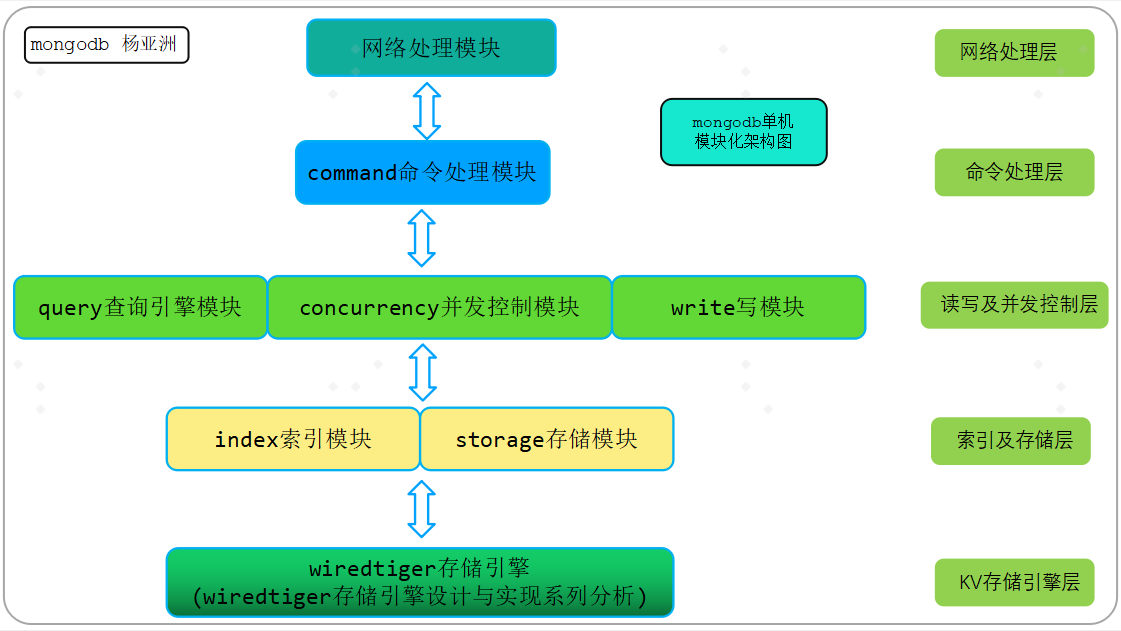
**storage存储模块设计与实现(一)-体验mongodb多引擎模块化及插件式支持设计**

前面我们分析了mongodb内核源码设计系列中的《网络处理模块设计与实现》、《command命令处理模块设计与实现》、《write(增、删、改）模块设计与实现》三个主要模块的核心设计。本文将分析改系列的第四个核心模块：《storage存储模块设计与实现》。



上图为mongodb单机核心模块细化拆分后的整体架构图，后续我们将围绕该架构图对剩余还没有分析的模块进行逐一详细分析。

# 抛出问题及疑问？

mongodb官方提供支持的存储引擎包括以下三种：inMemory、MMAPv1 、wiredTiger。此外，还支持第三方RocksDB、TerarkDB等引擎，mongodb内核默认数百万行，如何设计不同引擎模块插件化的方式嵌入mongodb内核将是本模块分析的一个重要原因。首先，我们抛出以下几个问题：

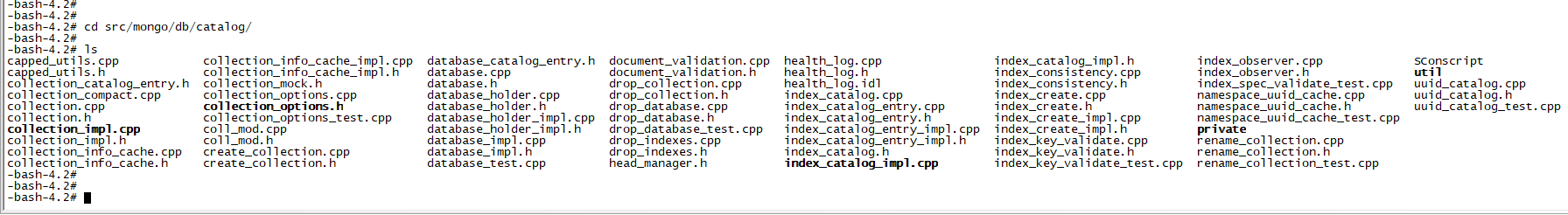
1. **mongodb存储storage模块如何通过分层设计支持不同的存储引擎？**
2. **如果自己有一个新的KV存储引擎，如何插件化嵌入到mongodb内核？**

# 2. 认识storage存储模块代码目录结构

index模块和storage模块很多功能都是复用相同代码，例如如何把数据写入wiredtiger存储引擎、如何从wiredtiger中读取KV数据、如何对数据进行KV封装以及封装过程等实现都大同小异。

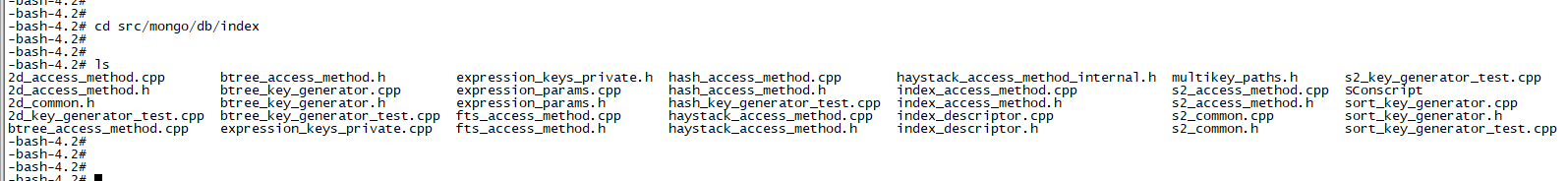
对底层wiredtiger存储引擎来说，他们本质上都是对数据表或者索引表的KV操作，索引index索引模块和storage存储模块相关代码很多都放在相同的目录，通过文件名区分属于那个模块。

下面我们来了解一下这两个模块主要代码目录结构，如下图所示：

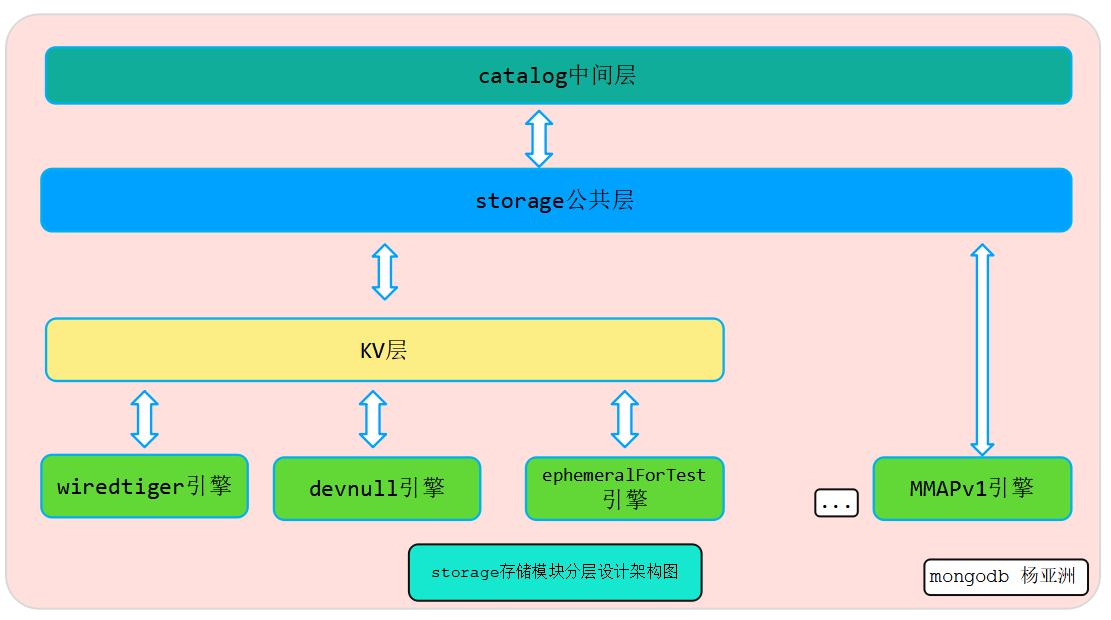




上面的src/mongo/db目录下面的catalog和storage目录就是Index模块和storage模块的核心代码实现相关目录。其中，由于index相比于storage模块拥有一些独有的特性功能，例如生成索引树KV中的KEY，索引解析等，因此index模块除了拥有storage存储模块相关目录结构外，还拥有其独有的index相关的目录，该目录为src/mongo/db/index，如下图所示：



从上面序言部分，我们大体可以看出storage模块化设计的代码结构，这种代码结构实际上有其深层次的遇意。mongodb为了满足不同存储引擎插件化方式潜入，内核按照模块化方法设计，分层架构图及功能说明如下：



* **catalog中间层**

catalog中间层把db库、collection集合、index索引相关接口抽象出来，从而对上层屏蔽掉底层存储模块实现细节。

* **storage公共层**

storage公共层主要是抽象各种不同存储存储(ephemeralForTest、MMAPv1 、devnull、wiredTiger、RocksDB、TerarkDB)操作接口，把共用的底层存储引擎接口统一抽象封装。

* **KV层**

该层只针对KV类型存储引擎，例如wiredtiger、rocksdb引擎底层都属于KV类型存储引擎，KV层主要负责把不同KV类型存储引擎共同接口功能抽象化。

* **具体存储引擎层**

该层完成不同存储引擎独有功能实现，不同引擎功能接口不一样，因此需要独立出来。

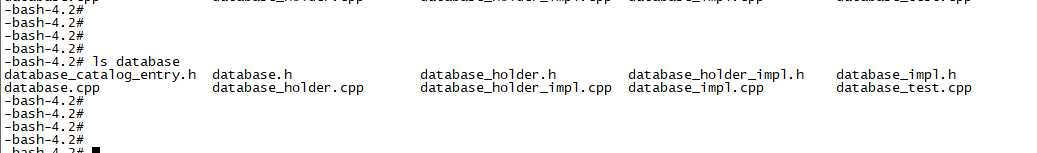
# 3. catalog中间层设计

文档存储层的读写包含对制定库的操作，也包括对制定表的操作。同时对库和表的操作有会和索引相关联，例如写数据除了写数据本身外，还需要写索引；此外，读数据也需要索引配合，这样才能快速检索获取数据。然后数据本身在存储层，所以也就在query读模块、write写(增删改)模块与storage存储层之间引入了catalog中间层。

catalog中间层主要完成以下功能：

## 3.1 database库级管理中间层

database相关库级管理主要由如下核心代码实现，如下图所示：



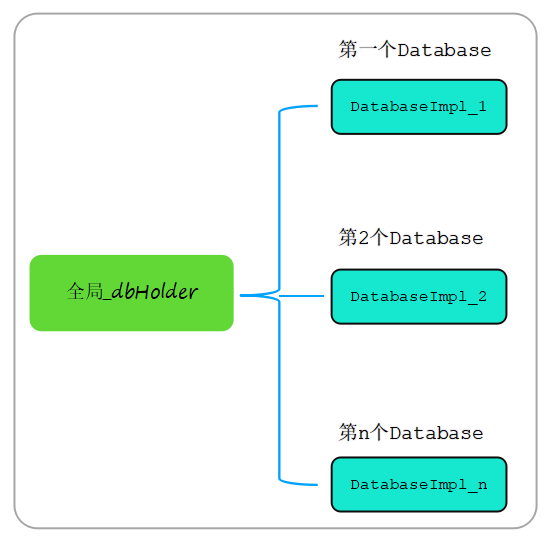
从上图代码结构可以看出，database库相关代码实现可以细分为如下几个子模块：

* **DatabaseHolder全局库管理子模块**

database相关代码文件实现所有db库的统一管理，通过一个全局\_dbHolder完成具体的封装，核心数据结构如下：

1. //定义一个全局的DatabaseHolder
2. DatabaseHolder& dbHolderImpl() {
3. static DatabaseHolder \_dbHolder;
4. return \_dbHolder;
5. }
7. class DatabaseHolderImpl : public DatabaseHolder::Impl {
8. public:
9. ......
11. //对应DatabaseImpl map表
12. typedef StringMap<Database\*> DBs;
13. //所有db保存到这里，通过DatabaseHolderImpl::openDb创建后保存到这里
14. DBs \_dbs;
15. }

从上面的核心代码可以看出，所有的Database通过DatabaseHolder全局结构统一管理起来，database\_holder\_impl.cpp代码文件主要完成Database库相关的生成、关闭、查找等，所有的Database全部存放在\_dbs这个StringMap表中，如下图所示：



database\_holder相关代码比较简单，这里不做分析，其核心接口功能说明如下表所示：

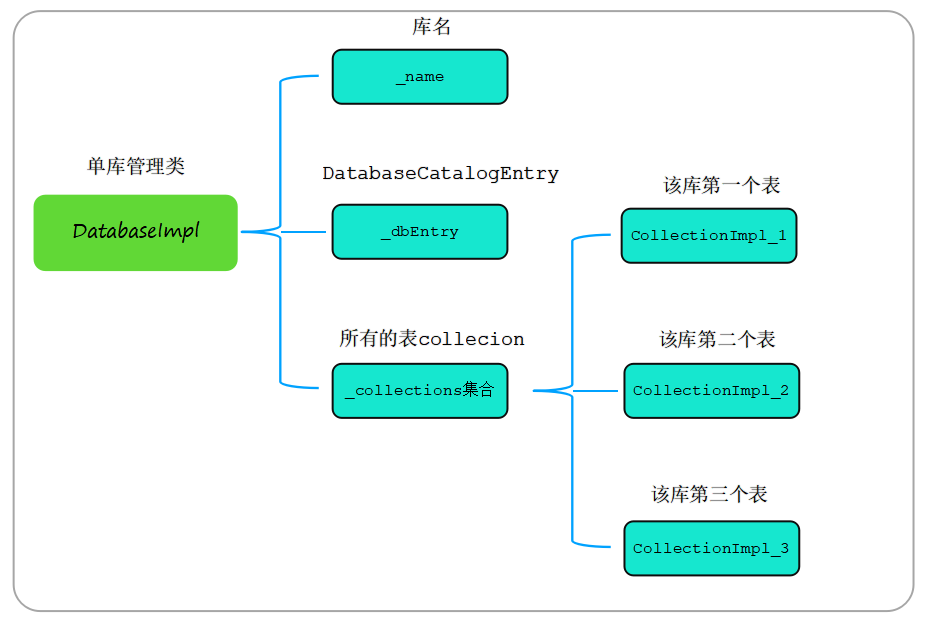
|  |  |  |
| --- | --- | --- |
| 类名 | 函数接口名 | 功能说明 |
| DatabaseHolderImpl | openDb(...) | 生成一个Database添加到\_dbs map表中 |
| close(...) | 从\_dbs map表中删除指定Database |
| closeAll(...) | 从\_dbs map表中删除所有Database |
| \_todb(...) | 从db.collection字符串中获取db信息 |
| get(...) | 从\_dbs map表中查找是否有指定Database |

* **DatabaseImpl单个库管理子模块**

DatabaseImpl对应一个指定得库，负责管理该库下面的所有表信息，其核心数据结构如下：

1. //指定库管理类，负责该库下面所有表处理
2. class DatabaseImpl : public Database::Impl {
3. public:
4. ......
5. private:
6. const std::string \_name;  // "dbname" 库名
8. //真正赋值在DatabaseHolderImpl::openDb， 默认为KVDatabaseCatalogEntryBase
9. //主要完成KV层的索引方法生成及该db库相关的元数据管理
10. DatabaseCatalogEntry\* \_dbEntry;  // not owned here
12. ......
13. //\_collections存储该DB下面所有的表
14. //DatabaseImpl::createCollection创建collection的时候添加到\_collections数组
15. CollectionMap \_collections;
16. }

该类主要包含三个核心成员：\_name、\_dbEntry、\_collections，其作用分表对应库名、该db相关元数据管理、表集合，如下图所示：



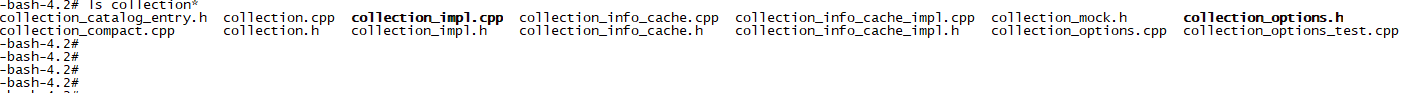
如上，一个库对应一个DatabaseImpl，每个DatabaseImpl下面可以包含多个表，这样库和表的逻辑关系就比较清晰展示了。

DatabaseImpl类核心接口功能说明如下表所示：

|  |  |  |
| --- | --- | --- |
| 类名 | 接口名 | 功能说明 |
| DatabaseImpl | init(...) | 从元数据文件\_mdb\_catalog.wt中获取所有的集合信息，然后初始化对应集合CollectionImpl信息 |
| \_getOrCreateCollectionInstance(...) | 初始化指定集合对应CollectionImpl |
| clearTmpCollections(...) | 清除临时集合信息 |
| setProfilingLevel(...) | 慢操作记录阀值设置，system.profile慢日志表创建 |
| getStats(...) | 库相关统计信息，提供给db.stats()命令使用 |
| dropCollection(...) | 删除集合，包括该集合相关的索引信息 |
| \_clearCollectionCache(...) | 从该db清除对应的指定集合缓存信息，也就是从DatabaseImpl.\_collections map表中清除某collection |
| getCollection(...) | 从db库中查找指定collection |
| renameCollection(...) | 集合重命名 |
| getOrCreateCollection(...) | 给该db下创建一个新集合，集合存在直接返回，不存在则创建 |
| dropDatabase(...) | 删除该db库及其下面的表 |
| mongo | userCreateNSImpl(...) | 建表命令参数合法性检查等 |

## 3.2 collecion集合管理中间层

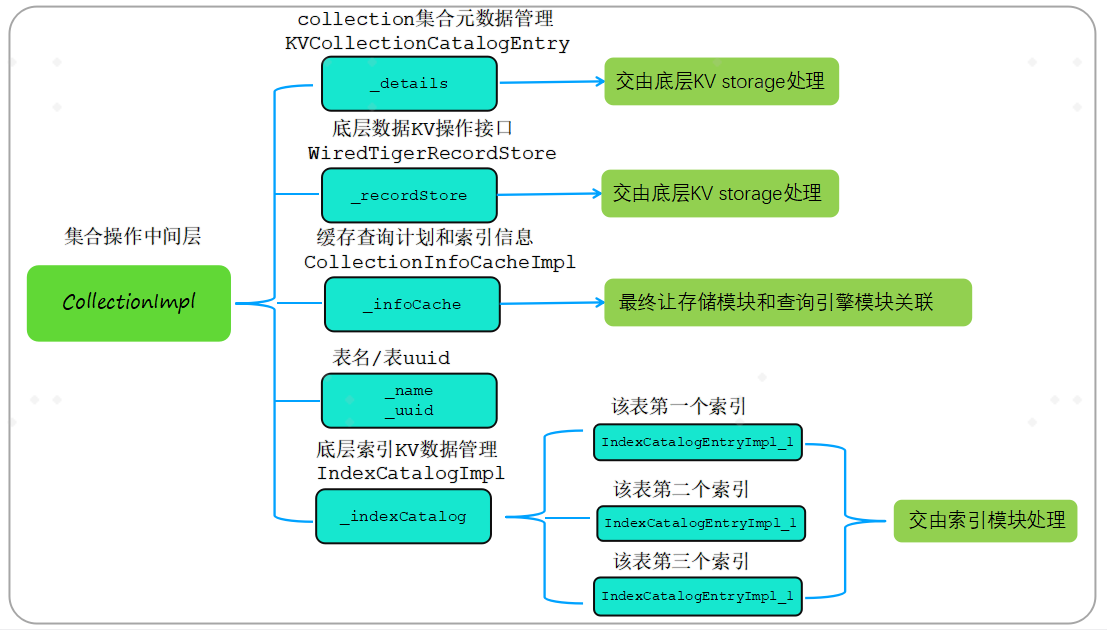
collecion表级管理核心实现主要由如下代码文件组成：



这些文件中最核心的是CollectionImpl类，该类接口主要完成底层KV数据封装，包括增、删、改、查及索引封装相关接口实现。该类主要成员变量如下：

1. class CollectionImpl {
2. ......
3. //表名
4. const NamespaceString \_ns;
5. //表对应uuid，一个表对应一个唯一uuid
6. OptionalCollectionUUID \_uuid;
8. ......
9. //数据行KV操作接口
10. RecordStore\* const \_recordStore;
11. //索引行KV操作接口实现
12. IndexCatalog \_indexCatalog;
13. ......
14. //对应CollectionInfoCacheImpl，缓存查询计划信息
15. CollectionInfoCache \_infoCache;
16. }

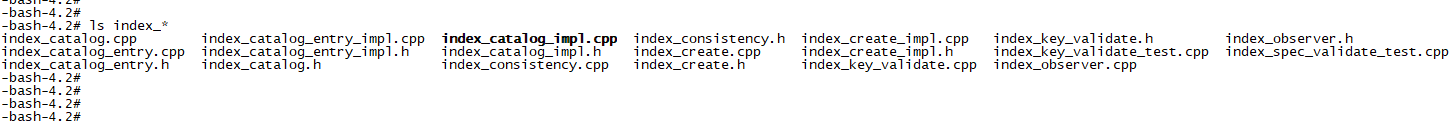
该类最核心的三个成员为\_recordStore、\_indexCatalog、\_infoCache、\_details，分别完成如下表对应功能：



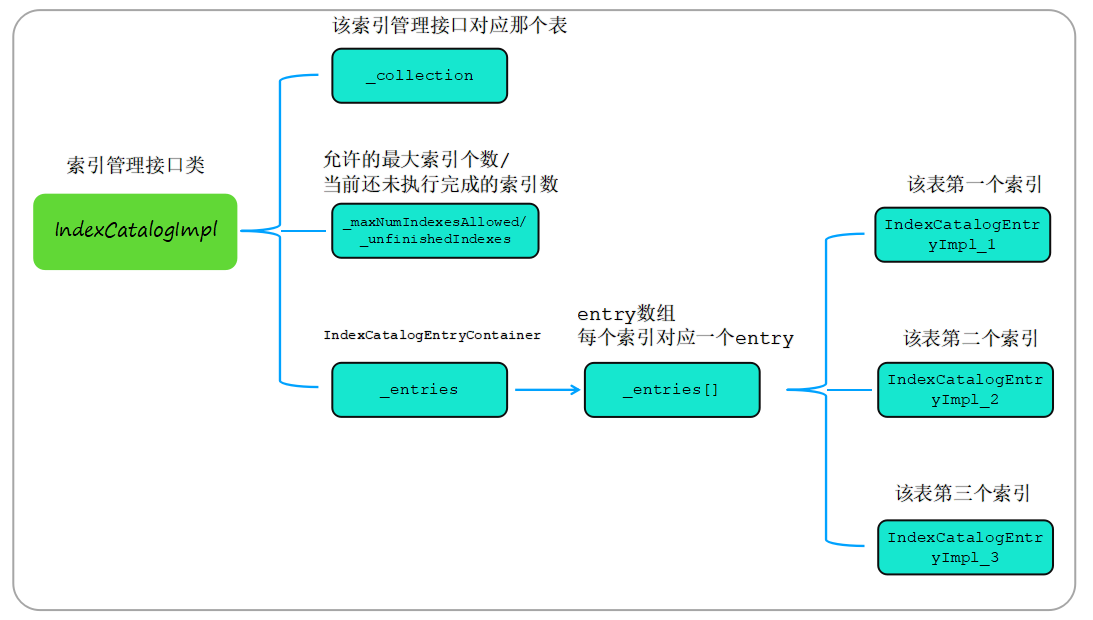
|  |  |
| --- | --- |
| 成员名 | 功能说明 |
| \_recordStore | 负责写入数据时候的KV转换组装，同时负责写入到WT存储引擎  该成员最终和KV引擎关联起来，最终实现在WiredTigerRecordStore |
| \_indexCatalog | 负责索引KV转换封装，最终负责把索引KV写入WT存储一引擎，索引模块相关章节详细说明 |
| \_details | collection元数据管理，主要针对存储引擎底层\_mdb\_catalog.wt文件对应的KV元数据管理 |
| \_infoCache | 缓存planCache和索引信息，这里和查询引擎关联起来 |

## 3.3 index索引管理中间层

index索引管理中间层核心代码文件如下：



索引中间层代码中最核心的功能由如下两个类实现：class index\_catalog\_impl{}和class index\_catalog\_entry\_impl{}，这两个类的关系如下图所示：



这两个类核心功能如下：

* index\_catalog\_impl类

该类记录管理指定表的所有索引信息。

* index\_catalog\_entry\_impl类

该类负责管理特定的某个索引。

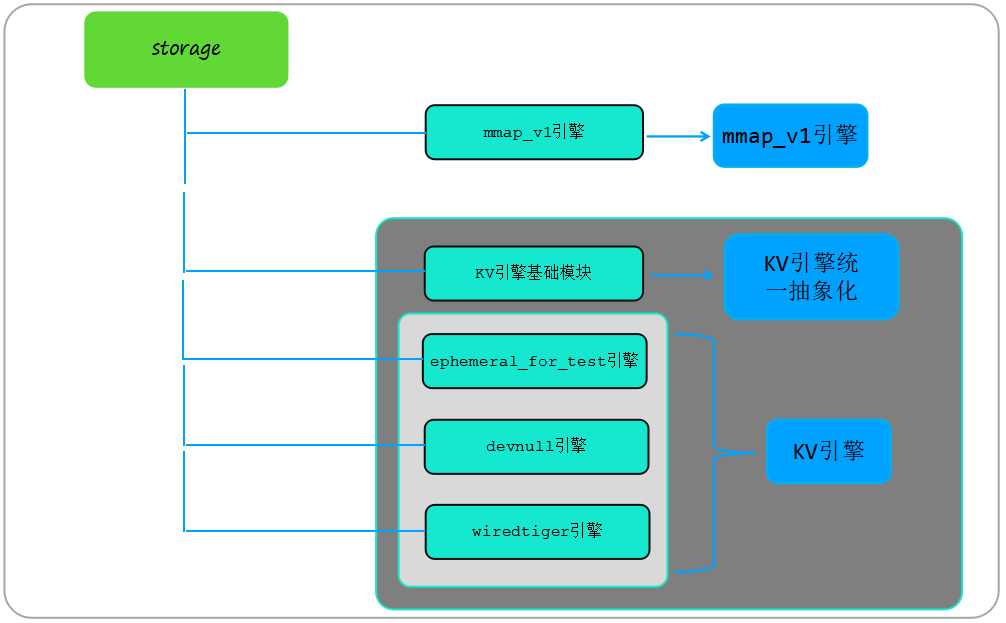
这两个类核心函数接口功能说明如下表所示：

|  |  |  |
| --- | --- | --- |
| 函数名 | 接口名 | 功能说明 |
| IndexCatalogImpl  IndexCatalogImpl | Init(...) | 从元数据文件"\_mdb\_catalog.wt"中获取索引信息,并构造索引对应的IndexCatalogEntryImpl添加到\_entries数组 |
| \_setupInMemoryStructures | 根据索引descriptor生成对应的IndexCatalogEntryImpl |
| checkUnfinished | 检查是否有未执行完成得索引 |
| \_getAccessMethodName | 确定该索引是什么类型(btree、text、2d索引等) |
| prepareSpecForCreate | 索引个数、索引冲突、索引名冲突、索引是否合规等检查 |
| createIndexOnEmptyCollection | 对IndexBuildBlock对应的索引进行初始化，创建索引对应的存储引擎底层索引文件 |
| IndexBuildBlock::fail() | 清除索引 |
| IndexBuildBlock::success() | 索引构建成功后的检查判断，并更新\_mdb\_catalog.wt元数据文件 |
| \_isSpecOk | spec索引内容有效性检查 |
| getDefaultIdIndexSpec | 获取id索引对应的spec |
| dropAllIndexes | 删除表下面的所有索引，包括cache中的entry、磁盘索引文件等 |
| dropIndex | 删除某个索引，包括cache中的entry、磁盘索引文件等 |
| \_dropIndex | dropIndex调用该接口，真正删除在这里，dropIndex相比\_dropIndex多一些检查判断 |
| \_deleteIndexFromDisk | 从磁盘删除索引文件，并清理元数据文件中的索引信息 |
| getAndClearUnfinishedIndexes | 实例异常挂掉情况下，当mongod重启后，需要清除这部分未执行成功的索引 |
| isMultikey | 判断指定索引是否是Multikey类型 |
| getMultikeyPaths | 获取指定索引对应的MultikeyPaths |
| haveAnyIndexes | 该表是否有至少有一个索引 |
| numIndexesTotal | 该表索引总数 |
| numIndexesReady | 当前已经执行完成的索引数 |
| haveIdIndex | 是否有id索引 |
| findIndexByName | 根据索引名查找索引 |
| getIndex | 获取desc对应IndexAccessMethod |
| getEntry | 获取desc对应IndexCatalogEntryImpl |
| \_indexFilteredRecords | 把bsonRecords对应的索引KV数据写入存储引擎 |
| \_indexRecords | 把bsonRecords对应的索引KV(如果需要过滤，则提前过滤)数据写入存储引擎 |
| \_unindexRecord | 删除loc对应的index索引数据 |
| \_fixIndexSpec | 从spec中获取索引信息,然后以bson格式返回 |
| IndexCatalogEntryImpl | init | 初始化entry对应的accessMethod |
| isMultikey | 是否multikey类型 |
| getMultikeyPaths | 获取该entry的MultikeyPaths信息 |
| setIsReady | 索引是否创建成功 |
| setMultikey | 根据MultikeyPaths获取Multikey |
| \_catalogIsReady | 获取IsReady标识信息 |
| \_catalogHead | 获取head信息 |
| \_catalogIsMultikey | 是否Multikey类型 |
| \_catalogGetPrefix | 获取prefix信息 |

# **storage公共层设计**

## 4.1 初识storage公共层及其存储引擎层目录结构

mongodb把catalog中间层以外的storage模块代码统一存放在src/mongo/db/storage目录，为了更好的理解模块化设计，首先看一下storage的子目录结构，大体可以归纳如下：



如上图所示，storage目录下面包含的子目录都和mongodb支持的存储引擎相关。mmap\_v1目录存放mmap\_v1存储引擎相关代码，devnull、wiredtiger、ephemeral\_for\_test都属于KV类存储引擎，因此把这几个存储引擎公共的kv操作相关接口抽象到了kv目录，所以storage目录多了一个kv目录。

从上面的目录结构大体可以看出，storage存储模块支持的存储引擎可以归类为两种类型：

* **KV引擎类：**官方默认包含devnull、wiredtiger、ephemeral\_for\_test三种。
* **MMAPV1Engine引擎类**

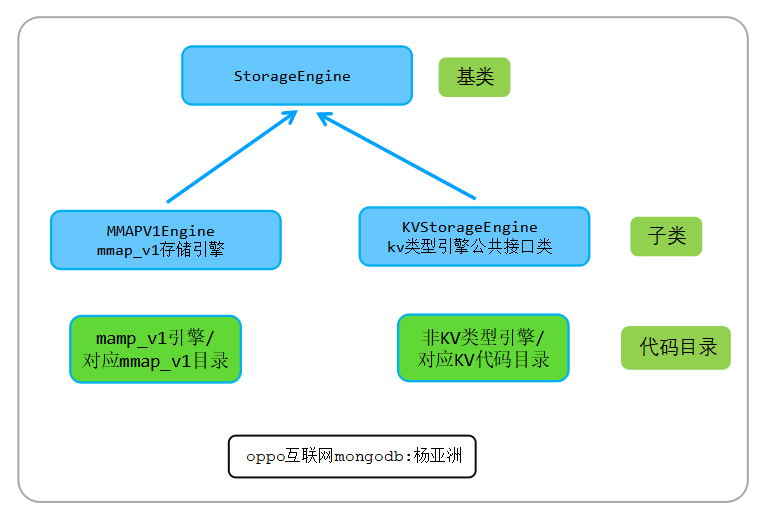
**说明：本系列文章只分析mongodb内核wiredtiger存储引擎实现。**

## 4.2 storage公共层核心代码实现

storage公共层核心代码主要包括src/mongo/db/storage这个目录中的.h文件和.cpp文件，该目录很多代码为test代码，忽略掉这一部分代码后，该公共层核心代码可以总结为如下：

* **storage\_engine.h(存储引擎基础类)**

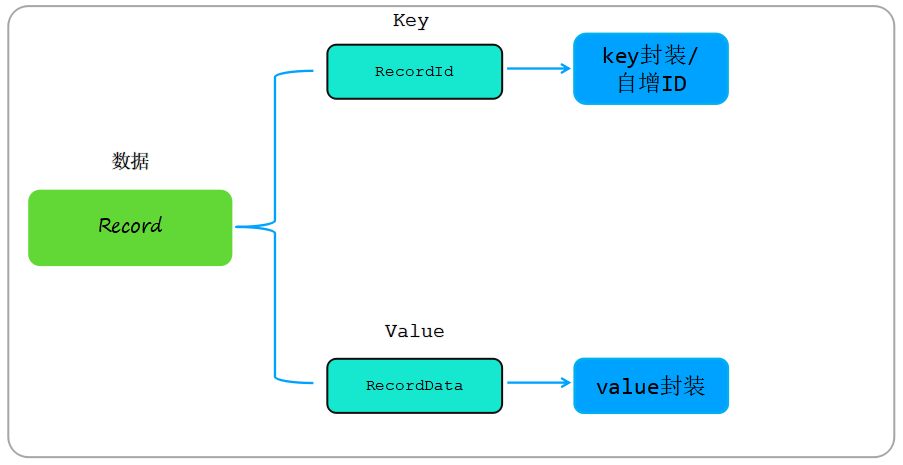
上面章节我们提到，mongodb在内核设计的时候，把存储引擎类型分为KV引擎和非KV引擎，class storage\_engine {}是不同类型引擎的基类。两种引擎各自对应一个子类来集成class storage\_engine {}，具体如下：



如上，mmap\_v1存储引擎相关核心代码实现主要由class MMAPV1Engine {}类实现，相关代码存放在mmap\_v1目录；其他的devnull、wiredtiger、ephemeral\_for\_test存储引擎都属于KV类型引擎，由class KVStorageEngine {}类实现，这些存储引擎的公共接口代码存放在kv目录。

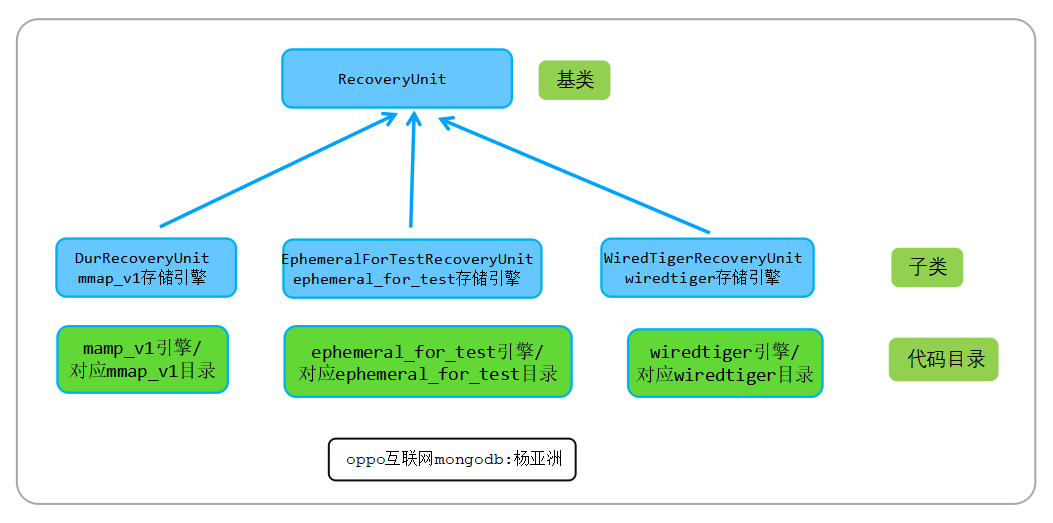
* **record\_store.h/record\_data.h (底层key:value数据封装)**

mongodb所有的数据在底层都是通过key:value的方式存储起来，不同存储引擎底层存储key:value方式不同，但是数据类型都是一样的，都是key:value类型。因此，storage中间层把客户端写入的数据最终通过key:value(简称KV)数据的封装过程抽象出来，如下图所示：



* **recovery\_unit.h**

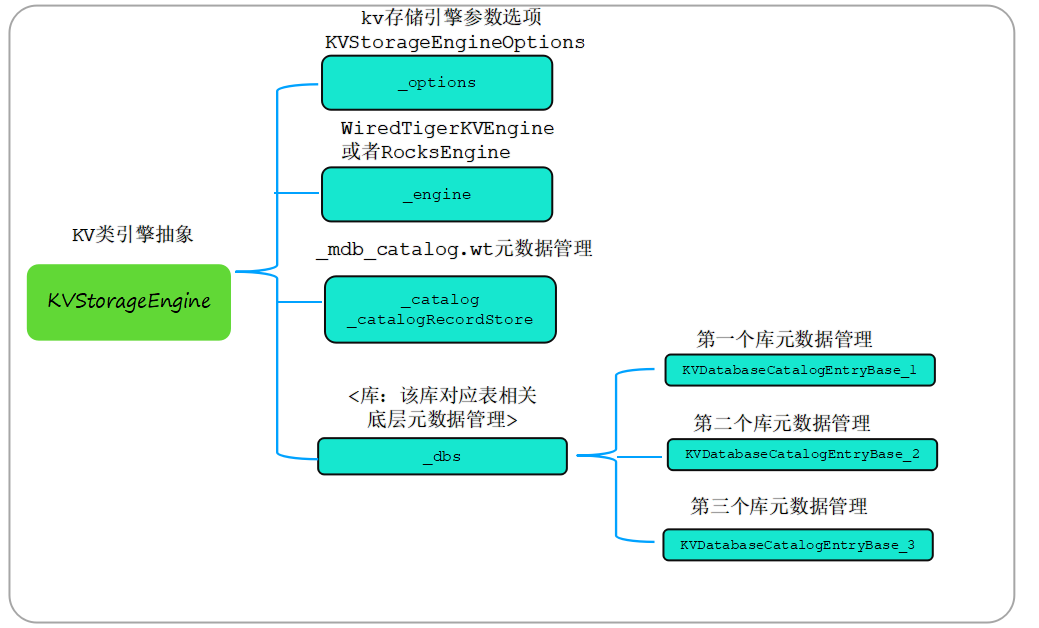
**recovery\_unit.h文件完成class recovery\_unit{}基础类定义，主要包括不同存储引擎层事务封装、快照管理等。各个不同存储引擎在各自对应代码目录中完成 class recovery\_unit{}类中的接口实现，如下图所示：**



# KV层设计

KV层主要完成KV类存储引擎(如wiredtiger、rocksdb等)元数据接口抽象，主要包括以下几个部分核心实现。

## 5.1 KV类引擎(KVStorageEngine)抽象管理



如上，KV引擎抽象类KVStorageEngine主要完成对应KVEngine管理，wiredtiger和rocksdb分别对应WiredTigerKVEngine和RocksEngine。此外，KV引擎抽象类还负责\_mdb\_catalog元数据文件初始化工作等。KVStorageEngine KV引擎抽象类核心接口功能说明如下表：

|  |  |  |
| --- | --- | --- |
| **类** | **接口** | **功能说明** |
| KVStorageEngine | KVStorageEngine(...) | 引擎初始化，获取元数据信息 |
| reconcileCatalogAndIdents | 元数据冲突判断检测 |
| cleanShutdown() | 实例shutdown引擎回收处理 |
| listDatabases | 获取所有db库名信息 |
| getDatabaseCatalogEntry | 获取DB对应元数据管理entry |
| dropDatabase | 删库相关的底层操作，删元数据，删ident目录数据等 |
| flushAllFiles | 把所有库表数据暴力写入磁盘 |
| getSnapshotManager | 获取快照管理模块 |
| setStableTimestamp | 后续分析配合WT引擎、repl一起分析，先跳过 |
| setOldestTimestamp | 后续分析配合WT引擎、repl一起分析，先跳过 |
| setInitialDataTimestamp | 后续分析配合WT引擎、repl一起分析，先跳过 |

## 5.2 kv\_catalog元数据底层操作管理

kv\_catalog元数据底层操作模块主要实现mongodb server层表、索引相关元数据管理，对应底层\_mdb\_catalog.wt文件。文件内容主要包括以下内容：

1. {
2. md: {
3. ns: "test.mycol",   //表名
4. options: {   //建表参数
5. uuid: UUID("75591c22-bd0f-4a56-ac95-ef90224cf3df"),
6. capped: true,
7. size: 6142976,
8. max: 10000,
9. autoIndexId: true
10. },
11. indexes: [{   //索引列表
12. spec: {
13. v: 2,
14. key: {
15. \_id: 1
16. },
17. name: "\_id\_",
18. ns: "test.mycol"
19. },
20. ready: false,
21. multikey: false,
22. multikeyPaths: {
23. \_id: BinData(0, 00)
24. },
25. head: 0,
26. prefix: -1
27. }],
28. prefix: -1
29. },
30. idxIdent: {   //表索引wt文件
31. \_id\_: "test/index/2--6948813758302814892"
32. },
33. ns: "test.mycol",   //表名
34. ident: "test/collection/1--6948813758302814892"   //表数据wt文件
35. }

如上，\_mdb\_catalog.wt文件内容主要包括以下几个部分：

* 表名
* 建表参数信息
* 表对应索引信息
* 存储索引数据的wt文件ident信息

也就是表对应的磁盘数据wt文件。

* 存储表数据的wt文件ident信息

也就是表对应的磁盘索引wt文件

**一个完整表数据包含以下两部分：**

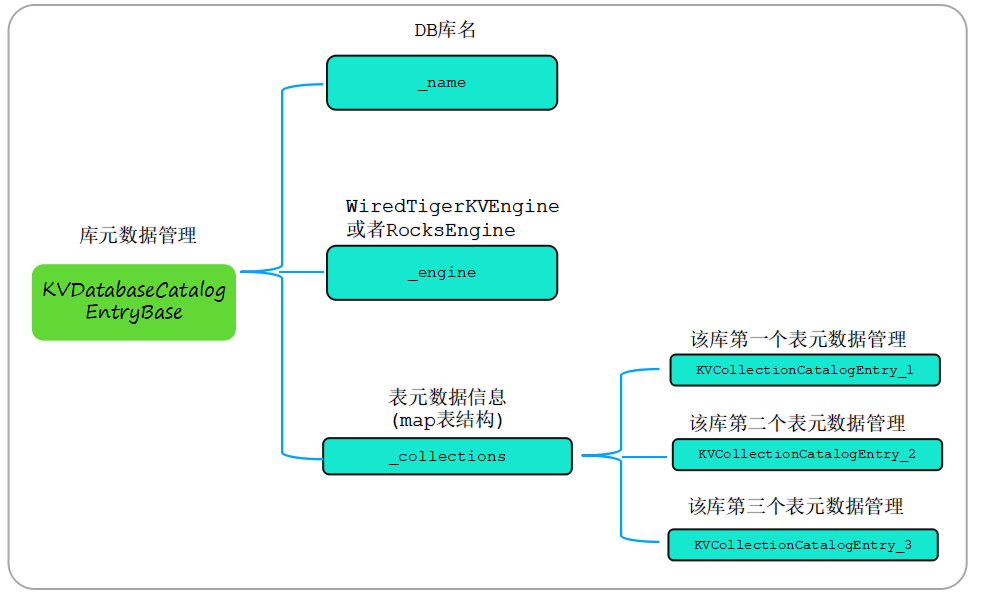
1. **数据表wt文件，一个表一个数据文件，也就是上面的ident对应目录文件，命令接口查看表对应数据ident方法：**
2. >
3. > db.mycol.stats().wiredTiger.uri
4. statistics:table:test/collection/1--6948813758302814892
5. >
6. **索引wt文件，一个索引对应一个索引文件，也就是上面的idxIdent对应文件目录，获取索引wt文件命令如下(说明：索引对应idxident没有通过stat直接提供，不过可以首先获取表索引磁盘占用信息，然后根据大小到对应目录查看文件大小即可确认该索引对应的索引uri)：**
7. //获取表索引信息，可以看到对应磁盘占用大小
8. shard\_4093D617:SECONDARY> db.opush\_message\_report.stats().indexSizes
9. { "\_id\_" : 45417041920, "\_id\_hashed" : 33888088064 }
10. shard\_4093D617:SECONDARY>
11. shard\_4093D617:SECONDARY>
12. //进入对应索引目录，根据数据大小即可对应起来，例如\_id\_索引对应index-13--799545853994112806.wt文件
13. -rw------- 1 root root  45417041920 Jul 16  2020 index-13--799545853994112806.wt
14. -rw------- 1 root root  33888088064 Dec 23 17:29 index-14--799545853994112806.wt
15. -rw------- 1 root root  27395637248 May 14 16:15 index-19--4995810767311230108.wt
16. -rw------- 1 root root      9977856 Dec 23 17:29 index-23--4995810767311230108.wt
17. [root@bjht3587 push\_stat]#

kv\_catalog.h和kv\_catalog.cpp主要就是完成对文件\_mdb\_catalog.wt中的元数据变更修改操作，这两个代码文件核心接口功能说明如下所示：

|  |  |  |
| --- | --- | --- |
| 类名 | 接口名 | 功能说明 |
| KVCatalog | \_newRand() | 生成随机数算法 |
| \_hasEntryCollidingWithRand(...) | ident文件是否以指定随机数结尾，必须把最大的随机数记录下来，以后有新表等从该随机数自动加1，避免重复 |
| \_newUniqueIdent(...) | ident生成，唯一性保证 |
| Init(...) | 从元数据文件\_mdb\_catalog.wt中获取元数据ident信息 |
| GetAllCollections(...) | 获取所有集合信息 |
| newCollection(...) | 建新表时候\_mdb\_catalog.wt中进行新表元数据初始化，一个表对应该文件一条记录 |
| getCollectionIdent(...) | 获取表对应的ident文件信息，也就是该表的数据wt文件 |
| getIndexIdent(...) | 获取索引对应的ident文件信息，也就是该索引对应索引wt文件 |
| \_findEntry(...) | 获取ns表对应元数据信息，再mdb\_catalog的第几行，内容是什么 |
| putMetaData(...) | 更新元数据信息 |
| renameCollection(...) | 表改名，则需要更新元数据信息，包括索引 |
| dropCollection (...) | 删表，则需要更新元数据信息，从元数据文件清除该表记录 |
| getAllIdents(...) | 获取所有表的ident数据文件信息 |
| isUserDataIdent(...) | 是否为数据wt文件或者索引wt文件 |

## 5.3 DatabaseCatalogEntry(指定库元数据管理)

一个DB库可以包含多个集合，库相关元数据管理实际上就是对该库下面多个表的元数据进行管理，该功能主要由DatabaseCatalogEntry基础类KVDatabaseCatalogEntryBase类相关接口实现，如下所示：

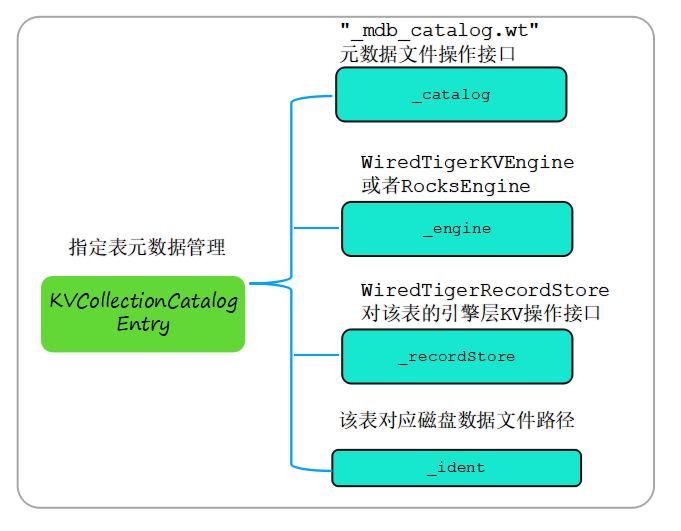


通过上面的KVDatabaseCatalogEntryBase类核心成员结构可以看出，该类基本上就是实现所属表的管理操作，该类核心接口功能如下：

|  |  |  |
| --- | --- | --- |
| **类名** | **核心接口** | **功能说明** |
| KVDatabaseCatalogEntryBase | exists() | 如果该库下面一个表都没有，则为false，否则为true |
| isEmpty() | 该库下面是否有表信息 |
| sizeOnDisk(...) | 库磁盘消耗=数据磁盘消耗+索引磁盘消耗 |
| getCollectionNamespaces(...) | 获取该库所有的表信息字符串 |
| getRecordStore(...) | 获取对该表进行底层数据KV操作的WiredTigerRecordStore |
| createCollection(...) | 指定库建表，需要对该表进行元数据等初始化管理以及相应Ident文件操作初始化 |
| initCollection(...) | \_mdb\_catalog.wt元数据文件初始化加载，如mongod实例重启过程 |
| renameCollection(...) | 表重命名相关元数据修改 |
| dropCollection(...) | 删表对应表元数据清除操作 |

## 5.4 CollectionCatalogEntry(指定表元数据管理)

指定表元数据管理主要通过class KVCollectionCatalogEntry{}相关接口实现，该类核心成员类型如下：

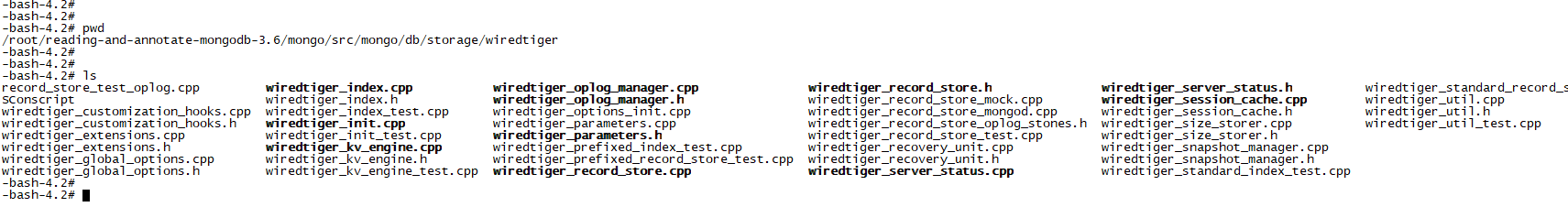


KVCollectionCatalogEntry类核心接口功能说明如下：

|  |  |  |
| --- | --- | --- |
| **类名** | **接口名** | **功能说明** |
| KVCollectionCatalogEntry | SetIndexIsMultikey(...) | 是否为Multikey Indexes索引类型，如果是则更新元数据 |
| removeIndex(...) | 删除表的指定索引对应底层元数据及索引文件清理 |
| prepareForIndexBuild(...) | 创建索引的准备工作 |
| indexBuildSuccess(...) | 索引执行完成后更新元数据 |
| updateTTLSetting(...) | TTL索引对应元数据更新 |
| addUUID(...) | UUID缓存中增加新的表uuid |
| removeUUID(...) | 删除缓存中对应uuid |
| isEqualToMetadataUUID(...) | 元数据中是否有该uuid |
| updateCappedSize(...) | 更新元数据固定表的size大小 |
| \_getMetaData(...) | 获取元数据信息 |

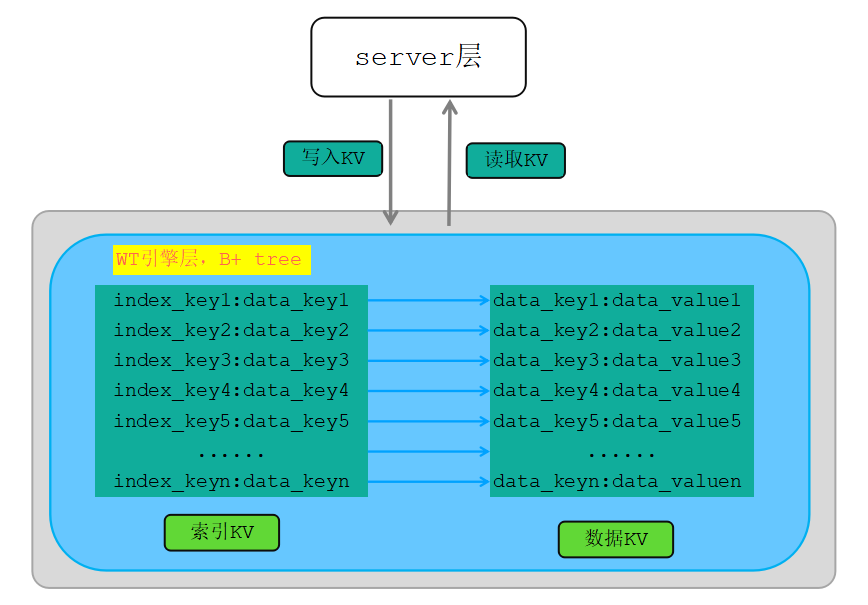
# Wiredtiger(简称WT)存储引擎层

wiredtiger存储引擎层主要完成mongodb server层和存储引擎层的衔接，用户数据和索引数据最终都转换为KV存入引擎层，引擎层核心代码目录结构如下：



## 6.1 WT引擎层和wiredtiger库衔接过程

WT引擎层代码最核心的功能就是提供server层写入KV数据及从引擎层查询KV数据，server层写入引擎层的的数据按照key进行B+ tree排序，也就是引擎层中的所有KV数据(包括用户KV数据和索引KV数据)都是排好序的。同样，server层查询指定数据，即可通过指定key在b+ tree中快速获取对应value。mongodb server层和引擎层关系架构如下图所示：



如上图，WT底层最终实现索引KV和数据KV的存储管理，索引KV数据的value指向数据KV的key，因此可以通过索引快速映射到数据KV的value，从而获取到数据。

* **说明1**

实际上，server层WT引擎层除了实现KV的读写外，还负责Oplog Journal管理、快照管理、WT session缓存管理等，由于这部分功能对WT内部细节有较高的依赖，因此先跳过，等后续《wiredtiger存储引擎设计与实现》系列分析完毕后，再从头回顾。

* **说明2(WT引擎底层为B+tree)**

WiredTiger maintains a table's data in memory using a data structure called a B-Tree ( B+ Tree to be specific), referring to the nodes of a B-Tree as pages. Internal pages carry only keys. The leaf pages store both keys and values.

详见：<http://source.wiredtiger.com/3.0.0/tune_page_size_and_comp.html>

## 6.2 WT引擎层读写接口使用方式

wiredtiger引擎库提供有各种example例子，其中最经典得一个例子就是KV写入、KV读取，代码如下：

[wiredtiger引擎KV操作demo](https://github.com/y123456yz/reading-and-annotate-wiredtiger-3.0.0/blob/master/wiredtiger/examples/c/ex_access.c)

1. access\_example(void)
2. {
3. ......
5. //获取一个库的全局connection
6. error\_check(wiredtiger\_open(home, NULL, "create", &conn));
8. //获取一个session
9. error\_check(conn->open\_session(conn, NULL, NULL, &session));
10. /\*! [access example connection] \*/
12. //创建一个table，对应mongo server层一个表
13. error\_check(session->create(
14. session, "table:access", "key\_format=S,value\_format=S"));
16. //获取一个table游标信息
17. error\_check(session->open\_cursor(
18. session, "table:access", NULL, NULL, &cursor));
20. //构造KV，然后写入
21. cursor->set\_key(cursor, "key1"); /\* Insert a record. \*/
22. cursor->set\_value(cursor, "value1");
23. error\_check(cursor->insert(cursor));
25. //重置游标
26. error\_check(cursor->reset(cursor));  /\* Restart the scan. \*/
27. //查询KV数据并打印
28. while ((ret = cursor->next(cursor)) == 0) {
29. error\_check(cursor->get\_key(cursor, &key));
30. error\_check(cursor->get\_value(cursor, &value));
31. printf("Got record: %s : %s\n", key, value);
32. }
33. //回收全局conn
34. error\_check(conn->close(conn, NULL));    /\* Close all handles. \*/

}

如上，一个table的数据KV写入和get获取过程可以总结为如下步骤：

1. 步骤一：获取该库的全局connection
2. 步骤二：获取一个session
3. 步骤三：通过session建表(wiredtiger称为table)
4. 步骤四：获取一个cursor游标，就可以通过该cursor进行KV数据操作了
5. 步骤五：根据cursor进行KV操作
6. 最后：回收全局connection

## 6.3 mongodb server引擎层操作接口实现

server引擎层操作接口就是按照上一节的例子进行wiredtiger底层接口调用，最常用的几个调用地方如下：

1. **步骤1：获取到数据目录的conn**
2. wiredtiger/wiredtiger\_kv\_engine.cpp
3. WiredTigerKVEngine::WiredTigerKVEngine->wiredtiger\_open(path.c\_str(), &\_eventHandler, config.c\_str(), &\_conn)
5. **步骤2：构造一个session**
6. conn->open\_session:
7. wiredtiger/wiredtiger\_session\_cache.cpp
8. invariantWTOK(conn->open\_session(conn, NULL, "isolation=snapshot", &\_session));
10. **步骤3：session和指定表或者索引关联,建表或者建索引**
11. s->create:
12. 第一个地方
13. wiredtiger/wiredtiger\_kv\_engine.cpp
14. WiredTigerKVEngine::createGroupedRecordStore-> s->create(s, uri.c\_str(), config.c\_str())
16. 第二个地方
17. wiredtiger/wiredtiger\_index.cpp
18. WiredTigerIndex::Create-> s->create(s, uri.c\_str(), config.c\_str());
20. **步骤4：  获取该session的一个cursor游标，这样就可以对指定表或者索引操作了**
21. session->open\_cursor：
22. 第一个地方
23. wiredtiger/wiredtiger\_index.cpp
24. WiredTigerIndex::BulkBuilder:openBulkCursor
25. wiredtiger/wiredtiger\_session\_cache.cpp
27. 第二个地方
28. wiredtiger/wiredtiger\_session\_cache.cpp:
29. //检查是否支持bulk功能
30. WiredTigerIndex::BulkBuilder :openBulkCursor(session, idx->uri().c\_str(), NULL, "bulk,checkpoint\_wait=false", &cursor);
31. WiredTigerIndex::BulkBuilder :session->open\_cursor(session, idx->uri().c\_str(), NULL, NULL, &cursor)