LevelDB

Options

Optins

ReadOptins

WriteOptins

LevelDB 重要数据结构

DB format

hello

WAL

Memtable

WAL

SSTable

SSTable 是 `Sorted String Table` 的缩写，是用于持久化有序键值对的数据结构。

这一部分涉及的代码为：

- `table/*`

在 LevelDB 中，SST 的文件包含以下几块：

1. data blocks：存放了 Key-Value 数据
2. meta blocks：在 LevelDB 中是存放了 BloomFilter 的数据
3. meta index block：
4. index block：
5. footer：
    1. `metaindex_handle`：指向 meta index block 的 `BlockHandle`
    2. `index_handle`：指向 index block 的 `BlockHandle`
    3. `padding`：
    4. `magic_number`：魔数，用于校验文件是否是 SST 文件

Block

`Block` 的作用：

1. 保存 `BlockContents` 转换后的数据，存储在 `Cache` 中。
2. 由于 SST 中存储的 Block 存在多个 item，因此需要一个迭代器来遍历。

Version

这一部分涉及的代码为：

- `db/version_edit.{h,cc}`
- `db/version_set.{h,cc}`

执行流程

打开流程

读流程

写流程

删除流程

快照

合并流程

组件

Slice

这部分涉及的代码为：

- `include/leveldb/slice.h`

在 LevelDB 中并没有使用 C++自带的 `std::string`，而是封装了一个 `Slice` 类，用于表示字符串片段。

`Slice` 包含了一个指向字符串的指针 `data_`和字符串的长度 `size_`。它不管理内存，只是对现有数据进行轻量级封装。正是因为不管理内存，所以 `Slice` 对象可以被轻松地拷贝和赋值，不回引入额外的内存开销，这对于高性能的数据库来说是非常重要的。

整个 `Slice` 的代码非常简单，主要有两个比较需要注意的函数：

- `remove_prefix`：删除前 `n` 个字节。
- `start_with`：判断是否以某个字符串开头。

Status

这部分涉及的代码为：

- `util/status.cc`

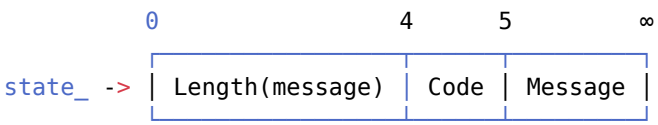
在 LevelDB 中，许多操作都需要通过返回码来表示操作的结果，并且通过返回码来决定下一步的工作。

LevelDB 中通过 `Status` 来表示操作的结果，`Status` 是一个类，它包含了一个状态码和一个消息。状态码是一个枚举类型，它表示了操作的结果，消息是一个字符串，它描述了操作的结果。

另外需要说明一下，在 LevelDB 中，`Code` 是不支持拓展的。

`Status` 中最主要的一个成员是 `state_`，它是一个指向状态码和消息的指针。`state_`的结构如下：

```
enum Code {
  kOk = 0,
  kNotFound = 1,
  kCorruption = 2,
  kNotSupported = 3,
  kInvalidArgument = 4,
  kIOError = 5
};
```



在看了 `Status` 的结构以后，我们可以看一下 `Status` 的几个函数的实现。

如果看懂了上面的这三个函数，那么关于 `Status` 的其他代码就不难理解了，这里就不多赘述了。

变长编码

这部分涉及的代码为：

- `util/coding.{h,cc}`

下面使用一个例子来演示 varint 的编解码过程。

编码

下面这个表示将 `123456` 编码为 `11000000 11000100 00000111`。

```
            123456 = 1 11100010 01000000   # 十进制变成二进制
                     111 1000100 1000000   # 分成7位一组
                 1000000 1000100 0000111   # 变成小端编码(little-endian)
              11000000 11000100 00000111   # 添加msb
              ^msb      ^msb     ^msb
```

解码

下面这个表示的是将 `10010110 00000001` 解码为 `150`。

```
                  10010110 00000001   # 原始数据
                  ^ msb    ^ msb
                   0010110  0000001   # 去掉 msb
                   0000001  0010110   # 变成大端编码(big-endian)
            00000010010110 = 150      # 拼接且转换为十进制
```

在知道了什么是变长编码以后，另一个问题随之而来，那就是为什么使用变长编码？这个问题可以参看在 stackoverflow 中有一个问题：为什么 Varint 是一种高效的数据表示方式？[1]

参考回答，可以总结出来几点：

- 优点：
  - 在实践中，大多数整数都是小整数，所以使用变长编码可以节省空间。
  - 由于变长编码是变长的，所以可以表示任意大小的整数。
- 缺点：
  - 由于变长编码是变长的，所以在解码的时候需要额外的计算。

下面来看一下 LevelDB 中是如何编码以及解码的。

上面两端代码其实完成了同一个功能，这两段代码的逻辑是一样的，但是 64 位整数的编码感觉更加优雅一些。

下面来看看解码过程。

在这里可以看到 LevelDB 的作者的巧思，`GetVarint32Ptr` 与 `GetVarint64Ptr` 两个函数并没有写的相同，而是假设了大多数情况下处理的都是比较小的数字，在 `GetVarint32Ptr` 中直接处理了一个字节的情况。只有当处理不了的时候（也就是 Varint 大于 1 个字节的时候），才会调用与 `GetVarint64Ptr` 类似的 `GetVarint32PtrFallback` 函数。

关于编码的其他的部分都比较简单，就不再说明了。

[1] https://stackoverflow.com/questions/24614553/why-is-varint-an-efficient-data-representation

迭代器 Iterator

在 LevelDB 中的迭代器的实现还是比较优雅的。

跳表

这部分涉及的代码为：

- db/skiplist.h

内存池 Arena

这一部分涉及的代码为：

- `util/arena.{h,cc}`

这里姑且将 Arena 翻译为内存池。之前我一直以为我没有英文环境，所以不太理解这个词的含义。直到我在 stackoverflow 看到了这个问题：What is the meaning of the term arena in relation to memory?[2]（arena 与内存有什么关系？），我才意识到好像 native English speaker 也不知道这个词的含义，笑。

[2] https://stackoverflow.com/questions/12825148/what-is-the-meaning-of-the-term-arena-in-relation-to-memory

布隆过滤器 BloomFilter

这一部分涉及的代码为：

- `util/bloom.cc`

LRU cache

这一部分涉及的代码为：

- `util/cache.cc`

跨平台以及可移植性

关于符号导出

在这个 commit 中[3]，定义了一个 `LEVELDB_EXPORT` 宏。并且在许多地方使用。

还有这个 commit[4]

在 Windows 平台上，我们需要在动态链接库中导出符号，以便于其他程序可以调用这些函数。在 Windows 平台上，我们需要使用 `__declspec(dllexport)` 来导出符号。在 Linux 平台上，我们需要使用 `__attribute__((visibility("default")))` 来导出符号。为了解决这个问题，我们可以使用 `#if` 来判断当前的编译环境，然后使用不同的宏来导出符号。

[3] https://github.com/google/leveldb/commit/4a7e7f50dcf661cfffe71737650b0fb18e195d18
[4] https://github.com/google/leveldb/commit/aece2068d7375f987685b8b145288c5557f9ce50

循环冗余校验 CRC

参考《A PAINLESS GUIDE TO CRC ERROR DETECTION ALGORITHMS》[5]

[5]http://www.ross.net/crc/download/crc_v3.txt

原子指针

在现在的 LevelDB 中早已移除，参考 commit[6]

附录

The Log-Structured Merge-Tree (LSM-Tree)

LSMTree[7]

LSM-based storage techniques: a survey

LSMTree 探讨论文[8]

RibbonFilter

CuckooFilter

参考

[6]https://github.com/google/leveldb/commit/7d8e41e49b8fddda66a2c5f0a6a47f1a916e8d26

[7]论文链接:https://www.cs.umb.edu/~poneil/lsmtree.pdf

[8]论文链接:https://doi.org/10.1007/s00778-019-00555-y