MongoDB

<https://metanit.com/nosql/mongodb/1.1.php>

**Онлайн-руководство по MongoDB**

**Последнее обновление: 28.03.2018**

1. [Глава 1. Введение в MongoDB](https://metanit.com/nosql/mongodb/1.1.php)
   1. [Что такое MongoDB](https://metanit.com/nosql/mongodb/1.1.php)
   2. [Начало работы с MongoDB](https://metanit.com/nosql/mongodb/1.2.php)
   3. [Графический клиент Compass](https://metanit.com/nosql/mongodb/1.3.php)
2. [Глава 2. Работа с базой данных MongoDB](https://metanit.com/nosql/mongodb/2.1.php)
   1. [Устройство базы данных. Документы](https://metanit.com/nosql/mongodb/2.1.php)
   2. [Установка и администрирование базы данных](https://metanit.com/nosql/mongodb/2.2.php)
   3. [Добавление данных](https://metanit.com/nosql/mongodb/2.3.php)
   4. [Выборка из БД](https://metanit.com/nosql/mongodb/2.4.php)
   5. [Команды группировки](https://metanit.com/nosql/mongodb/2.7.php)
   6. [Операторы выборки](https://metanit.com/nosql/mongodb/2.8.php)
   7. [Обновление данных](https://metanit.com/nosql/mongodb/2.9.php)
   8. [Удаление данных](https://metanit.com/nosql/mongodb/2.10.php)
   9. [Установка ссылок в БД](https://metanit.com/nosql/mongodb/2.11.php)
   10. [Работа с индексами](https://metanit.com/nosql/mongodb/2.12.php)
   11. [Управление коллекцией](https://metanit.com/nosql/mongodb/2.13.php)
3. [Глава 3. PHP и MongoDB](https://metanit.com/nosql/mongodb/3.1.php)
   1. [Установка драйвера для PHP](https://metanit.com/nosql/mongodb/3.1.php)
   2. [Подключение к MongoDB](https://metanit.com/nosql/mongodb/3.2.php)
   3. [Добавление данных](https://metanit.com/nosql/mongodb/3.3.php)
   4. [Выборка документов](https://metanit.com/nosql/mongodb/3.4.php)
   5. [Управление выборкой](https://metanit.com/nosql/mongodb/3.5.php)
   6. [Условные операторы](https://metanit.com/nosql/mongodb/3.6.php)
   7. [Обновление документов](https://metanit.com/nosql/mongodb/3.7.php)
   8. [Удаление данных](https://metanit.com/nosql/mongodb/3.8.php)
   9. [Работа с DBRef](https://metanit.com/nosql/mongodb/3.9.php)
   10. [Работа с GridFS](https://metanit.com/nosql/mongodb/3.10.php)
4. [Глава 4. C# и MongoDB](https://metanit.com/nosql/mongodb/4.1.php)
   1. [Установка драйвера и подключение к базе данных](https://metanit.com/nosql/mongodb/4.1.php)
   2. [Взаимодействие с коллекцией. Документы](https://metanit.com/nosql/mongodb/4.2.php)
   3. [Работа с моделями данных](https://metanit.com/nosql/mongodb/4.3.php)
   4. [Настройка модели с помощью атрибутов](https://metanit.com/nosql/mongodb/4.4.php)
   5. [Сохранение документов в базу данных](https://metanit.com/nosql/mongodb/4.5.php)
   6. [**Выборка из базы данных**](https://metanit.com/nosql/mongodb/4.6.php)
   7. [Фильтрация данных](https://metanit.com/nosql/mongodb/4.7.php)
   8. [Интерфейс IFindFluent и его методы](https://metanit.com/nosql/mongodb/4.8.php)
   9. [Агрегация и группировка](https://metanit.com/nosql/mongodb/4.9.php)
   10. [Редактирование и удаление документов](https://metanit.com/nosql/mongodb/4.10.php)
   11. [Метод BulkWriteAsync](https://metanit.com/nosql/mongodb/4.16.php)
   12. [Работа с GridFS](https://metanit.com/nosql/mongodb/4.11.php)
   13. [MongoDB и ASP.NET MVC. Создание проекта](https://metanit.com/nosql/mongodb/4.12.php)
   14. [MongoDB и ASP.NET MVC. Чтение и добавление данных](https://metanit.com/nosql/mongodb/4.13.php)
   15. [MongoDB и ASP.NET MVC. Редактирование и удаление](https://metanit.com/nosql/mongodb/4.14.php)
   16. [MongoDB и ASP.NET MVC. Работа с файлами изображений](https://metanit.com/nosql/mongodb/4.15.php)

**Введение в MongoDB**

**Что такое MongoDB**

**Последнее обновление: 30.10.2015**

MongoDB реализует новый подход к построению баз данных, где нет таблиц, схем, запросов SQL, внешних ключей и многих других вещей, которые присущи объектно-реляционным базам данных.

Со времен динозавров было обычным делом хранить все данные в реляционных базах данных (MS SQL, MySQL, Oracle, PostgresSQL). При этом было не столь важно, а подходят ли реляционные базы данных для хранения данного типа данных или нет.

В отличие от реляционных баз данных MongoDB предлагает документо-ориентированную модель данных, благодаря чему MongoDB работает быстрее, обладает лучшей масштабируемостью, ее легче использовать.

Но, даже учитывая все недостатки традиционных баз данных и достоинства MongoDB, важно понимать, что задачи бывают разные и методы их решения бывают разные. В какой-то ситуации MongoDB действительно улучшит производительность вашего приложения, например, если надо хранить сложные по структуре данные. В другой же ситуации лучше будет использовать традиционные реляционные базы данных. Кроме того, можно использовать смешенный подход: хранить один тип данных в MongoDB, а другой тип данных - в традиционных БД.

Вся система MongoDB может представлять не только одну базу данных, находящуюся на одном физическом сервере. Функциональность MongoDB позволяет расположить несколько баз данных на нескольких физических серверах, и эти базы данных смогут легко обмениваться данными и сохранять целостность.

**Формат данных в MongoDB**

Одним из популярных стандартов обмена данными и их хранения является JSON (JavaScript Object Notation). JSON эффективно описывает сложные по структуре данные. Способ хранения данных в MongoDB в этом плане похож на JSON, хотя формально JSON не используется. Для хранения в MongoDB применяется формат, который называется **BSON** (БиСон) или сокращение от binary JSON.

BSON позволяет работать с данными быстрее: быстрее выполняется поиск и обработка. Хотя надо отметить, что BSON в отличие от хранения данных в формате JSON имеет небольшой недостаток: в целом данные в JSON-формате занимают меньше места, чем в формате BSON, с другой стороны, данный недостаток с лихвой окупается скоростью.

**Кроссплатформенность**

MongoDB написана на C++, поэтому ее легко портировать на самые разные платформы. MongoDB может быть развернута на платформах Windows, Linux, MacOS, Solaris. Можно также загрузить исходный код и самому скомпилировать MongoDB, но рекомендуется использовать библиотеки с офсайта.

**Документы вместо строк**

Если реляционные базы данных хранят строки, то MongoDB хранит документы. В отличие от строк документы могут хранить сложную по структуре информацию. Документ можно представить как хранилище ключей и значений.

**Ключ** представляет простую метку, с которым ассоциировано определенный кусок данных.

Однако при всех различиях есть одна особенность, которая сближает MongoDB и реляционные базы данных. В реляционных СУБД встречается такое понятие как **первичный ключ**. Это понятие описывает некий столбец, который имеет уникальные значения. В MongoDB для каждого документа имеется уникальный идентификатор, который называется \_id. И если явным образом не указать его значение, то MongoDB автоматически сгенерирует для него значение.

Каждому ключу сопоставляется определенное значение. Но здесь также надо учитывать одну особенность: если в реляционных базах есть четко очерченная структура, где есть поля, и если какое-то поле не имеет значение, ему (в зависимости от настроек конкретной бд) можно присвоить значение NULL. В MongoDB все иначе. Если какому-то ключу не сопоставлено значение, то этот ключ просто опускается в документе и не употребляется.

**Коллекции**

Если в традиционном мире SQL есть таблицы, то в мире MongoDB есть коллекции. И если в реляционных БД таблицы хранят однотипные жестко структурированные объекты, то в коллекции могут содержать самые разные объекты, имеющие различную структуру и различный набор свойств.

**Репликация**

Система хранения данных в MongoDB представляет набор реплик. В этом наборе есть основной узел, а также может быть набор вторичных узлов. Все вторичные узлы сохраняют целостность и автоматически обновляются вместе с обновлением главного узла. И если основной узел по каким-то причинам выходит из строя, то один из вторичных узлов становится главным.

**Простота в использовании**

Отсутствие жесткой схемы базы данных и в связи с этим потребности при малейшем изменении концепции хранения данных пересоздавать эту схему значительно облегчают работу с базами данных MongoDB и дальнейшим их масштабированием. Кроме того, экономится время разработчиков. Им больше не надо думать о пересоздании базы данных и тратить время на построение сложных запросов.

**GridFS**

Одной из проблем при работе с любыми системами баз данных является сохранение данных большого размера. Можно сохранять данные в файлах, используя различные языки программирования. Некоторые СУБД предлагают специальные типы данных для хранения бинарных данных в БД (например, BLOB в MySQL).

В отличие от реляционных СУБД MongoDB позволяет сохранять различные документы с различным набором данных, однако при этом размер документа ограничивается 16 мб. Но MongoDB предлагает решение - специальную технологию **GridFS**, которая позволяет хранить данные по размеру больше, чем 16 мб.

Система GridFS состоит из двух коллекций. В первой коллекции, которая называется files, хранятся имена файлов, а также их метаданные, например, размер. А в другой коллекции, которая называется chunks, в виде небольших сегментов хранятся данные файлов, обычно сегментами по 256 кб.

Для тестирования GridFS можно использовать специальную утилиту **mongofiles**, которая идет в пакете mongodb.

**Установка и начало работы с MongoDB на Windows**

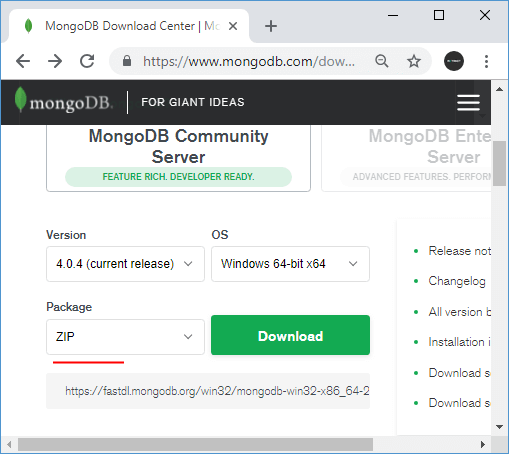
**Последнее обновление: 14.11.2018**

Для установки MongoDB загрузим один распространяемых пакетов с официального сайта <https://www.mongodb.com/download-center/community>.

Официальный сайт предоставляет пакеты дистрибутивов для различных платформ: Windows, Linux, MacOS, Solaris. И каждой платформы доступно несколько дистрибутивов. Причем есть два вида серверов - Community и Enterprise. В данном случае надо установить версию Community. Хотя Enterprise-версия обладает несколько большими возможностями, но она доступна только в триальном режиме или по подписке.

На момент написания данного материала последней версией платформы была версия **4.0.4**. Использование конкретной версии может несколько отличаться от применения иных версий платформы MongoDB.

Для загрузки нобходиомго функционала выберем нужную операционную систему и подходящий тип пакета:



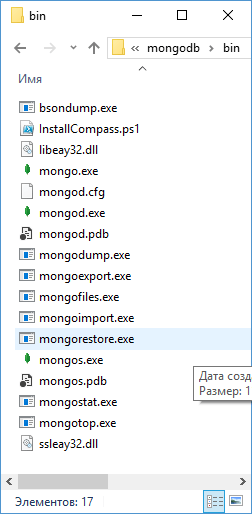
Для ОС Windows можно выбрать тип пакета "ZIP", то есть загрузить сервер в виде архива.

Если до установки уже была установлена более рання версия MongoDB, то ее необходимо удалить и также необходимо удалить все равне созданные базы данных.

После загрузки архивного пакета распакуем его в папку C:\mongodb.

**Содержимое пакета MongoDB**

Если после установки мы откроем папку *C:\mongodb\bin*, то сможем найти там кучу приложений, которые выполняют определенную роль. Вкратце рассмотрим их.



* **bsondump**: считывает содержимое BSON-файлов и преобразует их в читабельный формат, например, в JSON
* **mongo**: представляет консольный интерфейс для взаимодействия с базами данных, своего рода консольный клиент
* **mongod**: сервер баз данных MongoDB. Он обрабатывает запросы, управляет форматом данных и выполняет различные операции в фоновом режиме по управлению базами данных
* **mongodump**: утилита создания бэкапа баз данных
* **mongoexport**: утилита для экспорта данных в форматы JSON, TSV или CSV
* **mongofiles**: утилита, позволяющая управлять файлами в системе GridFS
* **mongoimport**: утилита, импорирующая данных в форматах JSON, TSV или CSV в базу данных MongoDB
* **mongorestore**: позволяет записывать данные из дампа, созданного mongodump, в новую или существующую базу данных
* **mongos**: служба маршрутизации MongoDB, которая помогает обрабатывать запросы и определять местоположение данных в кластере MongoDB
* **mongorestat**: представляет счетчики операций с бд
* **mongotop**: предоставляет способ подсчета времени, затраченного на операции чтения-записи в бд

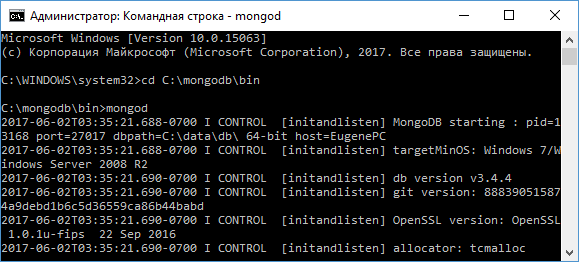
**Создание каталога для БД и запуск MongoDB**

После установки надо создать на жестком диске каталог, в котором будут находиться базы данных MongoDB.

В ОС Windows по умолчанию MongoDB хранит базы данных по пути *C:\data\db*, поэтому, если вы используете Windows, вам надо создать соответствующий каталог. В ОС Linux и MacOS каталогом по умолчанию будет */data/db*.

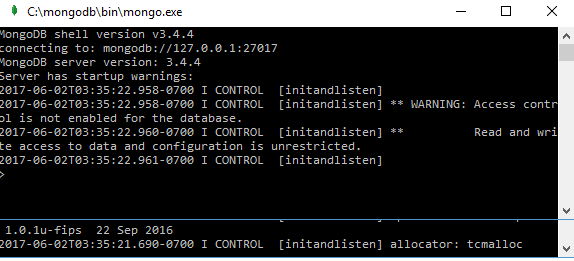
Если же возникла необходимость использовать какой-то другой путь к файлам, то его можно передать при запуске MongoDB во флаге --dbpath.

Итак, после создания каталога для хранения БД можно запустить сервер MongoDB. Сервер представляет приложение **mongod**, которое находится в папке bin. Для этого запустим командную строку (в Windows) или консоль в Linux и там введем соответствующие команды. Для ОС Windows это будет выглядеть так:



Командная строка отобразит нам ряд служебной информации, например, что сервер запускается на localhost на порту 27017.

И после удачного запуска сервера мы сможем производить операции с бд через оболочку **mongo**. Эта оболочка представляет файл **mongo.exe**, который располагается в выше рассмотренной папке установки. Запустим этот файл:



Это консольная оболочка для взаимодействия с сервером, через которую можно управлять данными. Второй строкой эта оболочка говорит о подключении к серверу mongod.

Теперь поизведем какие-либо простейшие действия. Введем в mongo последовательно следующие команды и после каждой команды нажмем на Enter:

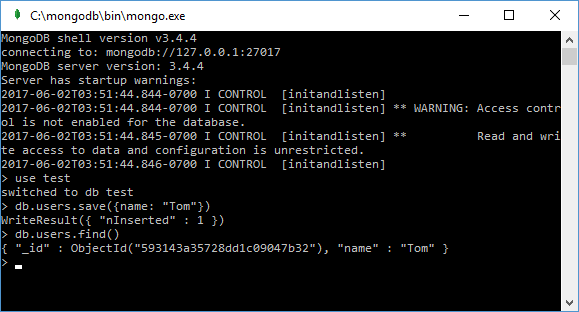
|  |  |
| --- | --- |
| 1  2  3 | use test  db.users.save( { name: "Tom" } )  db.users.find() |

Первая команда **use test** устанавливает в качестве используемой базу данных test. Даже если такой бд нет, то она создается автоматически. И далее db будет представлять текущую базу данных - то есть базу данных test. После db идет users - это коллекция, в которую затем мы добавляем новый объект. Если в SQL нам надо создавать таблицы заранее, то коллекции MongoDB создает самостоятельно при их отсутствии.

С помощью метода **db.users.save()** в коллекцию users базы данных test добавляется объект { name: "Tom" }. Описание добавляемого объекта определяется в формате, с которым вы возможно знакомы, если имели дело с форматом JSON. То есть в данном случае у объекта определен один ключ "name", которому сопоставляется значение "Tom". То есть мы добавляем пользователя с именем Tom.

Если объект был успешно добавлен, то консоль выведет результа в виде выражения WriteResult({ "nInserted" : 1 }).

А третья команда **db.users.find()** выводит на экран все объекты из бд test.



Из вывода вы можете увидеть, что к начальным значениям объекта было добавлено какое-то непонятно поле **ObjectId**. Как вы помните, MongoDB в качестве уникальных идентификаторов документа использует поле \_id. И в данном случае ObjectId как раз и представляет значение для идентификатора \_id.

**Установка драйверов MongoDB**

Конечно, мы можем работать и через консоль mongo, добавляя и отображая объекты в бд. Но нам также было бы неплохо, если бы mongoDB взаимодействовала бы с нашими приложениями, написанными на PHP, C++, C# и других языках программирования. И для этой цели нам потребуются специальные драйверы.

На офсайте на странице <https://docs.mongodb.com/ecosystem/drivers/> можно найти драйвера для таких языков программирования, как PHP, C++, C#, Java, Python, Perl, Ruby, Scala и др.

Далее уже, рассматривая взаимодействие отдельных языков программирования с MongoDB, мы подробнее рассмотрим установку и драйвера и всю необходимую конфигурацию для определенных языков программирования.

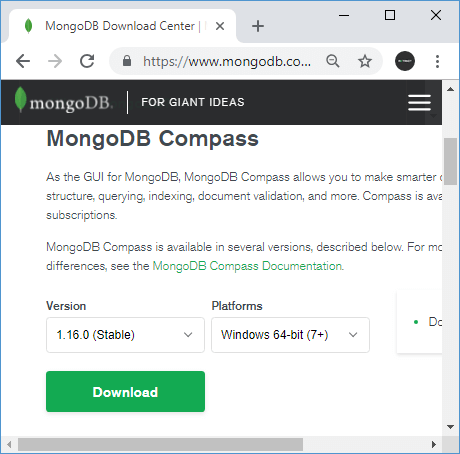
**Графический клиент Compass**

**Последнее обновление: 14.11.2018**

Для работы с MongoDB можно использовать официальный графический клиент **Compass**.

Если вдруг на этапе установки MongoDB, Mongo Compass не был установлен, то для его загрузки перейдем по адресу <https://www.mongodb.com/download-center/compass>.

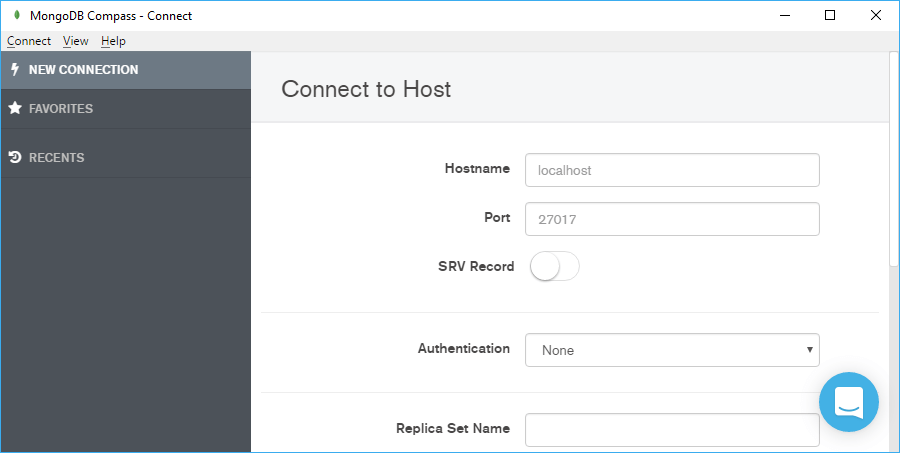
Укажем опции для загрузки - версию Compass и целевую операционную систему:

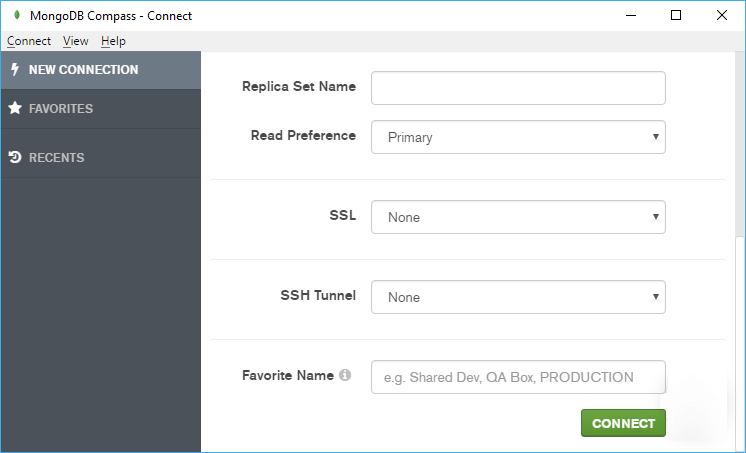


После загрузки программы установки запустим ее. Нам отобразится следующее окно, которое инфомирует о том, что произодится установка:



Установка производится очень быстро, и после ее окончания на рабочем столе и в меню Пуск (если наша ОС - Windows), отобразится соответствующая иконка. Кроме того, автоматически запустится сам Compass:



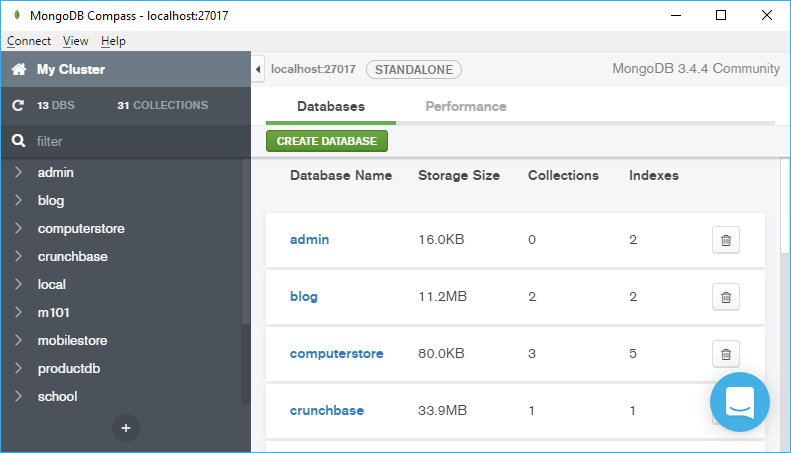


При запуске программы нам надо ввести ряд данных, чтобы подключиться к серверу.

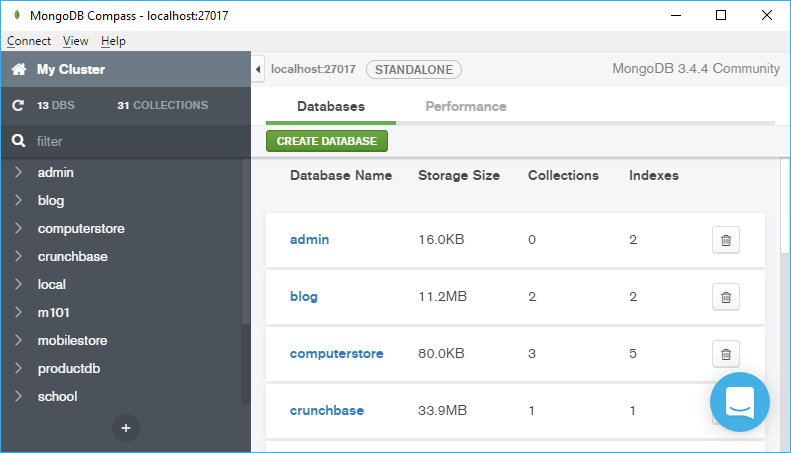
* **Hostname**: хост сервера
* **Port**: порт, по которому запущен сервер
* **SRV Record**: спецификация, которая определяет расположение сервера (например, адрес и порт).
* **Authentication**: тип применяемой аутентификации. Здеь есть следующие опции:
  + **None**
  + **Username / Password**
  + **X.509**
  + **Kerberos**
  + **LDAP**
* **Replica Set Name**: название реплики MongoDB, к которой происходит подключение
* **Read Preference**: определяет, как Compass управляет операциями чтения. Может принимать следующие опции: Primary, Primary Preferred, Secondary, Secondary Preferred и Nearest.
* **SSL**: указывает, будет ли использоваться защищенное подключение
* **SSH tunnel**: следует ли подключаться к кластеру MongoDB через туннель SSH
* **Favorite Name**: устанавливает имя подключения

Например, подключимся к локальному серверу. Для этого вначале запустим сам сервер MongoDB. В Compass оставим все настройки подключения по умолчанию (они как раз нацелены на локальный сервер, который запускается по адресу localhost:27017) и для подключения нажмем на кнопку Connect.

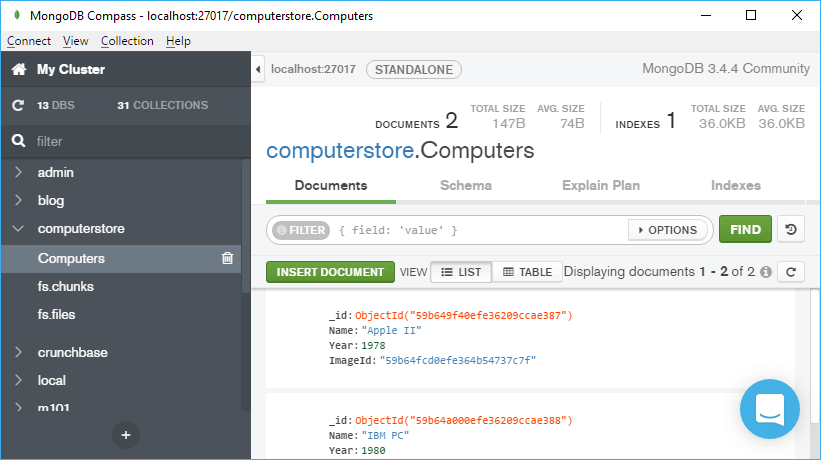
После этого нам откроется список баз данных, которые есть на сервере:



Мы можем выбрать определенную базу данных и получить по нему информацию, в частности, увидеть набор коллекций в бд, сколько они занимают данных.



Нажав на определенную коллекцию, можно увидеть графически все данные, которые есть в коллекции:



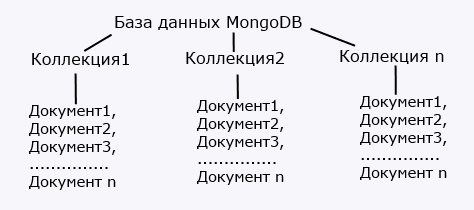
Используя графический интерфейс программы Compass, мы можем управлять этими данными, добавлять, изменять, удалять их.

**Работа с базой данных**

**Устройство базы данных. Документы**

**Последнее обновление: 25.03.2018**

Всю модель устройства базы данных в MongoDB можно представить следующим образом:



Если в реляционных бд содержимое составляют таблицы, то в mongodb база данных состоит из **коллекций**.

Каждая коллекция имеет свое уникальное имя - произвольный идентификатор, состоящий из не более чем 128 различных алфавитно-цифровых символов и знака подчеркивания.

В отличие от реляционных баз данных MongoDB не использует табличное устройство с четко заданным количеством столбцов и типов данных. MongoDB является документо-ориентированной системой, в которой центральным понятием является **документ**.

Документ можно представить как объект, хранящий некоторую информацию. В некотором смысле он подобен строкам в реляционных субд, где строки хранят информацию об отдельном элементе. Например, типичный документ:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | {      "name": "Bill",      "surname": "Gates",      "age": "48",      "company": {          "name" : "microsoft",          "year" : "1974",          "price" : "300000"          }  } |

Документ представляет набор пар ключ-значение. Например, в выражении "name": "Bill" name представляет ключ, а Bill - значение.

Ключи представляют строки. Значения же могут различаться по типу данных. В данном случае у нас почти все значения также представляют строковый тип, и лишь один ключ (company) ссылается на отдельный объект. Всего имеется следующие типы значений:

* **String**: строковый тип данных, как в приведенном выше примере (для строк используется кодировка UTF-8)
* **Array (массив)**: тип данных для хранения массивов элементов
* **Binary data (двоичные данные)**: тип для хранения данных в бинарном формате
* **Boolean**: булевый тип данных, хранящий логические значения TRUE или FALSE, например, {"married": FALSE}
* **Date**: хранит дату в формате времени Unix
* **Double**: числовой тип данных для хранения чисел с плавающей точкой
* **Integer**: используется для хранения целочисленных значений, например, {"age": 29}
* **JavaScript**: тип данных для хранения кода javascript
* **Min key/Max key**: используются для сравнения значений с наименьшим/наибольшим элементов BSON
* **Null**: тип данных для хранения значения Null
* **Object**: строковый тип данных, как в приведенном выше примере
* **ObjectID**: тип данных для хранения id документа
* **Regular expression**: применяется для хранения регулярных выражений
* **Symbol**: тип данных, идентичный строковому. Используется преимущественно для тех языков, в которых есть специальные символы.
* **Timestamp**: применяется для хранения времени

В отличие от строк документы могут содержать разнородную информацию. Так, рядом с документом, описанным выше, в одной коллекции может находиться другой объект, например:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | {      "name": "Tom",      "birthday": "1985.06.28",      "place" : "Berlin",      "languages" :[          "english",          "german",          "spanish"      ]  } |

Казалось бы разные объекты за исключением отдельных свойств, но все они могут находиться в одной коллекции.

Еще пара важных замечаний: в MongoDB запросы обладают регистрозависимостью и строгой типизацией. То есть следующие два документа не будут идентичны:

|  |  |
| --- | --- |
| 1  2 | {"age" : "28"}  {"age" : 28} |

Если в первом случае для ключа age определена в качестве значения строка, то во втором случае значением является число.

**Идентификатор документа**

Для каждого документа в MongoDB определен уникальный идентификатор, который называется \_id. При добавлении документа в коллекцию данный идентификатор создается автоматически. Однако разработчик может сам явным образом задать идентификатор, а не полагаться на автоматически генерируемые, указав соответствующий ключ и его значение в документе.

Данное поле должно иметь уникальное значение в рамках коллекции. И если мы попробуем добавить в коллекцию два документа с одинаковым идентификатором, то добавится только один из них, а при добавлении второго мы получим ошибку.

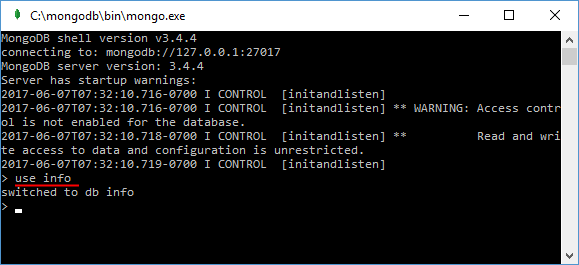
Если идентификатор не задан явно, то MongoDB создает специальное бинарное значение размером 12 байт. Это значение состоит из нескольких сегментов: значение типа timestamp размером 4 байта, идентификатор машины из 3 байт, идентификатор процесса из 2 байт и счетчик из 3 байт. Таким образом, первые 9 байт гарантируют уникальность среди других машин, на которых могут быть реплики базы данных. А следующие 3 байта гарантируют уникальность в течение одной секунды для одного процесса. Такая модель построения идентификатора гарантирует с высокой долей вероятности, что он будет иметь уникальное значение, ведь она позволяет создавать до 16 777 216 уникальных объектов ObjectId в секунду для одного процесса.

**Установка и администрирование базы данных**

**Последнее обновление: 25.03.2018**

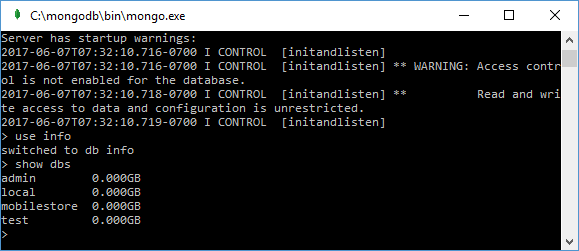
Начиная работать с MongoDB, первым делом надо установить нужную нам базу данных в качестве текущей, чтобы затем ее использовать. Для этого надо использовать команду use, после которой идет название базы данных. При этом не важно, существует ли такая бд или нет. Если ее нет, то MongoDB автоматически создаст ее при добавлении в нее данных. Например, запустим mongo.exe и введем там следующую команду:

|  |  |
| --- | --- |
| 1 | > use info |



Теперь в качестве текущей будет установлена БД info.

Если вы вдруг не уверены, а существует ли уже база данных с таким названием, то с помощью команды show dbs можно вывести названия всех имеющихся бд на консоль:



Для базы данных можно задать любое имя, однако есть некоторые ограничения. Например, в имени не должно быть символов /, \, ., ", \*, <, >, :, |, ?, $. Кроме того, имена баз данных ограничены 64 байтами.

Также есть зарезервированные имена, которые нельзя использовать: local, admin, config.

Причем как вы видите, бд info в данном списке нет, так как я в нее еще не добавил данные.

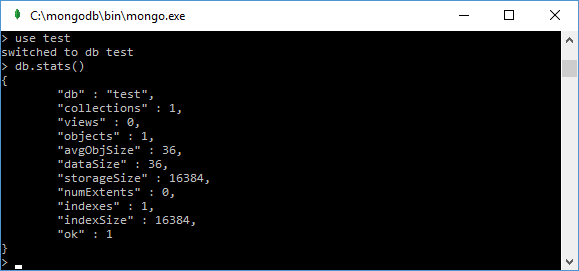
Если мы хотим узнать, какая бд используется в текущей момент, то мы можем воспользоваться командой db:

|  |  |
| --- | --- |
| 1  2 | > db  info |

Кроме баз данных мы можем просмотреть список всех коллекций в текущей бд с помощью команды show collections. Но так как бд info, которая указана текущей, еще не существует, то и коллекций она пока не содержит.

**Получение статистики**

Используя команду db.stats(), можно получить статистику по текущей базе данных. Например, у нас в качестве текущей установлена база данных test:



Похожим образом мы можем узнать всю статистику по отдельной коллекции. Например, узнаем статистику по коллекции users: db.users.stats()

**Добавление данных**

**Последнее обновление: 25.03.2018**

Установив бд, теперь мы можем добавить в нее данные. Все данные хранятся в бд в формате BSON, который близок к JSON, поэтому нам надо также вводить данные в этом формате. И хотя у нас, возможно, на данный момент нет ни одной коллекции, но при добавлении в нее данных она автоматически создается.

Как ранее говорилось, имя коллекции - произвольный идентификатор, состоящий из не более чем 128 различных алфавитно-цифровых символов и знака подчеркивания. В то же время имя коллекции не должно начинаться с префикса system., так как он зарезервирован для внутренних коллекций (например, коллекция system.users содержит всех пользователей базы данных). И также имя не должно содержать знака доллара - $.

Для добавления в коллекцию могут использоваться три ее метода:

* **insertOne()**: добавляет один документ
* **insertMany()**: добавляет несколько документов
* **insert()**: может добавлять как один, так и несколько документов

Итак, добавим один документ:

|  |  |
| --- | --- |
| 1 | > db.users.insertOne({"name": "Tom", "age": 28, languages: ["english",  "spanish"]}) |

Документ представляет набор пар ключ-значение. В данном случае добавляемый документ имеет три ключа: name, age, languages, и каждому из них сопоставляет определенное значение. Например, ключу languages в качесте значения сопоставляется массив.

Некоторые ограничения при использовании имен ключей:

* Символ $ не может быть первым символом в имени ключа
* Имя ключа не может содержать символ точки .

При добавлении данных, если мы явным образом не предоставили значение для поля "\_id" (то есть уникального идентификатора документа), то оно генерируется автоматически. Но в принципе мы можем сами установить этот идентификатор при добавлении данных:

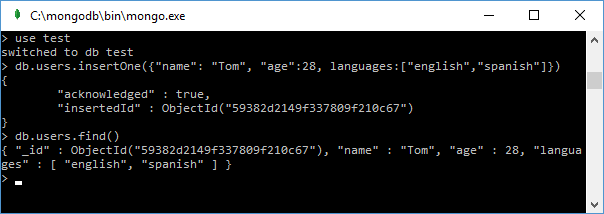
|  |  |
| --- | --- |
| 1 | > db.users.insertOne({"\_id": 123457, "name": "Tom", "age": 28,  languages: ["english", "spanish"]}) |
|  |  |

Стоит отметить, что названия ключей могут использоваться в кавычках, а могут и без кавычек.

В случае удачного добавления на консоль будет выведен идентификатор добавленного документа.

И чтобы убедиться, что документ в бд, мы его выводим функцией find.

|  |  |
| --- | --- |
| 1 | > db.users.find() |



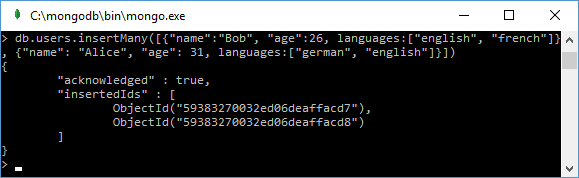
Чтобы вывести в более читабельном виде добавим метод pretty():

|  |  |
| --- | --- |
| 1 | > db.users.find().pretty() |

Если надо добавить ряд документов, то мы можем воспользоваться методом **insertMany()**:

|  |  |
| --- | --- |
| 1  2 | > db.users.insertMany([{"name": "Bob", "age": 26, languages: ["english",  "frensh"]},  {"name": "Alice", "age": 31, languages:["german", "english"]}]) |

После добавления консоль выводит идентификаторы добавленных документов:



И третий метод - **insert()** демонстрирует более универсальный способ добавления документов. При его вызове в него также передается добавляемый документ:

|  |  |
| --- | --- |
| 1 | > db.users.insert({"name": "Tom", "age": 28, languages: ["english",  "spanish"]}) |

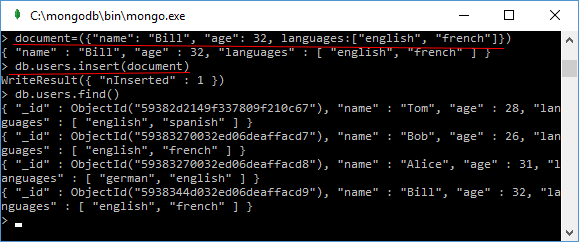
После его вызова на консоль выводится количество добавленных записей:

|  |  |
| --- | --- |
| 1 | WriteResult({ "nInserted" : 1 }) |

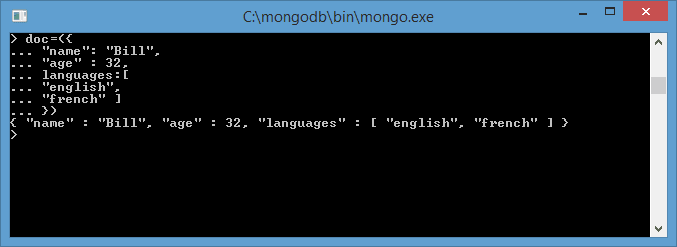
Есть еще один способ добавления в бд документа, который включает два этапа: определение документа (document = ( { ... } )) и собственно его добавление:

|  |  |
| --- | --- |
| 1  2 | > document=({"name": "Bill", "age": 32, languages: ["english", "french"]})  > db.users.insert(document) |

При желании опять же можно с помощью функции db.users.find() убедиться, что документ попал в бд.



Возможно, не всем будет удобно вводить в одну строчку все пары ключей и свойств. Но интеллектуальный интерпретатор MongoDB на основе javascript позволяет также вводить и многострочные команды. Если выражение не закончено (с точки зрения языка JavaScript), и вы нажимаете Enter, то ввод следующей части выражения автоматически переносится на следующую строку:



**Загрузка данных из файла**

Данные для базы данных mongodb можно определять в обычном текстовом файле, что довольно удобно, поскольку мы можем переносить или пересылать этот файл независимо от базы данных mongodb. Например, определим где-нибудь на жестком диске файл **users.js** со следующим содержимым:

|  |  |
| --- | --- |
| 1  2  3  4  5 | db.users.insertMany([  {"name": "Alice", "age": 31, languages: ["english", "french"]},  {"name": "Lene", "age": 29, languages: ["english", "spanish"]},  {"name": "Kate", "age": 30, languages: ["german", "russian"]}  ]) |

То есть здесь с помощью метода insertMany добавляются три документа в коллекцию users.

Для загузки файла в текущую базу данных применяется функция **load()**, в которую в качестве параметра передается путь к файлу:

|  |  |
| --- | --- |
| 1 | > load("D:/users.js") |

В данном случае предполагается, что файл располагается по пути "D:/users.js".

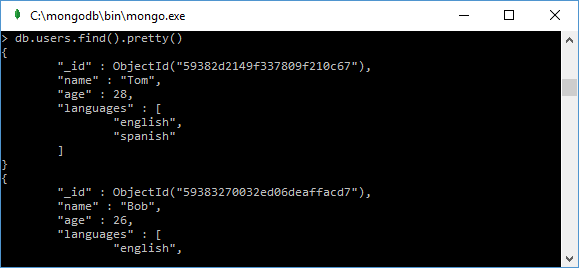
**Выборка из БД**

**Последнее обновление: 25.03.2018**

Наиболее простой способом получения содержимого БД представляет использование функции find. Действие этой функции во многом аналогично обычному запросу SELECT \* FROM Table, который извлекает все строки. Например, чтобы извлечь все документы из коллекции users, созданной в прошлой теме, мы можем использовать команду db.users.find().

Для вывода документов в более удобном наглядном представлении мы можем добавить вызов метода **pretty()**:

|  |  |
| --- | --- |
| 1 | > db.users.find().pretty() |



Однако что, если нам надо получить не все документы, а только те, которые удовлетворяют определенному требованию. Например, мы ранее в базу добавили следующие документы:

|  |  |
| --- | --- |
| 1  2  3 | > db.users.insertOne({"name": "Tom", "age": 28, languages: ["english",  "spanish"]})  > db.users.insertOne({"name": "Bill", "age": 32, languages: ["english",  "french"]})  > db.users.insertOne({"name": "Tom", "age": 32, languages: ["english",  "german"]}) |

Выведем все документы, в которых name=Tom:

|  |  |
| --- | --- |
| 1 | > db.users.find({name: "Tom"}) |

Такой запрос выведет нам два документа, в которых name=Tom.

Теперь более сложный запрос: нам надо вывести те объекты, у которых name=Tom и одновременно age=32. То есть на языке SQL это могло бы выглядеть так: SELECT \* FROM Table WHERE Name='Tom' AND Age=32. Данному критерию у нас соответствует последний добавленный объект. Тогда мы можем написать следующий запрос:

|  |  |
| --- | --- |
| 1 | > db.users.find({name: "Tom", age: 32}) |

Также несложно отыскать по элементу в массиве. Например, следующий запрос выводит все документы, у которых в массиве languages есть english:

|  |  |
| --- | --- |
| 1 | > db.users.find({languages: "english"}) |

Усложним запрос и получим те документы, у которых в массиве languages одновременно два языка: "english" и "german":

|  |  |
| --- | --- |
| 1 | > db.users.find({languages: ["english", "german"]}) |

Теперь выведем все документы, в которых "english" в массиве languages находится на первом месте:

|  |  |
| --- | --- |
| 1 | > db.users.find({"languages.0": "english"}) |

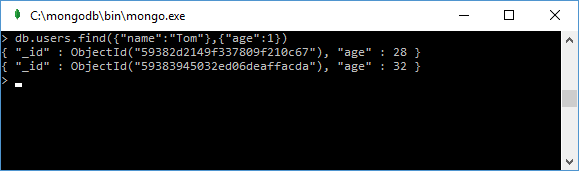
Соответственно если нам надо вывести документы, где english на втором месте (например, ["german", "english"]), то вместо нуля ставим единицу: languages.1.

**Проекция**

Документ может иметь множество полей, но не все эти поля нам могут быть нужны и важны при запросе. И в этом случае мы можем включить в выборку только нужные поля, использовав проекцию. Например, выведем только порцию информации, например, значения полей "age" у все документов, в которых name=Tom:

|  |  |
| --- | --- |
| 1 | > db.users.find({name: "Tom"}, {age: 1}) |

Использование единицы в качестве параметра {age: 1} указывает, что запрос должен вернуть только содержание свойства age.



И обратная ситуация: мы хотим найти все поля документа кроме свойства age. В этом случае в качестве параметра указываем 0:

|  |  |
| --- | --- |
| 1 | > db.persons.find({name: "Tom"}, {age: 0}) |

При этом надо учитывать, что даже если мы отметим, что мы хотим получить только поле name, поле \_id также будет включено в результирующую выборку. Поэтому, если мы не хотим видеть данное поле в выборке, то надо явным образом указать: {"\_id":0}

Альтернативно вместо 1 и 0 можно использовать true и false:

|  |  |
| --- | --- |
| 1 | > db.users.find({name: "Tom"}, {age: true, \_id: false}) |

Если мы не хотим при этом конкретизировать выборку, а хотим вывести все документы, то можно оставить первые фигурные скобки пустыми:

|  |  |
| --- | --- |
| 1 | > db.users.find({}, {age: 1, \_id: 0}) |

**Запрос к вложенным объектам**

Предыдущие запросы применялись к простым объектам. Но документы могут быть очень сложными по структуре. Например, добавим в коллекцию persons следующий документ:

|  |  |
| --- | --- |
| 1 | > db.users.insert({"name": "Alex", "age": 28, company: {"name":"microsoft",  "country":"USA"}}) |

Здесь определяется вложенный объект с ключом company. И чтобы найти все документы, у которых в ключе company вложенное свойство name=microsoft, нам надо использовать оператор точку:

|  |  |
| --- | --- |
| 1 | > db.users.find({"company.name": "microsoft"}) |

**Использование JavaScript**

MongoDB предоставляет замечательную возможность, создавать запросы, используя язык JavaScript. Например, создадим запрос, возвращающий те документы, в которых name=Tom. Для этого сначала объявляется функция:

|  |  |
| --- | --- |
| 1  2 | > fn = function() { return this.name=="Tom"; }  > db.users.find(fn) |



Этот запрос эквивалентен следующему:

|  |  |
| --- | --- |
| 1 | > db.users.find("this.name=='Tom'") |

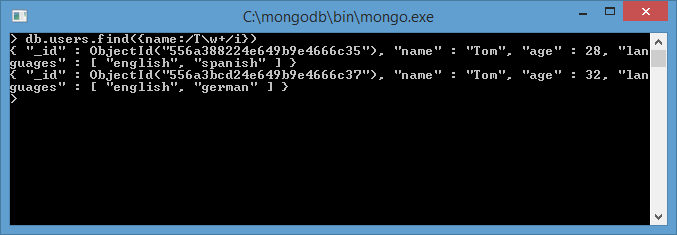
Собственно только запросами область применения JavaScript в консоли mongo не ограничена. Например, мы можем создать какую-нибудь функцию и применять ее:

|  |  |
| --- | --- |
| 1  2  3 | > function sqrt(n) { return n\*n; }  > sqrt(5)  25 |

**Использование регулярных выражений**

Еще одной замечательной возможностью при построении запросов является использование регулярных выражений. Например, найдем все документы, в которых значение ключа name начинается с буквы T:

> db.users.find({name:/T\w+/i})



**Настройка запросов и сортировка**

MongoDB представляет ряд функций, которые помогают управлять выборкой из бд. Одна из них - функция **limit**. Она задает максимально допустимое количество получаемых документов. Количество передается в виде числового параметра. Например, ограничим выборку тремя документами:

|  |  |
| --- | --- |
| 1 | > db.users.find().limit(3) |

В данном случае мы получим первые три документа (если в коллекции 3 и больше документов). Но что, если мы хотим произвести выборку не сначала, а пропустив какое-то количество документов? В этом нам поможет функция **skip**. Например, пропустим первые три записи:

|  |  |
| --- | --- |
| 1 | > db.users.find().skip(3) |

MongoDB предоствляет возможности отсортировать полученный из бд набор данных с помощью функции **sort**. Передавая в эту функцию значения 1 или -1, мы можем указать в каком порядке сортировать: по возрастанию (1) или по убыванию (-1). Во многом эта функция по действию аналогична выражению ORDER BY в SQL. Например, сортировка по возрастанию по полю name:

|  |  |
| --- | --- |
| 1 | > db.users.find().sort({name: 1}) |

Ну и в конце надо отметить, что мы можем совмещать все эти функции в одной цепочке:

|  |  |
| --- | --- |
| 1 | > db.users.find().sort({name: 1}).skip(3).limit(3) |

**Поиск одиночного документа**

Если все документы извлекаются функцией find, то одиночный документ извлекается функцией findOne. Ее действие аналогично тому, как если бы мы использовали функцию limit(1), которая также извлекает первый документ коллекции. А комбинация функций skip и limit извлечет документ по нужному местоположению.

**Параметр $natural**

Если вдруг нам надо отсортировать ограниченную коллекцию, то мы можем воспользоваться параметром $natural. Этот параметр позволяет задать сортировку: документы передаются в том порядке, в каком они были добавлены в коллекцию, либо в обратном порядке.

Например, отберем последние пять документов:

|  |  |
| --- | --- |
| 1 | > db.users.find().sort({ $natural: -1 }).limit(5) |

**Оператор $slice**

$slice является в некотором роде комбинацией функций limit и skip. Но в отличие от них $slice может работать с массивами.

Оператор $slice принимает два параметра. Первый параметр указывает на общее количество возвращаемых документов. Второй параметр необязательный, но если он используется, тогда первый параметр указывает на смещение относительно начала (как функция skip), а второй - на ограничение количества извлекаемых документов.

Например, в каждом документе определен массив languages для хранения языков, на которых говорит человек. Их может быть и 1, и 2, и 3 и более. И допустим, ранее мы добавили следующий объект:

|  |  |
| --- | --- |
| 1 | > db.users.insert({"name": "Tom", "age": "32", languages: ["english",  "german"]}) |

И мы хотим при выводе документов сделать так, чтобы в выборку попадал только один язык из массива languages, а не весь массив:

|  |  |
| --- | --- |
| 1 | > db.users.find ({name: "Tom"}, {languages: {$slice : 1}}) |

Данный запрос при извлечении документа оставит в результате только первый язык из массива languages, то есть в данном случае english.

Обратная ситуация: нам надо оставить в массиве также один элемент, но не с начала, а с конца. В этом случае необходимо передать в параметр отрицательное значение:

|  |  |
| --- | --- |
| 1 | > db.users.find ({name: "Tom"}, {languages: {$slice : -1}}); |

Теперь в массиве окажется german, так как он первый с конца в добавленном элементе.

Используем сразу два параметра:

|  |  |
| --- | --- |
| 1 | > db.users.find ({name: "Tom"}, {languages: {$slice : [-1, 1]}}); |

Первый параметр говорит начать выборку элементов с конца (так как отрицательное значение), а второй параметр указывает на количество возвращаемых элементов массива. В итоге в массиве language окажется "german"

**Курсоры**

Результат выборки, получаемой с помощью функции find, называется **курсором**:

|  |  |
| --- | --- |
| 1 | > var cursor = db.users.find(); null; |

Чтобы получить курсор и сразу же не выводить все содержащиеся в нем данные, после метода find() добавляет через точку с запятой выражение null;

Курсоры инкапсулируют в себе наборы получаемых из бд объектов. Используя синтаксис языка javascript и методы курсоров, мы можем вывести полученные документы на экран и как-то их обработать. Например:

|  |  |
| --- | --- |
| 1  2  3  4  5 | > var cursor = db.users.find();null;  > while(cursor.hasNext()){  ... obj = cursor.next();  ... print(obj["name"]);  ... } |

Курсор обладает методом **hasNext**, который показывает при переборе, имеется ли еще в наборе документ. А метод **next** извлекает текущий документ и перемещает курсор к следующему документу в наборе. В итоге в переменной obj оказывается документ, к полям которого мы можем получить доступ.

Также для перебора документов в курсоре в качестве альтернативы мы можем использовать конструкцию итератора javascript - **forEach**:

|  |  |
| --- | --- |
| 1  2  3  4 | > var cursor = db.users.find()  > cursor.forEach(function(obj){  ... print(obj.name);  ... }) |

Чтобы ограничить размер выборки, используется метод limit, принимающий количество документов:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | > var cursor = db.users.find();null;  null  > cursor.limit(5);null;  null  > cursor.forEach(function(obj){  ... print(obj.name);  ... }) |

Используя метод sort(), можно отсортировать документы в курсоре:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | > var cursor = db.users.find();null;  null  > cursor.sort({name:1});null;  null  > cursor.forEach(function(obj){  ... print(obj.name);  ... }) |

Выражение cursor.sort({name:1}) сортирует документы в курсоре по полю name по возрастанию. Если мы хотим отсортировать по убыванию, то вместо 1 используем -1: cursor.sort({name:-1})

И еще один метод skip() позволяет пропустить при выборке определенное количество документов:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | > var cursor = db.users.find();null;  null  > cursor.skip(2);null;  null  > cursor.forEach(function(obj){  ... print(obj.name);  ... }) |

В данном случае пропускаем два документа.

И кроме того, можно объединять все эти методы в цепочки:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | > var cursor = db.users.find();null;  null  > cursor.sort({name:1}).limit(3).skip(2);null;  null  > cursor.forEach(function(obj){  ... print(obj.name);  ... }) |

**Команды группировки**

**Последнее обновление: 30.10.2015**

Отдельно стоит рассмотреть команды группировки: count, group, distinct.

**Число элементов в коллекции**

С помощью функции count() можно получить число элементов в коллекции:

|  |  |
| --- | --- |
| 1 | > db.users.count() |

Можно группировать параметры поиска и функцию count, чтобы подсчитать, сколько определенных документов, например, у которых name=Tom:

|  |  |
| --- | --- |
| 1 | > db.users.find({name: "Tom"}).count() |

Более того мы можем создавать цепочки функций, чтобы конкретизировать условия подсчета:

|  |  |
| --- | --- |
| 1 | > db.users.find({name: "Tom"}).skip(2).count(true) |

Здесь надо отметить, что по умолчанию функция count не используется с функциями limit и skip. Чтобы их использовать, как в примере выше, в функцию count надо передать булевое значение true

**Функция distinct**

В коллекции могут быть документы, которые содержат одинаковые значения для одного или нескольких полей. Например, в нескольких документах определено name: "Tom". И нам надо найти только уникальные различающиеся значения для одного из полей документа. Для этого мы можем воспользоваться функцией **distinct**:

|  |  |
| --- | --- |
| 1  2 | > db.users.distinct("name")  ["Tom", "Bill", "Bob"] |

Здесь я запрашиваю только уникальные значения для поля name. И на следующей строке консоль выводит в виде массива найденные уникальные значения.

**Группировка и метод group**

Использование метода group аналогично применению выражения GROUP BY в SQL. Метод group принимает три параметра:

* **key**: указывает на ключ, по которому надо проводить группировку
* **initial**: инициализирует поля документа, который будет представлять группу документов
* **reduce**: представляет функцию, возвращающую количество элементов. Эта функция принимает в качестве аргументов два параметра: текущий элемент и агрегатный результирующий документ для текущей группы
* **keyf**: необязательный параметр. Используется вместо параметра key и представляет функцию, которая возвращает объект key
* **cond**: необязательный параметр. Представляет собой условие, которое должно возвращать true, иначе документ не примет участия в группировке. Если данный параметр неуказан, то в группировке участвуют все документы
* **finalize**: необязательный параметр. Представляет функцию, которая срабатывает перед тем, как будут возвращены результаты группировки.

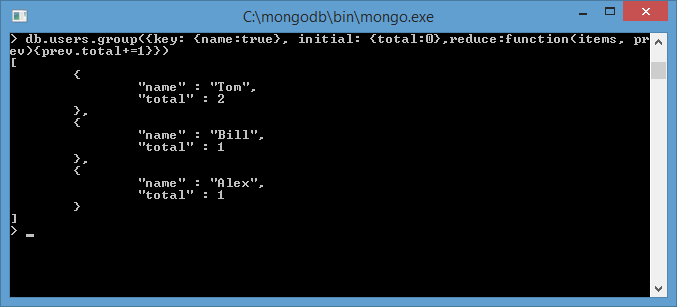
Например:

|  |  |
| --- | --- |
| 1  2 | > db.users.group ({key: {name : true}, initial: {total : 0},  reduce : function (curr, res){res.total += 1}}) |

Разберем выражение. Параметр key указывает, что группировка будет проводиться по ключу name: key: {name : true}

Значение параметра initial инициализирует поля результирующего документа, который будет представлять группу. Так, в данном случае устанавливается начальное значение поля total. Это поле будет представлять количество элементов для группы. И так как элементов может и не быть, то инициализируем нулем.

Параметр reduce представляет функцию, где параметр curr ссылается на текущий объект в группе, а res представляет текущую группу. Если найден еще один объект с определенным значением для поля name, то этот документ добавляется в группу, а в документе res на единицу увеличивается значение total.



Как видно из скриншота, в коллекции users оказалось два документа с name=Tom, и по одному документу с name=Alex и name=Bill

Используя группирование, следует учитывать, что функция group не работает в сегментированных кластерах ( sharded cluster). В этом случае вместо функции group следует использовать функциональность map-reduce.

**Операторы выборки**

**Последнее обновление: 28.03.2018**

**Условные операторы**

Условные операторы задают условие, которому должно соответствовать значение поля документа:

* **$eq** (равно)
* **$ne** (не равно)
* **$gt** (больше чем)
* **$lt** (меньше чем)
* **$gte** (больше или равно)
* **$lte** (меньше или равно)
* **$in** определяет массив значений, одно из которых должно иметь поле документа
* **$nin** определяет массив значений, которые не должно иметь поле документа

Например, найдем все документы, у которых значение ключа age меньше 30:

|  |  |
| --- | --- |
| 1 | > db.users.find ({age: {$lt : 30}}) |

Аналогично будет использование других операторов сравнения. Например, тот же ключ, только больше 30:

|  |  |
| --- | --- |
| 1 | > db.users.find ({age: {$gt : 30}}) |

Обратите внимание, что сравнение здесь проводится над целочисленными типами, а не строками. Если ключ age представляет строковые значения, то соответственно надо проводить сравнение над строками: db.users.find ({age: {$gt : "30"}}), однако результат будет тем же.

Но представим ситуацию, когда нам надо найти все объкты со значением поля age больше 30, но меньше 50. В этом случае мы можем комбинировать два оператора:

|  |  |
| --- | --- |
| 1 | > db.users.find ({age: {$gt : 30, $lt: 50}}) |

Найдем пользователей, возраст которых равен 22:

|  |  |
| --- | --- |
| 1 | > db.users.find ({age: {$eq : 22}}) |

По сути это аналогия следующего запроса:

|  |  |
| --- | --- |
| 1 | > db.users.find ({age: 22}) |

Обратная операция - найдем пользователей, возраст которых НЕ равен 22:

|  |  |
| --- | --- |
| 1 | > db.users.find ({age: {$ne : 22}}) |

Оператор $in определяет массив возможных выражений и ищет те ключи, значение которых имеется в массиве:

|  |  |
| --- | --- |
| 1 | > db.users.find ({age: {$in : [22, 32]}}) |

Противоположным образом действует оператор $nin - он определяет массив возможных выражений и ищет те ключи, значение которых отсутствует в этом массиве:

|  |  |
| --- | --- |
| 1 | > db.users.find ({age: {$nin : [22, 32]}}) |

**Логические операторы**

Логические операторы выполняются над условиями выборки:

* **$or**: соединяет два условия, и документ должен соответствовать одному из этих условий
* **$and**: соединяет два условия, и документ должен соответствовать обоим условиям
* **$not**: документ должен НЕ соответствовать условию
* **$nor**: соединяет два условия, и документ должен НЕ соответстовать обоим условиям

**Оператор $or**

Оператор **$or** представляет логическую операцию ИЛИ и определяет набор пар ключ-значение, которые должны иметься в документе. И если документ имеет хоть одну такую пару ключ-значение, то он соответствует данному запросу и извлекается из бд:

|  |  |
| --- | --- |
| 1 | > db.users.find ({$or : [{name: "Tom"}, {age: 22}]}) |

Это выражение вернет нам все документы, в которых либо name=Tom, либо age=22.

Другой пример вернет нам все документы, в которых name=Tom, а age равно либо 22, либо среди значений languages есть "german":

|  |  |
| --- | --- |
| 1 | > db.users.find ({name: "Tom", $or : [{age: 22}, {languages: "german"}]}) |

В подвыраженях or можно применять условные операторы:

|  |  |
| --- | --- |
| 1 | > db.users.find ({$or : [{name: "Tom"}, {age: {$gte:30}}]}) |

В данном случае мы выбираем все документы, где name="Tom" или поле age имеет значение от 30 и выше.

**Оператор $and**

Оператор **$and** представляет логическую операцию И (логическое умножение) и определяет набор критериев, которым обязателньо должен соответствовать документ. В отличие от оператора $or документ должен соответствовать всем указанным критериям. Например:

|  |  |
| --- | --- |
| 1 | > db.users.find ({$and : [{name: "Tom"}, {age: 32}]}) |

Здесь выбираемые документы обязательно должны имееть имя Tom и возраст 32 - оба этих признака.

**Поиск по массивам**

Ряд операторов предназначены для работы с массивами:

* **$all**: определяет набор значений, которые должны иметься в массиве
* **$size**: определяет количество элементов, которые должны быть в массиве
* **$elemMatch**: определяет условие, которым должны соответствовать элемены в массиве

**$all**

Оператор $all определяет массив возможных выражений и требует, чтобы документы имели весь определяемый набор выражений. Соответственно он применяется для поиску по массиву. Например, в документах есть массив languages, хранящий иностранные языки, на которых говорит пользователь. И чтобы найти всех людей, говорящих одновременно и по-английски, и по-французски, мы можем использовать следующее выражение:

|  |  |
| --- | --- |
| 1 | > db.users.find ({languages: {$all : ["english", "french"]}}) |

**Оператор $elemMatch**

Оператор **$elemMatch** позволяет выбрать документы, в которых массивы содержат элементы, попадающие под определенные условия. Например, пусть в базе данных будет коллекция, которая содержит оценки пользователей по определенным курсам. Добавим несколько документов:

|  |  |
| --- | --- |
| 1  2 | > db.grades.insertMany([{student: "Tom", courses:[{name: "Java", grade: 5},  {name: "MongoDB", grade: 4}]},  {student: "Alice", courses:[{name: "C++", grade: 3}, {name: "MongoDB",  grade: 5}]}]) |

Каждый документ имеет массив courses, который в свою очередь состоит из вложенных документов.

Теперь найдем студентов, которые для курса MongoDB имеют оценку выше 3:

|  |  |
| --- | --- |
| 1 | > db.grades.find({courses: {$elemMatch: {name: "MongoDB", grade: {$gt: 3}}}}) |

**Оператор $size**

Оператор $size используется для нахождения документов, в которых массивы имеют число элементов, равным значению $size. Например, извлечем все документы, в которых в массиве laguages два элемента:

|  |  |
| --- | --- |
| 1 | > db.users.find ({languages: {$size:2}}) |

Такой запрос будет соответствовать, например, следующему документу:

|  |  |
| --- | --- |
| 1 | {"name": "Tom", "age": 32, languages: ["english", "german"]} |

**Оператор $exists**

Оператор $exists позволяет извлечь только те документы, в которых определенный ключ присутствует или отсутствует. Например, вернем все документы, в который есть ключ company:

|  |  |
| --- | --- |
| 1 | > db.users.find ({company: {$exists:true}}) |

Если мы укажем у оператора $exists в качестве параметра false, то запрос вернет нам только те документы, в которых не определен ключ company.

**Оператор $type**

Оператор **$type** извлекает только те документы, в которых определенный ключ имеет значение определенного типа, например, строку или число:

|  |  |
| --- | --- |
| 1  2 | > db.users.find ({age: {$type:"string"}})  > db.users.find ({age: {$type:"number"}}) |

**Оператор $regex**

Оператор $regex задает регулярное выражение, которому должно соответствовать значение поля. Например, пусть поле name обязательно имеет букву "b":

|  |  |
| --- | --- |
| 1 | > db.users.find ({name: {$regex:"b"}}) |

Важно понимать, что $regex принимает не просто строки, а именно регулярные выражения, например: name: {$regex:"om$"} - значение name должно оканчиваться на "om".

**Обновление данных**

**Последнее обновление: 08.06.2017**

**Метод save**

Как и другие системы управления базами данных MongoDB предоставляет возможность обновления данных. Наиболее простым для использования является метод save. В качестве параметра этот метод принимает документ.

В этот документ в качестве поля можно передать параметр \_id. Если метод находит документ с таким значением \_id, то документ обновляется. Если же с подобным \_id нет документов, то документ вставляется.

Если параметр \_id не указан, то документ вставляется, а параметр \_id генерируется автоматически как при обычном добавлении через функцию insert:

|  |  |
| --- | --- |
| 1 | > db.users.save({name: "Eugene", age : 29, languages: ["english", "german",  "spanish"]}) |

В качестве результата функция возвращает объект WriteResult. Например, при успешном сохранении мы получим:

|  |  |
| --- | --- |
| 1 | WriteResult({"nInserted" : 1 }) |

**update**

Более детальную настройку при обновлении предлагает функция **update**. Она принимает три параметра:

* query: принимает запрос на выборку документа, который надо обновить
* objNew: представляет документ с новой информацией, который заместит старый при обновлении
* options: определяет дополнительные параметры при обновлении документов. Может принимать два аргумента: upsert и multi.

Если параметр upsert имеет значение true, что mongodb будет обновлять документ, если он найден, и создавать новый, если такого документа нет. Если же он имеет значение false, то mongodb не будет создавать новый документ, если запрос на выборку не найдет ни одного документа.

Параметр multi указывает, должен ли обновляться первый элемент в выборке (используется по умолчанию, если данный параметр не указан) или же должны обновляться все документы в выборке.

Например:

|  |  |
| --- | --- |
| 1 | > db.users.update({name : "Tom"}, {name: "Tom", age : 25}, {upsert: true}) |

Теперь документ, найденный запросом {name : "Tom"}, будет перезаписан документом {"name": "Tom", "age" :"25"}.

Функция update() также возвращает объект WriteResult. Например:

|  |  |
| --- | --- |
| 1 | WriteResult({"nMatched" : 1, "nUpserted": 0, "nModified": 1}) |

В данном случае результат говорит нам о том, что найден один документ, удовлетворяющий условию, и один документ был обновлен.

**Обновление отдельного поля**

Часто не требуется обновлять весь документ, а только значение одного из его ключей. Для этого применяется оператор **$set**. Если документ не содержит обновляемое поле, то оно создается.

|  |  |
| --- | --- |
| 1 | > db.users.update({name : "Tom", age: 29}, {$set: {age : 30}}) |

Если обновляемого поля в документе нет, до оно добавляется:

|  |  |
| --- | --- |
| 1 | > db.users.update({name : "Tom", age: 29}, {$set: {salary : 300}}) |

В данном случае обновлялся только один документ, первый в выборке. Указав значение multi:true, мы можем обновить все документы выборки:

|  |  |
| --- | --- |
| 1 | > db.users.update({name : "Tom"}, {$set: {name: "Tom", age : 25}},  {multi:true}) |

Для простого увеличения значения числового поля на определенное количество единиц применяется оператор **$inc**. Если документ не содержит обновляемое поле, то оно создается. Данный оператор применим только к числовым значениям.

|  |  |
| --- | --- |
| 1 | > db.users.update({name : "Tom"}, {$inc: {age:2}}) |

**Удаление поля**

Для удаления отдельного ключа используется оператор **$unset**:

|  |  |
| --- | --- |
| 1 | > db.users.update({name : "Tom"}, {$unset: {salary: 1}}) |

Если вдруг подобного ключа в документе не существует, то оператор не оказывает никакого влияния. Также можно удалять сразу несколько полей:

|  |  |
| --- | --- |
| 1 | > db.users.update({name : "Tom"}, {$unset: {salary: 1, age: 1}}) |

**updateOne и updateMany**

Метод **updateOne** похож на метод update за тем исключением, что он обновляет только один документ.

|  |  |
| --- | --- |
| 1 | > db.users.updateOne({name : "Tom", age: 29}, {$set: {salary : 360}}) |

Если необходимо обновить все документы, соответствующие некоторому критерию, то применяется метод **updateMany()**:

|  |  |
| --- | --- |
| 1 | > db.users.updateMany({name : "Tom"}, {$set: {salary : 560}}) |

**Обновление массивов**

**Оператор $push**

Оператор $push позволяет добавить еще одно значение к уже существующему. Например, если ключ в качестве значения хранит массив:

|  |  |
| --- | --- |
| 1 | > db.users.updateOne({name : "Tom"}, {$push: {languages: "russian"}}) |

Если ключ, для которого мы хотим добавить значение, не представляет массив, то мы получим ошибку Cannot apply $push/$pushAll modifier to non-array.

Используя оператор $each, можно добавить сразу несколько значений:

|  |  |
| --- | --- |
| 1 | > db.users.update({name : "Tom"}, {$push: {languages: {$each: ["russian",  "spanish", "italian"]}}}) |

Еще пара операторов позволяет настроить вставку. Оператор **$position** задает позицию в массиве для вставки элементов, а оператор **$slice** указывает, сколько элементов оставить в массиве после вставки.

|  |  |
| --- | --- |
| 1 | > db.users.update({name : "Tom"}, {$push: {languages: {$each: ["german",  "spanish", "italian"], $position:1, $slice:5}}}) |

В данном случае элементы ["german", "spanish", "italian"] будут вставляться в массив languages с 1-го индекса, и после вставки, в массиве останутся только 5 первых элементов.

**Оператор $addToSet**

Оператор $addToSet подобно оператору $push добавляет объекты в массив. Отличие состоит в том, что $addToSet добавляет данные, если их еще нет в массиве:

|  |  |
| --- | --- |
| 1 | > db.users.update({name : "Tom"}, {$addToSet: {languages: "russian"}}) |

**Удаление элемента из массива**

Оператор $pop позволяет удалять элемент из массива:

> db.users.update({name : "Tom"}, {$pop: {languages: 1}})

Указывая для ключа languages значение 1, мы удаляем первый элемент с конца. Чтобы удалить первый элемент сначала массива, надо передать отрицательное значение:

|  |  |
| --- | --- |
| 1 | > db.users.update({name : "Tom"}, {$pop: {languages: -1}}) |

Несколько иное действие предполагает оператор $pull. Он удаляет каждое вхождение элемента в массив. Например, через оператор $push мы можем добавить одно и то же значение в массив несколько раз. И теперь с помощью $pull удалим его:

|  |  |
| --- | --- |
| 1 | > db.users.update({name : "Tom"}, {$pull: {languages: "english"}}) |

А если мы хотим удалить не одно значение, а сразу несколько, тогда мы можем применить оператор $pullAll:

|  |  |
| --- | --- |
| 1 | > db.users.update({name : "Tom"}, {$pullAll: {languages: ["english", "german",  "french"]}}) |

**Удаление данных**

**Последнее обновление: 30.10.2015**

Для удаления документов в MongoDB предусмотрен метод remove:

|  |  |
| --- | --- |
| 1 | > db.users.remove({name : "Tom"}) |

Метод remove() возвращает объект WriteResult. При успешном удалении одного документа результат будет следующим:

|  |  |
| --- | --- |
| 1 | WriteResult({"nRemoved" : 1}) |

В итоге все найденные документы с name=Tom будут удалены. Причем, как и в случае с find, мы можем задавать условия выборки для удаления различными способами (в виде регулярных выражений, в виде условных конструкций и т.д.):

|  |  |
| --- | --- |
| 1  2 | > db.users.remove({name : /T\w+/i})  > db.users.remove({age: {$lt : 30}}) |

Метод remove также может принимать второй необязательный параметр булевого типа, который указывает, надо удалять один элемент или все элементы, соответствующие условию. Если этот параметр равен true, то удаляется только один элемент. По умолчанию он равен false:

|  |  |
| --- | --- |
| 1 | > db.users.remove({name : "Tom"}, true) |

Чтобы удалить разом все документы из коллекции, надо оставить пустым параметр запроса:

|  |  |
| --- | --- |
| 1 | > db.users.remove({}) |

**Удаление коллекций и баз данных**

Мы можем удалять не только документы, но и коллекции и базы данных. Для удаления коллекций используется функция **drop**:

|  |  |
| --- | --- |
| 1 | > db.users.drop() |

И если удаление коллекции пройдет успешно, то консоль выведет:

true

Чтобы удалить всю базу данных, надо воспользоваться функцией **dropDatabase()**:

|  |  |
| --- | --- |
| 1 | > db.dropDatabase() |



**Установка ссылок в БД**

**Последнее обновление: 30.10.2015**

В реляционных базах данных можно устанавливать внешние ключи, когда поля из одной таблицы ссылаются на поля в другой таблице. В MongoDB также можно устанавливать ссылки.

**Ручная установка ссылок**

Ручная установка ссылок сводится к присвоению значения поля \_id одного документа полю другого документа. Допустим, у нас могут быть коллекции, представляющие компании и работников, работающих в этих компаниях. Итак, сначала добавим в коллекцию companies документ представляющий компанию:

|  |  |
| --- | --- |
| 1 | > db.companies.insert({"\_id" : "microsoft", year: 1974}) |

Теперь добавим в коллекцию persons документ, представляющий работника. В этом документе будет поле company, представляющее компанию, где работает работник. И очень важно, что в качестве значения для этого поля мы устанавливаем не объект company, а значение ключа \_id добавленного выше документа:

|  |  |
| --- | --- |
| 1 | > db.users.insert({name: "Tom", age: 28, company: "microsoft"}) |

Теперь получим документ из коллекции users:

|  |  |
| --- | --- |
| 1 | > user = db.users.findOne() |

В данном случае имеется в виду, что выше добавленный элемент будет единственным в коллекции.

После этого консоль выводит полученный документ. И теперь найдем ссылку на его компанию в коллекции companies:

|  |  |
| --- | --- |
| 1 | > db.companies.findOne({\_id: user.company}) |

И если документ с таким идентификатором обнаружен, он отображается на консоли:



**Автоматическое связывание**

Используя функциональность **DBRef**, мы можем установить автоматическое связывание между документами. Посмотрим на примере применение данной функциональности. Вначале добавим новый документ в коллекцию companies:

|  |  |
| --- | --- |
| 1  2 | > apple=({"name" : "apple", "year": 1976})  > db.companies.save(apple) |

Обратите внимание, что в данном случае сохранение идет с помощью метода save, не insert. Метод save при добавлении нового документа генерирует \_id. И после сохранения мы можем вывести документ на консоль: > apple

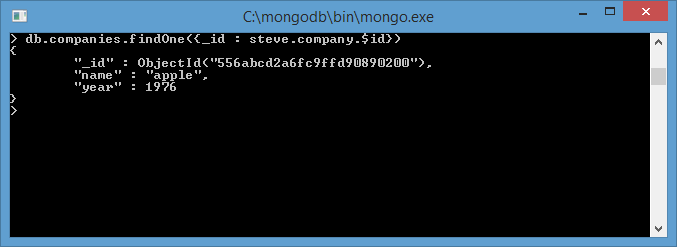
Теперь создадим новый документ для коллекции person, у которого ключ company свяжем с только что добавленным документом apple:

|  |  |
| --- | --- |
| 1  2 | > steve = ({"name": "Steve", "age": 25, company: new DBRef('companies', apple.\_id)})  > db.users.save(steve) |



И мы можем протестировать:

|  |  |
| --- | --- |
| 1 | > db.companies.findOne({\_id: steve.company.$id}) |



Посмотрев на примере, теперь разберем организацию ссылок между документами. Для связывания с документом apple использовалось следующее выражение company: new DBRef('companies', apple.\_id)}). Формальный синтаксис DBRef следующий:

|  |  |
| --- | --- |
| 1 | { "$ref" : название\_коллекции, "$id": значение [, "$db" : название\_бд ]} |

Первый параметр $ref указывает на коллекцию, где хранится связанный документ. Второй параметр указывает на значение, которое и будет представлять что-то типа внешнего ключа. Третий необязательный параметр указывает на базу данных.

При тестировании в качестве запроса на выборку указывается выражение \_id: steve.company.$id. Так как person.company представляет теперь объект new DBRef('companies', apple.\_id)}), то нам надо конкретизировать параметр steve.company.**$id**

**Работа с индексами**

**Последнее обновление: 08.07.2016**

При поиске документов в небольших коллекциях мы не испытаем особых проблем. Однако когда коллекции содержат миллионы документов, а нам надо сделать выборку по определенному полю, то поиск нужных данных может занять некоторое время, которое может оказаться критичным для нашей задачи. В этом случае нам могут помочь индексы.

Индексы позволяют упорядочить данные по определенному полю, что впоследствии ускорит поиск. Например, если мы в своем приложении или задаче, как правило, выполняем поиск по полю name, то мы можем индексировать коллекцию по этому полю:

|  |  |
| --- | --- |
| 1 | > db.users.createIndex({"name" : 1}) |

Таким образом с помощью метода createIndex устанавливается индекс по полю name. MongoDB позволяет установить до 64 индексов на одну коллекцию.

**Настройка индексов**

Если мы просто определим индекс для коллекции, например, db.users.createIndex({"name" : 1}), то мы все еще сможем добавлять в коллекцию документы с одинаковым значением ключа name. Однако, если нам потребуется, чтобы в коллекцию можно было добавлять документ с одним и тем же значением ключа только один раз, мы можем установить флаг unique:

|  |  |
| --- | --- |
| 1 | > db.users.createIndex({"name" : 1}, {"unique" : true}) |

Теперь, если мы попытаемся добавить в коллекцию два документа с одним и тем же значением name, то мы получим ошибку.

В тоже время тут есть свои тонкости. Так, документ может не иметь ключа name. В этом случае для добавляемого документа автоматически создается ключ name со значением null. Поэтому при добавлении второго документа, в котором не определен ключ name, будет выброшено исключение, так как ключ name со значением null уже присутствует в коллекции.

Также можно задать уникальный индекс сразу для двух полей:

|  |  |
| --- | --- |
| 1 | > db.users.createIndex({"name" : 1, "age" : 1}, {"unique" : true}) |

Однако в этом случае все добавляемые документы должны иметь уникальные значения для обоих полей.

Кроме того, тут есть свои ограничения. Например, значение поля, по которому идет индексация, не должно быть больше 1024 байт.

**Управление индексами**

Все индексы базы данных хранятся в системной коллекции *system.indexes*. Обратившись к ней, мы можем получить все индексы и связанную с ними информацию:

|  |  |
| --- | --- |
| 1 | > db.system.indexes.find() |

Также мы можем воспользоваться методом getIndexes и вывести всю информацию об индексах для конкретной коллекции:

|  |  |
| --- | --- |
| 1 | > db.users.getIndexes() |

Данная команда вернет вывод наподобие следующего:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18 | [      {          "v" : 1,          "key" : {                  "\_id" : 1          },          "ns" : "test.users",          "name" : "\_id\_"      },      {          "v" : 1,          "key" : {                  "name" : 1          },          "ns" : "test.users",          "name" : "name\_1"      }  ] |

Как мы видим, здесь для коллекции users (из бд test) определено 2 индекса: id и name. Поле key используется для поиска максимального и минимального значений, для различных операций, где надо применять данный индекс. Поле name применяется в качестве идентификатора для операций администрирования, например, для удаления индекса:

|  |  |
| --- | --- |
| 1 | > db.users.dropIndex("name\_1") |

**Управление коллекцией**

**Последнее обновление: 25.03.2018**

**Явное создание коллекции**

В предыдущих темах коллекция создавалась неявно автоматически при добавлении в нее первых данных. Но мы также можем создать ее явным образом, применив метод **db.createCollection(name, options)**, где name - название коллекции, а options - необязательный объект с дополнительными настройками инициализации. Например:

|  |  |
| --- | --- |
| 1  2 | > db.createCollection("accounts")  {"ok" : 1} |

Таким образом, создается коллекция accounts.

**Переименование коллекции**

В процессе работы, возможно, потребуется изменить название коллекции. Например, если при первом добавлении данных в ее названии была опечатка. И чтобы не удалять и затем пересоздавать коллекцию, следует использовать функцию renameCollection:

|  |  |
| --- | --- |
| 1 | > db.users.renameCollection("новое\_название") |

И если переименование пройдет удачно, то консоль отобразит:

{"ok" : 1}

**Ограниченные коллекции**

Когда мы отправляем запрос к бд на выборку, то MongoDB возвращает нам документы в том порядке, как правило, в котором они были добавлены. Однако такой порядок не всегда гарантируется, так как данные могут быть удалены, перемещены, изменены. Поэтому в MongoDB существует понятие **ограниченной коллекции** (capped collection). Подобная коллекция гарантирует, что документы будут располагаться в том же порядке, в котором они добавлялись в коллекцию. Ограниченные коллекции имеют фиксированный размер. И когда в коллекции уже нет места, наиболее старые документы удаляются, и в конец добавляются новые данные.

В отличие от обычных коллекций ограниченные мы можем задать явным образом. Например, создадим ограниченную коллекцию с названием profile и зададим для нее размер в 9500 байт:

|  |  |
| --- | --- |
| 1 | > db.createCollection("profile", {capped:true, size:9500}) |

И после удачного создания коллекции консоль выведет:

{"ok":1}

Также можно ограничить количество документов в коллекции, указав его в параметре max:

|  |  |
| --- | --- |
| 1 | > db.createCollection("profile", {capped:true, size:9500, max: 150}) |

Однако при таком способе создания коллекции следует учитывать, что если все место под коллекцию заполнено (например, выделенные нами 9500 байтов), а количество документов еще не достигло максимума, в данном случае 150, то в этом случае при добавлении нового документа самый старый документ будет удаляться, а на его место будет вставляться новый документ.

При обновлении документов в таких коллекциях надо помнить, что документы не должны расти в размерах, иначе обновление не удастся произвести.

Также нельзя удалять документы из подобных коллекций, можно только удалить всю коллекцию.

**Подколлекции**

Для упрощения организации данных в коллекциях мы можем использовать подколлекции. Например, данные по коллекции users надо разграничить на профили и учетные данные. И мы можем использовать создать коллекции db.users.profiles и db.users.accounts. При этом они не будут никак связаны с коллекцией users. То есть в итоге будут три разные коллекции, однако в плане логической организации хранения данных, возможно, для кого-то так будет проще.

<https://metanit.com/nosql/mongodb/3.1.php>