MongoDB internals

Table of Contents

[Patterns used 1](#__RefHeading__417_2029184438)

[Binaries which are built 1](#__RefHeading__419_2029184438)

[Major classes 1](#__RefHeading__421_2029184438)

[Server (mongod) execution flow 3](#__RefHeading__423_2029184438)

[Directory Overview 3](#__RefHeading__425_2029184438)

# Patterns used

There are classes which function as

1. Factory (used to create new instances of a class)
2. Builder ( used to mutate an instance)
3. Impl (multiple implementations of the same class)
4. Interface (abstract trait)
5. Listener (to listen to changes on an instance)

There are some singleton objects in every binary. Search for MONGO\_INITIALIZER to find them. See calls to mongo::runGlobalInitializersOrDie() at start of each executable.

# Binaries which are built

mongod (server) : from db/db.cpp

mongos (sharding proxy) : from s/server.cpp

mongo (client shell) : shell/dbshell.cpp

Functions which are shared by mongos and mongod are placed in library “db/mongodandmongos”

Code specific to mongod goes into “serverOnlyFiles” in the 'Sconscript” file.

# Major classes

**DbMessage/DbResponse** encapsulate the wire protocol between mongo clients and server.

**ServiceContext** : Represents the context in which binary is functioninig (.e.g. mongo sharding proxy or mongo server). “ServerContextMongod” is the singleton instance on the “mongod” server. The singleton is retrieved using getGlobalServiceContext(). It owns one or more clients.

**ClientBasic** : From it are derived Client(used in server) and ClientInfo (used in sharding proxy) classes. Client object in the server binary represents a client connection. It has atmost one OperationContext. See calls to ServiceContext::makeClient().

**OperationContext** is the equivalent of Transaction in Mongodb. Every OperationContext in a server with CurOp support has a stack of CurOp objects. The entry at the top of the stack is used to record timing and resource statistics for the executing operation or suboperation.

**RecordCursor** : Each getmore on a cursor is a separate OperationContext. Storage engines only need to implement the derived SeekableRecordCursor

**WriteUnitOfWork**

**WriteConcern**

**RecoveryUnit**

**CursorManager** : singleton

**Command** : represents a command executed by the proxy or mongo server. All derived commands reside in “commands” dir.

**SnapshotManager**

**Database** has n Collections

Each **Collection** has IndexCatalog, CollectionCatalogEntry, RecordStore

**IndexCatalog** has IndexAccessMethod

**IndexAccessMethod** points to SortedDataInterface

**StorageEngine** <- KVStorageEngine, MMapV1Engine

**KVStorageEngine** has KVEngine as member

**KVEngine** <- WiredTiger, RocksDB

**KVCatalog** <- RecordStore

# Server (mongod) execution flow

MessageServer::run()

MessageHandler

Request::process

execCommand

parseQuery : convert raw string -> CanonicalQuery -> QuerySolution tree -> PlanStage tree.

PlanExecutor : execute PlanStages against RecordStore and IndexAccessMethod classes.

# Directory Overview

**base**

**bson**

**crypto**

**client**

mongo shell code (clientdriver library)

**executor**

AsyncStreamFactoryInterface, AsyncStreamInterface,

AsyncTimerFactoryInterface, AsyncTimerInterface,

NetworkInterface

TaskExecutor

ThreadPoolInterface

ConnectionPool

**logger**

**platform**

**rpc**

**s** - sharding related code

**scripting**

**stdx** – related to standard C++ library classes

**util**

**util/concurrency**

Locks and threadpool

**util/net**

MessageHandler

MessageServer

View

Message

**db**

ServiceContext

CurOp

OperationContext

**db/auth**

**db/catalog** holds engine-independent code to represent/manipulate a column family or index. Calls code in db/storage.

Collection

IndexCatalog

**db/commands** (in turn calls db/query)

executes commands received from the client

calls getExecutor() and its variants.

**db/concurrency** holds lock manager

LockManager - singleton

**db/exec** contains code for various PlanStages.

PlanStage (and its 34 derived classes) : This represents a tree of data access and data transforms needed to satisfy a command.

WorkingSet – an operation is executed in many stages. All stages share the working set (i.e. data on which the stages operate).

**db/ftdc** – stands for full time diagnostic data capture. It takes a set of BSON documents containing metrics, and compresses them into a highly compressed buffers.

**db/fts** – code to implement text search

**db/geo** - code for geo indexing

**db/index**

IndexAccessMethod and derived classes, which call on engine-specific SortedDataInterface implementations.

IndexDescriptor

**db/matcher** - compares json with pattern

MatchExpression and derived classes for different expressions.

MatchableDocument

Matcher

**db/modules** – is symbolically linked to other storage engine

**db/ops**

**db/pipeline** holds code for query execution

Expression - derived classes

Pipeline - is used in mongos and mongod

PipelineD is used in mongod

**db/query** (calls db/exec and db/catalog) : holds code to parse a query string and create query plan

CanonicalQuery : any query is transformed into a canonical representation

PlanExecutor : iterates over tree of PlanStages.

QuerySolution : holds tree of QuerySolutionNodes.

getExecutorUpdate/getExecutorDelete/getExecutorFind all execute in two major steps

-> CanonicalQuery::canonicalize

-> getExecutor

CanonicalQuery::canonicalize : convert raw BSON string -> CanonicalQuery

getExecutor : convert CanonicalQuery -> PlanExecutor

-> prepareForExecution : convert CanonicalQuery->QuerySolution

-> StageBuilder::build : convert QuerySolution -> PlanStage tree

-> PlanExecutor::make : convert CanonicalQuery -> PlanExecutor

LiteParsedQuery

convert BSONObj -> LiteParsedQuery

QueryPlanner::plan

CanonicalQuery -> QuerySolution

PlanExecutor::executePlan

**db/repl**

DatabaseCloner

CollectionCloner

QuorumChecker

**db/s** (sharding related)

**db/sorter**

**db/stats**

**db/storage** (storage engine-specific code)

StorageEngine

RecordStore

SnapshotManager