1. LMDB基本使用方式

1.1 打开数据库流程

STEP1: 创建environment对象：mdb_env_create() ->

STEP2: 设置environment对象最大数据库数量：mdb_env_set_maxdbs() ->

STEP3：设置Max file size：mdb_env_set_mapsize() ->

STEP4: 打开environment对象： mdb_env_open() ->

STEP5: 创建transaction对象：mdb_txn_begin() ->

STEP6: 打开数据库： mdb_dbi_open() ->

STEP7: 提交transaction：mdb_txn_commit()

注意：

1. 在同一个进程中不要多次调用mdb_env_create()打开同一个database；否则，当同一个进程中多个线程都调用mdb_env_create()获取锁打开同一个database，其中某一个线程close这个数据库environment对象以后会释放掉这个锁，但是其他线程还在正常使用这个数据库，这时另一个进程调用mdb_env_create()然后打开这个database，那么就会有多个写进程，造成数据冲突。

2. mdb_env_set_maxdbs()必须在mdb_env_create()之后，mdb_env_open()之前调用

1.2 数据操作

1.2.1 读数据

STEP1: 创建transaction对象：mdb_txn_begin() ->

STEP2: 读数据： mdb_get() ->

STEP3: 提交transaction：mdb_txn_commit() 或者 mdb_txn_abort()

1.2.2 写数据

STEP1: 创建transaction对象：mdb_txn_begin() ->

STEP2: 更新数据： mdb_put() ->

STEP3: 提交transaction：mdb_txn_commit()

1.2.3 删除数据

STEP1: 创建transaction对象：mdb_txn_begin() ->

STEP2: 删除数据： mdb_del() ->

STEP3: 提交transaction：mdb_txn_commit()

1.2.4 Batch operation

实际上，上述读写删除操作都是数据量为一个record的batch 操作，当有多个数据操作时，方法如下：

STEP1: 创建transaction对象：mdb_txn_begin() ->

STEP2: mdb_put() ->

STEP N：...

STEP N+1：mdb_del() ->

STEP N+2：提交transaction：mdb_txn_commit()

1.2.5 游标遍历

STEP1: 创建transaction对象：mdb_txn_begin() ->

STEP2: 打开游标：mdb_cursor_open()

STEP3：mdb_cursor_get()

STEP4：游标还有数据转STEP3，否则STEP5

STEP5：abort或者提交transaction：mdb_txn_commit()/mdb_txn_abort()

1.3 关闭数据库

STEP1: close数据库对象：mdb_dbi_close() ->

STEP2: close environment对象：mdb_env_close()

注意：

1. 不要修改从LMDB API返回的MDB_val对象（比如 free其中的data），这些数据可能是直接从mmap上出来的数据，并且，如果在本次transaction结束之后还需要用这些数据，提前拷贝一份，不然这些数据在transaction结束以后就不可用了。

2. 在mdb_env_close()之前需要保证所有transaction，database和cursor都已经关闭

2. LMDB参数设置与性能调优

2.1 mdb_env_create()参数设置

```
* <ul>
      *   <li>#MDB_WRITEMAP
      *       Use a writeable memory map unless MDB_RDONLY is set. This uses
```

```
 *      fewer mallocs but loses protection from application bugs
 *      like wild pointer writes and other bad updates into the database.
 *      This may be slightly faster for DBs that fit entirely in RAM, but
 *      is slower for DBs larger than RAM.
 *      Incompatible with nested transactions.
 *      Do not mix processes with and without MDB_WRITEMAP on the same
 *      environment.  This can defeat durability (#mdb_env_sync etc).
 *  <li>#MDB_NOMETASYNC
 *      Flush system buffers to disk only once per transaction, omit the
 *      metadata flush. Defer that until the system flushes files to disk,
 *      or next non-MDB_RDONLY commit or #mdb_env_sync(). This optimization
 *      maintains database integrity, but a system crash may undo the last
 *      committed transaction. I.e. it preserves the ACI (atomicity,
 *      consistency, isolation) but not D (durability) database property.
 *      This flag may be changed at any time using #mdb_env_set_flags().
 *  <li>#MDB_NOSYNC
 *      Don't flush system buffers to disk when committing a transaction.
 *      This optimization means a system crash can corrupt the database or
 *      lose the last transactions if buffers are not yet flushed to disk.
 *      The risk is governed by how often the system flushes dirty buffers
 *      to disk and how often #mdb_env_sync() is called.  However, if the
 *      filesystem preserves write order and the #MDB_WRITEMAP flag is not
 *      used, transactions exhibit ACI (atomicity, consistency, isolation)
 *      properties and only lose D (durability).  I.e. database integrity
 *      is maintained, but a system crash may undo the final transactions.
 *      Note that (#MDB_NOSYNC | #MDB_WRITEMAP) leaves the system with no
 *      hint for when to write transactions to disk, unless #mdb_env_sync()
 *      is called. (#MDB_MAPASYNC | #MDB_WRITEMAP) may be preferable.
 *      This flag may be changed at any time using #mdb_env_set_flags().
 *  <li>#MDB_MAPASYNC
 *      When using #MDB_WRITEMAP, use asynchronous flushes to disk.
 *      As with #MDB_NOSYNC, a system crash can then corrupt the
 *      database or lose the last transactions. Calling #mdb_env_sync()
 *      ensures on-disk database integrity until next commit.
 *      This flag may be changed at any time using #mdb_env_set_flags().
 *  <li>#MDB_NOTLS
 *      Don't use Thread-Local Storage. Tie reader locktable slots to
 *      #MDB_txn objects instead of to threads. I.e. #mdb_txn_reset() keeps
```

```
	 *		the slot reseved for the #MDB_txn object. A thread may use parallel
	 *		read-only transactions. A read-only transaction may span threads if
	 *		the user synchronizes its use. Applications that multiplex many
	 *		user threads over individual OS threads need this option. Such an
	 *		application must also serialize the write transactions in an OS
	 *		thread, since LMDB's write locking is unaware of the user threads.
	 *	<li>#MDB_NOLOCK
	 *		Don't do any locking. If concurrent access is anticipated, the
	 *		caller must manage all concurrency itself. For proper operation
	 *		the caller must enforce single-writer semantics, and must ensure
	 *		that no readers are using old transactions while a writer is
	 *		active. The simplest approach is to use an exclusive lock so that
	 *		no readers may be active at all when a writer begins.
	 *	<li>#MDB_NORDAHEAD
	 *		Turn off readahead. Most operating systems perform readahead on
	 *		read requests by default. This option turns it off if the OS
	 *		supports it. Turning it off may help random read performance
	 *		when the DB is larger than RAM and system RAM is full.
	 *		The option is not implemented on Windows.
	 *	<li>#MDB_NOMEMINIT
	 *		Don't initialize malloc'd memory before writing to unused spaces
	 *		in the data file. By default, memory for pages written to the data
	 *		file is obtained using malloc. While these pages may be reused in
	 *		subsequent transactions, freshly malloc'd pages will be initialized
	 *		to zeroes before use. This avoids persisting leftover data from other
	 *		code (that used the heap and subsequently freed the memory) into the
	 *		data file. Note that many other system libraries may allocate
	 *		and free memory from the heap for arbitrary uses. E.g., stdio may
	 *		use the heap for file I/O buffers. This initialization step has a
	 *		modest performance cost so some applications may want to disable
	 *		it using this flag. This option can be a problem for applications
	 *		which handle sensitive data like passwords, and it makes memory
	 *		checkers like Valgrind noisy. This flag is not needed with
#MDB_WRITEMAP,
	 *		which writes directly to the mmap instead of using malloc for pages.
The
	 *		initialization is also skipped if #MDB_RESERVE is used; the
	 *		caller is expected to overwrite all of the memory that was
	 *		reserved in that case.
```

```
        *       This flag may be changed at any time using #mdb_env_set_flags().
 * </ul>
```

上面摘抄了对性能有影响的option，下面对其中几个选项就我自己的理解做一些简单分析：

1. 对于MDB_WRITEMAP选项，我的理解是在数据写入时，直接使用mmap映射对应的虚存地址，将更新后的数据写到对应page中，后续由操作系统或者mdb_env_sync()将这个page刷盘。如果没有指定MDB_WRITEMAP，LMDB在数据写入会malloc申请内存页，然后commit的时候会将malloc申请的page刷到内核缓冲，然后由操作系统或者调用mdb_env_sync()刷盘。显然，使用MDB_WRITEMAP会减少malloc/free的调用，并且减少一次page的内存拷贝。

2. MDB_NORDAHEAD选项会关闭预读，在数据库大小超过RAM大小的情况下能提高性能。

3. MDB_NOSYNC 和 MDB_NOMETASYNC组合，在transaction提交的时候不会触发将数据刷盘的操作，而是推迟到操作系统或者调用mdb_env_sync()来刷盘。

但是，由于MDB_WRITEMAP会使写入直接在mmap上的地址上操作，在内存容量小于数据库大小是可能会造成频繁的内存换入换出，影响程序性能。所以初步使用如下组合：

(MDB_NOSYNC | MDB_NOMETASYNC | MDB_NORDAHEAD)

2.2 针对read only transaction的操作的优化

1. read only 的transaction指定MDB_RDONLY选项。因为LMDB以MVCC支持并发读，但是只能单个写，指定MDB_RDONLY能够很大提高读写效率。

2. 对于read only的transaction使用mdb_txn_reset()代替mdb_txn_commit()，这样LMDB可以重用alloc的transaction，但是只针对read only的transaction。

3. 对于read only的transaction内打开的cursor在transaction完成以后也可以重用，但是也只能应用到新的read only transaction。

这样，针对上述3点我们可以对代码进行部分优化。

2.3 针对Bulk Loading的优化

```
<li>#MDB_APPEND - append the given key/data pair to the end of the
        *       database. This option allows fast bulk loading when keys are
        *       already known to be in the correct order. Loading unsorted keys
        *       with this flag will cause a #MDB_KEYEXIST error.
```

不适用以太坊，其要求所有的keys都是排序了的。