 block handle 的 key，而是可以选择"the r"，因为它更短，而且满足了”>= 当期 block 的所有 key 及<后续的 block 的所有 key”这一条件。具体会通过 Comparator 的 FindShortestSeparator 方法找到这个 key。对于最后的那个 block 的 block handle 的处理要特殊些，因为它后面没有 block 了，所以它是通过 Comparator 的 FindShortSuccessor 来确定的。

在 Finish 操作中，会进行 meta_index_block，index_block 和 footer 的写入。

4. 与 HFile 的对比分析

HBase 的 HFile 目前已经有两个版本，从最初的 V1 到现在的 V2，格式发生了很大的变化。与 V1 版本相比，V2 版本主要做了两个改进，这两个改进都主要是为了降低内存使用和启动时间，思路都是将索引信息分散到多个 block，填满就写出去，而不是集中存放到文件末尾，这也降低了 writer 的内存占用。

1. 增加了树状结构的数据块索引(data block index)支持。原因是在数据块的索引很大时，很难全部 load 到内存，比如当前的一个 data block 会在 data block indxe 区域对应一个数据项，假设每个 block 64KB，每个索引项 64Byte，这样如果每条机器上存放了 6TB 数据，那么索引数据就得有 6GB，因此这个占用的内存还是很高的。通过对这些索引以树状结构进行组织，只让顶层索引常驻内存，其他索引按需读取并通过 LRU cache 进行缓存，这样就不需要将全部索引加载到内存。

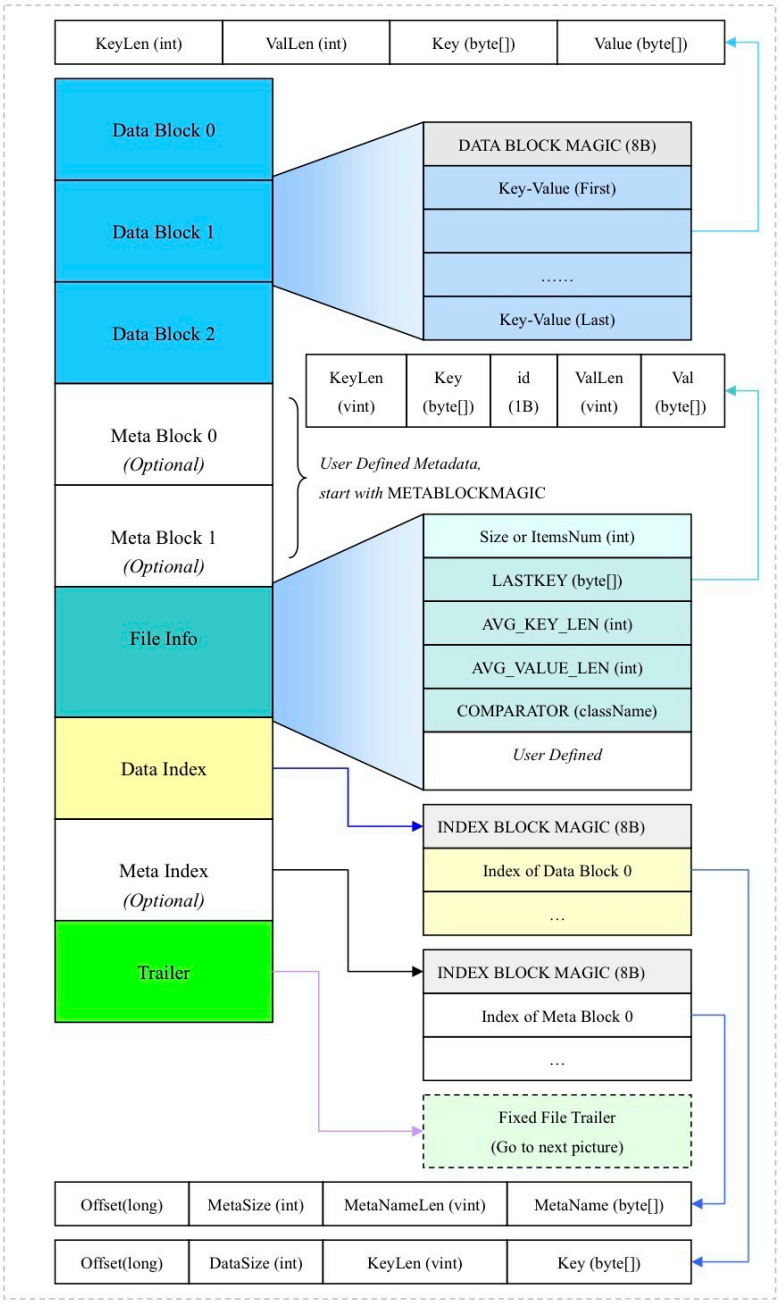
相当于把原先平坦的索引结构以树状的结构进行分散化的组织。现在的 index block 也是与 data block 一样是散布到整个文件之中，而不再是单纯的在结尾处。同时为支持对序列化数据进行二分查找，还为非 root 的 block index 设计了新的"non-root index block"。

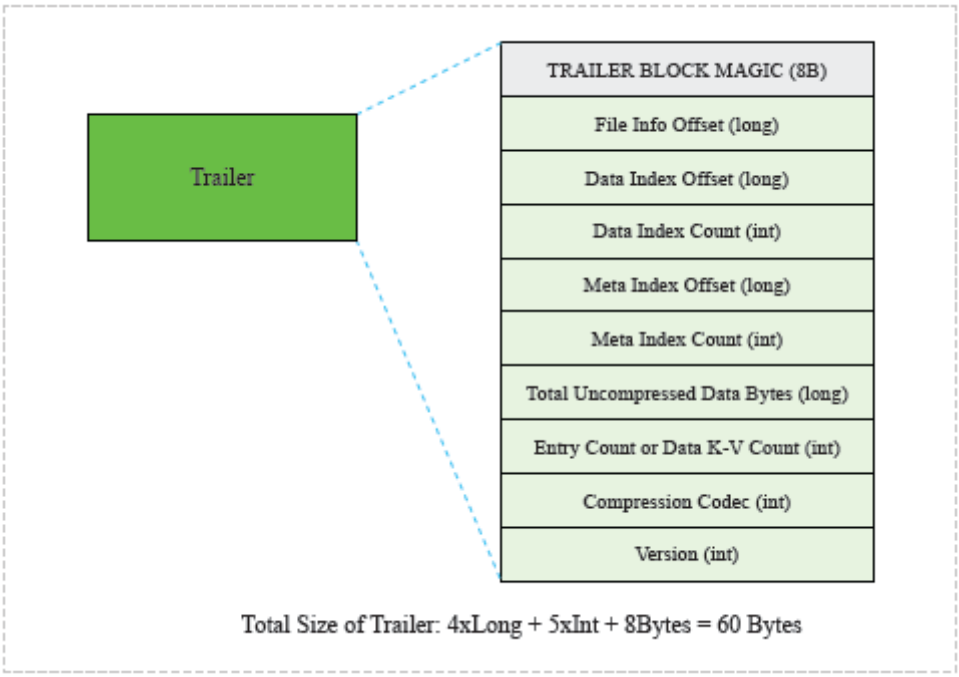
具体实现来看，比如在写入 HFile 时，在内存中会存放当前的 inline block index，当 inline block index 大小达到一定阈值(比如 128KB)时就直接 flush 到磁盘，而不再是最后做一次 flush，这样就不需要在内存中一直保持所有的索引数据。当所有的 inline block index 生成之后，HFile writer 会生成更上一级的 block index，它里面的内容就是这些 inline block index 的 offset，依次递归，逐步生成更上层的 block index，上层的包含的就是下层的 offset，直到最顶层大小小于阈值时为止。所以整个过程就是自底向上的通过下层 index block 逐步构建出上层 index block。

2. 之前的 bloom filter 数据是存放在单独的一个 meta block 里，新版里它将可以被存为多个 block。优点类似于第一个改动，这就允许在处理一个查询时不必将所有的数据都 load 到内存。

此处我们主要拿 HFile V1 与 LevelDB SSTable 进行对比，因为这两者渊源更深一些，也更类似一些。

4.1. HFile V1 文件格式





4.2. 对比分析

- SSTable 对 Key 采用了前缀压缩，HFile V1 不支持前缀压缩，HFile V2 中已经支持前缀压缩。对于 Key 的查找来说，在单个 block 内部，HFile V1 需要顺序遍历，而 SSTable 可以进行二分。
- SSTable 中随处可见 varint 编码，HFile V1 中只有个别字段是 varint。
- SSTable 支持 CRC 校验，HFile 不支持 CRC 校验，当然实际上底层的 HDFS 已经提供了类似功能。
- SSTable 将压缩类型存在了每个 Block 的元信息中，HFile V1 则存在了底部的 Trailer 中。SSTable 对于索引中的 key 进行了优化，尽量采用短的字串。
- SSTable 将文件统计信息作为一个 MetaBlock 来处理，而 HFile V1 具有一个独立的 FileInfo 部分。
- SSTable 中对于 MetaBlock 支持还不具备，没有 BloomFilter，而 HFile 中有，可以快速检测一个 key 是否在该文件中。

总结：SSTable 的结构简单，不同类型的 Block 都是同构的，对于数据的压缩校验考虑的比较多，但目前开源的这个版本，缺少对 BloomFilter(关于 BloomFilter 的设计方案已经提出，估计很快就会加入)等 MetaBlock 的支持。HFile 的结构繁杂不够清晰，且本身缺乏某些重要特性，比如 CRC 校验。

5. 性能因素

影响到性能且与 SSTable 文件直接相关的参数主要由如下几个：
block_size，block_restart_interval，compression，verify_checksums。
其中除 verify_checksums 是在读时设置外，其余均是在写时设置。

5.1. Block 大小

SSTable 的 block 大小默认是 64KB。HFile 的 Block 大小默认也是 64KB(or 65535 bytes)。下面是 HFile JavaDoc 中的注释：
“Minimum block size。通常的使用情况下，我们推荐将最小的 block 大小设为 8KB 到 1MB。如果文件主要用于顺序访问，应该用大一点的 block 大小。但是，这会导致低效的随机访问(因为有更多的数据需要进行解压)。对于随机访问来说，小一点的 block 大小会好些，但是这可能需要更多的内存来保存 block index，同时可能在创建文件时会变慢(因为我们必须针对每个 data block 进行压缩器的 flush)。另外，由于压缩编码器的内部缓存机制的影响，最小可能的 block 大小大概是 20KB-30KB”。

也就是说 block 大小受用户访问模式，数据本身特点，压缩算法，实际中可通过实验来选择合适的 block 大小。通常来说单个 KeyValue 越大，所选择的 Block 大小也应该越大。

5.2. 重启点区间大小

重启点区间长度要>=1，如果=1，实际上相当于不进行前缀压缩。同时对于 Block 来说，末尾的重启点数组承担着 Block 内部 KeyValue 索引的作用，随机读取时会据此进行二分查找。因此重启点区间越大，则前缀压缩越充分，但二分查找的粒度会变大，key 的查找速度会下降。

5.3. 压缩

压缩是以 CPU 换 IO 的，是否开启压缩要看瓶颈在哪里，如果 IO 是瓶颈，那么应该开启压缩，如果 CPU 是瓶颈则可不开压缩，具体问题还要具体分析。

5.4. CRC

用户可以选择是否在读取时打开 CRC 校验，但是无法选择在生成文件时不进行 CRC 校验。

6. 参考文献

http://code.google.com/p/leveldb/

http://leveldb.googlecode.com/svn/trunk/doc/index.html

HFile：A Block-Indexed File Format to Store Sorted Key-Value Pairs

Data Block Encoding of KeyValues (aka delta encoding / prefix compression)

7. 附录

7.1. crc32c_defs.h

```
#ifndef CRC32C_DEFS_H_
#define CRC32C_DEFS_H_

/*
 * This is the CRC32c polynomial, as outlined by Castagnoli.
 *
x^32+x^28+x^27+x^26+x^25+x^23+x^22+x^20+x^19+x^18+x^14+x^13+x^11+x^10+
x^9+
 * x^8+x^6+x^0
 */
#define CRC32C_POLY_LE 0x82F63B78

/* How many bits at a time to use.    Valid values are 1, 2, 4, 8, 32 and 64. */
/* For less performance-sensitive, use 4 */
#ifdef CONFIG_CRC32C_SLICEBY8
# define CRC32C_BITS 64
#endif
#ifdef CONFIG_CRC32C_SLICEBY4
# define CRC32C_BITS 32
#endif
#ifdef CONFIG_CRC32C_SARWATE
# define CRC32C_BITS 8
#endif
#ifdef CONFIG_CRC32C_BIT
# define CRC32C_BITS 1
#endif
```

```
#ifndef CRC32C_BITS
# define CRC32C_BITS 64
#endif

/*
 * Little-endian CRC computation.  Used with serial bit streams sent
 * lsbit-first.  Be sure to use cpu_to_le32() to append the computed CRC.
 */
#if CRC32C_BITS > 64 || CRC32C_BITS < 1 || CRC32C_BITS == 16 || \
    CRC32C_BITS & CRC32C_BITS-1
# error "CRC32C_BITS must be one of {1, 2, 4, 8, 32, 64}"
#endif

#endif /* CRC32C_DEFS_H_ */
```

7.2. gen_crc32ctable.c

```
#include <stdio.h>
#include "crc32c_defs.h"
#include <inttypes.h>

#define ENTRIES_PER_LINE 4

#if CRC32C_BITS <= 8
#define LE_TABLE_SIZE (1 << CRC32C_BITS)
#else
#define LE_TABLE_SIZE 256
#endif

static uint32_t crc32c_table[8][256];

/**
 * crc32c_init() - allocate and initialize LE table data
 *
 * crc is the crc of the byte i; other entries are filled in based on the
 * fact that crctable[i^j] = crctable[i] ^ crctable[j].
 *
 */
```

```
static void crc32c_init(void)
{
    unsigned i, j;
    uint32_t crc = 1;

    crc32c_table[0][0] = 0;

    for (i = LE_TABLE_SIZE >> 1; i; i >>= 1) {
        crc = (crc >> 1) ^ ((crc & 1) ? CRC32C_POLY_LE : 0);
        for (j = 0; j < LE_TABLE_SIZE; j += 2 * i)
            crc32c_table[0][i + j] = crc ^ crc32c_table[0][j];
    }
    for (i = 0; i < LE_TABLE_SIZE; i++) {
        crc = crc32c_table[0][i];
        for (j = 1; j < 8; j++) {
            crc = crc32c_table[0][crc & 0xff] ^ (crc >> 8);
            crc32c_table[j][i] = crc;
        }
    }
}

static void output_table(uint32_t table[8][256], int len, char trans)
{
    int i, j;

    for (j = 0 ; j < 8; j++) {
        printf("static const u32 t%d_%ce[] = {", j, trans);
        for (i = 0; i < len - 1; i++) {
            if ((i % ENTRIES_PER_LINE) == 0)
                printf("\n");
            printf("to%ce(0x%8.8xL),", trans, table[j][i]);
            if ((i % ENTRIES_PER_LINE) != (ENTRIES_PER_LINE - 1))
                printf(" ");
        }
        printf("to%ce(0x%8.8xL)};\n\n", trans, table[j][len - 1]);

        if ((j+1)*8 >= CRC32C_BITS)
            break;
    }
}

int main(int argc, char **argv)
{
    printf("/*\n");
```

```
    printf(" * crc32c_table.h - CRC32c tables\n");
    printf(" *        this file is generated - do not edit\n");
    printf(" *    # gen_crc32ctable > crc32c_table.h\n");
    printf(" *        with\n");
    printf(" *    CRC32C_BITS = %d\n", CRC32C_BITS);
    printf(" */\n");

    if (CRC32C_BITS > 1) {
            crc32c_init();
            output_table(crc32c_table, LE_TABLE_SIZE, 'l');
    }

    return 0;
}
```