MongoDB 入门教程

董旭阳

2024-09-09

第 1 篇 MongoDB 简介

1.1 MongoDB 特性

MongoDB 是一个开源、跨平台、分布式文档数据库，属于 NoSQL（Not Only SQL）数据库的一种。

简单易用

MongoDB 是一个面向文档的数据库，使用文档（document）对象存储数据，这种方式比关系型数据库（RDBMS）中的数据行格式更加灵活。文档可以支持在单个记录中表示复杂的层级关系。

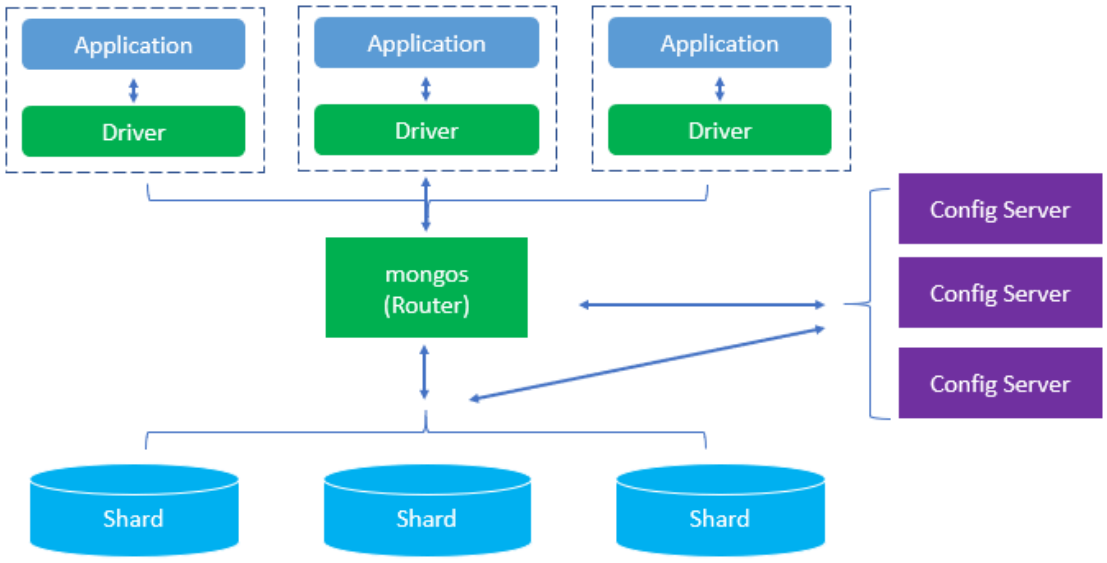
MongoDB 不需要预定义模式结构（schema），可以更加方便快捷地增加或删除文档中的字段。

可扩展

随着数据量的增长，我们将会面临系统可扩展性的挑战。一般来说，扩展方式有两种：

- 纵向扩展（Scaling up），升级服务器的资源（CPU、RAM 等）。
- 横向扩展（Scaling out），为集群增加更多的服务器。这种方式比纵向扩展更加经济、可扩展性更好，缺点就是管理更加复杂。

MongoDB 天生具有横向扩展性。MongoDB 可以将数据分布式存储到多个服务器中，同时可以自动管理跨节点的负载均衡，将数据操作路由到相应的服务器。下图演示了 MongoDB 使用 sharding 分片进行横向扩展的示意：



功能丰富

作为一个数据库管理系统，MongoDB 支持数据的插入、更新、删除以及查询。除此之外，MongoDB 还提供了以下功能：

- 索引
- 聚合
- 指定集合与索引的类型
- 文件存储

我们将会在后面的教程中详细介绍这些功能。

高性能

MongoDB 提供了高性能的数据存储，例如，嵌入式数据模型可以减少 I/O 操作，索引可以加速查询。

MongoDB 的理念就是创建一个可扩展、灵活且高性能的全功能数据库。

1.2 MongoDB 版本

MongoDB 提供了三种不同的版本：社区版、企业版以及云数据库（Atlas）。

MongoDB 社区版

MongoDB 社区版可以免费使用，支持 Windows、Linux 以及 macOS 平台。

MongoDB 社区版使用(SSPL)许可证，意味着如果我们将 MongoDB 作为一个公共服务提供给其他人使用，必须开源支持该服务的软件代码，例如管理或者监控该服务的工具。否则，我们需要购买企业版。

如果我们使用 MongoDB 社区版作为应用程序的一个组件，而不是最终的产品，可以免费使用。

MongoDB 企业版

MongoDB 企业版是 MongoDB 的商业版本，也是 MongoDB 企业高级订阅的一部分。

MongoDB 企业版提供了很多社区版不支持的功能，例如：

- 内存（In-Memory）存储引擎
- 审计功能
- Kerberos 认证
- LDAP 代理认证和 LDAP 授权
- 静态加密

MongoDB Atlas

MongoDB Atlas 是一个全球云数据库服务。它是一种数据库即服务（database as a service），可以让我们专注于应用开发，而不是管理数据库。

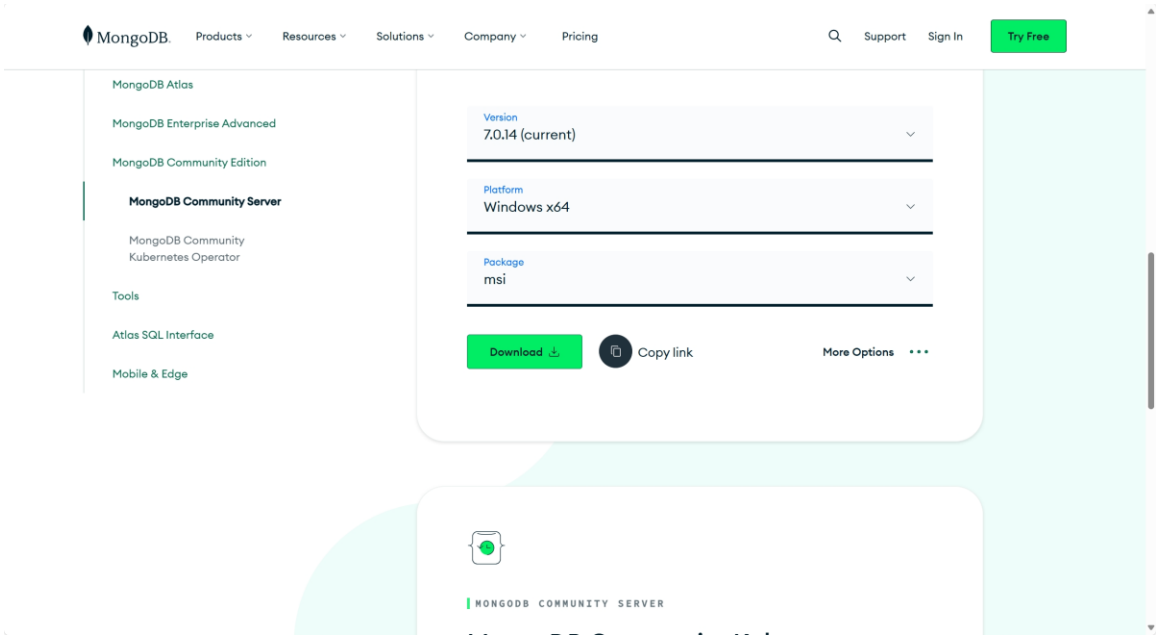
MongoDB Atlas 支持各种常见的云平台，包括 AWS、Azure 以及 GCP，国内的阿里云、腾讯云也提供了相应的服务。MongoDB Atlas 为个人提供了用于学习的免费数据库服务。

第 2 篇 MongoDB 安装

本篇我们将会学习如何安装 MongoDB 数据库服务器和管理开发工具。

2.1 下载 MongoDB 社区版

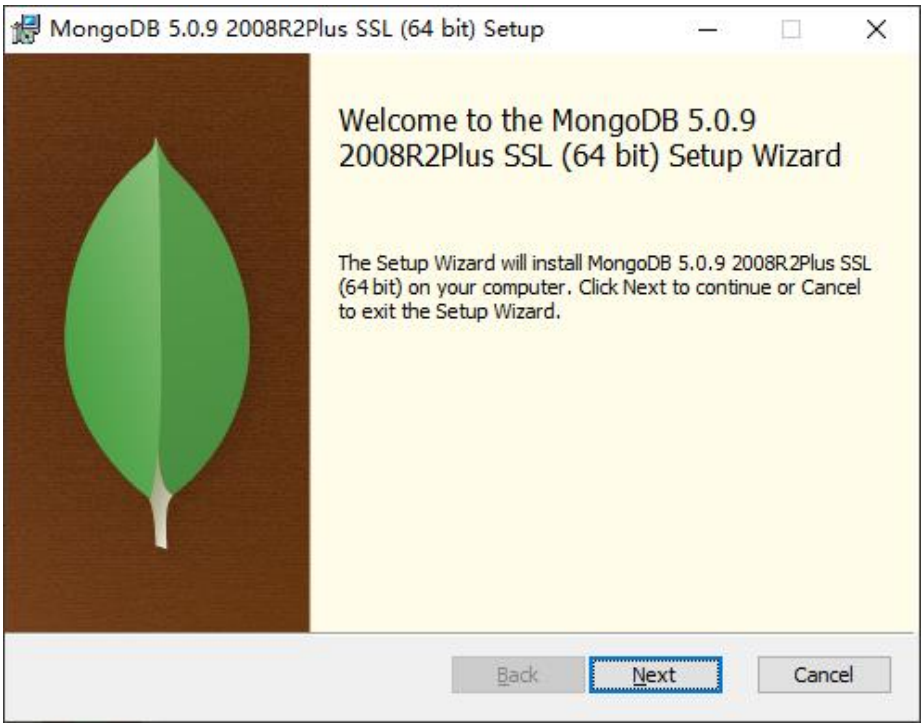
首先，打开 MongoDB 官方网站中的下载页面，然后选择 MongDB Community Server。



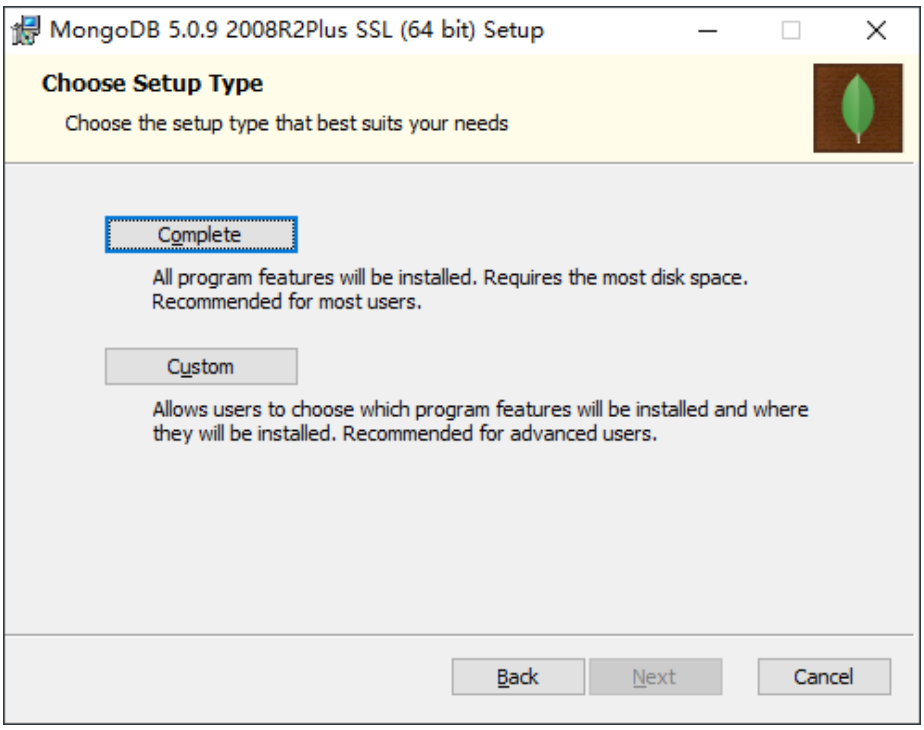
在页面右侧选择版本、平台和安装包类型，点击“Download”下载安装文件。

2.2 安装 MongoDB 社区版

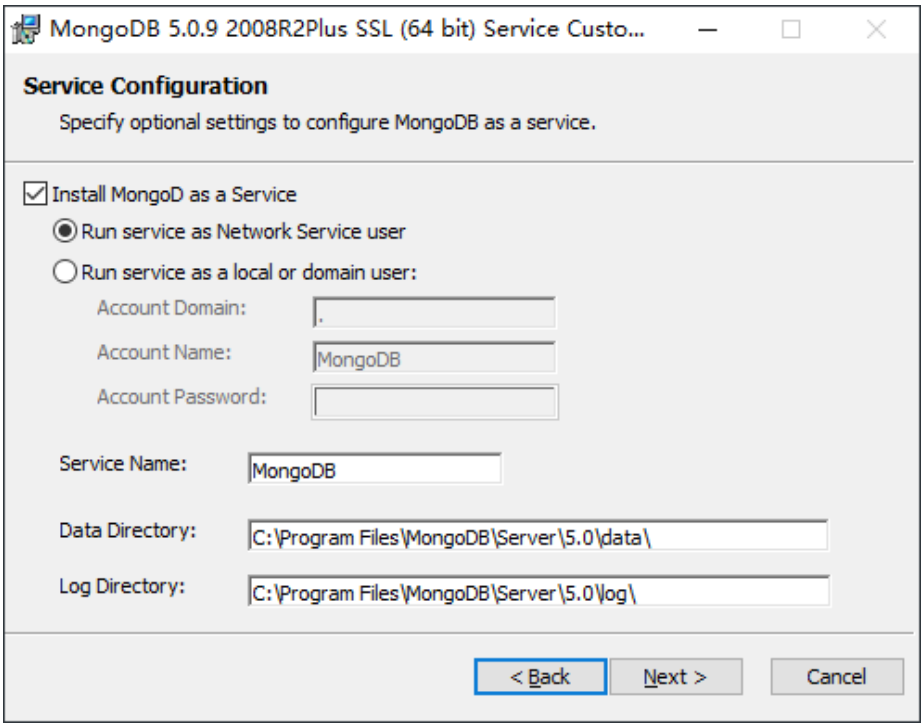
下载完成后双击安装文件，启动安装向导。



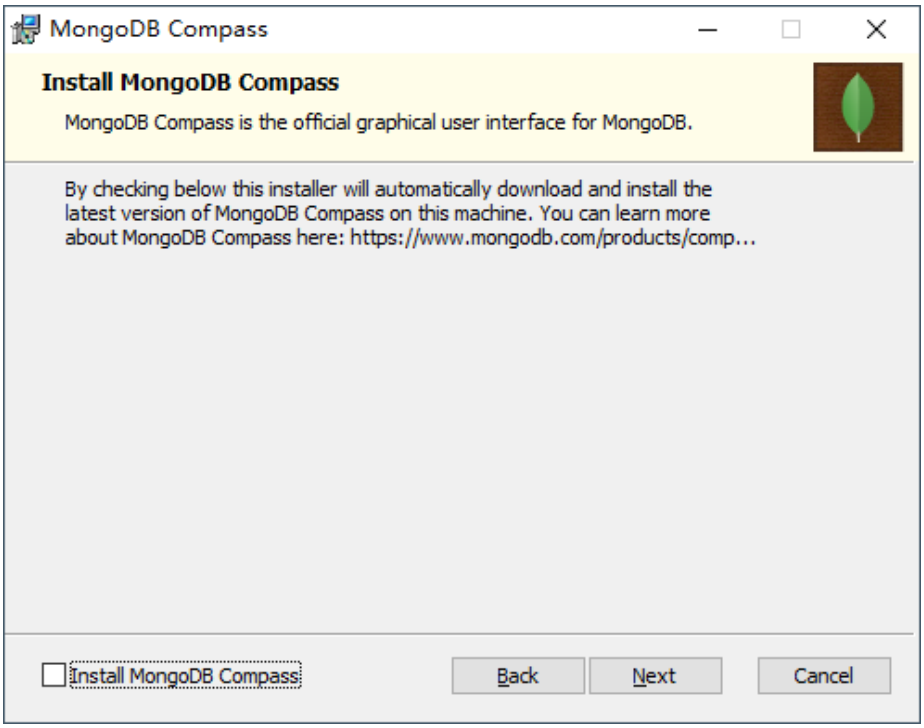
点击“Next”按钮：勾选“I accept the terms in the License Agreement”，点击“Next”按钮：



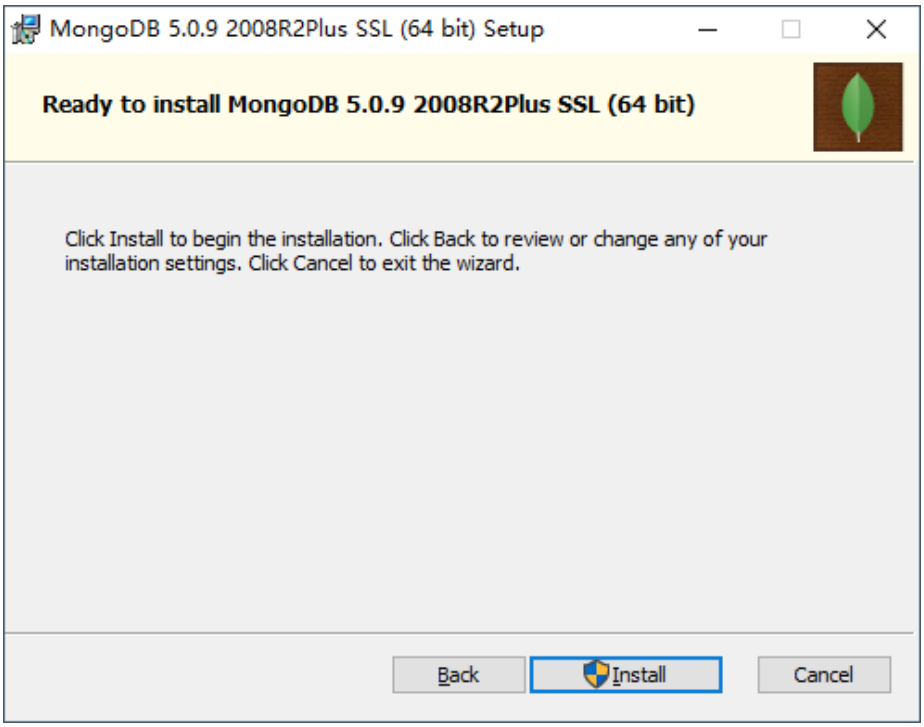
点击“Complete”按钮安装全部功能。如果想要自定义安装指定的功能，可以选择“Custom”按钮，仅推荐高级用户使用该选项。



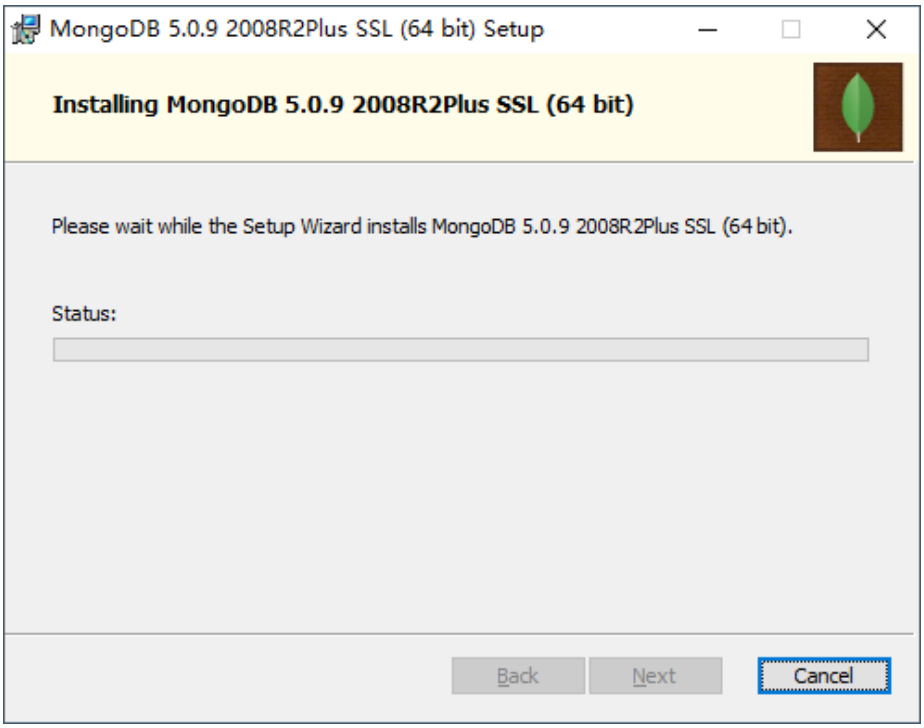
保留默认配置，点击“Next”按钮：



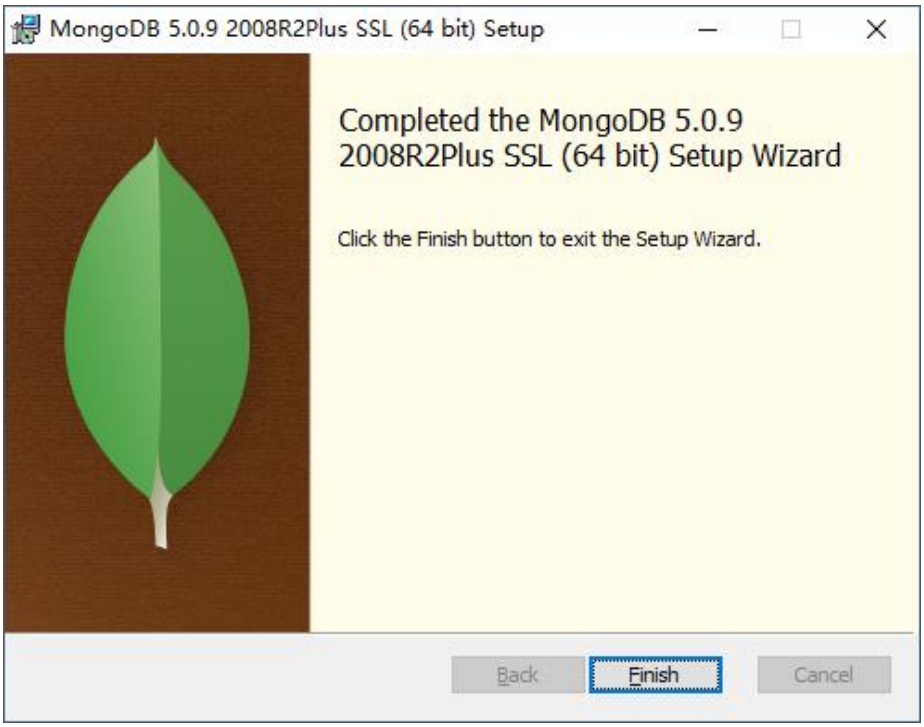
去掉“Install MongoDB Compass”复选框，点击“Next”按钮。如果我们选择了该复选框，安装程序会同时下载并安装客户端工具 MongoDB Compass。我们会在安装 MongoDB 服务器之后单独安装该功能。



点击“Install”按钮开始安装。



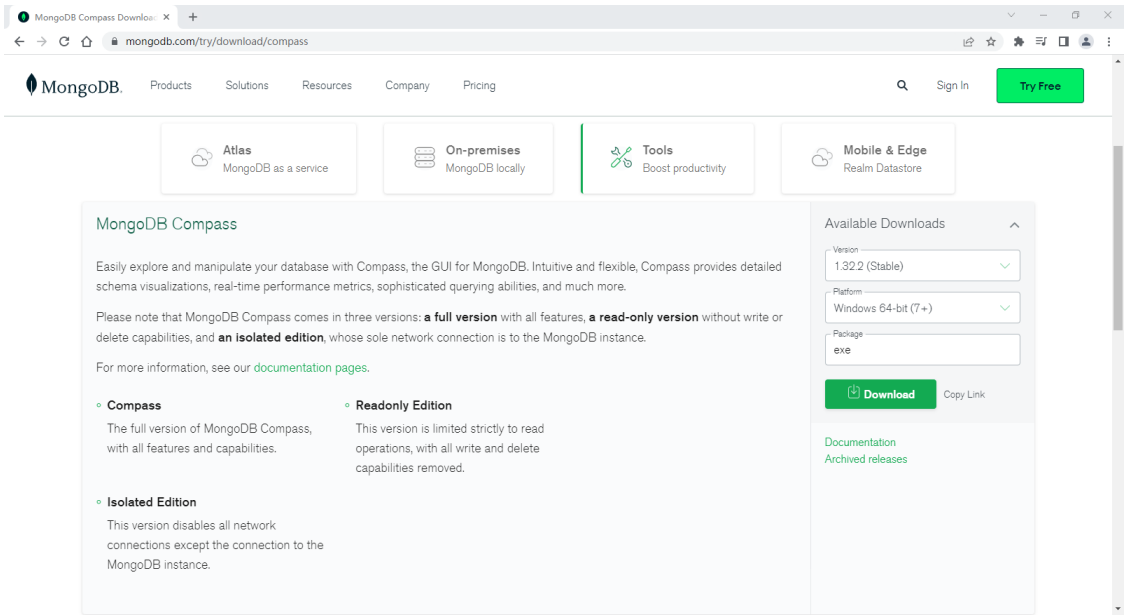
安装完成之后，点击“Finish”按钮退出。



接下来我们安装 MongoDB Compass。MongoDB Compass 是一个官方提供的 GUI 工具，可以用于连 MongoDB 服务器，执行查询、索引、文档验证等操作。

2.3 下载 MongoDB Compass

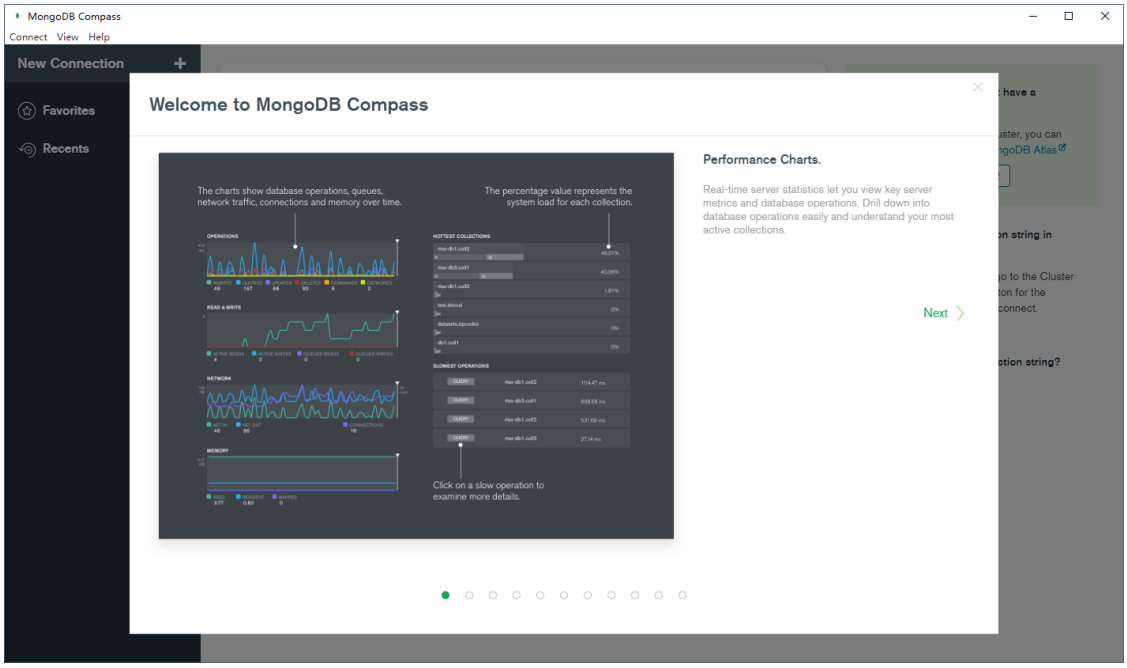
首先，点击官方网站上的下载页面。



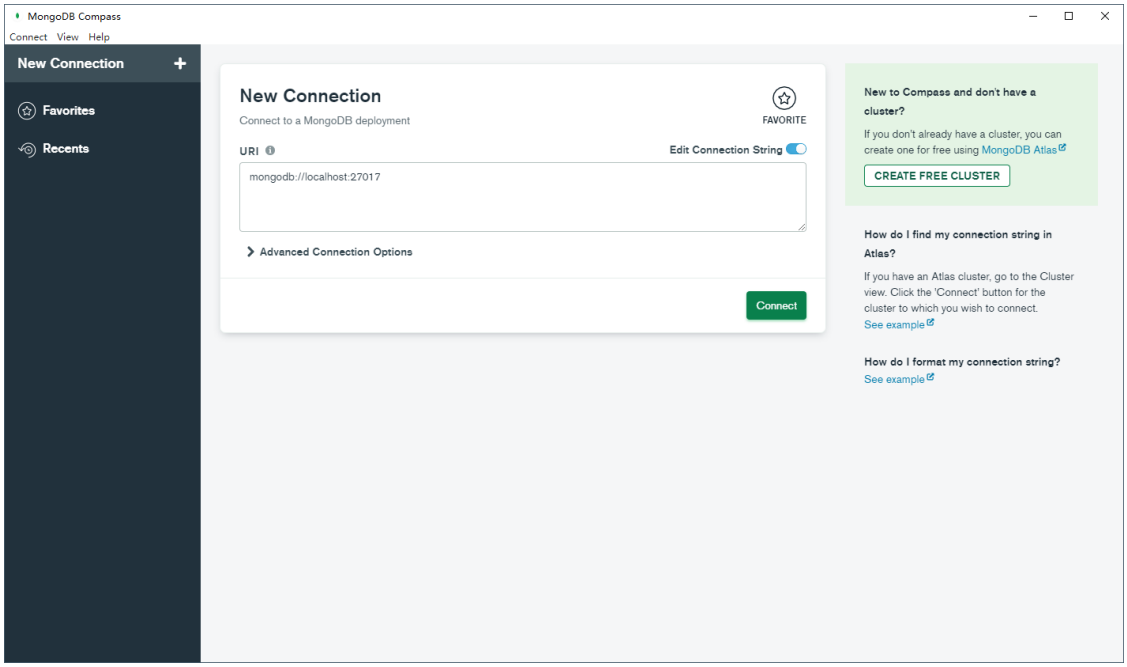
选择 MongoDB Compass 版本之后，点击“Download”按钮下载安装文件。

2.4 安装 MongoDB Compass

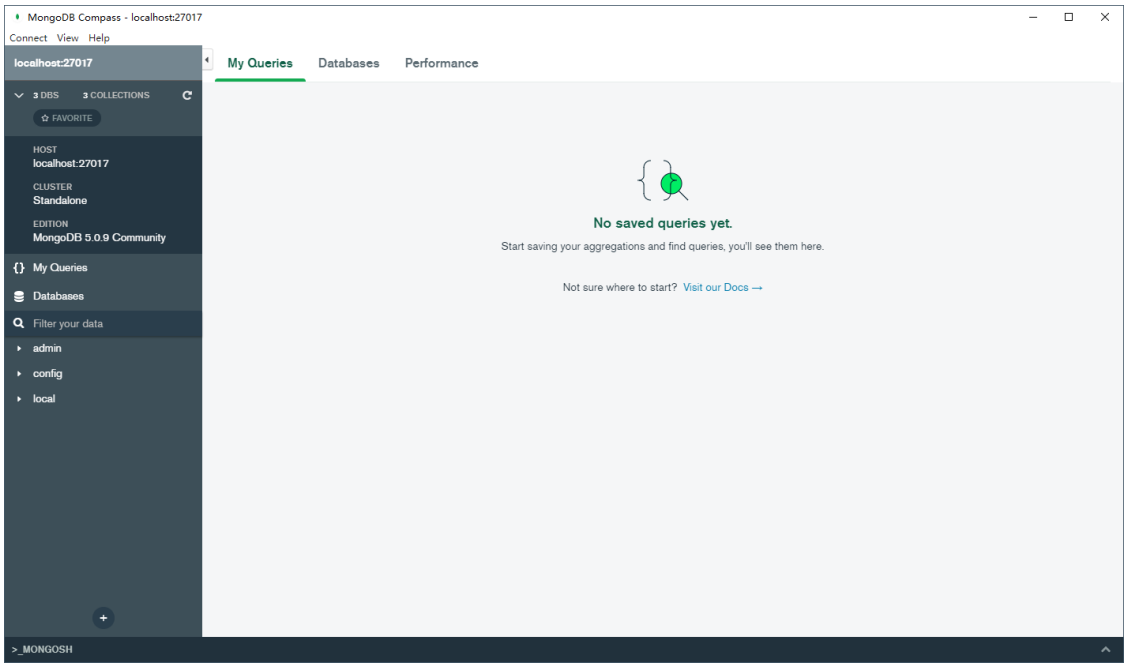
下载完成后双击安装文件，启动安装向导。安装程序会自动完成安装，然后打开欢迎界面。



关闭欢迎界面，进入连接界面。



如果是连接本地 MongoDB 服务器，直接点击“Connect”按钮即可。



点击左侧的“Databases”节点，可以看到默认创建的三个数据库：admin、config 以及 local。

至此，我们已经完成了 MongoDB 服务器和 Compass 工具的安装，可以正式开始学习了。

第 3 篇 MongoDB 基本概念

本篇将会介绍 MongoDB 中的一些基本概念，例如文档、集合、数据库以及命名空间等。

3.1 数据格式

在 MongoDB 中，数据使用 JSON 或者 BSON 格式进行处理和存储。

JSON

JSON 全称为 JavaScript Object Notation，是一种轻量级的数据交换格式。JSON 语法基于 JavaScript ECMA-262 3rd edition 的一个子集。

一个 JSON 文档就是一个由字段和值组成的集合，遵循特定的格式。例如：

```
{
   "first_name": "John",
   "last_name": "Doe",
   "age": 22,
   "skills": ["Programming","Databases", "API"]
}
```

BSON

BSON 代表 Binary JSON，它是一种类 JSON 文档的二进制序列化。BSON 更注重存储和处理的效率。

3.2 文档

MongoDB 以 BSON 文档的形式存储数据记录，简称为文档（Document）。

文档是一个由字段-值组成的集合，结构如下：

```
{
   field_name1: value1,
   field_name2: value2,
   field_name3: value3,
   ...
}
```

以上语法中，字段名是字符串，字段值可以是数字、字符串、对象、数组等。例如：

```
{
    _id: ObjectId("5f339953491024badf1138ec"),
    title: "SQL 编程思想",
    isbn: "9787121421402",
    publisher: "电子工业出版社",
    published_date: new Date('2021-10-01'),
    author: { first_name: "旭阳", last_name: "董"}
}
```

该文档包含以下字段-值：

- _id 是一个 ObjectId；
- title 是一个字符串；
- isbn 是一个字符串；
- publisher 是一个字符串；
- published_date 是一个 Date 类型数据；
- author 是一个嵌入式文档，包含了两个字段：first_name 和 last_name。

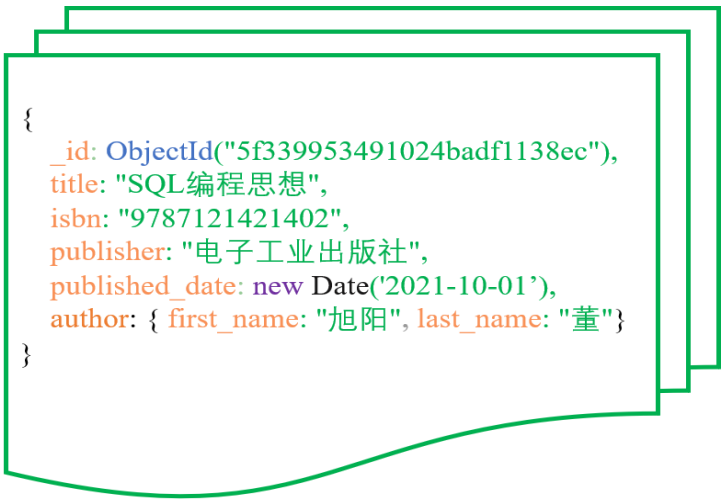
如果你了解关系型数据，可以看出文档就像是表中的一行记录，但是它比数据行更具有表达性。

文档中的字段名需要遵循以下规则：

- MongoDB 保留字段_id 用于唯一标识一个文档；
- 字段名不能包含 null 字符；
- 最外层的字段名不能进以$符号开始。

3.3 集合

MongoDB 使用集合（Colletion）存储文档。一个集合就是一组文档。



MongoDB 中的集合类似于关系型数据库中的表。

MongoDB	RDBMS
文档	行
集合	表

表拥有固定的模式（字段定义），但是集合的模式是动态的。动态模式意味着集合中的多个文档结构可能完全不同。例如，以下文档可以存储在同一个集合中：

```
{
    title: "SQL 编程思想",
    published_date: new Date('2021-10-01')
}

{
    title: "MongoDB 从入门到商业实战",
    published_date: new Date('2019-09-01'),
    isbn": "9787121372247"
}
```

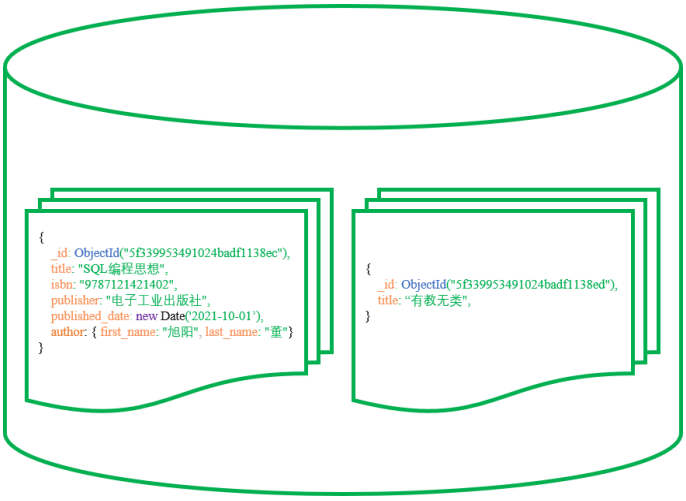
第 2 个文档比第 1 个文档多个一个字段。理论上来说，每个文档都可以有不同的字段。

一个集合拥有一个名字，例如 books。集合名词不能包含以下内容：

- $字符；
- null（\0）字符；
- 空字符串；
- 以 system 开头，因为 system*是 MongoDB 内部集合的保留名。

3.4 数据库

MongoDB 集合存储在数据库中。单个 MongoDB 实例可以包含多个数据库（Database）。



数据库可以通过名字进行引用，例如 bookdb。数据库名不能为以下内容：

- 一个空字符串；
- 包含以下字符：/、 、.、“、*、<、>、:、|、?、$、空格或者\0（空字符）；
- 长度超过 64 字节。

MongoDB 还包含一些保留的数据库名称，例如 admin、local 以及 config，我们不能使用这些名称创建新的数据库。

3.5 命名空间

命名空间（Namespace）由数据库名加上其中的集合名组成。命名空间可以用于完整引用一个集合。

例如，对于数据库 bookdb 中的集合 books，命名空间为 bookdb.books。

第 4 篇 MongoDB 客户端

本篇我们介绍 MongoDB 客户端工具 mongo 的使用。

4.1 mongo shell

mongo shell 是一个用于连接 MongoDB 的交互式 JavaScript 接口。mongo shell 可以用于操作 MongoDB 中的数据，也可以执行一些管理任务。

mongo shell 类似于 MySQL 数据库客户端 mysql，PostgreSQL 客户端 psql，或者 Oracle 数据库中的 SQL*Plus 工具。

> 注意，MongoDB v5.0 开始默认弃用了随着 MongoDB 一起安装的 mongo shell，推荐使用新的 mongsh。

在使用 mongo shell 之前，我们需要下载并安装该工具。安装完成之后，在命令行中输入以下命令：

```
mongosh
```

mongo shell 会自动连接到本地（localhost）默认端口（27017）上的 MongoDB 服务。

mongo shell 即可以作为一个功能完备的 JavaScript 解释器，也可以作为一个 MongoDB 客户端工具。

JavaScript 解释器

mongo shell 是一个功能完备的 JavaScript 解释器，所以我们可以用它执行 JavaScript 代码。例如：

```
> Math.max(10,20,30);
30
```

mongo shell 允许输入跨行命令，当我们输入回车时它会检测 JavaScript 语句是否完整。如果语句不完整，我们可以在下一行继续输入：

```
> function add(a, b) {
... return a + b;
... }
> add(10,20);
30
```

输入 console.clear()命令可以清空屏幕：

```
console.clear()
```

MongoDB 客户端

mongo shell 是一个 MongoDB 客户端。默认情况下，它会连接本地 MongoDB 服务中的 test 数据库，并且将数据库连接设置为全局变量 db。

db 变量可以用于查看当前数据库：

```
> db
test
```

除了 JavaScript 语法之外，mongosh 还提供了很多方便我们与 MongoDB 数据库服务器交换的命令。例如，shows dbs 命令可以列出服务中的全部数据库：

```
test> show dbs
admin        41 kB
config     73.7 kB
local      81.9 kB
```

以上输出结果显示了 3 个数据库。

如果想要切换当前数据库，可以使用 use 命令。例如，以下命令可以将当前数据库切换为 bookdb 数据库：

```
test> use bookdb
switched to db bookdb
```

注意，我们可以切换到一个不存在的数据库。此时，当我们第一次保存数据时，MongoDB 会自动创建这个数据库。

执行以上命令之后，变量 db 的值为 bookdb：

```
> db
bookdb
```

此时，我们可以通过变量 db 访问数据库 bookdb 中的集合 books：

```
> db.books
bookdb.books
```

4.2 基本的 CRUD 操作

下面我们演示一下如何创建（Create）、读取（Read）、更新（Update）以及删除（Delete）文档。这些操作也被称为 CRUD。

本篇只涉及简单的 CRUD 操作，我们将会在后续教程中学习详细 CRUD 操作。

创建文档

如果想要在集合中创建一个新的文档，可以使用 insertOne()方法。

以下命令为集合 books 增加了一个新的文档（一本新书）：

```
db.books.insertOne({
   title: "SQL 编程思想",
   published_year: 2021
})
```

输出结果如下：

```
{
        "acknowledged" : true,
        "insertedId" : ObjectId("62bb0962874071c64b1f9b7b")
}
```

当我们输入回车时，mongo shell 会将命令发送到 MongoDB 服务器。如果命令有效，MongoDB 会插入文档并返回结果。

以上示例中，返回对象包含两个键：acknowledged 和 insertedId。insertedId 的值就是文档的_id 字段。

如果我们增加文档时没有指定_id 字段，MongoDB 会自动为文档指定一个唯一的 ObjectId 作为_id 字段的值。

MongoDB 使用_id 字段唯一标识集合中的文档。

查找文档

如果想要查找集合中的文档，可以使用 findOne()方法。例如：

```
db.books.findOne()
```

输出结果如下；

```
{
  _id: ObjectId("62bb0962874071c64b1f9b7b"),
  title: 'SQL 编程思想',
  published_year: 2021
}
```

pretty()方法可以将输出结果进行格式化显示，例如：

```
db.books.find().pretty()
{
        "_id" : ObjectId("62bb0962874071c64b1f9b7b"),
        "title" : "SQL 编程思想",
        "published_year" : 2021
}
```

更新文档

如果想要更新某个文档中的内容，可以使用 updateOne()方法。该方法至少需要提供两个参数：

- 第一个参数指定了需要更新的文档。

- 第二个参数指定了更新操作的内容。

以下示例更新了标题为“SQL 编程思想”的文档的 published_year 字段：

```
db.books.updateOne(
    { title: "SQL 编程思想"},
    { $set: { published_year: 2022 }}
)
```

其中，第一个参数表示更新标题为“SQL 编程思想”的第一个文档。第二个参数使用$set 操作符更新字段 published_year 的值。返回结果如下：

```
{
  acknowledged: true,
  insertedId: null,
  matchedCount: 1,
  modifiedCount: 1,
  upsertedCount: 0
}
```

删除文档

如果想要删除集合中的某个文档，可以使用 deleteOne()方法。该方法包含一个参数，用于指定要删除的文档。

以下示例使用 deleteOne()方法删除集合 books 中标题为“SQL 编程思想”的第一个文档：

```
db.books.deleteOne({title: "SQL 编程思想"});
输出结果如下：
{
    "acknowledged": true,
    "deletedCount": 1
}
```

返回结果中的 deletedCount 表示成功删除了一个文档。

如果想要查看当前数据库中的所有集合，可以使用 show collections 命令：

```
show collections

books
```

数据库 bookdb 中目前只包含一个集合。

第 5 篇 MongoDB 数据类型

本篇将会介绍 MongoDB 中最常用的一些数据类型。

5.1 空类型

null 类型用于表示空值（null）和不存在的字段值。例如：

```
{
   "isbn": null
}
```

5.2 布尔类型

boolean 类型包含两个值：true 和 false。例如：

```
{
   "best_seller": true
}
```

5.3 数字类型

默认情况下，mongo shell 使用 64 位浮点数。例如：

```
{
    "price": 9.95,
    "pages": 851
}
```

NumberInt 和 NumberLong 类分别表示 4 字节和 8 字节整数。例如：

```
{
   "year": NumberInt("2020"),
   "words": NumberLong("95403")
}
```

5.4 字符串类型

string 类型表示 UTF-8 字符组成的字符串。例如：

```
{
   "title": "MongDB Data Types"
}
```

5.5 日期类型

date 类型以 8 字节整数存储了 Unix 纪元（1970 年 1 月 1 日）以来的毫秒数，不包含时区。例如：

```
{
```

```
    "updated_at":  new Date()
}
```

在 JavaScript 中，Date 类用于表示 MongoDB 中的日期类型。

注意，创建 Date 对象时需要调用 new Date()，而不仅仅是 Date()，因为 Date() 返回的是日期字符串而不是 Date 对象。

mongo shell 使用本地时区设置显示日期，但是 MongoDB 不会存储日期中的时区信息。如果想要存储时区，可以使用额外的字段，例如 timezone。

5.6 正则表达式

MongoDB 支持存储 JavaScript 正则表达式（regular expression），例如：

```
{
    "pattern": /\d+/
}
```

示例中的//是一个正则表达式，用于匹配一个或多个十进制数字。

5.7 数组类型

array 类型可以存储由任何类型的数据组成的列表。列表中每个值得类型可以不同，例如：

```
{
    "title": "MongoDB Array",
    "reviews": ["John", 3.5, "Jane", 5]
}
```

MongoDB 可以理解文档中得数组结构，支持基于数组元素的操作。例如，我们可以查找 reviews 数组中包含元素值为 5 所有文档。另外，我们还可以基于 reviews 数组创建一个索引，提高查询的性能。

5.8 嵌入式文档

文档中的字段可以是另一个文档，即嵌入式文档（embedded document）。

以下是一个 book 文档，包含了一个嵌入的 author 文档：

```
{
    "title": "MongoDB 入门教程",
    "pages": 200,
    "author": {
       "first_name": "John",
       "last_name": "Who"
    }
}
```

示例中的 author 文档拥有它自己的键值对，包括 first_name 和 last_name。

5.9 对象 ID

在 MongoDB 中，每个文档都有一个“_id”键。“_id”键的值可以是任何类型，但是它默认为一个 ObjectId。

同一个集合中，每个“_id”键的值必须唯一，MongoDB 使用它标识集合中的每个文档。

ObjectId 类是“_id”键的默认类型，它可以产生跨服务器的全局唯一值。因为 MongoDB 原生支持分布式，在共享环境中确保标识符唯一很重要。

ObjectId 占用 12 字节存储，每个字节以 2 个十六进制数字表示，因此 ObjectId 包含 24 个十六进制数字。

12 字节 ObjectId 值包含以下内容：

- 4 字节的时间戳，表示 ObjectId 的产生时间，数值为 Unix 纪元以来的秒数。
- 5 字节的随机值。
- 3 字节的增量计数器，使用随机数进行初始化。

ObjectId 前面 9 字节可以确保跨服务器和进程每秒钟的唯一性，最后 3 个字节可以确保每个进程每秒钟内的唯一性。因此，单个进程每秒钟可以产生 256^3（16,777,216）个唯一 ObjectId。

如果我们插入文档时没有指定“_id”键的值，MongoDB 会自动为文档生成一个唯一 id。例如：

```
db.books.insertOne({
   "title": "SQL 编程思想"
});

{
        "acknowledged" : true,
        "insertedId" : ObjectId("62bb0962874071c64b1f9b7b")
}
```

MongoDB 生成了一个值为 ObjectId(“62bb0962874071c64b1f9b7b”)的唯一 id。我们可以查看插入的文档：

```
db.books.find().pretty()

{
        "_id" : ObjectId("62bb0962874071c64b1f9b7b"),
        "title" : "SQL 编程思想",
        "published_year" : 2021
}
```

第 6 篇 CRUD 之创建文档

本篇开始将会介绍 MongoDB 中的基本 CRUD 操作，首先我们来学习一下如何创建文档。

6.1 使用 insertOne()方法创建单个文档

集合的 insertOne()方法可以用于创建单个文档。该方法的语法如下：

```
db.collection.insertOne(
   <document>,
   { writeConcern: <document>}
)
```

insertOne()方法包含两个参数：

- document 是想要创建的文档内容，这是一个必选参数。
- writeConcern 是一个可选参数，用于指定插入操作结果的确认级别。我们会在后续教程中详细讨论该选项。

insertOne()方法的返回结果包含以下字段信息：

- acknowledged，一个布尔值。如果插入文档时指定了安全写入（writeConcern），该字段的值为 true；如果禁用了安全写入机制，该字段的值为 false。
- insertedId，存储了文档的_id 字段的值。

注意，如果集合不存在，insertOne()方法同时会创建该集合并插入文档。

如果插入文档时没有指定_id 字段，MongoDB 会自动增加该字段并产生一个唯一的 ObjectId。如果显式指定了_id 字段的值，需要确保数据在集合内的唯一性。否则，MongoDB 将会返回一个重复键错误。

接下来我们看几个示例。

以下示例使用 insertOne()方法为集合 books 创建了一个新的文档，插入文档时没有指定_id 字段：

```
db.books.insertOne({
    title: 'MongoDB insertOne',
    year: '2022'
});
```

输出结果如下：

```
{
        "acknowledged" : true,
```

```
            "insertedId" : ObjectId("621489fcf514a446bf1a98ea")
    }
```

我们可以使用 find()方法查询插入的文档：

```
    db.books.find()
```

输出结果如下：

```
    [
      {
        _id: ObjectId("621489fcf514a446bf1a98ea"),
        title: 'MongoDB insertOne',
        year: '2022'
      }
    ]
```

接下来的示例在插入文档时指定了自定义的_id 字段值：

```
    db.books.insertOne({
       _id: 1,
       title: "Mastering Big Data",
       year: "2022"
    });
```

输出结果如下：

```
    { "acknowledged" : true, "insertedId" : 1 }
```

如果此时我们再插入一个文档，而且它的_id 字段值也等于 1：

```
    db.books.insertOne({
       _id: 1,
       title: "MongoDB for JS Developers",
       year: "2021"
    });
```

由于集合 books 中的_id:1 已经存在，MongoDB 将会返回以下错误：

```
    WriteError({
            "index" : 0,
            "code" : 11000,
            "errmsg" : "E11000 duplicate key error collection: bookstore.books
index: _id_ dup key: { _id: 1.0 }",
            "op" : {
                    "_id" : 1,
                    "title" : "MongoDB for JS Developers",
                    "year" : "2021"
            }
    })
```

6.2 使用 insertMany()方法创建多个文档

insertMany()方法可以一次插入多个文档，该方法的语法如下：

```
db.collection.insertMany(
   [document1, document2, ...],
   {
      writeConcern: <document>,
      ordered: <boolean>
   }
)
```

insertMany()方法包含两个参数。第一个参数是一个文档数组，代表了需要插入的文档。第二个参数是一个文档，包含了两个可选的字段。其中 writeConcern 指定写入操作的安全确认级别，ordered 用于指定是否需要按照顺序写入文档。

如果 ordered 设置为 true，MongoDB 将会按照第一个参数中的顺序写入文档，这是默认值。如果 ordered 设置为 false，MongoDB 可能会为了提高性能重新组织文档的顺序。

insertMany()方法的返回结果包含了以下信息：

- 如果插入文档时指定了安全写入（writeConcern），acknowledged 字段的值为 true；如果禁用了安全写入机制，该字段的值为 false。
- 一个由文档_id 字段的值组成的数组，代表了成功插入的文档。

注意，如果集合不存在，insertMany()方法会创建该集合并插入文档。不过，只有插入操作成功时才会创建集合。

如果插入文档时没有指定_id 字段，MongoDB 会自动增加该字段并产生一个唯一的 ObjectId。如果显式指定了_id 字段的值，需要确保数据在集合内的唯一性。否则，MongoDB 将会返回一个重复键错误。

如果产生错误，insertMany()会抛出一个 BulkWriteError 异常。此时，顺序插入操作会停止，但是无顺序插入操作会继续处理其他的文档。

以下语句使用 insertMany()方法一次插入多个文档，没有指定_id 字段：

```
db.books.insertMany([
   { title:  "NoSQL Distilled", year: "2020"},
   { title:  "NoSQL in 7 Days", year: "2019"},
   { title:  "NoSQL Database", year: "2018"},
]);
```

输出结果如下：

```
{
  acknowledged: true,
  insertedIds: {
```

```
      '0': ObjectId("62148d16f514a446bf1a98f1"),
      '1': ObjectId("62148d16f514a446bf1a98f2"),
      '2': ObjectId("62148d16f514a446bf1a98f3")
    }
  }
```

以下语句使用自定义的_id 字段插入多个文档：

```
  db.books.insertMany([
     { _id: 1, title:  "SQL Basics", year: "2021"},
     { _id: 2, title:  "SQL Advanced", year: "2022"}
  ]);
```

输出结果如下：

```
  { acknowledged: true, insertedIds: { '0': 1, '1': 2 } }
```

下面的语句插入了已经存在的_id 字段：

```
  db.books.insertMany([
     { _id: 2, title:  "SQL Performance Tuning", year: "2020"},
     { _id: 3, title:  "SQL Index", year: "2020"}
  ]);
```

由于_id:2 已经存在，MongoDB 返回了以下错误：

```
  Uncaught:
  MongoBulkWriteError: E11000 duplicate key error collection: bookdb.books
index: _id_ dup key: { _id: 2 }
  Result: BulkWriteResult {
    result: {
      ok: 1,
      writeErrors: [
        WriteError {
          err: {
            index: 0,
            code: 11000,
            errmsg: 'E11000 duplicate key error collection: bookdb.books index:
_id_ dup key: { _id: 2 }',
            errInfo: undefined,
            op: {
              _id: 2,
              title: 'SQL Performance Tuning',
              year: '2020'
            }
          }
        }
      ],
      writeConcernErrors: [],
      insertedIds: [ { index: 0, _id: 2 }, { index: 1, _id: 3 } ],
      nInserted: 0,
      nUpserted: 0,
      nMatched: 0,
```

```
    nModified: 0,
    nRemoved: 0,
    upserted: []
  }
}
```

接下来的示例使用了无序插入方式：

```
db.books.insertMany(
   [{ _id: 3, title:  "SQL Performance Tuning", year: "2020"},
   { _id: 3, title:  "SQL Trees", year: "2019"},
   { _id: 4, title:  "SQL Graph", year: "2021"},
   { _id: 5, title:  "NoSQL Pros", year: "2022"}],
   { ordered: false }
);
```

以上示例中的_id:3 是重复值，MongoDB 返回了错误。不过，因为使用了无序插入，_id 为 4 和 5 的文档仍然能够插入集合中。以下语句返回了所有的文档：

```
db.books.find()

[
  {
    _id: ObjectId("62148d16f514a446bf1a98f1"),
    title: 'NoSQL Distilled',
    isbn: '0-4696-7030-4'
  },
  {
    _id: ObjectId("62148d16f514a446bf1a98f2"),
    title: 'NoSQL in 7 Days',
    isbn: '0-4086-6859-8'
  },
  {
    _id: ObjectId("62148d16f514a446bf1a98f3"),
    title: 'NoSQL Database',
    isbn: '0-2504-6932-4'
  },
  { _id: 1, title: 'SQL Basics', year: '2021' },
  { _id: 2, title: 'SQL Advanced', year: '2022' },
  { _id: 3, title: 'SQL Performance Tuning', year: '2020' },
  { _id: 4, title: 'SQL Graph', year: '2021' },
  { _id: 5, title: 'NoSQL Pros', year: '2022' }
]
```

第 7 篇 CRUD 之查找文档

本篇将会介绍如何利用集合的 findOne()和 find()方法查找文档。

7.1 使用 findOne()方法查找单个文档

findOne()方法用于返回集合中满足条件的单个文档，该方法的语法如下：

```
db.collection.findOne(query, projection)
```

findOne()方法包含两个可选的参数：

- query 用于指定一个选择标准；
- projection 用于指定返回的字段。

如果省略 query 参数，findOne()返回磁盘中存储的第一个文档。如果省略 projection 参数，默认返回文档中的全部字段。

如果想要指定是否返回某个字段，可以使用以下格式指定 projection 参数：

```
{field1: value, field1: value, ... }
```

其中，value 设置为 true 或者 1 表示返回该字段；如果 value 设置为 false 或者 0，MongoDB 不会返回该字段。

默认情况下，MongoDB 总是返回_id 字段。如果不需要返回该字段，可以在 projection 参数中明确指定_id:0。

如果存在多个满足查询条件的文档，findOne()只会返回磁盘中找到的第一个文档。接下来我们看几个示例，首先创建一个集合 products：

```
db.products.insertMany([
    { "_id" : 1, "name" : "xPhone", "price" : 799, "releaseDate":
ISODate("2011-05-14"), "spec" : { "ram" : 4, "screen" : 6.5, "cpu" :
2.66 },"color":["white","black"],"storage":[64,128,256]},
    { "_id" : 2, "name" : "xTablet", "price" : 899, "releaseDate":
ISODate("2011-09-01") , "spec" : { "ram" : 16, "screen" : 9.5, "cpu" :
3.66 },"color":["white","black","purple"],"storage":[128,256,512]},
    { "_id" : 3, "name" : "SmartTablet", "price" : 899, "releaseDate":
ISODate("2015-01-14"), "spec" : { "ram" : 12, "screen" : 9.7, "cpu" :
3.66 },"color":["blue"],"storage":[16,64,128]},
    { "_id" : 4, "name" : "SmartPad", "price" : 699, "releaseDate":
ISODate("2020-05-14"),"spec" : { "ram" : 8, "screen" : 9.7, "cpu" :
1.66 },"color":["white","orange","gold","gray"],"storage":[128,256,1024]},
    { "_id" : 5, "name" : "SmartPhone", "price" : 599,"releaseDate":
ISODate("2022-09-14"), "spec" : { "ram" : 4, "screen" : 5.7, "cpu" :
1.66 },"color":["white","orange","gold","gray"],"storage":[128,256]}
  ])
```

以下示例使用 findOne()方法查找集合 products 中的第一个文档：

```
db.products.findOne()
```

查询返回了文档中的全部字段：

```
{
  _id: 1,
  name: 'xPhone',
  price: 799,
  releaseDate: ISODate("2011-05-14T00:00:00.000Z"),
  spec: { ram: 4, screen: 6.5, cpu: 2.66 },
  color: [ 'white', 'black' ],
  storage: [ 64, 128, 256 ]
}
```

省略 query 参数和指定一个空文档参数的结果相同：

```
db.products.findOne({})
```

以下语句使用 findOne()方法查找_id 等于 2 的文档：

```
 db.products.findOne({_id:2})
```

返回结果如下：

```
{
  _id: 2,
  name: 'xTablet',
  price: 899,
  releaseDate: ISODate("2011-09-01T00:00:00.000Z"),
  spec: { ram: 16, screen: 9.5, cpu: 3.66 },
  color: [ 'white', 'black', 'purple' ],
  storage: [ 128, 256, 512 ]
}
```

以下示例使用 findOne()方法查找_id 等于 5 的文档，并且只返回了_id 和 name 字段：

```
db.products.findOne({_id: 5}, {name: 1})

{ "_id" : 5, "name" : "SmartPhone" }
```

从返回结果可以看出，MongoDB 默认返回了_id 字段。

如果想要从返回结果中去掉_id 字段，可以明确指定_id:0，例如：

```
db.products.findOne({ _id: 5 }, {name: 1, _id: 0})

{ "name" : "SmartPhone" }
```

7.2 使用 find()方法查找文档

find()方法用于查找满足指定条件的文档，并且返回一个指向这些文档的游标（指针）。该方法的语法如下：

```
db.collection.find(query, projection)
```

find()方法包含两个可选的参数：

- query 用于指定一个选择标准。如果省略该参数或者指定一个空文档参数，将会返回集合中的全部文档。
- projection 用于指定返回的字段。如果省略该参数，将会返回文档中的全部字段。

默认情况下，MongoDB 总是返回_id 字段。如果不需要返回该字段，可以在 projection 参数中明确指定_id:0。

由于 mongo shell 自动遍历 find()方法返回的游标，我们不需要执行额外的操作就可以获取游标中的文档。默认情况下，mongo shell 只显示前 20 篇文档，输入 it 命令可以显示更多文档。

下面我们使用一个新的集合 books 作为演示：

```
db.books.insertMany([
    { "_id" : 1, "title" : "Unlocking Android", "isbn" : "1933988673",
"categories" : [ "Open Source", "Mobile" ] },
    { "_id" : 2, "title" : "Android in Action, Second Edition", "isbn" :
"1935182722", "categories" : [ "Java" ] },
    { "_id" : 3, "title" : "Specification by Example", "isbn" : "1617290084",
"categories" : [ "Software Engineering" ] },
    { "_id" : 4, "title" : "Flex 3 in Action", "isbn" : "1933988746",
"categories" : [ "Internet" ] },
    { "_id" : 5, "title" : "Flex 4 in Action", "isbn" : "1935182420",
"categories" : [ "Internet" ] },
    { "_id" : 6, "title" : "Collective Intelligence in Action", "isbn" :
"1933988312", "categories" : [ "Internet" ] },
    { "_id" : 7, "title" : "Zend Framework in Action", "isbn" : "1933988320",
"categories" : [ "Web Development" ] },
    { "_id" : 8, "title" : "Flex on Java", "isbn" : "1933988797",
"categories" : [ "Internet" ] },
    { "_id" : 9, "title" : "Griffon in Action", "isbn" : "1935182234",
"categories" : [ "Java" ] },
    { "_id" : 10, "title" : "OSGi in Depth", "isbn" : "193518217X",
"categories" : [ "Java" ] },
    { "_id" : 11, "title" : "Flexible Rails", "isbn" : "1933988509",
"categories" : [ "Web Development" ] },
    { "_id" : 13, "title" : "Hello! Flex 4", "isbn" : "1933988762",
"categories" : [ "Internet" ] },
    { "_id" : 14, "title" : "Coffeehouse", "isbn" : "1884777384",
"categories" : [ "Miscellaneous" ] },
```

```
       { "_id" : 15, "title" : "Team Foundation Server 2008 in Action", "isbn" :
"1933988592", "categories" : [ "Microsoft .NET" ] },
       { "_id" : 16, "title" : "Brownfield Application Development in .NET",
"isbn" : "1933988711", "categories" : [ "Microsoft" ] },
       { "_id" : 17, "title" : "MongoDB in Action", "isbn" : "1935182870",
"categories" : [ "Next Generation Databases" ] },
       { "_id" : 18, "title" : "Distributed Application Development with
PowerBuilder 6.0", "isbn" : "1884777686", "categories" : [ "PowerBuilder" ] },
       { "_id" : 19, "title" : "Jaguar Development with PowerBuilder 7",
"isbn" : "1884777864", "categories" : [ "PowerBuilder", "Client-Server" ] },
       { "_id" : 20, "title" : "Taming Jaguar", "isbn" : "1884777686",
"categories" : [ "PowerBuilder" ] },
       { "_id" : 21, "title" : "3D User Interfaces with Java 3D", "isbn" :
"1884777902", "categories" : [ "Java", "Computer Graphics" ] },
       { "_id" : 22, "title" : "Hibernate in Action", "isbn" : "193239415X",
"categories" : [ "Java" ] },
       { "_id" : 23, "title" : "Hibernate in Action (Chinese Edition)",
"categories" : [ "Java" ] },
       { "_id" : 24, "title" : "Java Persistence with Hibernate", "isbn" :
"1932394885", "categories" : [ "Java" ] },
       { "_id" : 25, "title" : "JSTL in Action", "isbn" : "1930110529",
"categories" : [ "Internet" ] },
       { "_id" : 26, "title" : "iBATIS in Action", "isbn" : "1932394826",
"categories" : [ "Web Development" ] },
       { "_id" : 27, "title" : "Designing Hard Software", "isbn" : "133046192",
"categories" : [ "Object-Oriented Programming", "S" ] },
       { "_id" : 28, "title" : "Hibernate Search in Action", "isbn" :
"1933988649", "categories" : [ "Java" ] },
       { "_id" : 29, "title" : "jQuery in Action", "isbn" : "1933988355",
"categories" : [ "Web Development" ] },
       { "_id" : 30, "title" : "jQuery in Action, Second Edition", "isbn" :
"1935182323", "categories" : [ "Java" ] }
   ]);
```

以下示例使用不带参数的 find()方法返回集合 books 中的全部文档：

```
   db.books.find()
```

mongo shell 显示了前 20 篇文档，包含了全部字段。输入 it 命令并回车，将会看到后续 20 篇文档。

以下示例返回了_id 等于 10 的文档，包含了文档的全部字段：

```
   db.books.find({_id: 10})

   [
     {
       _id: 10,
       title: 'OSGi in Depth',
       isbn: '193518217X',
       categories: [ 'Java' ]
     }
```

```
]
```

以下示例返回了 category 等于“Java”的所有文档，返回结果中包含了_id、title 以及 isbn 三个字段：

```
db.books.find({ categories: 'Java'}, { title: 1,isbn: 1})

[
  {
    _id: 2,
    title: 'Android in Action, Second Edition',
    isbn: '1935182722'
  },
  { _id: 9, title: 'Griffon in Action', isbn: '1935182234' },
  { _id: 10, title: 'OSGi in Depth', isbn: '193518217X' },
  {
    _id: 21,
    title: '3D User Interfaces with Java 3D',
    isbn: '1884777902'
  },
  { _id: 22, title: 'Hibernate in Action', isbn: '193239415X' },
  { _id: 23, title: 'Hibernate in Action (Chinese Edition)' },
  {
    _id: 24,
    title: 'Java Persistence with Hibernate',
    isbn: '1932394885'
  },
  { _id: 28, title: 'Hibernate Search in Action', isbn: '1933988649' },
  {
    _id: 30,
    title: 'jQuery in Action, Second Edition',
    isbn: '1935182323'
  }
]
```

7.3 利用投影操作返回指定字段

在 MongoDB 中，投影（projection）表示在查询中返回指定的字段。

默认情况下 find()和 findOne()方法会返回文档中的全部字段，但是大多数情况下我们不需要查询全部字段。

如果想要选择返回某些字段，可以在一个文档中指定这些字段并将该文档作为参数传递给 find()和 findOne()方法。该参数文档被称为投影文档。指定返回字段的语法如下：

```
{ <field>: value, ...}
```

如果 value 设置为 1 或者 true，表示返回字段；如果 value 设置为 0 或者 false，表示不返回该字段。如果投影文档为空（{}），表示返回全部字段。

对于嵌入式文档中的字段，可以使用点号指定：

```
{ "<embeddedDocument>.<field>": value, ... }
```

以下示例返回了_id 等于 1 的文档中的 name、price 以及_id 字段，同时还返回了嵌入式文档 spec 中的 screen 字段：

```
db.products.find({_id:1}, {
    name: 1,
    price: 1,
    "spec.screen": 1
})
```

输出结果如下：

```
[ { _id: 1, name: 'xPhone', price: 799, spec: { screen: 6.5 } } ]
```

MongoDB 4.4 以及更高版本还支持嵌套形式指定返回的字段：

```
db.products.find({_id:1}, {
    name: 1,
    price: 1,
    spec : { screen: 1 }
})
```

与此类似，对于数组中的字段，也可以使用点号指定：

```
{ "<arrayField>.field": value, ...}
```

以下示例返回了_id、name 以及数组 inventory 中的 qty 字段：

```
db.products.find({}, {
    name: 1,
    "inventory.qty": 1
});
```

输出结果如下：

```
[
  { _id: 1, name: 'xPhone', inventory: [ { qty: 1200 } ] },
  { _id: 2, name: 'xTablet', inventory: [ { qty: 300 } ] },
  {
    _id: 3, name: 'SmartTablet', inventory: [ { qty: 400 }, { qty: 200 } ]
  },
  { _id: 4, name: 'SmartPad', inventory: [ { qty: 1200 } ] },
  { _id: 5, name: 'SmartPhone' }
]
```

第 8 篇 比较运算符

本篇将会介绍 MongoDB 中查找文档时常用的一些比较运算符，包括$eq、$gt、$gte、$lt、$lte、$ne、$in 以及$nin。

8.1 $eq 运算符

$eq 运算符用于匹配字段等于（=）指定值的文档。$eq 运算符的语法如下：

```
{ <field>: { $eq: <value> } }
```

以上语法等价于下面的写法：

```
{<field>: <value>}
```

我们创建一个集合 products 作为下文中的演示：

```
db.products.insertMany([
    { "_id" : 1, "name" : "xPhone", "price" : 799, "releaseDate":
ISODate("2011-05-14"), "spec" : { "ram" : 4, "screen" : 6.5, "cpu" :
2.66 },"color":["white","black"],"storage":[64,128,256]},
    { "_id" : 2, "name" : "xTablet", "price" : 899, "releaseDate":
ISODate("2011-09-01") , "spec" : { "ram" : 16, "screen" : 9.5, "cpu" :
3.66 },"color":["white","black","purple"],"storage":[128,256,512]},
    { "_id" : 3, "name" : "SmartTablet", "price" : 899, "releaseDate":
ISODate("2015-01-14"), "spec" : { "ram" : 12, "screen" : 9.7, "cpu" :
3.66 },"color":["blue"],"storage":[16,64,128]},
    { "_id" : 4, "name" : "SmartPad", "price" : 699, "releaseDate":
ISODate("2020-05-14"),"spec" : { "ram" : 8, "screen" : 9.7, "cpu" :
1.66 },"color":["white","orange","gold","gray"],"storage":[128,256,1024]},
    { "_id" : 5, "name" : "SmartPhone", "price" : 599,"releaseDate":
ISODate("2022-09-14"), "spec" : { "ram" : 4, "screen" : 9.7, "cpu" :
1.66 },"color":["white","orange","gold","gray"],"storage":[128,256]}
 ])
```

以下示例使用$eq 运算符查找 products 集合中 price 字段等于 899 的所有文档：

```
db.products.find({
    price: {
        $eq: 899
    }
}, {
    name: 1,
    price: 1
})
```

我们也可以使用以下等价写法：

```
db.products.find({
    price: 899
```

```
}, {
    name: 1,
    price: 1
})
```

以上两个示例的返回结果相同：

```
[
  { _id: 2, name: 'xTablet', price: 899 },
  { _id: 3, name: 'SmartTablet', price: 899 }
]
```

下面的示例使用$eq 运算符查找嵌套文档 spec 中字段 ram 的值等于 4 的文档：

```
db.products.find({
    "spec.ram": {
        $eq: 4
    }
}, {
    name: 1,
    "spec.ram": 1
})
```

查询返回的文档如下：

```
[
  { _id: 1, name: 'xPhone', spec: { ram: 4 } },
  { _id: 5, name: 'SmartPhone', spec: { ram: 4 } }
]
```

接下来的示例使用$eq 运算符查找 products 集合中数组 color 包含“black”元素的文档：

```
db.products.find({
    color: {
        $eq: "black"
    }
}, {
    name: 1,
    color: 1
})
```

查询返回的文档如下：

```
[
  { _id: 1, name: 'xPhone', color: [ 'white', 'black' ] },
  { _id: 2, name: 'xTablet', color: [ 'white', 'black', 'purple' ] }
]
```

8.2 $gt 运算符

$gt 运算符用于匹配字段大于（>）指定值的文档。$gt 运算符的语法如下：

```
{ field: { $gt: value}}
```

以下示例使用$gt 运算符查找集合 products 中 price 大于 699 的文档：

```
db.products.find({
    price: {
        $gt: 699
    }
}, {
    name: 1,
    price: 1
})
```

查询返回的结果如下：

```
[
  { _id: 1, name: 'xPhone', price: 799 },
  { _id: 2, name: 'xTablet', price: 899 },
  { _id: 3, name: 'SmartTablet', price: 899 }
]
```

8.3 $gte 运算符

$gte 运算符用于匹配字段大于等于（>=）指定值的文档。$gte 运算符的语法如下：

```
{field: {$gte: value} }
```

以下示例使用$gte 运算符查找集合 products 中嵌入文档 spec 的字段 screen 大于或者等于 9.5 的文档：

```
db.products.find({
    "spec.screen": {
        $gte: 9.5
    }
}, {
    name: 1,
    "spec.screen": 1
})
```

返回结果如下：

```
[
  { _id: 2, name: 'xTablet', spec: { screen: 9.5 } },
  { _id: 3, name: 'SmartTablet', spec: { screen: 9.7 } },
  { _id: 4, name: 'SmartPad', spec: { screen: 9.7 } },
  { _id: 5, name: 'SmartPhone', spec: { screen: 9.7 } }
]
```

8.4 $lt 运算符

$lt 运算符用于匹配字段小于（<）指定值的文档。$lt 运算符的语法如下：

```
{field: {$lt: value} }
```

以下示例使用$lt 运算符查找集合 products 中数组字段 storage 至少包含一个小于 128 的元素的文档：

```
db.products.find({
    storage: {
        $lt: 128
    }
}, {
    name: 1,
    storage: 1
})
```

查询返回的文档如下：

```
[
  { _id: 1, name: 'xPhone', storage: [ 64, 128, 256 ] },
  { _id: 3, name: 'SmartTablet', storage: [ 16, 64, 128 ] }
]
```

8.5 $lte 运算符

$lte 运算符用于匹配字段小于等于（<=）指定值的文档。$lte 运算符的语法如下：

```
{field: {$lte: value} }
```

以下示例使用$lte 运算符查找集合 products 中发布日期早于或者等于 2015-01-11 的所有文档：

```
db.products.find({
    "releaseDate": {
        $lte: new ISODate('2015-01-01')
    }
}, {
    name: 1,
    releaseDate: 1
});
```

查询返回的文档如下：

```
[
  {
    _id: 1,
    name: 'xPhone',
    releaseDate: ISODate("2011-05-14T00:00:00.000Z")
  },
  {
    _id: 2,
    name: 'xTablet',
    releaseDate: ISODate("2011-09-01T00:00:00.000Z")
```

```
    }
]
```

8.6 $ne 运算符

$ne 运算符匹配字段不等于（<>）指定值的文档。$ne 运算符的语法如下：

```
{ field: {$ne: value}}
```

以下示例使用$ne 运算符查找集合 products 中 price 不等于 899 的文档：

```
db.products.find({
    price: {
        $ne: 899
    }
}, {
    name: 1,
    price: 1
})
```

查询返回的文档如下：

```
[
  { _id: 1, name: 'xPhone', price: 799 },
  { _id: 4, name: 'SmartPad', price: 699 },
  { _id: 5, name: 'SmartPhone', price: 599 },
  { _id: 6, name: 'xWidget' }
]
```

8.7 $in 运算符

$in 运算符匹配字段等于（=）数组中任意值的文档。$in 运算符的语法如下：

```
{ field: { $in: [<value1>, <value2>,...] }}
```

如果 field 只有一个值，$in 运算符匹配该字段等于数组中任意值的文档。如果 field 也是一个数组，$in 运算符匹配该数组包含数组[value1, value2,…]中任意值的文档。数组列表 value1, value2, …可以是一个常量列表或者正则表达式列表。

> 提示：正则表达式是一组定义搜索模式的字符，例如正则表达式//匹配任何数组，包括 1，123，1234 等。

以下示例使用$in 运算符查找 products 集合中 price 字段等于 599 或者 799 的文档：

```
db.products.find({
    price: {
        $in: [699, 799]
    }
}, {
    name: 1,
```

```
    price: 1
})
```

查询返回的文档如下：

```
[
  { _id: 1, name: 'xPhone', price: 799 },
  { _id: 4, name: 'SmartPad', price: 699 }
]
```

8.8 $nin 运算符

$nin 运算符匹配字段不等于（!=）数组中任意值的文档，或者指定字段不存在的文档。$nin 运算符的语法如下：

```
{ field: { $nin: [ <value1>, <value2> ...]} }
```

以下示例使用$nin 运算符查找 products 集合中 price 字段既不等于 599 也不等于 799 的文档：

```
db.products.find({
    price: {
        $nin: [699, 799]
    }
}, {
    name: 1,
    price: 1
})
```

查询返回的文档如下：

```
[
  { _id: 2, name: 'xTablet', price: 899 },
  { _id: 3, name: 'SmartTablet', price: 899 },
  { _id: 5, name: 'SmartPhone', price: 599 }
]
```

以下示例使用$nin 运算符查找 color 数组字段中不包含任何匹配正则表达式/^g+/或者/^w+/的元素的文档：

```
db.products.find({
    color: {
        $nin: [/^g+/, /^w+/]
    }
}, {
    name: 1,
    color: 1
})
```

查询返回的文档如下：

```
[ { _id: 3, name: 'SmartTablet', color: [ 'blue' ] } ]
```

第 9 篇 逻辑运算符

本篇将会介绍 MongoDB 中的逻辑运算符，包括$and、$or、$not 以及$nor。

9.1 $and 运算符

$and 是一个逻辑查询运算符，可以基于一个或多个表达式执行逻辑 AND 操作。

$and 运算符的语法如下：

```
   $and :[{expression1}, {expression2},...]
```

如果所有的表达式结果都为 true，$and 运算符返回 true。

$and 运算符从左至右计算表达式的值，只要有一个表达式为 false，就会结束运算并返回 false。这种实现方法被称为短路运算（short-circuit evaluation）。

另外，当我们使用逗号分隔的表达式列表时，MongoDB 也会执行隐式逻辑 AND 操作：

```
   { field: { expression1, expression2, ... }
```

接下来的示例我们将会使用 products 集合：

```
   db.products.insertMany([
       { "_id" : 1, "name" : "xPhone", "price" : 799, "releaseDate" :
ISODate("2011-05-14T00:00:00Z"), "spec" : { "ram" : 4, "screen" : 6.5, "cpu" :
2.66 }, "color" : [ "white", "black" ], "storage" : [ 64, 128, 256 ] },
       { "_id" : 2, "name" : "xTablet", "price" : 899, "releaseDate" :
ISODate("2011-09-01T00:00:00Z"), "spec" : { "ram" : 16, "screen" : 9.5, "cpu" :
3.66 }, "color" : [ "white", "black", "purple" ], "storage" : [ 128, 256,
512 ] },
       { "_id" : 3, "name" : "SmartTablet", "price" : 899, "releaseDate" :
ISODate("2015-01-14T00:00:00Z"), "spec" : { "ram" : 12, "screen" : 9.7, "cpu" :
3.66 }, "color" : [ "blue" ], "storage" : [ 16, 64, 128 ] },
       { "_id" : 4, "name" : "SmartPad", "price" : 699, "releaseDate" :
ISODate("2020-05-14T00:00:00Z"), "spec" : { "ram" : 8, "screen" : 9.7, "cpu" :
1.66 }, "color" : [ "white", "orange", "gold", "gray" ], "storage" : [ 128, 256,
1024 ] },
       { "_id" : 5, "name" : "SmartPhone", "price" : 599, "releaseDate" :
ISODate("2022-09-14T00:00:00Z"), "spec" : { "ram" : 4, "screen" : 9.7, "cpu" :
1.66 }, "color" : [ "white", "orange", "gold", "gray" ], "storage" : [ 128,
256 ] },
       { "_id" : 6, "name" : "xWidget", "spec" : { "ram" : 64, "screen" : 9.7,
"cpu" : 3.66 }, "color" : [ "black" ], "storage" : [ 1024 ] }
   ])
```

下面的示例使用$and 运算符查找集合 products 中满足以下条件的文档：

- price 字段的值等于 899；
- 并且 field 字段包含“white”或者“black”。

```
db.products.find({
    $and: [{
        price: 899
    }, {
        color: {
            $in: ["white", "black"]
        }
    }]
}, {
    name: 1,
    price: 1,
    color: 1
})
```

查询返回的文档如下：

```
[
  {
    _id: 2,
    name: 'xTablet',
    price: 899,
    color: [ 'white', 'black', 'purple' ]
  }
]
```

以下示例使用$and 运算符查找 price 字段值等 699 并且该字段存在的文档：

```
db.products.find({
    $and: [{
        price: 699
    }, {
        price: {
            $exists: true
        }
    }]
}, {
    name: 1,
    price: 1,
    color: 1
})
```

输出结果如下：

```
[
  {
    _id: 4,
    name: 'SmartPad',
    price: 699,
    color: [ 'white', 'orange', 'gold', 'gray' ]
  }
```

```
]
```

我们也可以使用隐式逻辑 AND 运算符实现相同的效果：

```
db.products.find({
    price: {
        $eq: 699,
        $exists: true
    }
}, {
    name: 1,
    price: 1,
    color: 1
})
```

$exists 运算符用于判断指定字段是否存在。

9.2 $or 运算符

$or 是一个逻辑查询运算符，可以基于一个或多个表达式执行逻辑 OR 操作，返回至少满足一个表达式的文档。

$or 运算符的语法如下：

```
$or:[{expression1}, {expression2},...]
```

下面的示例使用$or 运算符查找集合 products 中价格等于 799 或者 899 的产品：

```
db.products.find({
    $or: [{
        price: 799
    }, {
        price: 899
    }]
}, {
    name: 1,
    price: 1
})
```

查询返回的文档如下：

```
[
  { _id: 1, name: 'xPhone', price: 799 },
  { _id: 2, name: 'xTablet', price: 899 },
  { _id: 3, name: 'SmartTablet', price: 899 }
]
```

以上示例使用了相同的字段进行过滤，因此我们也可以使用$in 运算符：

```
db.products.find({
    price: {
        $in: [799, 899]
```

```
    }
}, {
    name: 1,
    price: 1
})
```

下面的示例使用$or 运算符查找价格小于 699 或者大于 799 的产品：

```
db.products.find({
    $or: [
        { price: {$lt: 699} },
        { price: {$gt: 799} }
    ]
}, {
    name: 1,
    price: 1
})
```

输出结果如下：

```
[
  { _id: 2, name: 'xTablet', price: 899 },
  { _id: 3, name: 'SmartTablet', price: 899 },
  { _id: 5, name: 'SmartPhone', price: 599 }
]
```

9.3 $not 运算符

$not 是一个逻辑查询运算符，可以将一个表达式的结果进行逻辑 NOT 运算，返回不满足条件的文档，包括那些不存在指定字段的文档。

$not 运算符的语法如下：

```
{ field: { $not: { <expression> } } }
```

以下示例使用$not 运算符查找 price 字段小于等于 699，或者不存在 price 字段的文档：

```
db.products.find({
    price: {
        $not: {
            $gt: 699
        }
    }
}, {
    name: 1,
    price: 1
})
```

查询返回的文档如下：

```
[
```

```
  { _id: 4, name: 'SmartPad', price: 699 },
  { _id: 5, name: 'SmartPhone', price: 599 },
  { _id: 6, name: 'xWidget' }
]
```

注意，{ $not: { $gt: 699 } }的逻辑和$lte 运算符不同。{ $lte : 699 }返回的是 price 字段**存在**并且小于等于 699 的文档。

以下示例使用$not 运算符查找 name 字段不匹配正则表达式/^Smart+/的文档：

```
db.products.find({
    name: {
        $not: /^Smart+/
    }
}, {
    name: 1
})
```

正则表达式/^Smart+/匹配以“Smart”开头的字符串。查询返回的文档如下：

```
[
  { _id: 1, name: 'xPhone' },
  { _id: 2, name: 'xTablet' },
  { _id: 6, name: 'xWidget' }
]
```

9.4 $nor 运算符

$nor 是一个逻辑查询运算符，可以基于一个或多个表达式执行逻辑 NOR 运算，返回不满足所有条件的文档。

$nor 运算符的语法如下：

```
{ $nor: [ { <expression1> }, { <expression2> },...] }
```

以下示例使用$nor 运算符查询价格不等于 899，并且颜色不包含“gold”的产品：

```
db.products.find({
    $nor :[
        { price: 899},
        { color: "gold"}
    ]
}, {
    name: 1,
    price: 1,
    color: 1
})
```

与$not 运算符类似，$nor 运算符也会返回指定字段不存在的文档。

查询返回的结果如下：

```
   { "_id" : 1, "name" : "xPhone", "price" : 799, "color" : [ "white",
"black" ] }
   { "_id" : 6, "name" : "xWidget", "color" : [ "black" ] }
```

第 10 篇 元素运算符

本篇将会介绍 MongoDB 中的两个元素查询运算符：$exists 以及$type。

10.1 $exists 运算符

$exists 是一个元素查询运算符，语法如下：

```
{ field: { $exists: <boolean_value> } }
```

如果设置为 true，$exists 运算符将会匹配指定字段存在数值的文档，数值可以是 null。

如果设置为 false，$exists 运算符将会匹配不包含指定字段的文档。

提示：MongoDB 中的$exists 运算符并不等价于 SQL 中的 EXISTS 运算符。从 MongoDB 4.2 开始，$type: 0 不等价于$exists:false。

接下来的示例将会使用以下 products 集合：

```
db.products.insertMany([
    { "_id" : 1, "name" : "xPhone", "price" : 799, "releaseDate" :
ISODate("2011-05-14T00:00:00Z"), "spec" : { "ram" : 4, "screen" : 6.5, "cpu" :
2.66 }, "color" : [ "white", "black" ], "storage" : [ 64, 128, 256 ] },
    { "_id" : 2, "name" : "xTablet", "price" : 899, "releaseDate" :
ISODate("2011-09-01T00:00:00Z"), "spec" : { "ram" : 16, "screen" : 9.5, "cpu" :
3.66 }, "color" : [ "white", "black", "purple" ], "storage" : [ 128, 256,
512 ] },
    { "_id" : 3, "name" : "SmartTablet", "price" : 899, "releaseDate" :
ISODate("2015-01-14T00:00:00Z"), "spec" : { "ram" : 12, "screen" : 9.7, "cpu" :
3.66 }, "color" : [ "blue" ], "storage" : [ 16, 64, 128 ] },
    { "_id" : 4, "name" : "SmartPad", "price" : 699, "releaseDate" :
ISODate("2020-05-14T00:00:00Z"), "spec" : { "ram" : 8, "screen" : 9.7, "cpu" :
1.66 }, "color" : [ "white", "orange", "gold", "gray" ], "storage" : [ 128, 256,
1024 ] },
    { "_id" : 5, "name" : "SmartPhone", "price" : 599, "releaseDate" :
ISODate("2022-09-14T00:00:00Z"), "spec" : { "ram" : 4, "screen" : 9.7, "cpu" :
1.66 }, "color" : [ "white", "orange", "gold", "gray" ], "storage" : [ 128,
256 ] },
    { "_id" : 6, "name" : "xWidget", "spec" : { "ram" : 64, "screen" : 9.7,
"cpu" : 3.66 }, "color" : [ "black" ], "storage" : [ 1024 ] },
    { "_id" : 7, "name" : "xReader","price": null, "spec" : { "ram" : 64,
"screen" : 6.7, "cpu" : 3.66 }, "color" : [ "black", "white" ], "storage" :
[ 128 ] }
])
```

以下示例使用$exists 运算符查找包含 price 字段的文档：

```
db.products.find(
```

```
    {
 price: {
 $exists: true
 }
},
    {
 name: 1,
 price: 1
 }
)
```

查询返回的文档如下：

```
{ "_id" : 1, "name" : "xPhone", "price" : 799 }
{ "_id" : 2, "name" : "xTablet", "price" : 899 }
{ "_id" : 3, "name" : "SmartTablet", "price" : 899 }
{ "_id" : 4, "name" : "SmartPad", "price" : 699 }
{ "_id" : 5, "name" : "SmartPhone", "price" : 599 }
{ "_id" : 7, "name" : "xReader", "price" : null }
```

以下查询使用$exists 运算符查找包含 price 字段并且价格大于 799 的文档：

```
db.products.find({
    price: {
        $exists: true,
        $gt: 699
    }
}, {
    name: 1,
    price: 1
});
```

输出结果如下：

```
{ "_id" : 1, "name" : "xPhone", "price" : 799 }
{ "_id" : 2, "name" : "xTablet", "price" : 899 }
{ "_id" : 3, "name" : "SmartTablet", "price" : 899 }
```

下面的示例使用$exists 运算符查找不包含 price 字段的文档：

```
db.products.find({
    price: {
        $exists: false
    }
}, {
    name: 1,
    price: 1
});
```

查询返回了没有价格的产品：

```
{ "_id" : 6, "name" : "xWidget" }
```

10.2 $type 运算符

有时候我们需要处理非结构化的数据，而它们没有明确的数据类型。此时，我们就需要使用$type 运算符。

$type 是一个元素查询运算符，可以查找字段为指定 BSON 类型的文档。

$type 运算符的语法如下：

```
{ field: { $type: <BSON type> } }
```

$type 运算符也支持 BSON 类型组成的列表参数：

```
{ field: { $type: [ <BSON type1> , <BSON type2>, ... ] } }
```

以上语法中，$type 运算符可以查找字段属于参数列表中任一 BSON 类型的文档。

MongoDB 提供了三种指定 BSON 类型的方法：类型名称、数字编号以及别名。下表列出了这三种方法对应的 BSON 类型：

类型	编号	别名
Double	1	“double”
String	2	“string”
Object	3	“object”
Array	4	“array”
Binary data	5	“binData”
ObjectId	7	“objectId”
Boolean	8	“bool”
Date	9	“date”
Null	10	“null”
Regular Expression	11	“regex”
JavaScript	13	“javascript”
32-bit integer	16	“int”
Timestamp	17	“timestamp”
64-bit integer	18	“long”
Decimal128	19	“decimal”
Min key	-1	“minKey”
Max key	127	“maxKey”

另外，别名“number”可以匹配以下 BSON 类型：

- double

- 32-bit integer
- 64-bit integer
- decimal

下面的示例我们需要使用新的 products 集合：

```
   db.products.insertMany([
       { "_id" : 1, "name" : "xPhone", "price" : "799", "releaseDate" :
ISODate("2011-05-14T00:00:00Z"), "spec" : { "ram" : 4, "screen" : 6.5, "cpu" :
2.66 }, "color" : [ "white", "black" ], "storage" : [ 64, 128, 256 ] },
       { "_id" : 2, "name" : "xTablet", "price" : NumberInt(899),
"releaseDate" : ISODate("2011-09-01T00:00:00Z"), "spec" : { "ram" : 16,
"screen" : 9.5, "cpu" : 3.66 }, "color" : [ "white", "black", "purple" ],
"storage" : [ 128, 256, 512 ] },
       { "_id" : 3, "name" : "SmartTablet", "price" : NumberLong(899),
"releaseDate" : ISODate("2015-01-14T00:00:00Z"), "spec" : { "ram" : 12,
"screen" : 9.7, "cpu" : 3.66 }, "color" : [ "blue" ], "storage" : [ 16, 64,
128 ] },
       { "_id" : 4, "name" : "SmartPad", "price" : [599, 699, 799],
"releaseDate" : ISODate("2020-05-14T00:00:00Z"), "spec" : { "ram" : 8,
"screen" : 9.7, "cpu" : 1.66 }, "color" : [ "white", "orange", "gold", "gray" ],
"storage" : [ 128, 256, 1024 ] },
       { "_id" : 5, "name" : "SmartPhone", "price" : ["599",699],
"releaseDate" : ISODate("2022-09-14T00:00:00Z"), "spec" : { "ram" : 4,
"screen" : 9.7, "cpu" : 1.66 }, "color" : [ "white", "orange", "gold", "gray" ],
"storage" : [ 128, 256 ] },
       { "_id" : 6, "name" : "xWidget", "spec" : { "ram" : 64, "screen" : 9.7,
"cpu" : 3.66 }, "color" : [ "black" ], "storage" : [ 1024 ] }
   ])
```

products 集合中的 price 字段可能是 int、double 或者 long 类型。

以下示例使用$type 运算符查找 price 字段属于字符串类型，或者 price 字段是包含字符串元素的数组的文档：

```
   db.products.find({
       price: {
           $type: "string"
       }
   }, {
       name: 1,
       price: 1
   })
```

查询返回的结果如下：

```
   { "_id" : 1, "name" : "xPhone", "price" : "799" }
   { "_id" : 5, "name" : "SmartPhone", "price" : [ "599", 699 ] }
```

字符串类型对应编号为 2，我们也可以在查询中使用数字编号 2 实现相同的结果：

```
db.products.find({
    price: {
        $type: 2
    }
}, {
    name: 1,
    price: 1
})
```

以下示例使用$type 运算符和别名“number”查找 price 字段属于 int、long 或者 double 类型，或者 price 字段是包含数字的数组的文档：

```
db.products.find({
    price: {
        $type: "number"
    }
}, {
    name: 1,
    price: 1
})
```

查询返回的文档如下：

```
{ "_id" : 2, "name" : "xTablet", "price" : 899 }
{ "_id" : 3, "name" : "SmartTablet", "price" : NumberLong(899) }
{ "_id" : 4, "name" : "SmartPad", "price" : [ 599, 699, 799 ] }
{ "_id" : 5, "name" : "SmartPhone", "price" : [ "599", 699 ] }
```

以下示例使用$type 运算符查找 price 字段属于数字或者字符串类型，或者 price 字段是包含数字或者字符串元素的数组的文档：

```
db.products.find({
    price: {
        $type: ["number", "string"]
    }
}, {
    name: 1,
    price: 1
})
```

查询返回的结果如下：

```
{ "_id" : 1, "name" : "xPhone", "price" : "799" }
{ "_id" : 2, "name" : "xTablet", "price" : 899 }
{ "_id" : 3, "name" : "SmartTablet", "price" : NumberLong(899) }
{ "_id" : 4, "name" : "SmartPad", "price" : [ 599, 699, 799 ] }
{ "_id" : 5, "name" : "SmartPhone", "price" : [ "599", 699 ] }
```

查询结果中没有包含_id 等于 6 的文档，因为该文档中没有 price 字段。

第 11 篇 数组运算符

本篇将会介绍 MongoDB 中查找数组元素相关的运算符，包括$size、$all 以及$elemMatch。

11.1 $size 运算符

$size 是一个数组查询运算符，可以判断文档的字段是否包含指定数量的元素。

$size 运算符的语法如下：

```
{ array_field: {$size: element_count} }
```

其中，array_field 是字段名，element_count 表示该字段包含的元素数量。

接下来的示例将会使用以下集合：

```
db.products.insertMany([
    { "_id" : 1, "name" : "xPhone", "price" : 799, "releaseDate" :
ISODate("2011-05-14T00:00:00Z"), "spec" : { "ram" : 4, "screen" : 6.5, "cpu" :
2.66 }, "color" : [ "white", "black" ], "storage" : [ 64, 128, 256 ] },
    { "_id" : 2, "name" : "xTablet", "price" : 899, "releaseDate" :
ISODate("2011-09-01T00:00:00Z"), "spec" : { "ram" : 16, "screen" : 9.5, "cpu" :
3.66 }, "color" : [ "white", "black", "purple" ], "storage" : [ 128, 256,
512 ] },
    { "_id" : 3, "name" : "SmartTablet", "price" : 899, "releaseDate" :
ISODate("2015-01-14T00:00:00Z"), "spec" : { "ram" : 12, "screen" : 9.7, "cpu" :
3.66 }, "color" : [ "blue" ], "storage" : [ 16, 64, 128 ] },
    { "_id" : 4, "name" : "SmartPad", "price" : 699, "releaseDate" :
ISODate("2020-05-14T00:00:00Z"), "spec" : { "ram" : 8, "screen" : 9.7, "cpu" :
1.66 }, "color" : [ "white", "orange", "gold", "gray" ], "storage" : [ 128, 256,
1024 ] },
    { "_id" : 5, "name" : "SmartPhone", "price" : 599, "releaseDate" :
ISODate("2022-09-14T00:00:00Z"), "spec" : { "ram" : 4, "screen" : 9.7, "cpu" :
1.66 }, "color" : [ "white", "orange", "gold", "gray" ], "storage" : [ 128,
256 ] },
    { "_id" : 6, "name" : "xWidget", "spec" : { "ram" : 64, "screen" : 9.7,
"cpu" : 3.66 }, "color" : [ "black" ], "storage" : [ 1024 ] }
])
```

以下示例使用$size 运算符查找数组字段 color 包含两个元素的文档：

```
db.products.find({
    color: {
        $size: 2
    }
}, {
    name: 1,
    color: 1
})
```

查询返回的文档如下：

```
{ "_id" : 1, "color" : [ "white", "black" ], "name" : "xPhone" }
```

以下示例同时使用了$size 运算符和$or 运算符查找数组字段 color 包含一个或者两个元素的文档：

```
db.products.find({
    $or: [{
            color: {
                $size: 1
            }
        },
        {
            color: {
                $size: 2
            }
        }
    ]
}, {
    name: 1,
    color: 1
})
```

查询返回的结果如下：

```
{ "_id" : 1, "color" : [ "white", "black" ], "name" : "xPhone" }
{ "_id" : 3, "color" : [ "blue" ], "name" : "SmartTablet" }
```

11.2 $all 运算符

$all 是一个数组查询运算符，可以判断文档的字段是否包含指定的所有元素。

$all 运算符的语法如下：

```
{ <arrayField>: { $all: [element1, element2, ...]} }
```

如果$all 运算符后面的数组为空，不会匹配任何文档。

如果$all 运算符只有一个元素，应该使用表达式，而不是数组：

```
{ <arrayField>: element1 }
```

$all 运算符可以使用等价的$and 运算符实现：

```
{ $and: [{ arrayField: element1}, {arrayField: element2} ]}
```

以下示例使用$all 运算符查找 color 字段同时包含“black”和“white”两个元素的文档：

```
db.products.find({
    color: {
        $all: ["black", "white"]
```

```
    }
}, {
    name: 1,
    color: 1
})
```

查询返回的结果如下：

```
{ "_id" : 1, "name" : "xPhone", "color" : [ "white", "black" ] }
{ "_id" : 2, "name" : "xTablet", "color" : [ "white", "black", "purple" ] }
```

上面的示例也可以使用$and 运算符实现：

```
db.products.find({
    $and: [
        {color: "black"},
        {color: "white"}
    ]
}, {
    name: 1,
    color: 1
})
```

11.3 $elemMatch 运算符

$elemMatch 也是一个数组查询运算符，可以判断文档是否包含指定数组字段，并且该字段至少包含一个满足条件的元素。

$elemMatch 运算符的语法如下：

```
{ <arrayField>: {$elemMatch: { <query1>, <query2>, ...} } }
```

注意，$elemMatch 运算符不支持$where 表达式或者$text 查询表达式。

以下示例使用$elemMatch 运算符查询 products 集合中的文档：

```
db.products.find({
    storage: {
        $elemMatch: {
            $lt: 128
        }
    }
}, {
    name: 1,
    storage: 1
});
```

查询返回了数组字段 storage 中至少包含一个小于 128 的元素的文档：

```
[
  { _id: 1, name: 'xPhone', storage: [ 64, 128, 256 ] },
  { _id: 3, name: 'SmartTablet', storage: [ 16, 64, 128 ] }
]
```

第 12 篇 查询结果排序

本篇将会介绍 MongoDB 中的游标 sort()方法，实现查询结果的排序功能。

12.1 sort()方法

sort()方法可以为查询返回的文档指定指定一个显示顺序：

```
cursor.sort({field1: order, field2: order, ...})
```

sort()方法支持多字段排序，每个字段都可以指定升序或者降序排序。

{ field: 1 }表示按照字段的升序排序：

```
cursor.sort({ field: 1 })
```

{ field: -1}表示按照字段的降序排序：

```
cursor.sort({field: -1})
```

下面的语法表示先按照字段 field1 的升序排序，然后再按照字段 field2 的降序排序：

```
cursor.sort({field1: 1, field2: -1});
```

相同类型的数据比较很简单，但是不同 BSON 类型的数据比较略微有些复杂。

MongoDB 使用以下从小到大的顺序比较不同的 BSON 类型：

1. MinKey（内部类型）
2. 空类型
3. 数字（整数、长整数、双精度浮点数、十进制数字）
4. 符号、字符串
5. 对象
6. 数组
7. BinData
8. ObjectId
9. 布尔类型
10. 日期
11. 时间戳
12. 正则表达式
13. MaxKey（内部类型）

更多详细信息可以参考官方文档。

12.2 sort()示例

接下来我们将会使用以下 products 集合演示 sort()方法的使用。

```
db.products.insertMany([
    { "_id" : 1, "name" : "xPhone", "price" : 799, "releaseDate" :
ISODate("2011-05-14T00:00:00Z"), "spec" : { "ram" : 4, "screen" : 6.5, "cpu" :
2.66 }, "color" : [ "white", "black" ], "storage" : [ 64, 128, 256 ] },
    { "_id" : 2, "name" : "xTablet", "price" : 899, "releaseDate" :
ISODate("2011-09-01T00:00:00Z"), "spec" : { "ram" : 16, "screen" : 9.5, "cpu" :
3.66 }, "color" : [ "white", "black", "purple" ], "storage" : [ 128, 256,
512 ] },
    { "_id" : 3, "name" : "SmartTablet", "price" : 899, "releaseDate" :
ISODate("2015-01-14T00:00:00Z"), "spec" : { "ram" : 12, "screen" : 9.7, "cpu" :
3.66 }, "color" : [ "blue" ], "storage" : [ 16, 64, 128 ] },
    { "_id" : 4, "name" : "SmartPad", "price" : 699, "releaseDate" :
ISODate("2020-05-14T00:00:00Z"), "spec" : { "ram" : 8, "screen" : 9.7, "cpu" :
1.66 }, "color" : [ "white", "orange", "gold", "gray" ], "storage" : [ 128, 256,
1024 ] },
    { "_id" : 5, "name" : "SmartPhone", "price" : 599, "releaseDate" :
ISODate("2022-09-14T00:00:00Z"), "spec" : { "ram" : 4, "screen" : 9.7, "cpu" :
1.66 }, "color" : [ "white", "orange", "gold", "gray" ], "storage" : [ 128,
256 ] },
    { "_id" : 6, "name" : "xWidget", "spec" : { "ram" : 64, "screen" : 9.7,
"cpu" : 3.66 }, "color" : [ "black" ], "storage" : [ 1024 ] },
    { "_id" : 7, "name" : "xReader", "price" : null, "spec" : { "ram" : 64,
"screen" : 6.7, "cpu" : 3.66 }, "color" : [ "black", "white" ], "storage" :
[ 128 ] }
])
```

示例一：单个字段排序

下面的查询返回了所有存在 price 字段的文档，返回的字段包含_id、name 以及 price：

```
db.products.find({
    'price': {
        $exists: 1
    }
}, {
    name: 1,
    price: 1
})
```

输出结果如下：

```
[
  { _id: 1, name: 'xPhone', price: 799 },
  { _id: 2, name: 'xTablet', price: 899 },
  { _id: 3, name: 'SmartTablet', price: 899 },
  { _id: 4, name: 'SmartPad', price: 699 },
  { _id: 5, name: 'SmartPhone', price: 599 },
```

```
  { _id: 7, name: 'xReader', price: null }
]
```

如果想要按照价格从低到高对返回的结果进行排序，可以使用以下 sort()方法：

```
db.products.find({
    'price': {
        $exists: 1
    }
}, {
    name: 1,
    price: 1
}).sort({
    price: 1
})
```

输出结果如下：

```
[
  { _id: 7, name: 'xReader', price: null },
  { _id: 5, name: 'SmartPhone', price: 599 },
  { _id: 4, name: 'SmartPad', price: 699 },
  { _id: 1, name: 'xPhone', price: 799 },
  { _id: 2, name: 'xTablet', price: 899 },
  { _id: 3, name: 'SmartTablet', price: 899 }
]
```

以上示例中，价格为空的产品排在最前面，然后依次从低到高进行排列。

如果想要按照价格从高到低进行排序，可以将 sort()方法中的 price 字段设置为 -1：

```
db.products.find({
    'price': {
        $exists: 1
    }
}, {
    name: 1,
    price: 1
}).sort({
    price: -1
})
```

返回结果如下：

```
[
  { _id: 2, name: 'xTablet', price: 899 },
  { _id: 3, name: 'SmartTablet', price: 899 },
  { _id: 1, name: 'xPhone', price: 799 },
  { _id: 4, name: 'SmartPad', price: 699 },
  { _id: 5, name: 'SmartPhone', price: 599 },
  { _id: 7, name: 'xReader', price: null }
```

```
]
```

示例二：多个字段排序

以下示例使用 sort()方法按照 name 和 price 字段从小到大的顺序对查询结果进行排序：

```
db.products.find({
    'price': {
        $exists: 1
    }
}, {
    name: 1,
    price: 1
}).sort({
    price: 1,
    name: 1
});
```

输出结果如下：

```
[
  { _id: 7, name: 'xReader', price: null },
  { _id: 5, name: 'SmartPhone', price: 599 },
  { _id: 4, name: 'SmartPad', price: 699 },
  { _id: 1, name: 'xPhone', price: 799 },
  { _id: 3, name: 'SmartTablet', price: 899 },
  { _id: 2, name: 'xTablet', price: 899 }
]
```

在以上示例中，sort()方法首先按照价格进行排序，然后对价格相同的结果再按照名称进行排序。_id 为 3 和 2 的产品价格相同，都是 899；它们按照名称进行升序排列，SmartTablet 排在了 xTablet 之前。

以下示例按照价格升序、名称降序的方式对返回的产品进行排序：

```
db.products.find({
    'price': {
        $exists: 1
    }
}, {
    name: 1,
    price: 1
}).sort({
    price: 1,
    name: -1
})

[
  { _id: 7, name: 'xReader', price: null },
  { _id: 5, name: 'SmartPhone', price: 599 },
  { _id: 4, name: 'SmartPad', price: 699 },
```

```
  { _id: 1, name: 'xPhone', price: 799 },
  { _id: 2, name: 'xTablet', price: 899 },
  { _id: 3, name: 'SmartTablet', price: 899 }
]
```

此时，xTable 排在了 SmartTablet 之前。

示例三：基于日期排序

以下示例按照日期字段 releaseDate 对 products 集合中的文档进行排序，查询只返回了存在 releaseDate 字段的文档，并且只返回了_id、name 以及 releaseDate 字段：

```
db.products.find({
    releaseDate: {
        $exists: 1
    }

}, {
    name: 1,
    releaseDate: 1
}).sort({
    releaseDate: 1
});
```

查询返回的结果如下：

```
[
  {
    _id: 1,
    name: 'xPhone',
    releaseDate: ISODate("2011-05-14T00:00:00.000Z")
  },
  {
    _id: 2,
    name: 'xTablet',
    releaseDate: ISODate("2011-09-01T00:00:00.000Z")
  },
  {
    _id: 3,
    name: 'SmartTablet',
    releaseDate: ISODate("2015-01-14T00:00:00.000Z")
  },
  {
    _id: 4,
    name: 'SmartPad',
    releaseDate: ISODate("2020-05-14T00:00:00.000Z")
  },
  {
    _id: 5,
    name: 'SmartPhone',
```

```
    releaseDate: ISODate("2022-09-14T00:00:00.000Z")
  }
]
```

示例四：基于嵌入字段排序

以下示例使用嵌入式文档 spec 中的 ram 字段对产品进行排序：

```
db.products.find({}, {
    name: 1,
    spec: 1
}).sort({
    "spec.ram": 1
});
```

输出结果如下：

```
[
  { _id: 1, name: 'xPhone', spec: { ram: 4, screen: 6.5, cpu: 2.66 } },
  {
    _id: 5,
    name: 'SmartPhone',
    spec: { ram: 4, screen: 9.7, cpu: 1.66 }
  },
  {
    _id: 4,
    name: 'SmartPad',
    spec: { ram: 8, screen: 9.7, cpu: 1.66 }
  },
  {
    _id: 3,
    name: 'SmartTablet',
    spec: { ram: 12, screen: 9.7, cpu: 3.66 }
  },
  {
    _id: 2,
    name: 'xTablet',
    spec: { ram: 16, screen: 9.5, cpu: 3.66 }
  },
  {
    _id: 6,
    name: 'xWidget',
    spec: { ram: 64, screen: 9.7, cpu: 3.66 }
  },
  {
    _id: 7,
    name: 'xReader',
    spec: { ram: 64, screen: 6.7, cpu: 3.66 }
  }
]
```

第 13 篇 限制返回结果数量

本篇我们学习 MongoDB 中的 limit()方法，它可以限制查询返回的文档数量。

13.1 limit()方法

find()方法可能会查找出大量的文档，但是应用程序不一定需要所有的返回结果。为了限制返回文档的数量，可以使用 limit()方法：

```
db.collection.find(<query>).limit(<documentCount>)
```

其中，文档数量的取值范围为-2^{31} 到 2^{31}。如果指定了超出该范围的参数，limit()返回的结果不确定。

如果指定了负数值，limit()返回结果和对应正数参数的结果相同。另外，系统在返回一个批次的文档之后将会关闭游标。如果查询结果的数量不止一个批次，实际返回的结果数量将会小于指定的数量。

如果将文档数量设置为 0，相当于没有限制返回结果的数量。

提示：limit()方法和 SQL 语句中的 LIMIT 子句作用类似。

如果想要使用 limit()方法获取确定内容的结果，需要先对结果进行排序：

```
cursor
   .sort({...})
   .limit(<documentCount>)
```

在实际应用中，我们通常结合 limit()和 skip()方法实现分页功能。skip()方法用于指定从哪一个文档开始返回结果：

```
cursor.skip(<offset>)
```

以下语法用于返回第 pageNo 页中的文档，每页包含 documentCount 个文档：

```
db.collection.find({...}
).sort({...}
).skip(pageNo > 0 ? ( ( pageNo - 1 ) * documentCount) : 0
).limit(documentCount);
```

skip()意味着 MongoDB 服务器需要从结果集中扫描并忽略指定数量的文档，然后开始返回结果。因此，随着页数的增加，skip()方法会越来越慢。

提示：limit()和 skip()方法和 SQL 中的 LIMIT OFFSET 子句作用类似。

13.2 limit()示例

接下来的示例将会使用以下产品集合：

```
db.products.insertMany([
    { "_id" : 1, "name" : "xPhone", "price" : 799, "releaseDate" :
ISODate("2011-05-14T00:00:00Z"), "spec" : { "ram" : 4, "screen" : 6.5, "cpu" :
2.66 }, "color" : [ "white", "black" ], "storage" : [ 64, 128, 256 ] },
    { "_id" : 2, "name" : "xTablet", "price" : 899, "releaseDate" :
ISODate("2011-09-01T00:00:00Z"), "spec" : { "ram" : 16, "screen" : 9.5, "cpu" :
3.66 }, "color" : [ "white", "black", "purple" ], "storage" : [ 128, 256,
512 ] },
    { "_id" : 3, "name" : "SmartTablet", "price" : 899, "releaseDate" :
ISODate("2015-01-14T00:00:00Z"), "spec" : { "ram" : 12, "screen" : 9.7, "cpu" :
3.66 }, "color" : [ "blue" ], "storage" : [ 16, 64, 128 ] },
    { "_id" : 4, "name" : "SmartPad", "price" : 699, "releaseDate" :
ISODate("2020-05-14T00:00:00Z"), "spec" : { "ram" : 8, "screen" : 9.7, "cpu" :
1.66 }, "color" : [ "white", "orange", "gold", "gray" ], "storage" : [ 128, 256,
1024 ] },
    { "_id" : 5, "name" : "SmartPhone", "price" : 599, "releaseDate" :
ISODate("2022-09-14T00:00:00Z"), "spec" : { "ram" : 4, "screen" : 9.7, "cpu" :
1.66 }, "color" : [ "white", "orange", "gold", "gray" ], "storage" : [ 128,
256 ] },
    { "_id" : 6, "name" : "xWidget", "spec" : { "ram" : 64, "screen" : 9.7,
"cpu" : 3.66 }, "color" : [ "black" ], "storage" : [ 1024 ] },
    { "_id" : 7, "name" : "xReader", "price" : null, "spec" : { "ram" : 64,
"screen" : 6.7, "cpu" : 3.66 }, "color" : [ "black", "white" ], "storage" :
[ 128 ] }
])
```

示例一：查找最昂贵的产品

以下示例使用 limit()方法获取产品集合中最贵的产品，返回结果中包含了_id、name 以及 price 字段：

```
db.products.find({}, {
    name: 1,
    price: 1
}).sort({
    price: -1
}).limit(1);
```

查询返回的结果如下：

```
[ { _id: 2, name: 'xTablet', price: 899 } ]
```

在示例中，我们按照价格从高到低对产品进行排序，然后使用 limit()返回了排名第一的产品。产品集合中包含了 2 个价格为 899 的产品，因此返回的文档取决于它们在磁盘中的存储顺序。

为了返回确定性的结果，排序结果必须具有唯一性。例如：

```
db.products.find({}, {
    name: 1,
    price: 1
}).sort({
```

```
    price: -1,
    name: 1
}).limit(1);
```

查询返回的结果如下：

```
[ { _id: 3, name: 'SmartTablet', price: 899 } ]
```

示例同时使用了价格和名称进行排序，当价格相同时按照名称升序进行排列，此时查询返回了确定的结果。

示例二：实现分页查询功能

假如我们想要将产品进行分页展示，每页显示 2 个产品。以下查询使用 skip() 和 limit()方法获取了第 2 页中的文档：

```
db.products.find({}, {
    name: 1,
    price: 1
}).sort({
    price: -1,
    name: 1
}).skip(2).limit(2);
```

查询结果如下：

```
[
  { _id: 1, name: 'xPhone', price: 799 },
  { _id: 4, name: 'SmartPad', price: 699 }
]
```

第 14 篇 CRUD 之更新文档

本篇我们将会介绍如何使用集合的 updateOne()和 updateMany()方法更新文档。

14.1 updateOne()方法

updateOne()方法可以更新满足条件的单个文档，语法如下：

```
db.collection.updateOne(filter, update, options)
```

其中，

- filter 用于指定一个查询条件。如果多个文档都满足条件，updateOne()只会更新第一个满足条件的文档。如果指定一个空文档（{}）作为查询条件，updateOne()将会更新集合中返回的第一个文档。
- update 用于指定更新操作。
- options 参数用于指定一些更新选项，本篇不涉及相关内容。

updateOne()方法将会返回一个结果文档，包括但不限于以下字段：

- matchedCount 字段返回了满足条件的文档数量。
- modifiedCount 字段返回了被更新的文档数量。对于 updateOne()方法，结果为 0 或者 1。

$set 操作符

$set 操作符用于替换指定字段的值，语法如下：

```
{ $set: { <field1>: <value1>, <field2>: <value2>, ...}}
```

如果指定的字段不存在，$set 操作符将会为文档增加一个新的字段，只要新的字段不违法类型约束。

如果指定字段时使用了点符合，例如 embededDoc.field，同时字段 field 不存在，$set 操作符将会创建一个嵌入式文档。

14.2 updateOne()示例

接下来的示例将会使用以下 products 集合：

```
db.products.insertMany([
    { "_id" : 1, "name" : "xPhone", "price" : 799, "releaseDate":
ISODate("2011-05-14"), "spec" : { "ram" : 4, "screen" : 6.5, "cpu" :
2.66 },"color":["white","black"],"storage":[64,128,256]},
    { "_id" : 2, "name" : "xTablet", "price" : 899, "releaseDate":
ISODate("2011-09-01") , "spec" : { "ram" : 16, "screen" : 9.5, "cpu" :
3.66 },"color":["white","black","purple"],"storage":[128,256,512]},
```

```
       { "_id" : 3, "name" : "SmartTablet", "price" : 899, "releaseDate":
ISODate("2015-01-14"), "spec" : { "ram" : 12, "screen" : 9.7, "cpu" :
3.66 },"color":["blue"],"storage":[16,64,128]},
       { "_id" : 4, "name" : "SmartPad", "price" : 699, "releaseDate":
ISODate("2020-05-14"),"spec" : { "ram" : 8, "screen" : 9.7, "cpu" :
1.66 },"color":["white","orange","gold","gray"],"storage":[128,256,1024]},
       { "_id" : 5, "name" : "SmartPhone", "price" : 599,"releaseDate":
ISODate("2022-09-14"), "spec" : { "ram" : 4, "screen" : 5.7, "cpu" :
1.66 },"color":["white","orange","gold","gray"],"storage":[128,256]}
    ])
```

示例一：更新单个文档

以下示例使用 updateOne()方法更新_id 等于 1 的文档中的 price 字段：

```
db.products.updateOne({
    _id: 1
}, {
    $set: {
        price: 899
    }
})
```

其中，

- { _id : 1 }是查找文档的条件。
- { $set: { price: 899 } }指定了更新操作，利用$set 操作符将 price 字段修改为 899。

示例返回的结果如下：

```
{
  acknowledged: true,
  insertedId: null,
  matchedCount: 1,
  modifiedCount: 1,
  upsertedCount: 0
}
```

在返回结果中，matchedCount 代表了满足查询条件的文档数量（1），modifiedCount 代表了被修改的文档数量（1）。

我们可以使用 findOne()方法验证修改后的文档内容：

```
db.products.findOne({ _id: 1 }, { name: 1, price: 1 })

{ _id: 1, name: 'xPhone', price: 899 }
```

示例二：更新第一个匹配的文档

以下查询返回了价格等于 899 的所有产品：

```
db.products.find({ price: 899 }, { name: 1, price: 1 })

[
  { _id: 1, name: 'xPhone', price: 899 },
  { _id: 2, name: 'xTablet', price: 899 },
  { _id: 3, name: 'SmartTablet', price: 899 }
]
```

以下示例使用 updateOne()方法更新价格等于 899 的第一个文档：

```
db.products.updateOne({ price: 899 }, { $set: { price: null } })

{
  acknowledged: true,
  insertedId: null,
  matchedCount: 1,
  modifiedCount: 1,
  upsertedCount: 0
}
```

按照默认的返回顺序，第一个价格等于 899 的产品_id 等于 1，它的价格被更新为 null：

```
db.products.find({ _id: 1}, { name: 1, price: 1 })

[ { _id: 1, name: 'xPhone', price: null } ]
```

示例三：更新嵌入式文档

以下查询返回了_id 等于 4 的文档：

```
db.products.find({ _id: 4 }, { name: 1, spec: 1 })

[
  {
    _id: 4,
    name: 'SmartPad',
    spec: { ram: 8, screen: 9.7, cpu: 1.66 }
  }
]
```

下面的示例使用 updateOne()方法更新该文档中的嵌入式文档 spec 的 ram、screen 以及 cpu 字段：

```
db.products.updateOne({
    _id: 4
}, {
    $set: {
        "spec.ram": 16,
        "spec.screen": 10.7,
        "spec.cpu": 2.66
    }
```

```
})
```

操作返回的结果如下：

```
{
  acknowledged: true,
  insertedId: null,
  matchedCount: 1,
  modifiedCount: 1,
  upsertedCount: 0
}
```

再次查询文档中的内容：

```
db.products.find({ _id: 4 }, { name: 1, spec: 1 })

[
  {
    _id: 4,
    name: 'SmartPad',
    spec: { ram: 16, screen: 10.7, cpu: 2.66 }
  }
]
```

示例四：更新数组元素

以下示例使用 updateOne() 方法更新文档（_id: 4）中数组字段 storage 的第一个和第二个元素：

```
db.products.updateOne(
 { _id: 4},
 {
    $set: {
        "storage.0": 16,
        "storage.1": 32
    }
 }
)
```

返回结果如下：

```
{
  acknowledged: true,
  insertedId: null,
  matchedCount: 1,
  modifiedCount: 1,
  upsertedCount: 0
}
```

再次查询该文档中的内容：

```
db.products.find({ _id: 4 }, { name: 1, storage: 1 });
```

```
[ { _id: 4, name: 'SmartPad', storage: [ 16, 32, 1024 ] } ]
```

14.3 updateMany() 方法

updateMany() 方法可以更新满足条件的所有文档，语法如下：

```
db.collection.updateMany(filter, update, options)
```

其中，

- filter 用于指定一个查询条件。如果指定一个空文档（{}）作为查询条件，将会更新集合中的全部文档。
- update 用于指定更新操作。
- options 参数用于指定一些更新选项，本篇不涉及相关内容。

updateMany() 方法将会返回一个结果文档，包括但不限于以下字段：

- matchedCount 字段返回了满足条件的文档数量。
- modifiedCount 字段返回了被更新的文档数量。

updateMany() 同样使用 $set 操作符执行更新操作。

14.4 updateMany() 示例

示例一：更新多个文档

以下示例使用 updateMany() 方法更新价格为 899 的所有产品：

```
db.products.updateMany(
    {  price: 899},
    { $set: {  price: 895 }}
)
```

其中，

- { price : 899 } 指定了查找文档的条件。
- { $set: { price: 895} } 指定了更新操作，利用 $set 操作符将 price 字段修改为 895。

查询返回的结果如下：

```
{
  acknowledged: true,
  insertedId: null,
  matchedCount: 2,
  modifiedCount: 2,
  upsertedCount: 0
}
```

在返回结果中，matchedCount 代表了满足查询条件的文档数量（2），modifiedCount 代表了被更新的文档数量（2）。

再次通过价格（895）查询产品：

```
db.products.find({
    price: 895
}, {
    name: 1,
    price: 1
})
```

查询返回的结果如下：

```
[
  { _id: 2, name: 'xTablet', price: 895 },
  { _id: 3, name: 'SmartTablet', price: 895 }
]
```

示例二：更新嵌入式文档

以下示例使用 updateMany() 更新价格大于 700 的产品中嵌入式文档 spec 的 ram、screen 以及 cpu 字段：

```
db.products.updateMany({
    price: { $gt: 700}
}, {
    $set: {
        "spec.ram": 32,
        "spec.screen": 9.8,
        "spec.cpu": 5.66
    }
})
```

返回结果显示成功更新 3 个文档：

```
{
  acknowledged: true,
  insertedId: null,
  matchedCount: 3,
  modifiedCount: 3,
  upsertedCount: 0
}
```

示例三：更新数组元素

以下示例使用 updateMany() 方法更新文档（_id 等于 1、2、3）中的数组字段 storage 的第一个元素和第二个元素：

```
db.products.updateMany({
    _id: {
        $in: [1, 2, 3]
```

```
    }
}, {
    $set: {
        "storage.0": 16,
        "storage.1": 32
    }
})
```

返回结果如下：

```
{
  acknowledged: true,
  insertedId: null,
  matchedCount: 3,
  modifiedCount: 3,
  upsertedCount: 0
}
```

第 15 篇 文档更新之$inc 操作符

本篇将会介绍如何在 update() 方法中使用 $inc 操作符增加指定字段的值。

15.1 $inc 操作符

有时候我们需要增加文档中某些字段的值， $inc 操作符可以实现这个功能。

$inc 操作符的语法如下：

```
{ $inc: {<field1>: <amount1>, <field2>: <amount2>, ...} }
```

其中，amount 可以是正数或者负数。正数表示增加字段的值，负数表示减少字段的值。

如果指定的字段不存在，$inc 操作符将会创建并设置该字段的值。

15.2 $inc 示例

我们将会使用以下集合进行演示：

```
db.products.insertMany([
    { "_id" : 1, "name" : "xPhone", "price" : 799, "releaseDate":
ISODate("2011-05-14"), "spec" : { "ram" : 4, "screen" : 6.5, "cpu" :
2.66 },"color":["white","black"],"storage":[64,128,256]},
    { "_id" : 2, "name" : "xTablet", "price" : 899, "releaseDate":
ISODate("2011-09-01") , "spec" : { "ram" : 16, "screen" : 9.5, "cpu" :
3.66 },"color":["white","black","purple"],"storage":[128,256,512]},
    { "_id" : 3, "name" : "SmartTablet", "price" : 899, "releaseDate":
ISODate("2015-01-14"), "spec" : { "ram" : 12, "screen" : 9.7, "cpu" :
3.66 },"color":["blue"],"storage":[16,64,128]},
    { "_id" : 4, "name" : "SmartPad", "price" : 699, "releaseDate":
ISODate("2020-05-14"),"spec" : { "ram" : 8, "screen" : 9.7, "cpu" :
1.66 },"color":["white","orange","gold","gray"],"storage":[128,256,1024]},
    { "_id" : 5, "name" : "SmartPhone", "price" : 599,"releaseDate":
ISODate("2022-09-14"), "spec" : { "ram" : 4, "screen" : 5.7, "cpu" :
1.66 },"color":["white","orange","gold","gray"],"storage":[128,256]}
  ])
```

示例一：增加指定字段的值

以下示例使用 $inc 操作符将 products 集合中文档（_id: 1）的 price 字段的值增加 50：

```
db.products.updateOne({
    _id: 1
}, {
    $inc: {
        price: 50
```

```
    }
})
```

返回结果如下：

```
{
  acknowledged: true,
  insertedId: null,
  matchedCount: 1,
  modifiedCount: 1,
  upsertedCount: 0
}
```

再次查询该文档，可以看到价格的变化：

```
db.products.find(
    {_id: 1},
    {name: 1, price: 1}
)

[ { _id: 1, name: 'xPhone', price: 849 } ]
```

示例二：减少指定字段的值

下面的示例使用 $inc 操作符将文档（_id: 1）中的 price 字段的值减少 150：

```
db.products.updateOne({
    _id: 1
}, {
    $inc: {
        price: -150
    }
})
```

返回结果如下：

```
{
  acknowledged: true,
  insertedId: null,
  matchedCount: 1,
  modifiedCount: 1,
  upsertedCount: 0
}
```

再次查询该产品的价格：

```
db.products.find(
    { _id: 1 },
    { name: 1, price: 1 }
)

[ { _id: 1, name: 'xPhone', price: 699 } ]
```

示例三：更新多个字段的值

以下示例使用 $inc 操作符更新了嵌入式文档 spec 中的 price 字段和 ram 字段的值：

```
db.products.updateOne({
    _id: 1
}, {
    $inc: {
        price: 50,
        "spec.ram": 4
    }
})
```

输出结果如下：

```
{
  acknowledged: true,
  insertedId: null,
  matchedCount: 1,
  modifiedCount: 1,
  upsertedCount: 0
}
```

再次查询该文档，验证更新后的结果：

```
db.products.find(
    {_id: 1},
    {name: 1, price: 1, "spec.ram": 1}
)

[ { _id: 1, name: 'xPhone', price: 749, spec: { ram: 8 } } ]
```

第 16 篇 文档更新之 min/max 操作符

本篇将会介绍如何在 update() 方法中使用 $min 和 $max 操作符更新指定字段的值。

16.1 $min 操作符

$min 是一个字段更新操作符，如果指定的数值小于（<）字段当前值，将字段的值修改为指定值。

$min 操作符的语法如下：

```
{ $min: {<field1>: <value1>, ...} }
```

如果字段当前值大于等于指定的值，不会执行更新操作。

如果指定的字段不存在，$min 操作符将会创建字段并设置字段的值。

16.2 $min 示例

下文中的示例将会使用以下集合：

```
db.products.insertMany([
    { "_id" : 1, "name" : "xPhone", "price" : 799, "releaseDate":
ISODate("2011-05-14"), "spec" : { "ram" : 4, "screen" : 6.5, "cpu" :
2.66 },"color":["white","black"],"storage":[64,128,256]},
    { "_id" : 2, "name" : "xTablet", "price" : 899, "releaseDate":
ISODate("2011-09-01") , "spec" : { "ram" : 16, "screen" : 9.5, "cpu" :
3.66 },"color":["white","black","purple"],"storage":[128,256,512]},
    { "_id" : 3, "name" : "SmartTablet", "price" : 899, "releaseDate":
ISODate("2015-01-14"), "spec" : { "ram" : 12, "screen" : 9.7, "cpu" :
3.66 },"color":["blue"],"storage":[16,64,128]},
    { "_id" : 4, "name" : "SmartPad", "price" : 699, "releaseDate":
ISODate("2020-05-14"),"spec" : { "ram" : 8, "screen" : 9.7, "cpu" :
1.66 },"color":["white","orange","gold","gray"],"storage":[128,256,1024]},
    { "_id" : 5, "name" : "SmartPhone", "price" : 599,"releaseDate":
ISODate("2022-09-14"), "spec" : { "ram" : 4, "screen" : 5.7, "cpu" :
1.66 },"color":["white","orange","gold","gray"],"storage":[128,256]}
 ])
```

以下示例使用 $min 操作符更新文档（_id: 5）的 price 字段：

```
db.products.updateOne({
    _id: 5
}, {
    $min: {
        price: 699
    }
})
```

查询匹配了一个文档，但是没有执行更新操作：

```
{
  acknowledged: true,
  insertedId: null,
  matchedCount: 1,
  modifiedCount: 0,
  upsertedCount: 0
}
```

原因在于 699 大于字段的当前值（599）。

以下示例使用 $min 操作符将文档（_id: 5）的 price 字段更新为 499：

```
db.products.updateOne({
    _id: 5
}, {
    $min: {
        price: 499
    }
})
```

此时，查询更新了一个文档：

```
{
  acknowledged: true,
  insertedId: null,
  matchedCount: 1,
  modifiedCount: 1,
  upsertedCount: 0
}
```

验证文档更新后的内容：

```
db.products.find({ _id: 5 }, { name: 1, price: 1 })

[ { _id: 5, name: 'SmartPhone', price: 499 } ]
```

16.3 $max 操作符

$max 是一个字段更新操作符，如果指定的数值大于（>）字段当前值，将字段的值修改为指定值。

$max 操作符的语法如下：

```
{ $max: {<field1>: <value1>, ...} }
```

如果字段当前值小于等于指定的值，不会执行更新操作。

如果指定的字段不存在，$max 操作符将会创建字段并设置字段的值。

16.4 $max 示例

以下示例使用 $max 操作符更新文档（_id: 1）的 price 字段：

```
db.products.updateOne({
    _id: 1
}, {
    $max: {
        price: 699
    }
})
```

查询匹配了一个文档，但是没有执行更新操作：

```
{
  acknowledged: true,
  insertedId: null,
  matchedCount: 1,
  modifiedCount: 0,
  upsertedCount: 0
}
```

原因在于 699 小于字段的当前值（799）。

以下示例使用 $min 操作符将文档（_id: 1）的 price 字段更新为 899：

```
db.products.updateOne({
    _id: 1
}, {
    $max: {
        price: 899
    }
})
```

此时，查询更新了一个文档：

```
{
  acknowledged: true,
  insertedId: null,
  matchedCount: 1,
  modifiedCount: 1,
  upsertedCount: 0
}
```

验证文档更新后的内容：

```
db.products.find({ _id: 1 }, { name: 1, price: 1 })

[ { _id: 1, name: 'xPhone', price: 899 } ]
```

第 17 篇 文档更新之$mul 操作符

本篇将会介绍如何使用 MongoDB $mul 操作符将字段的值乘以一个倍数。

17.1 $mul 操作符

$mul 是一个字段更新操作符，可以将指定字段的值乘以一个倍数。

$mul 操作符的语法如下：

```
{ $mul: { <field1>: <number1>, <field2>: <number2>, ... } }
```

被更新的字段必须是一个数字字段。如果需要更新嵌入式文档中的字段或者数组中的元素，可以使用点符号，例如 <embedded_doc>.<field> 或者<array>.<index>。

如果被更新的字段不存在，$mul 操作符将会创建字段并将其值设置为 0。

17.2 $mul 示例

接下来的示例我们将会使用一下集合：

```
db.products.insertMany([
    { "_id" : 1, "name" : "xPhone", "price" : 799, "releaseDate":
ISODate("2011-05-14"), "spec" : { "ram" : 4, "screen" : 6.5, "cpu" :
2.66 },"color":["white","black"],"storage":[64,128,256]},
    { "_id" : 2, "name" : "xTablet", "price" : 899, "releaseDate":
ISODate("2011-09-01") , "spec" : { "ram" : 16, "screen" : 9.5, "cpu" :
3.66 },"color":["white","black","purple"],"storage":[128,256,512]},
    { "_id" : 3, "name" : "SmartTablet", "price" : 899, "releaseDate":
ISODate("2015-01-14"), "spec" : { "ram" : 12, "screen" : 9.7, "cpu" :
3.66 },"color":["blue"],"storage":[16,64,128]},
    { "_id" : 4, "name" : "SmartPad", "price" : 699, "releaseDate":
ISODate("2020-05-14"),"spec" : { "ram" : 8, "screen" : 9.7, "cpu" :
1.66 },"color":["white","orange","gold","gray"],"storage":[128,256,1024]},
    { "_id" : 5, "name" : "SmartPhone", "price" : 599,"releaseDate":
ISODate("2022-09-14"), "spec" : { "ram" : 4, "screen" : 5.7, "cpu" :
1.66 },"color":["white","orange","gold","gray"],"storage":[128,256]}
  ])
```

示例一：将字段的值乘以一个倍数

以下示例使用 $mul 操作符将文档（_id: 5）的 price 字段的值增加 10%：

```
db.products.updateOne({ _id: 5 }, { $mul: { price: 1.1 } })
```

返回结果显示查询匹配并更新了一个文档：

```
{
  acknowledged: true,
```

```
  insertedId: null,
  matchedCount: 1,
  modifiedCount: 1,
  upsertedCount: 0
}
```

以下查询验证了更新后的结果：

```
db.products.find({
    _id: 5
}, {
    name: 1,
    price: 1
})

[ { _id: 5, name: 'SmartPhone', price: 658.9000000000001 } ]
```

示例二：将数组元素乘以一个倍数

以下查询使用 $mul 操作符将文档（_id: 1）中数组 storage 的第一个、第二个以及第三个元素的值增加一倍：

```
db.products.updateOne({
    _id: 1
}, {
    $mul: {
        "storage.0": 2,
        "storage.1": 2,
        "storage.2": 2
    }
})
```

查询返回结果如下：

```
{
  acknowledged: true,
  insertedId: null,
  matchedCount: 1,
  modifiedCount: 1,
  upsertedCount: 0
}
```

以下查询验证了更新之后的文档内容：

```
db.products.findOne({ _id: 1 }, { name: 1, storage: 1 })

{ _id: 1, name: 'xPhone', storage: [ 128, 256, 512 ] }
```

示例三：将嵌入式文档中的字段值乘以一个倍数

以下示例使用 $mul 操作符将 products 集合中全部文档的 spec 中的 ram 字段值增加一倍：

```
db.products.updateMany({}, {
    $mul: {
        "spec.ram": 2
    }
})
```

查询返回结果如下：

```
{
  acknowledged: true,
  insertedId: null,
  matchedCount: 5,
  modifiedCount: 5,
  upsertedCount: 0
}
```

以下查询返回了 products 集合中的全部文档：

```
db.products.find({}, { name: 1, "spec.ram": 1 })

[
  { _id: 1, name: 'xPhone', spec: { ram: 8 } },
  { _id: 2, name: 'xTablet', spec: { ram: 32 } },
  { _id: 3, name: 'SmartTablet', spec: { ram: 24 } },
  { _id: 4, name: 'SmartPad', spec: { ram: 16 } },
  { _id: 5, name: 'SmartPhone', spec: { ram: 8 } }
]
```

第 18 篇 文档更新之$unset 操作符

本篇将会介绍如何利用 $unset 操作符删除文档中的字段。

18.1 $unset 操作符

$unset 是一个字段更新操作符，用于删除文档中的指定字段。

$unset 操作符的语法如下：

```
{ $unset: {<field>: "", ... }}
```

参数中的字段值不影响操作结果，可以指定任意值。如果指定字段不存在，不会执行任何操作，也不会返回错误或者警告。

如果想要指定嵌入书文档中的字段，可以使用点符号：

```
{ $unset: { "<embedded_doc>.<field>": "", ... }}
```

需要注意的是，$unset 操作符不会删除任何数组元素，而是将数组元素设置为空。

```
{ $unset: {"<array>.<index>": "", ...}
```

这种实现可以保持数组大小和元素位置的一致性。

18.2 $unset 示例

下文中的示例将会使用 products 集合：

```
db.products.insertMany([
    { "_id" : 1, "name" : "xPhone", "price" : 799, "releaseDate":
ISODate("2011-05-14"), "spec" : { "ram" : 4, "screen" : 6.5, "cpu" :
2.66 },"color":["white","black"],"storage":[64,128,256]},
    { "_id" : 2, "name" : "xTablet", "price" : 899, "releaseDate":
ISODate("2011-09-01") , "spec" : { "ram" : 16, "screen" : 9.5, "cpu" :
3.66 },"color":["white","black","purple"],"storage":[128,256,512]},
    { "_id" : 3, "name" : "SmartTablet", "price" : 899, "releaseDate":
ISODate("2015-01-14"), "spec" : { "ram" : 12, "screen" : 9.7, "cpu" :
3.66 },"color":["blue"],"storage":[16,64,128]},
    { "_id" : 4, "name" : "SmartPad", "price" : 699, "releaseDate":
ISODate("2020-05-14"),"spec" : { "ram" : 8, "screen" : 9.7, "cpu" :
1.66 },"color":["white","orange","gold","gray"],"storage":[128,256,1024]},
    { "_id" : 5, "name" : "SmartPhone", "price" : 599,"releaseDate":
ISODate("2022-09-14"), "spec" : { "ram" : 4, "screen" : 5.7, "cpu" :
1.66 },"color":["white","orange","gold","gray"],"storage":[128,256]}
 ])
```

示例一：删除文档中的字段

以下示例使用 $unset 操作符删除文档（_id:1）中的 price 字段：

```
db.products.updateOne({
    _id: 1
}, {
    $unset: {
        price: ""
    }
})
```

返回结果如下：

```
{
  acknowledged: true,
  insertedId: null,
  matchedCount: 1,
  modifiedCount: 1,
  upsertedCount: 0
}
```

其中，modifiedCount 表示修改了一个文档。

查询集合 products 中的全部文档：

```
db.products.find({}, { name: 1, price: 1 })

[
  { _id: 1, name: 'xPhone' },
  { _id: 2, name: 'xTablet', price: 899 },
  { _id: 3, name: 'SmartTablet', price: 899 },
  { _id: 4, name: 'SmartPad', price: 699 },
  { _id: 5, name: 'SmartPhone', price: 599 }
]
```

从返回结果可以看出，文档（_id:1）中已经没有了 price 字段。

示例二：删除嵌入式文档中的字段

以下示例使用 $unset 操作符删除集合 products 中嵌入式文档 spec 中的 ram 字段：

```
db.products.updateMany({}, {
    $unset: {
        "spec.ram": ""
    }
})
```

返回结果如下：

```
{
```

```
  acknowledged: true,
  insertedId: null,
  matchedCount: 5,
  modifiedCount: 5,
  upsertedCount: 0
}
```

查询集合 products 中的全部文档：

```
db.products.find({}, {
    spec: 1
})

[
  { _id: 1, spec: { screen: 6.5, cpu: 2.66 } },
  { _id: 2, spec: { screen: 9.5, cpu: 3.66 } },
  { _id: 3, spec: { screen: 9.7, cpu: 3.66 } },
  { _id: 4, spec: { screen: 9.7, cpu: 1.66 } },
  { _id: 5, spec: { screen: 5.7, cpu: 1.66 } }
]
```

示例三：将数组元素设置为空

接下来的示例使用 $unset 操作符将数组 storage 中的第一个元素设置为 null：

```
db.products.updateMany({}, { $unset: { "storage.0": "" } })
```

返回结果如下：

```
{
  acknowledged: true,
  insertedId: null,
  matchedCount: 5,
  modifiedCount: 5,
  upsertedCount: 0
}
```

查看数组 storage 中的内容：

```
db.products.find({}, { "storage":1})

[
  { _id: 1, storage: [ null, 128, 256 ] },
  { _id: 2, storage: [ null, 256, 512 ] },
  { _id: 3, storage: [ null, 64, 128 ] },
  { _id: 4, storage: [ null, 256, 1024 ] },
  { _id: 5, storage: [ null, 256 ] }
]
```

示例四：删除多个字段

以下示例使用 $unset 操作符一次性删除了文档中的 releaseDate 和 spec 字段：

```
db.products.updateMany({}, {
    $unset: {
        releaseDate: "",
        spec: ""
    }
})
```

返回结果如下：

```
{
  acknowledged: true,
  insertedId: null,
  matchedCount: 5,
  modifiedCount: 5,
  upsertedCount: 0
}
```

验证更新后的文档内容：

```
db.products.find({}, {
    name: 1,
    storage: 1,
    releaseDate: 1,
    spec: 1
})

[
  { _id: 1, name: 'xPhone', storage: [ null, 128, 256 ] },
  { _id: 2, name: 'xTablet', storage: [ null, 256, 512 ] },
  { _id: 3, name: 'SmartTablet', storage: [ null, 64, 128 ] },
  { _id: 4, name: 'SmartPad', storage: [ null, 256, 1024 ] },
  { _id: 5, name: 'SmartPhone', storage: [ null, 256 ] }
]
```

查询返回的结果中没有 releaseDate 和 spec 字段。

第 19 篇 文档更新之$rename 操作符

本篇将会介绍 MongoDB $rename 操作符，它可以用于重命名文档中的字段。

19.1 $rename 操作符

$rename 是一个字段更新操作符，可以用于修改文档的字段名。

$rename 操作符的语法如下：

```
{ $rename: { <field_name>: <new_field_name>, ...}}
```

其中，新的字段名 必须和原字段名 不同。

如果文档中已经存在一个名为 的字段，$rename 操作符将会删除该字段，然后将字段 改名为 。

如果文档中不存在名为 的字段，$rename 操作符不会执行任何操作，也不会返回任何警告或者错误。

$rename 操作符可以修改嵌入式文档中的字段名，也可以将字段移入或者移出嵌入式文档。

19.2 $rename 示例

首先创建以下集合：

```
db.products.insertMany([
       { "_id" : 1, "nmea" : "xPhone", "price" : 799, "releaseDate":
ISODate("2011-05-14"), "spec" : { "ram" : 4, "screen" : 6.5, "cpu" :
2.66 },"color":["white","black"],"storage":[64,128,256]},
       { "_id" : 2, "nmea" : "xTablet", "price" : 899, "releaseDate":
ISODate("2011-09-01") , "spec" : { "ram" : 16, "screen" : 9.5, "cpu" :
3.66 },"color":["white","black","purple"],"storage":[128,256,512]},
       { "_id" : 3, "nmea" : "SmartTablet", "price" : 899, "releaseDate":
ISODate("2015-01-14"), "spec" : { "ram" : 12, "screen" : 9.7, "cpu" :
3.66 },"color":["blue"],"storage":[16,64,128]},
       { "_id" : 4, "nmea" : "SmartPad", "price" : 699, "releaseDate":
ISODate("2020-05-14"),"spec" : { "ram" : 8, "screen" : 9.7, "cpu" :
1.66 },"color":["white","orange","gold","gray"],"storage":[128,256,1024]},
       { "_id" : 5, "nmea" : "SmartPhone", "price" : 599,"releaseDate":
ISODate("2022-09-14"), "spec" : { "ram" : 4, "screen" : 5.7, "cpu" :
1.66 },"color":["white","orange","gold","gray"],"storage":[128,256]}
    ])
```

示例一：修改字段的名称

以下示例使用 $rename 操作符将拼写错误的字段名“nmea”修改为“name”：

```
db.products.updateMany({}, {
    $rename: {
        nmea: "name"
    }
})
```

返回的结果如下：

```
{
  acknowledged: true,
  insertedId: null,
  matchedCount: 5,
  modifiedCount: 5,
  upsertedCount: 0
}
```

查询 products 集合中的文档，验证修改后的字段名：

```
db.products.find({}, { name: 1 })

[
  { _id: 1, name: 'xPhone' },
  { _id: 2, name: 'xTablet' },
  { _id: 3, name: 'SmartTablet' },
  { _id: 4, name: 'SmartPad' },
  { _id: 5, name: 'SmartPhone' }
]
```

示例二：修改嵌入式文档中的字段名称

以下示例使用 $rename 操作符将嵌入式文档 spec 中的字段 size 重命名为 screenSize：

```
db.products.updateMany({}, {
    $rename: {
        "spec.screen": "spec.screenSize"
    }
})
```

返回结果如下：

```
{
  acknowledged: true,
  insertedId: null,
  matchedCount: 5,
  modifiedCount: 0,
  upsertedCount: 0
}
```

再次查询 products 集合中的文档：

```
db.products.find({}, {
    spec: 1
```

```
})

[
  { _id: 1, spec: { ram: 4, cpu: 2.66, screenSize: 6.5 } },
  { _id: 2, spec: { ram: 16, cpu: 3.66, screenSize: 9.5 } },
  { _id: 3, spec: { ram: 12, cpu: 3.66, screenSize: 9.7 } },
  { _id: 4, spec: { ram: 8, cpu: 1.66, screenSize: 9.7 } },
  { _id: 5, spec: { ram: 4, cpu: 1.66, screenSize: 5.7 } }
]
```

示例三：移出嵌入式文档中的字段

以下示例使用 $rename 操作符将文档（_id: 1）中的嵌入式文档 spec 的字段 cpu 修改为顶层字段：

```
db.products.updateOne({
 _id: 1
},
{
    $rename: {
        "spec.cpu": "cpu"
    }
})
```

返回结果如下：

```
{
  acknowledged: true,
  insertedId: null,
  matchedCount: 1,
  modifiedCount: 1,
  upsertedCount: 0
}
```

再次查询该文档：

```
db.products.find({ _id: 1})

[
  {
    _id: 1,
    price: 799,
    releaseDate: ISODate("2011-05-14T00:00:00.000Z"),
    spec: { ram: 4, screenSize: 6.5 },
    color: [ 'white', 'black' ],
    storage: [ 64, 128, 256 ],
    name: 'xPhone',
    cpu: 2.66
  }
]
```

示例四：替代已有字段

接下来的示例使用 $rename 操作符将文档（_id: 2）中的 color 字段重命名为 storage。不过，该文档中已经存在一个名为 storage 的字段。因此， $rename 操作符会将原有的 storage 字段删除，并且将原有的 color 字段重命名为 storage：

```
db.products.updateOne({
    _id: 2
}, {
    $rename: {
        "color": "storage"
    }
})
```

返回结果如下：

```
{
  acknowledged: true,
  insertedId: null,
  matchedCount: 1,
  modifiedCount: 0,
  upsertedCount: 0
}
```

查看该文档中的内容：

```
db.products.find({ _id: 2 })

[
  {
    _id: 2,
    price: 899,
    releaseDate: ISODate("2011-09-01T00:00:00.000Z"),
    spec: { ram: 16, cpu: 3.66, screenSize: 9.5 },
    storage: [ 'white', 'black', 'purple' ],
    name: 'xTablet'
  }
]
```

第 20 篇 文档更新之 UPSERT

第 14 篇中介绍了更新文档的 updateOne() 和 updateMany() 方法，它们还支持一种特殊的操作：更新插入（UPSERT）。

20.1 更新插入

更新插入包含了两个操作，更新文档和插入文档：

- 如果存在匹配的文档，更新该文档；
- 否则，插入一个新文档。

如果想要实现更新插入，可以将 updateOne() 或者 updateMany() 方法中的 upsert 选项设置为 true：

```
db.collection.updateOne(filter, update, { upsert: true} )

document.collection.updateMany(filter, update, { upsert: true} )
```

默认情况下，第三个参数中的 upsert 字段为 false。意味着只有匹配查询条件的文档会被更新。

20.2 示例

创建以下文档集合：

```
db.products.insertMany([
    { "_id" : 1, "nmea" : "xPhone", "price" : 799, "releaseDate":
ISODate("2011-05-14"), "spec" : { "ram" : 4, "screen" : 6.5, "cpu" :
2.66 },"color":["white","black"],"storage":[64,128,256]},
    { "_id" : 2, "nmea" : "xTablet", "price" : 899, "releaseDate":
ISODate("2011-09-01") , "spec" : { "ram" : 16, "screen" : 9.5, "cpu" :
3.66 },"color":["white","black","purple"],"storage":[128,256,512]},
    { "_id" : 3, "nmea" : "SmartTablet", "price" : 899, "releaseDate":
ISODate("2015-01-14"), "spec" : { "ram" : 12, "screen" : 9.7, "cpu" :
3.66 },"color":["blue"],"storage":[16,64,128]},
    { "_id" : 4, "nmea" : "SmartPad", "price" : 699, "releaseDate":
ISODate("2020-05-14"),"spec" : { "ram" : 8, "screen" : 9.7, "cpu" :
1.66 },"color":["white","orange","gold","gray"],"storage":[128,256,1024]},
    { "_id" : 5, "nmea" : "SmartPhone", "price" : 599,"releaseDate":
ISODate("2022-09-14"), "spec" : { "ram" : 4, "screen" : 5.7, "cpu" :
1.66 },"color":["white","orange","gold","gray"],"storage":[128,256]}
])
```

下面的查询使用 updateMany() 方法更新文档（_id: 6）的 price 字段：

```
db.products.updateMany(
    {_id: 6 },
```

```
    { $set: {price: 999} }
)
```

查询没有匹配任何文档，因此不会更新任何文档：

```
{
  acknowledged: true,
  insertedId: null,
  matchedCount: 0,
  modifiedCount: 0,
  upsertedCount: 0
}
```

如果将以上示例中 updateMany() 方法的 upsert 选项设置为 true，将会插入一个新文档。例如：

```
db.products.updateMany(
    { _id: 6 },
    { $set: {price: 999} },
    { upsert: true}
)
```

查询返回的结果如下：

```
{
  acknowledged: true,
  insertedId: 6,
  matchedCount: 0,
  modifiedCount: 0,
  upsertedCount: 1
}
```

结果显示没有匹配任何文档（matchedCount: 0），updateMany() 方法没有更新任何文档。但是该方法插入了一个文档并返回了新文档的 id（upsertedId: 6）。

查询集合 products 中的文档，可以返回新的文档：

```
db.products.find({_id:6})

[ { _id: 6, price: 999 } ]
```

第 21 篇 CRUD 之删除文档

本篇将会介绍如何利用 deleteOne() 和 deleteMany() 方法删除满足指定条件的文档。

21.1 deleteOne() 方法

deleteOne() 方法用于删除集合中的单个文档，语法如下：

```
db.collection.deleteOne(filter, option)
```

该方法包含两个参数：

- filter 是一个必选参数，用于指定一个查询条件，满足条件的文档才会被删除。如果指定一个空文档（{}）作为查询条件，将会删除集合中的第一个文档。
- option 是一个可选参数，用于指定删除选项，本篇不涉及相关内容。

deleteOne() 方法返回一个结果文档，其中的 deleteCount 字段存储了被删除文档的数量。

21.2 deleteOne() 示例

本篇将会使用以下示例文档：

```
db.products.insertMany([
    { "_id" : 1, "name" : "xPhone", "price" : 799, "releaseDate":
ISODate("2011-05-14"), "spec" : { "ram" : 4, "screen" : 6.5, "cpu" :
2.66 },"color":["white","black"],"storage":[64,128,256]},
    { "_id" : 2, "name" : "xTablet", "price" : 899, "releaseDate":
ISODate("2011-09-01") , "spec" : { "ram" : 16, "screen" : 9.5, "cpu" :
3.66 },"color":["white","black","purple"],"storage":[128,256,512]},
    { "_id" : 3, "name" : "SmartTablet", "price" : 899, "releaseDate":
ISODate("2015-01-14"), "spec" : { "ram" : 12, "screen" : 9.7, "cpu" :
3.66 },"color":["blue"],"storage":[16,64,128]},
    { "_id" : 4, "name" : "SmartPad", "price" : 699, "releaseDate":
ISODate("2020-05-14"),"spec" : { "ram" : 8, "screen" : 9.7, "cpu" :
1.66 },"color":["white","orange","gold","gray"],"storage":[128,256,1024]},
    { "_id" : 5, "name" : "SmartPhone", "price" : 599,"releaseDate":
ISODate("2022-09-14"), "spec" : { "ram" : 4, "screen" : 9.7, "cpu" :
1.66 },"color":["white","orange","gold","gray"],"storage":[128,256]}
])
```

示例一：删除单个文档

以下示例使用 deleteOne() 方法删除 products 集合中 _id 等于 1 的文档：

```
db.products.deleteOne({ _id: 1 })
```

查询返回的结果如下：

```
{ "acknowledged" : true, "deletedCount" : 1 }
```

其中，deleteCount 字段的值为 1，表示删除了一个文档。

示例二：删除第一个文档

以下查询使用 deleteOne() 方法删除 products 集合中的第一个文档：

```
db.products.deleteOne({})
```

返回结果如下：

```
{ "acknowledged" : true, "deletedCount" : 1 }
```

21.3 deleteMany() 方法

deleteMany() 方法用于删除满足条件的所有文档，语法如下：

```
db.collection.deleteMany(filter, option)
```

其中，

- filter 是一个必选参数，用于指定一个查询条件，满足条件的文档才会被删除。如果指定一个空文档（{}）作为查询条件，将会删除集合中的全部文档。
- option 是一个可选参数，用于指定删除选项，本篇不涉及相关内容。

21.4 deleteMany() 示例

示例一：删除多个文档

以下示例使用 deleteMany() 方法删除 price 等于 899 的所有文档：

```
db.products.deleteMany({ price: 899 })
```

返回结果如下：

```
{ "acknowledged" : true, "deletedCount" : 2 }
```

deleteCount 字段表示删除了 2 个文档。

示例二：删除全部文档

以下示例使用 deleteMany() 方法删除了 products 集合中的全部文档：

```
db.products.deleteMany({})
```

返回结果如下：

```
{ acknowledged: true, deletedCount: 3 }
```

第 22 篇 聚合操作

本篇将会介绍如何使用 MongoDB 聚合操作对文档进行分组，以及聚合表达式的使用。

22.1 聚合操作

MongoDB 聚合操作可以处理多个文档并返回计算后的结果。聚合操作通常用于按照指定字段的值进行分组并计算汇总结果。例如，聚合操作可以按照不同的产品计算订单中的总销售额。

聚合管道用于执行聚合操作，一个聚合管道包含一个或者多个处理文档的阶段。每个阶段都会基于它的输入文档执行操作，然后返回输出文档；输出文档会传递给下一个阶段，最后一个阶段返回最终的结果。

每个阶段的操作可以是以下内容之一：

- $project - 选择输出结果中包含的字段；
- $match - 选择要处理的文档；
- $limit - 限制传递到下一阶段的文档数量；
- $skip - 忽略指定数量的文档；
- $sort - 文档排序；
- $group - 文档分组；
- …

以下是定义聚合管道的语法：

```
db.collection.aggregate([{ $match:...},{$group:...},{$sort:...}]);
```

其中，

- aggregate() 方法用于指定聚合操作；
- 传递一个文档数组作为参数，每个文档描述了管道中的一个阶段。

MongoDB 4.2 及更高版本支持使用聚合管道更新文档。

22.2 聚合示例

首先，使用数据库 coffeeshop 存储咖啡的销量信息：

```
use coffeeshop
```

其次，为集合 sales 创建一些测试文档：

```
db.sales.insertMany([
    { "_id" : 1, "item" : "Americanos", "price" : 5, "size": "Short",
"quantity" : 22, "date" : ISODate("2022-01-15T08:00:00Z") },
    { "_id" : 2, "item" : "Cappuccino", "price" : 6, "size":
"Short","quantity" : 12, "date" : ISODate("2022-01-16T09:00:00Z") },
    { "_id" : 3, "item" : "Lattes", "price" : 15, "size":
"Grande","quantity" : 25, "date" : ISODate("2022-01-16T09:05:00Z") },
    { "_id" : 4, "item" : "Mochas", "price" : 25,"size": "Tall", "quantity" :
11, "date" : ISODate("2022-02-17T08:00:00Z") },
    { "_id" : 5, "item" : "Americanos", "price" : 10, "size":
"Grande","quantity" : 12, "date" : ISODate("2022-02-18T21:06:00Z") },
    { "_id" : 6, "item" : "Cappuccino", "price" : 7, "size":
"Tall","quantity" : 20, "date" : ISODate("2022-02-20T10:07:00Z") },
    { "_id" : 7, "item" : "Lattes", "price" : 25,"size": "Tall", "quantity" :
30, "date" : ISODate("2022-02-21T10:08:00Z") },
    { "_id" : 8, "item" : "Americanos", "price" : 10, "size":
"Grande","quantity" : 21, "date" : ISODate("2022-02-22T14:09:00Z") },
    { "_id" : 9, "item" : "Cappuccino", "price" : 10, "size":
"Grande","quantity" : 17, "date" : ISODate("2022-02-23T14:09:00Z") },
    { "_id" : 10, "item" : "Americanos", "price" : 8, "size":
"Tall","quantity" : 15, "date" : ISODate("2022-02-25T14:09:00Z")}
]);
```

然后，使用聚合管道查找“Americanos”品牌的咖啡的销量信息，按照杯型分组统计销量，最后按照销量总高到低进行排序：

```
db.sales.aggregate([
    {
        $match: { item: "Americanos" }
    },
    {
        $group: {
            _id: "$size",
            totalQty: {$sum: "$quantity"}
        }
    },
    {
        $sort: { totalQty : -1}
    }
]);
```

返回结果如下：

```
[
  { _id: 'Grande', totalQty: 33 },
  { _id: 'Short', totalQty: 22 },
  { _id: 'Tall', totalQty: 15 }
]
```

示例中的聚合管道包含以下三个阶段：

- 阶段 1：$match 阶段过滤品牌为“Americanos”的咖啡，将结果传递给 $group 阶段；
- 阶段 2：$group 阶段按照咖啡杯型进行分组，使用 $sum 计算每个组的总销量。该阶段创建了一个新的文档集合，每个文档包含了 _id 和 totalQty 两个字段，然后将结果文档传递给 $sort 阶段；
- 阶段 3：$sort 阶段基于 totalQty 字段从大到小进行排序并返回结果文档。

22.3 SQL 与 MongoDB 聚合操作对比

以上聚合管道的等价 SQL 语句如下：

```
select
   name as _id,
   sum(quantity) as totalQty
from
   sales
where name = 'Americanos'
group by name
order by totalQty desc;
```

下表列出了 SQL 和 MongoDB 聚合操作的比对：

SQL 子句	**MongoDB 聚合**
select	$project
from	db.collection.aggregate(…)
join	$unwind
where	$match
group by	$group
聚合函数：avg、count、sum、max、min	表达式：$avg、$count、$sum、$max、$min
having	$match

第 23 篇 聚合统计之$sum 表达式

本篇将会介绍如何利用 $sum 表达式计算一组数值的总和。

23.1 $sum 表达式

MongoDB 支持 $sum 表达式，用于返回一组数值的总和。该表达式的语法如下：

```
{ $sum: <expression> }
```

如果汇总的字段包含非数值内容， $sum 表达式会忽略相应的内容。如果所有文档中都不存在指定的汇总字段，$sum 表达式的结果为 0。

23.2 $sum 示例

首先创建一个 sales 集合：

```
db.sales.insertMany([
    { "_id" : 1, "item" : "Americanos", "price" : 5, "size": "Short",
"quantity" : 22, "date" : ISODate("2022-01-15T08:00:00Z") },
    { "_id" : 2, "item" : "Cappuccino", "price" : 6, "size":
"Short","quantity" : 12, "date" : ISODate("2022-01-16T09:00:00Z") },
    { "_id" : 3, "item" : "Lattes", "price" : 15, "size":
"Grande","quantity" : 25, "date" : ISODate("2022-01-16T09:05:00Z") },
    { "_id" : 4, "item" : "Mochas", "price" : 25,"size": "Tall", "quantity" :
11, "date" : ISODate("2022-02-17T08:00:00Z") },
    { "_id" : 5, "item" : "Americanos", "price" : 10, "size":
"Grande","quantity" : 12, "date" : ISODate("2022-02-18T21:06:00Z") },
    { "_id" : 6, "item" : "Cappuccino", "price" : 7, "size":
"Tall","quantity" : 20, "date" : ISODate("2022-02-20T10:07:00Z") },
    { "_id" : 7, "item" : "Lattes", "price" : 25,"size": "Tall", "quantity" :
30, "date" : ISODate("2022-02-21T10:08:00Z") },
    { "_id" : 8, "item" : "Americanos", "price" : 10, "size":
"Grande","quantity" : 21, "date" : ISODate("2022-02-22T14:09:00Z") },
    { "_id" : 9, "item" : "Cappuccino", "price" : 10, "size":
"Grande","quantity" : 17, "date" : ISODate("2022-02-23T14:09:00Z") },
    { "_id" : 10, "item" : "Americanos", "price" : 8, "size":
"Tall","quantity" : 15, "date" : ISODate("2022-02-25T14:09:00Z")}
]);
```

示例一：简单汇总

以下示例计算了全部咖啡的总销量：

```
db.sales.aggregate([
  {
    $group: {
      _id: null,
```

```
        totalQty: { $sum: '$quantity' },
      },
    },
  ]);
```

输出结果如下：

```
[ { _id: null, totalQty: 185 } ]
```

以下示例使用 $project 操作删除了返回结果中的 _id：

```
db.sales.aggregate([
  {
    $group: {
      _id: null,
      totalQty: { $sum: '$quantity' },
    },
  },
  { $project: { _id: 0 } },
]);


[ { totalQty: 185 } ]
```

示例二：分组汇总

以下示例使用 $sum 表达式计算了不同产品的销量汇总：

```
db.sales.aggregate([
  {
    $group: {
      _id: '$item',
      totalQty: { $sum: '$quantity' },
    },
  },
]);
```

返回结果如下：

```
[
  { _id: 'Cappuccino', totalQty: 49 },
  { _id: 'Lattes', totalQty: 55 },
  { _id: 'Americanos', totalQty: 70 },
  { _id: 'Mochas', totalQty: 11 }
]
```

示例三：汇总与排序

以下示例使用 $sum 表达式计算了不同产品的销量汇总，并且基于每个产品的总销量（totalQty）从高到低进行了排序：

```
db.sales.aggregate([
  {
```

```
    $group: {
      _id: '$item',
      totalQty: { $sum: '$quantity' },
    },
  },
  { $sort: { totalQty: -1 } },
]);
```

返回结果如下：

```
[
  { _id: 'Americanos', totalQty: 70 },
  { _id: 'Lattes', totalQty: 55 },
  { _id: 'Cappuccino', totalQty: 49 },
  { _id: 'Mochas', totalQty: 11 }
]
```

示例四：汇总与过滤

以下示例使用 $sum 表达式计算了不同产品的销量汇总，并且返回了总销量大于 50 的产品：

```
db.sales.aggregate([
  {
    $group: {
      _id: '$item',
      totalQty: { $sum: '$quantity' },
    },
  },
  { $match: { totalQty: { $gt: 50 } } },
  { $sort: { totalQty: -1 } },
]);
```

返回结果如下：

```
[
  { _id: 'Americanos', totalQty: 70 },
  { _id: 'Lattes', totalQty: 55 }
]
```

示例五：基于表达式的汇总

以下示例使用 $sum 表达式计算了不同杯型咖啡的销售金额汇总，销售金额等于销量乘以价格：

```
db.sales.aggregate([
  {
    $group: {
      _id: '$size',
      totalAmount: { $sum: { $multiply: ['$price', '$quantity'] } },
    },
  },
```

```
    { $sort: { totalAmount: -1 } },
  ]);
```

返回结果如下：

```
  [
    { _id: 'Tall', totalAmount: 1285 },
    { _id: 'Grande', totalAmount: 875 },
    { _id: 'Short', totalAmount: 182 }
  ]
```

第 24 篇 聚合统计之$count 表达式

本篇将会介绍 $count 表达式，它可以返回一组文档的数量。

24.1 $count 表达式

MongoDB $count 表达式的作用就是返回文档的数量，语法如下：

```
{ $count: {} }
```

$count 表达式不需要任何参数。

$count 表达式等价于以下形式的 $sum 表达式：

```
{ $sum: 1 }
```

24.2 $count 示例

接下来我们将会使用以下集合进行演示：

```
db.sales.insertMany([
    { "_id" : 1, "item" : "Americanos", "price" : 5, "size": "Short",
"quantity" : 22, "date" : ISODate("2022-01-15T08:00:00Z") },
    { "_id" : 2, "item" : "Cappuccino", "price" : 6, "size":
"Short","quantity" : 12, "date" : ISODate("2022-01-16T09:00:00Z") },
    { "_id" : 3, "item" : "Lattes", "price" : 15, "size":
"Grande","quantity" : 25, "date" : ISODate("2022-01-16T09:05:00Z") },
    { "_id" : 4, "item" : "Mochas", "price" : 25,"size": "Tall", "quantity" :
11, "date" : ISODate("2022-02-17T08:00:00Z") },
    { "_id" : 5, "item" : "Americanos", "price" : 10, "size":
"Grande","quantity" : 12, "date" : ISODate("2022-02-18T21:06:00Z") },
    { "_id" : 6, "item" : "Cappuccino", "price" : 7, "size":
"Tall","quantity" : 20, "date" : ISODate("2022-02-20T10:07:00Z") },
    { "_id" : 7, "item" : "Lattes", "price" : 25,"size": "Tall", "quantity" :
30, "date" : ISODate("2022-02-21T10:08:00Z") },
    { "_id" : 8, "item" : "Americanos", "price" : 10, "size":
"Grande","quantity" : 21, "date" : ISODate("2022-02-22T14:09:00Z") },
    { "_id" : 9, "item" : "Cappuccino", "price" : 10, "size":
"Grande","quantity" : 17, "date" : ISODate("2022-02-23T14:09:00Z") },
    { "_id" : 10, "item" : "Americanos", "price" : 8, "size":
"Tall","quantity" : 15, "date" : ISODate("2022-02-25T14:09:00Z")}
]);
```

示例一：分组统计文档的数量

以下示例使用 $count 表达式计算不同种类咖啡的数量：

```
db.sales.aggregate([
  {
    $group: {
```

```
        _id: '$item',
        itemCount: { $count: {} },
      },
    },
  ])
```

返回结果如下：

```
  [
    { _id: 'Mochas', itemCount: 1 },
    { _id: 'Americanos', itemCount: 4 },
    { _id: 'Lattes', itemCount: 2 },
    { _id: 'Cappuccino', itemCount: 3 }
  ]
```

其中，

- _id: “$item” 用于将文档按照 item 字段进行分组，返回 4 个组；
- $count: {} 用于统计每个分组内的文档数据，并将结果赋予 itemCount 字段。

示例二：统计与过滤

以下示例使用 $count 表达式计算不同种类咖啡的数量，并且返回数量大于 2 的结果：

```
  db.sales.aggregate([
    {
      $group: {
        _id: '$item',
        itemCount: { $count: {} },
      },
    },
    {
      $match: { itemCount: { $gt: 2 } },
    },
  ]);
```

返回结果如下：

```
  [
    { _id: 'Americanos', itemCount: 4 },
    { _id: 'Cappuccino', itemCount: 3 }
  ]
```

第 25 篇 聚合统计之$avg 表达式

本篇将会介绍如何利用 $avg 表达式返回一组数字的平均值。

25.1 $avg 表达式

MongoDB $avg 表达式的作用是返回一组数据的平均值，语法如下：

```
{ $avg: <expression> }
```

$avg 表达式将会忽略任何非数字数据和缺失的数据。如果所有的数据都是非数字数据，表达式将会返回 null。

25.2 $avg 示例

以下示例将会使用 sales 集合：

```
db.sales.insertMany([
      { "_id" : 1, "item" : "Americanos", "price" : 5, "size": "Short",
"quantity" : 22, "date" : ISODate("2022-01-15T08:00:00Z") },
      { "_id" : 2, "item" : "Cappuccino", "price" : 6, "size":
"Short","quantity" : 12, "date" : ISODate("2022-01-16T09:00:00Z") },
      { "_id" : 3, "item" : "Lattes", "price" : 15, "size":
"Grande","quantity" : 25, "date" : ISODate("2022-01-16T09:05:00Z") },
      { "_id" : 4, "item" : "Mochas", "price" : 25,"size": "Tall", "quantity" :
11, "date" : ISODate("2022-02-17T08:00:00Z") },
      { "_id" : 5, "item" : "Americanos", "price" : 10, "size":
"Grande","quantity" : 12, "date" : ISODate("2022-02-18T21:06:00Z") },
      { "_id" : 6, "item" : "Cappuccino", "price" : 7, "size":
"Tall","quantity" : 20, "date" : ISODate("2022-02-20T10:07:00Z") },
      { "_id" : 7, "item" : "Lattes", "price" : 25,"size": "Tall", "quantity" :
30, "date" : ISODate("2022-02-21T10:08:00Z") },
      { "_id" : 8, "item" : "Americanos", "price" : 10, "size":
"Grande","quantity" : 21, "date" : ISODate("2022-02-22T14:09:00Z") },
      { "_id" : 9, "item" : "Cappuccino", "price" : 10, "size":
"Grande","quantity" : 17, "date" : ISODate("2022-02-23T14:09:00Z") },
      { "_id" : 10, "item" : "Americanos", "price" : 8, "size":
"Tall","quantity" : 15, "date" : ISODate("2022-02-25T14:09:00Z")}
   ]);
```

示例一：平均销售数量

以下示例将文档按照 T item 字段进行分组，然后使用 $avg 表达式计算每个组的平均销售数量：

```
db.sales.aggregate([
  {
    $group: {
      _id: '$item',
```

```
        averageQty: { $avg: '$quantity' },
      },
    },
  ]);
```

输出结果如下：

```
[
  { _id: 'Americanos', averageQty: 17.5 },
  { _id: 'Lattes', averageQty: 27.5 },
  { _id: 'Mochas', averageQty: 11 },
  { _id: 'Cappuccino', averageQty: 16.333333333333332 }
]
```

示例二：平均销售金额

以下示例将文档按照 T item 字段进行分组，然后使用 $avg 表达式计算每个组的平均销售金额：

```
db.sales.aggregate([
  {
    $group: {
      _id: '$item',
      averageAmount: { $avg: { $multiply: ['$quantity', '$price'] } },
    },
  },
  { $sort: { averageAmount: 1 } },
])
```

返回结果如下：

```
[
  { _id: 'Cappuccino', averageAmount: 127.33333333333333 },
  { _id: 'Americanos', averageAmount: 140 },
  { _id: 'Mochas', averageAmount: 275 },
  { _id: 'Lattes', averageAmount: 562.5 }
]
```

示例三：平均值与过滤

以下示例使用 $avg 表达式计算每个组的平均销售金额，并且返回了平均销售金额大于 150 的产品：

```
db.sales.aggregate([
  {
    $group: {
      _id: '$item',
      averageAmount: { $avg: { $multiply: ['$quantity', '$price'] } },
    },
  },
  { $match: { averageAmount: { $gt: 150 } } },
  { $sort: { averageAmount: 1 } },
```

```
]);
```

返回的结果如下：

```
[
  { _id: 'Mochas', averageAmount: 275 },
  { _id: 'Lattes', averageAmount: 562.5 }
]
```

第 26 篇 聚合统计之*max*/min 表达式

本篇将会介绍两个 MongoDB 表达式，返回一组数据中最大值的 $max 表达式，以及返回一组数据中最小值的 $min 表达式。

26.1 $max 表达式

$max 表达式用于返回一组数据中的最大值，语法如下：

```
{ $max: <expression> }
```

$max 表达式在执行操作时会忽略 null 或者缺失的数据。

如果表达式的参数全部为 null 或者缺失的数据，$max 表达式将会返回 null。

26.2 $max 示例

首先创建以下 sales 集合：

```
db.sales.insertMany([
    { "_id" : 1, "item" : "Americanos", "price" : 5, "size": "Short",
"quantity" : 22, "date" : ISODate("2022-01-15T08:00:00Z") },
    { "_id" : 2, "item" : "Cappuccino", "price" : 6, "size":
"Short","quantity" : 12, "date" : ISODate("2022-01-16T09:00:00Z") },
    { "_id" : 3, "item" : "Lattes", "price" : 15, "size":
"Grande","quantity" : 25, "date" : ISODate("2022-01-16T09:05:00Z") },
    { "_id" : 4, "item" : "Mochas", "price" : 25,"size": "Tall", "quantity" :
11, "date" : ISODate("2022-02-17T08:00:00Z") },
    { "_id" : 5, "item" : "Americanos", "price" : 10, "size":
"Grande","quantity" : 12, "date" : ISODate("2022-02-18T21:06:00Z") },
    { "_id" : 6, "item" : "Cappuccino", "price" : 7, "size":
"Tall","quantity" : 20, "date" : ISODate("2022-02-20T10:07:00Z") },
    { "_id" : 7, "item" : "Lattes", "price" : 25,"size": "Tall", "quantity" :
30, "date" : ISODate("2022-02-21T10:08:00Z") },
    { "_id" : 8, "item" : "Americanos", "price" : 10, "size":
"Grande","quantity" : 21, "date" : ISODate("2022-02-22T14:09:00Z") },
    { "_id" : 9, "item" : "Cappuccino", "price" : 10, "size":
"Grande","quantity" : 17, "date" : ISODate("2022-02-23T14:09:00Z") },
    { "_id" : 10, "item" : "Americanos", "price" : 8, "size":
"Tall","quantity" : 15, "date" : ISODate("2022-02-25T14:09:00Z")}
]);
```

以下示例使用 $max 表达式查找所有文档中的最大销量：

```
db.sales.aggregate([
  {
    $group: {
      _id: null,
      maxQty: { $max: '$quantity' },
```

```
    },
  },
  {
    $project: {
      _id: 0,
    },
  },
]);
```

返回结果如下：

```
[ { maxQty: 30 } ]
```

以下示例使用 $max 表达式查找不同产品的最大销量：

```
db.sales.aggregate([
  {
    $group: {
      _id: '$item',
      maxQty: { $max: '$quantity' },
    },
  },
]);
```

返回结果如下：

```
[
  { _id: 'Mochas', maxQty: 11 },
  { _id: 'Americanos', maxQty: 22 },
  { _id: 'Lattes', maxQty: 30 },
  { _id: 'Cappuccino', maxQty: 20 }
]
```

以下示例使用 $max 表达式查找不同产品的最大销售金额：

```
db.sales.aggregate([
  {
    $group: {
      _id: '$item',
      maxQty: { $max: { $multiply: ['$quantity', '$price'] } },
    },
  },
]);
```

返回结果如下：

```
[
  { _id: 'Mochas', maxQty: 275 },
  { _id: 'Cappuccino', maxQty: 170 },
  { _id: 'Americanos', maxQty: 210 },
  { _id: 'Lattes', maxQty: 750 }
]
```

$min 表达式

MongoDB $min 表达式可以返回一组数据中的最小值，语法如下：

```
{ $min: <expression> }
```

$min 表达式在执行操作时会忽略 null 或者缺失的数据。

如果表达式的参数全部为 null 或者缺失的数据，$min 表达式将会返回 null。

$min 示例

以下示例使用 $min 表达式查找所有文档中的最小销量：

```
db.sales.aggregate([
  {
    $group: {
      _id: null,
      maxQty: { $min: '$quantity' },
    },
  },
  {
    $project: {
      _id: 0,
    },
  },
]);
```

返回结果如下：

```
[ { minQty: 11 } ]
```

以下示例使用 $max 表达式查找不同产品的最小销量：

```
db.sales.aggregate([
  {
    $group: {
      _id: '$item',
      minQty: { $min: '$quantity' },
    },
  },
]);
```

返回结果如下：

```
[
  { _id: 'Mochas', minQty: 11 },
  { _id: 'Americanos', minQty: 12 },
  { _id: 'Lattes', minQty: 25 },
  { _id: 'Cappuccino', minQty: 12 }
]
```

以下示例使用 $max 表达式查找不同产品的最小销售金额：

```
db.sales.aggregate([
  {
    $group: {
      _id: '$item',
      maxQty: { $min: { $multiply: ['$quantity', '$price'] } },
    },
  },
]);
```

返回结果如下：

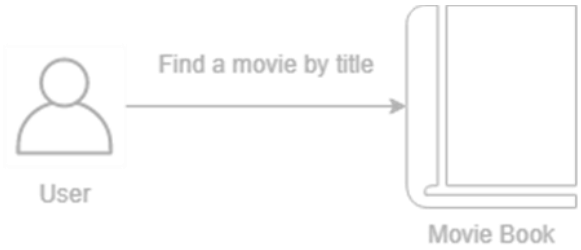
```
[
  { _id: 'Cappuccino', minQty: 72 },
  { _id: 'Americanos', minQty: 110 },
  { _id: 'Lattes', minQty: 375 },
  { _id: 'Mochas', minQty: 275 }
]
```

第 27 篇 创建索引

本篇将会介绍 MongoDB 中的索引概念，以及如何利用 createIndex() 方法创建索引。

27.1 索引简介

假设存在一本包含介绍各种电影的图书。



如果想要查找一部名为“Pirates of Silicon Valley”的电影，我们需要翻阅每一页，直到发现该电影的介绍为止。

显然，这种查找方法效率低下。如果这本书包含一个内容索引，记录了电影的标题及对应的页码，我们就可以通过索引快速找到相应的电影介绍：

```
Pimpernel' Smith              1
...
Pirates of Silicon Valley     201
...
Twas the Night                300
```

在上面的索引信息中，电影“Pirates of Silicon Valley”位于这本书的 201 页。因此，我们可以直接打开 201 页查看相应的介绍：

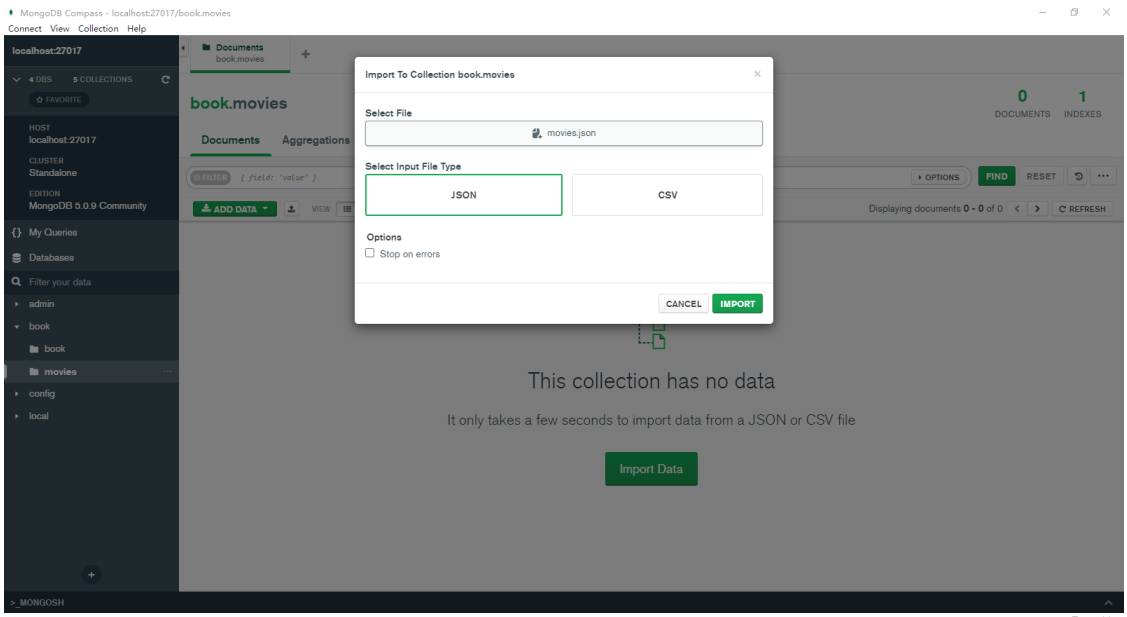
从这个示例可以看出，索引可以更快地查找数据。

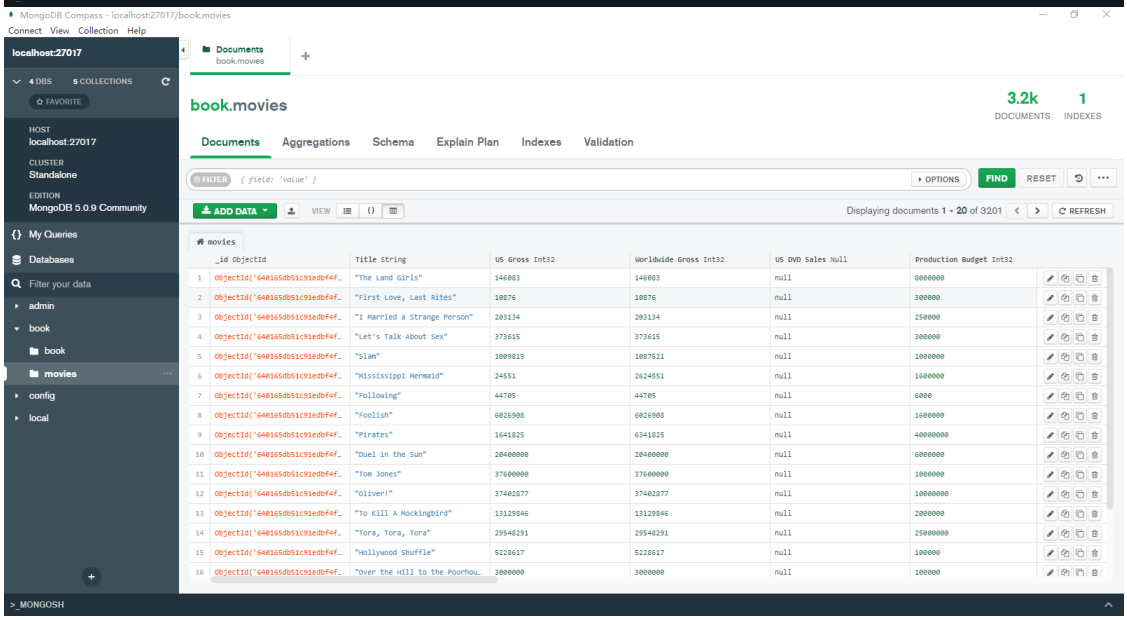
MongoDB 索引的工作方式和上面的示例类似，我们可以基于文档集合中的指定字段创建索引。MongoDB 使用 B-树 结构存储索引。

另一方面，当我们插入、更新或者删除文档时， MongoDB 需要维护相应的索引。也就是说，索引提高了文件的检索性能，但是需要以额外的写入和存储空间为代价。因此，应该建立合适的索引，而不是尽可能多的索引。

27.2 查看索引

我们首先创建一个新的 movies 集合，然后通过 MongoDB Compass 导入初始化数据（movies.json）：





默认情况下，所有的集合都拥有一个 _id 字段索引。getIndexes() 方法可以用于查看集合中的索引，语法如下：

```
db.collection.getIndexes()
```

以下命令可以查看集合 movies 中的索引：

```
db.movies.getIndexes()

[ { v: 2, key: { _id: 1 }, name: '_id_' } ]
```

输出结果中的索引名称为“_id_”，索引字段为 _id。{ _id : 1 } 中的数字 1 代表了升序索引。

27.3 执行计划

以下查询用于查找电影“The Lake House”：

```
db.movies.find({
   Title: 'The Lake House'
})

{ _id: ObjectId("640165db51c91edbf4fa413f"),
  Title: 'The Lake House',
  'US Gross': 52330111,
  'Worldwide Gross': 114830111,
  'US DVD Sales': 39758509,
  'Production Budget': 40000000,
  'Release Date': 'Jun 16 2006',
  'MPAA Rating': 'PG',
  'Running Time min': null,
  Distributor: 'Warner Bros.',
  Source: 'Remake',
  'Major Genre': 'Drama',
  'Creative Type': 'Fantasy',
  Director: null,
  'Rotten Tomatoes Rating': 36,
  'IMDB Rating': 6.8,
  'IMDB Votes': 36613 }
```

为了查找该电影，MongoDB 需要扫描 movies 集合。在执行查询之前，MongoDB 查询计划器会选择最有效的执行计划。explain() 方法可以用于获取执行计划的相关信息。例如：

```
db.movies.find({
   Title: 'The Lake House'
}).explain('executionStats')

{ explainVersion: '1',
  queryPlanner:
   { namespace: 'book.movies',
```

```
    indexFilterSet: false,
    parsedQuery: { Title: { '$eq': 'The Lake House' } },
    maxIndexedOrSolutionsReached: false,
    maxIndexedAndSolutionsReached: false,
    maxScansToExplodeReached: false,
    winningPlan:
     { stage: 'COLLSCAN',
       filter: { Title: { '$eq': 'The Lake House' } },
       direction: 'forward' },
    rejectedPlans: [] },
  executionStats:
   { executionSuccess: true,
     nReturned: 1,
     executionTimeMillis: 2,
     totalKeysExamined: 0,
     totalDocsExamined: 3201,
     executionStages:
      { stage: 'COLLSCAN',
        filter: { Title: { '$eq': 'The Lake House' } },
        nReturned: 1,
        executionTimeMillisEstimate: 0,
        works: 3203,
        advanced: 1,
        needTime: 3201,
        needYield: 0,
        saveState: 4,
        restoreState: 4,
        isEOF: 1,
        direction: 'forward',
        docsExamined: 3201 } },
  command:
   { find: 'movies',
     filter: { Title: 'The Lake House' },
     '$db': 'book' },
  serverInfo:
   { host: 'LAPTOP-DGRB6HD9',
     port: 27017,
     version: '5.0.9',
     gitVersion: '6f7dae919422dcd7f4892c10ff20cdc721ad00e6' },
  serverParameters:
   { internalQueryFacetBufferSizeBytes: 104857600,
     internalQueryFacetMaxOutputDocSizeBytes: 104857600,
     internalLookupStageIntermediateDocumentMaxSizeBytes: 104857600,
     internalDocumentSourceGroupMaxMemoryBytes: 104857600,
     internalQueryMaxBlockingSortMemoryUsageBytes: 104857600,
     internalQueryProhibitBlockingMergeOnMongoS: 0,
     internalQueryMaxAddToSetBytes: 104857600,
     internalDocumentSourceSetWindowFieldsMaxMemoryBytes: 104857600 },
  ok: 1 }
```

explain() 方法返回了大量的信息，我们首先需要关注 winningPlan 部分：

```
...
     winningPlan:
      { stage: 'COLLSCAN',
        filter: { Title: { '$eq': 'The Lake House' } },
        direction: 'forward' },
...
```

winningPlan 返回了查询优化器最终选择的执行计划。示例中的 COLLSCAN 代表了集合扫描。

另外，executionStats 显示查询结果中包含 1 个文档，执行时间为 2 毫秒。

27.4 创建索引

createIndex() 方法可以用于创建新的索引。例如，以下命令可以为 movies 集合的 Title 字段创建索引：

```
db.movies.createIndex({Title:1})

'Title_1'
```

参数 { Title: 1} 包含了字段名和一个数值：

- Title 字段是索引键；
- 数值 1 表示按照字段的值从小到大创建升序索引，-1 表示从大到小创建降序索引。

createIndex() 方法返回了索引的名称。示例中创建的索引 Title_1 由字段名和数值 1（表示升序）组成。

再次查看集合 movies 中的索引：

```
db.movies.getIndexes()

[
  { v: 2, key: { _id: 1 }, name: '_id_' },
  { v: 2, key: { Title: 1 }, name: 'Title_1' }
]
```

再次使用 explain() 方法查看上文中查询语句的执行计划和统计信息：

```
db.movies.find({
   Title: 'The Lake House'
}).explain('executionStats')

{ explainVersion: '1',
  queryPlanner:
   { namespace: 'book.movies',
     indexFilterSet: false,
     parsedQuery: { Title: { '$eq': 'The Lake House' } },
     maxIndexedOrSolutionsReached: false,
```

```
        maxIndexedAndSolutionsReached: false,
        maxScansToExplodeReached: false,
        winningPlan:
         { stage: 'FETCH',
           inputStage:
            { stage: 'IXSCAN',
              keyPattern: { Title: 1 },
              indexName: 'Title_1',
              isMultiKey: false,
              multiKeyPaths: { Title: [] },
              isUnique: false,
              isSparse: false,
              isPartial: false,
              indexVersion: 2,
              direction: 'forward',
              indexBounds:  {  Title:  [  '["The  Lake  House",  "The  Lake
House"]' ] } } },
        rejectedPlans: [] },
     executionStats:
      { executionSuccess: true,
        nReturned: 1,
        executionTimeMillis: 0,
        totalKeysExamined: 1,
        totalDocsExamined: 1,
        executionStages:
         { stage: 'FETCH',
           nReturned: 1,
           executionTimeMillisEstimate: 0,
           works: 2,
           advanced: 1,
           needTime: 0,
           needYield: 0,
           saveState: 0,
           restoreState: 0,
           isEOF: 1,
           docsExamined: 1,
           alreadyHasObj: 0,
           inputStage:
            { stage: 'IXSCAN',
              nReturned: 1,
              executionTimeMillisEstimate: 0,
              works: 2,
              advanced: 1,
              needTime: 0,
              needYield: 0,
              saveState: 0,
              restoreState: 0,
              isEOF: 1,
              keyPattern: { Title: 1 },
              indexName: 'Title_1',
              isMultiKey: false,
```

```
              multiKeyPaths: { Title: [] },
              isUnique: false,
              isSparse: false,
              isPartial: false,
              indexVersion: 2,
              direction: 'forward',
              indexBounds:  {  Title:  [  '["The  Lake  House",  "The  Lake
House"]' ] },
              keysExamined: 1,
              seeks: 1,
              dupsTested: 0,
              dupsDropped: 0 } } },
     command:
      { find: 'movies',
        filter: { Title: 'The Lake House' },
        '$db': 'book' },
     serverInfo:
      { host: 'LAPTOP-DGRB6HD9',
        port: 27017,
        version: '5.0.9',
        gitVersion: '6f7dae919422dcd7f4892c10ff20cdc721ad00e6' },
     serverParameters:
      { internalQueryFacetBufferSizeBytes: 104857600,
        internalQueryFacetMaxOutputDocSizeBytes: 104857600,
        internalLookupStageIntermediateDocumentMaxSizeBytes: 104857600,
        internalDocumentSourceGroupMaxMemoryBytes: 104857600,
        internalQueryMaxBlockingSortMemoryUsageBytes: 104857600,
        internalQueryProhibitBlockingMergeOnMongoS: 0,
        internalQueryMaxAddToSetBytes: 104857600,
        internalDocumentSourceSetWindowFieldsMaxMemoryBytes: 104857600 },
     ok: 1 }
```

此时，查询优化器选择了索引扫描（IXSCAN），而不是集合扫描（COLLSCAN）。执行时间下降到了 0 毫秒。

第 28 篇 删除索引

本篇将会介绍 MongoDB 删除索引的 dropIndex() 方法。

28.1 dropIndex() 方法

集合的 dropIndex() 方法可以用于删除索引，语法如下：

```
db.collection.dropIndex(index)
```

其中，index 代表了想要删除的索引，它可以是一个指定索引名称的字符串，也可以是描述索引定义的文档。

注意，_id 字段上的默认索引无法被删除。

28.2 dropIndex() 示例

示例一：删除单个索引

首先，使用以下命令基于 movies 集合中的“Release Date”字段创建一个索引：

```
db.movies.createIndex({'Release Date': 1})

'Release Date_1'
```

以上语句创建了一个名为“”的索引。

其次，利用 getIndexes() 方法查看 movies 集合中的全部索引：

```
db.movies.getIndexes()

[
  { v: 2, key: { _id: 1 }, name: '_id_' },
  { v: 2, key: { Title: 1 }, name: 'Title_1' }
  { v: 2, key: { 'Release Date': 1 }, name: 'Release Date_1' }
]
```

接下来，使用 dropIndex() 删除索引“Release Date_1”：

```
db.movies.dropIndex('Release Date_1')

{ nIndexesWas: 3, ok: 1 }
```

返回结果显示已经删除了一个索引。

最后，再次使用 getIndexes() 方法查看索引：

```
db.movies.getIndexes()

[
```

```
    { v: 2, key: { _id: 1 }, name: '_id_' },
    { v: 2, key: { Title: 1 }, name: 'Release Date_1' }
  ]
```

示例二：基于定义删除索引

首先，重新创建基于“Release Date”字段的索引：

```
db.movies.createIndex({'Release Date': 1})

'Release Date_1'
```

查看 movies 集合上的索引：

```
db.movies.getIndexes()

[
  { v: 2, key: { _id: 1 }, name: '_id_' },
  { v: 2, key: { Title: 1 }, name: 'Title_1' }
  { v: 2, key: { 'Release Date': 1 }, name: 'Release Date_1' }
]
```

然后，基于索引“Release Date_1”的定义删除该索引：

```
db.movies.dropIndex({'Release Date': 1});

{ nIndexesWas: 3, ok: 1 }
```

示例三：删除所有非主键索引

从 MongoDB 4.2 开始，无法使用集合的 dropIndex('*') 删除所有非主键（_id）索引，而是需要使用 dropIndexes() 方法：

```
db.collection.dropIndexes()
```

首先，重新创建基于“Release Date”字段的索引：

```
db.movies.createIndex({'Release Date': 1})

'Release Date_1'
```

查看 movies 集合上的索引：

```
db.movies.getIndexes()

[
  { v: 2, key: { _id: 1 }, name: '_id_' },
  { v: 2, key: { Title: 1 }, name: 'Title_1' }
  { v: 2, key: { 'Release Date': 1 }, name: 'Release Date_1' }
]
```

最后，使用 dropIndexes()方法删除_id 之外的其他索引：

```
db.movies.dropIndexes()

{
  nIndexesWas: 3,
  msg: 'non-_id indexes dropped for collection',
  ok: 1
}
```

第 29 篇 复合索引

本篇将会介绍 MongoDB 复合索引的概念，以及如何创建复合索引。

29.1 复合索引

复合索引（compound index）是指基于多个字段的索引，通常可以用于优化匹配多个字段的查询。

复合索引同样可以使用 createIndex() 方法创建，语法如下：

```
db.collection.createIndex({
    field1: type,
    field2: type,
    field3: type,
    ...
});
```

其中，field1、field2、field3 都是字段；type 代表了类型，1 表示索引中的数据按照升序排列，-1 表示按照降序排列。

提示：MongoDB 复合索引最多包含 32 个字段。

复合索引中的字段顺序至关重要。如果一个复合索引包含字段 field1、field2，索引首先按照 field1 进行排序，如果 field1 相同，再按照 field2 排序。

复合索引遵循最左匹配原则。例如，一个复合索引包含字段 field1、field2，可以支持以下查询优化：

- 基于字段 field1 的匹配
- 基于字段 field1 以及 field2 的匹配

但是，它不支持基于字段 field2 的查询优化。

29.2 优化示例

首先，我们为集合 movies 创建一个基于 Title 以及 Release Date 的复合索引：

```
db.movies.createIndex({ Title: 1, 'Release Date': 1 })

'Title_1_Release Date_1'
```

然后查找名称包含 batman，并且在 2005 年 1 月 15 日发行的电影：

```
db.movies.find({Title: /batman/i, "Release Date": 'Jun 15 2005'}).explain('executionStats');

...
```

```
      inputStage: {
        stage: 'IXSCAN',
        filter: {
          Title: BSONRegExp("batman", "i")
        },
        nReturned: 1,
        executionTimeMillisEstimate: 5,
        works: 3180,
        advanced: 1,
        needTime: 3178,
        needYield: 0,
        saveState: 3,
        restoreState: 3,
        isEOF: 1,
        keyPattern: {
          Title: 1,
          'Release Date': 1
        },
        indexName: 'Title_1_Release Date_1',
...
```

查询使用了索引 Title_1_Release Date_1，而不是扫描整个集合。

然后查找名称包含 batman 的电影：

```
db.movies.find({Title: /batman/i}).explain('executionStats');

...
      inputStage: {
        stage: 'IXSCAN',
        filter: {
          Title: BSONRegExp("batman", "i")
        },
        nReturned: 6,
        executionTimeMillisEstimate: 0,
        works: 3192,
        advanced: 6,
        needTime: 3185,
        needYield: 0,
        saveState: 3,
        restoreState: 3,
        isEOF: 1,
        keyPattern: {
          Title: 1,
          'Release Date': 1
        },
        indexName: 'Title_1_Release Date_1',
...
```

以上查询只匹配了 Title，优化器仍然能够使用复合索引进行优化。

然后查找在 2005 年 1 月 15 日发行的电影：

```
db.movies.find({"Release Date": 'Jun 15 2005'}).explain('executionStats');

...
    executionStages: {
      stage: 'COLLSCAN',
      filter: {
        'Release Date': {
          '$eq': 'Jun 15 2005'
        }
      },
      nReturned: 1,
...
```

这种情况下，查询优化器无法使用复合索引，而是通过扫描整个集合（COLLSCAN）查找数据。

第 30 篇 唯一索引

本篇将会介绍 MongoDB 唯一索引，它可以用于确保文档字段值的唯一性。

30.1 唯一索引

很多时候我们需要确保文档中某个字段值的唯一性，例如 email 或者 username。唯一索引（unique index）可以帮助我们实现这种业务规则。实际上，MongoDB 使用唯一索引确保主键 _id 的唯一性。

创建唯一索引的方法和普通索引相同，只需要额外指定 {unique: true} 选项：

```
db.collection.createIndex({ field: 1}, {unique: true});
```

30.2 索引示例

首先，创建一个新的集合 users，插入一些文档：

```
db.users.insertMany([
   { name: "张三", dob: "1990-01-01", email: "zhangsan@test.com"},
   { name: "李四", dob: "1992-06-30", email: "lisi@test.com"}
]);
```

然后，我们基于 email 字段创建一个唯一索引：

```
db.users.createIndex({email:1},{unique:true});
```

接下来我们尝试插入一个已经存在的文档：

```
db.users.insertOne(
    { name: "张叁", dob: "1995-03-12", email: "zhangsan@test.com"}
);
```

此时，MongoDB 将会返回以下错误：

```
MongoServerError: E11000 duplicate key error collection: book.users index:
email_1 dup key: { email: 'zhangsan@test.com' }
```

下面我们演示一下集合已经存在重复数据时如何创建唯一索引。首先删除并重建集合 users：

```
db.users.drop()

db.users.insertMany([
   { name: "张三", dob: "1990-01-01", email: "zhangsan@test.com"},
   { name: "张叁", dob: "1995-03-12", email: "zhangsan@test.com"},
   { name: "李四", dob: "1992-06-30", email: "lisi@test.com"}
]);
```

然后，基于 email 字段创建一个唯一索引：

```
db.users.createIndex({email: 1},{unique:true})

MongoServerError: Index build failed: 95f78956-d5d0-4882-bfe0-2d856df18c61: Collection book.users ( 6da472db-2884-4608-98b6-95a003b4f29c ) :: caused by :: E11000 duplicate key error collection: mflix.users index: email_1 dup key: { email: zhangsan@test.com }
```

以上错误的原因在于 email 中存在重复的记录。

通常，我们会在插入数据之前创建唯一索引，可以从头开始确保数据的唯一性。如果基于已有数据创建唯一索引，可能会由于重复数据导致索引创建失败。为此，我们需要删除重复的数据之后再创建索引。

例如，我们可以首先删除重复的 user：

```
db.users.deleteOne({ name: " 张 叁 ", dob: "1995-03-12", email: "zhangsan@test.com"});

{ acknowledged: true, deletedCount: 1 }
```

然后基于 email 字段创建唯一索引：

```
db.users.createIndex({email: 1},{unique:true})

email_1
```

30.3 复合唯一索引

基于多个字段创建的唯一索引就是复合唯一索引（unique compound index）。复合唯一索引可以确保多个字段值的组合唯一。例如，基于字段 field1 和 field2 创建复合唯一索引，以下数据具有唯一性：

field1	field2	组合
1	1	(1,1)
1	2	(1,2)
2	1	(2,1)
2	2	(2,2)

以下数据存在重复值：

field1	field2	组合
1	1	(1,1)
1	1	(1,1) 重复
2	1	(2,1)
2	1	(2,1) 重复

接下来我们看一个示例，首先创建一个集合 locations。

```
db.locations.insertOne({
    address: "北京市丰台区莲花池东路 118 号北京西站",
    latitude: 39.894793,
    longitude: 116.321592
})
```

然后，基于 latitude 和 longitude 创建一个复合唯一索引：

```
db.locations.createIndex({
    latitude: 1,
    longitude: 1
},{ unique: true });

'latitude_1_longitude_1'
```

插入一个经度相同的地点：

```
db.locations.insertOne({
    address: "北京市海淀区复兴路甲 9 号中华世纪坛",
    lat: 39.910577,
    long: 116.321592
})
```

操作执行成功，因为复合索引 latitude_1_longitude_1 校验的是经度和纬度的组合值。

最后，插入一个经纬度已经存在的地点：

```
db.locations.insertOne({
    address: "北京市丰台区莲花池东路 118 号北京西站",
    latitude: 39.894793,
    longitude: 116.321592
})
```

此时，MongoDB 返回重复键错误：

```
MongoServerError: E11000 duplicate key error collection: book.locations
index: latitude_1_longitude_1 dup key: { lat: 39.894793, long: 116.321592 }
```

第 31 篇 数据导入和导出

本篇将会介绍如何利用 mongoimport 工具将文件导入本地 MongoDB 数据库服务器，以及如何利用 mongoexport 工具将 MongoDB 中的数据导出到文件中。

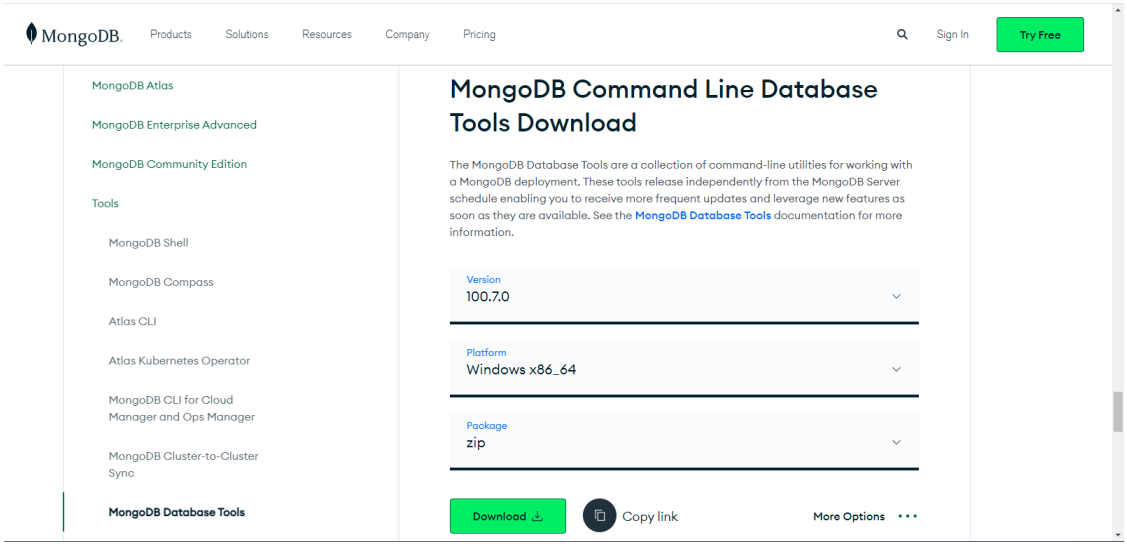
31.1 MongoDB 数据库工具

MongoDB 提供了一系列数据库工具，包括：

- 二进制导入/导出工具 mongodump、mongorestore 以及 bsondump；
- 数据导入/导出工具 mongoimport 以及 mongoexport；
- 诊断工具 mongostat 以及 mongotop；
- GridFS 工具 mongofiles。

MongoDB 4.4 版本开始，这些数据库工具不再随着服务器一起发布安装，而是使用单独的软件包。

首先，通过下载页面选择相应的版本、平台以及安装文件格式。



然后运行安装文件，按照提示进行安装即可。

同时，我们还需要点击下载示例文件 movies.json。

31.2 使用 mongoimport 导入文件

mongoimport 工具可以将 JSON、CSV 以及 TSV 文件导入 MongoDB 服务器。

首先，在命令行中进入安装目录。Windows 平台默认为 C:Files\100。

```
cd "C:\Program Files\MongoDB\Tools\100\bin"
```

然后，执行 mongoimport 命令将 movies.json 文件导入 MongoDB 服务器：

```
mongoimport.exe D:\data\movies.json -d bookdb -c movies --drop
```

其中，-d 用于指定目标数据库，-c 用于指定目标集合，--drop 表示如何该集合已经存在则删除后再导入。

导入成功后会显示以下信息：

```
2023-03-30T11:49:43.954+0800    connected to: mongodb://localhost/
2023-03-30T11:49:44.028+0800    dropping: bookdb.movies
2023-03-30T11:49:45.298+0800      3201 document(s) imported successfully. 0
document(s) failed to import.
```

然后连接到 MongoDB 服务器并查询导入的 movies 集合：

```
db.movies.countDocuments()
3201

db.movies.findOne()
{
  _id: ObjectId("642506d80e4683c5c14d7fa8"),
  Title: "Let's Talk About Sex",
  'US Gross': 373615,
  'Worldwide Gross': 373615,
  'US DVD Sales': null,
  'Production Budget': 300000,
  'Release Date': 'Sep 11 1998',
  'MPAA Rating': null,
  'Running Time min': null,
  Distributor: 'Fine Line',
  Source: null,
  'Major Genre': 'Comedy',
  'Creative Type': null,
  Director: null,
  'Rotten Tomatoes Rating': 13,
  'IMDB Rating': null,
  'IMDB Votes': null
}
```

31.3 使用 mongoexport 导出文件

mongoexport 工具可以将 MongoDB 数据库中的内容导出为 JSON 或者 CSV 文件。

导出 MongoDB 实例中的数据

以下命令用于将本地 27017 端口 MongoDB 数据库 bookdb 中的集合 movies 数据导出到 movies.json 文件：

```
mongoexport.exe --collection=movies --db=bookdb --out=movies.json
```

其中，--collection 用于指定要导出的集合，--db 指定了集合所在的数据库，--out 用于指定导出的文件路径和名称。

如果想要导出远程 MongoDB 实例中的数据，需要指定 --uri 连接字符串，例如：

```
   mongoexport.exe  --uri="mongodb://mongodb0.remote.server:27017/bookdb"    --collection=movies --out=movies.json
```

另外，我们也可以通过 --host 以及 --port 参数指定服务器地址和端口。例如：

```
   mongoexport.exe     --host="mongodb0.remote.server"     --port=27017     --collection=movies --db=bookdb --out=movies.json
```

导出副本集中的数据

如果想要导出副本集中的数据，可以在 --uri 连接字符串中指定副本集和成员:

```
   mongoexport.exe                                                                --uri="mongodb://mongodb0.remote.server:27017,mongodb1.remote.server:27017,mongodb2.remote.server:27017/bookdb?replicaSet=myReplicaSetName"   --collection=movies --out=movies.json
```

或者，也可以在 --host 参数中指定副本集和成员：

```
   mongoexport.exe                                                                --host="myReplicaSetName/mongodb0.remote.server:27017,mongodb1.remote.server:27017,mongodb2.remote.server:27017"     --collection=movies     --db=bookdb     --out=movies.json
```

默认情况下，mongoexport 通过副本集的主节点读取数据。不过，我们可以通过指定读优先级修改这个配置。例如：

```
   mongoexport.exe                                                                --uri="mongodb://mongodb0.remote.server:27017,mongodb1.remote.server:27017,mongodb2.remote.server:27017/bookdb?replicaSet=myReplicaSetName&readPreference=secondary" --collection=movies --out=movies.json
```

以上命令将会从副本集的从节点读取数据。

或者，也可以通过 --readPreference 参数指定读取的节点：

```
   mongoexport.exe                                                                --host="myReplicaSetName/mongodb0.remote.server:27017,mongodb1.remote.server:27017,mongodb2.remote.server:27017"  --readPreference=secondary  --collection=movies --out=movies.json
```

导出分片集群中的数据

如果想要导出分片集群中的数据，可以在 --uri 连接字符串中指定 mongos 实例的地址。例如：

```
   mongoexport.exe    --uri="mongodb://mongos0.remote.server:27017/bookdb"    --
collection=movies --out=movies.json
```

或者也可以在 --host 参数中指定 mongos 实例的地址和端口：

```
   mongoexport.exe  --host="mongos0.remote.server:27017"  --collection=movies  --
db=bookdb --out=movies.json
```