# 官网

https://www.mongodb.com/zh-cn

# 下载地址

https://www.mongodb.com/download-center/community

# 基本概念

MongoDB 是一个流行的开源 NoSQL 数据库，特别适用于处理大量非结构化或半结构化数据。与传统的关系型数据库（如 MySQL、PostgreSQL）相比，MongoDB 采用的是面向文档的存储方式，而不是基于表的关系型结构。

# BSON

## 基本概念

BSON（Binary JSON）是 MongoDB 使用的一种二进制格式，用于存储和传输数据。它是 JSON（JavaScript Object Notation）格式的二进制扩展，保留了 JSON 的易读性和结构化特点，同时也对数据的存储和传输做了优化，使其更加适合高效处理大规模数据。

## BSON 与 JSON 的对比

### JSON

**文本格式**：JSON 是纯文本格式，人类可以直接读取，适合用于数据交换。

**数据类型限制**：JSON 只支持有限的数据类型，例如字符串、数字、布尔值、数组、对象和 null。

**无二进制数据支持**：JSON 本身不支持二进制数据的存储，所有的二进制数据必须转换为字符串（例如 Base64 编码）。

### BSON

**二进制格式**：BSON 是二进制格式，比 JSON 更紧凑，适合高效的存储和传输。

**扩展数据类型**：除了 JSON 支持的数据类型外，BSON 还支持额外的类型，例如 Date、Binary（二进制数据）和 ObjectId（MongoDB 特有的 ID 类型）。

**优化存储和查询性能**：由于是二进制格式，BSON 在存储上比 JSON 更加高效，尤其是在需要存储大量数据时。

## 结构

BSON 数据的结构非常类似于 JSON，但在内部采用二进制格式表示。BSON 中的每个文档由若干个字段组成，每个字段都有一个类型和对应的值。字段的存储顺序与 JSON 中字段的顺序类似，具体字段的类型和数据形式如下：

### ****BSON 文档结构****

1、字段：每个 BSON 文档由一组键值对组成，其中键（Key）是字符串类型，值（Value）是对应的数据类型（例如整数、字符串、数组等）。

2、文档标识符：文档通常由 "\_id" 字段来标识，MongoDB 会自动生成一个 ObjectId 类型的值作为文档的主键。

### ****数据类型****

BSON 支持的主要数据类型包括：

Null (0x0A)：表示一个空值。

Boolean (0x08)：表示布尔值，true 或 false。

Integer (0x10)：支持 32 位整数（int32）和 64 位整数（int64）。

Double (0x01)：表示双精度浮点数。

String (0x02)：表示字符串，BSON 中的字符串采用 UTF-8 编码。

Object (0x03)：表示嵌套的文档（类似于 JSON 中的对象）。在 BSON 中，嵌套文档是一个有效的数据类型。

Array (0x04)：表示数组。BSON 中的数组是有序的元素集合。

Binary Data (0x05)：用于存储二进制数据，例如文件、图片、视频等。

ObjectId (0x07)：MongoDB 特有的 ID 类型，用于唯一标识每个文档。

Date (0x09)：表示日期和时间，以毫秒为单位。

Regular Expression (0x0B)：表示正则表达式。

JavaScript (0x0D)：表示 JavaScript 代码。

Symbol (0x0E)：表示符号类型，类似于字符串，但通常用于表示短小的字符串。

Code with Scope (0x0F)：表示带有作用域的 JavaScript 代码。

Decimal128 (0x13)：表示高精度浮点数，用于处理非常精确的数值，例如金融应用中常用的精确小数。

### ****BSON 文档的示例****



在 BSON 中，"\_id" 字段会以 ObjectId 类型存储，"address" 是一个嵌套文档，"tags" 是一个数组，"createdAt" 是一个日期对象，"isActive" 是布尔类型，"age" 是整数。

## ****BSON 序列化和反序列化****

### 序列化

将数据结构（如文档或对象）转换为 BSON 格式，MongoDB 在插入数据时会自动将 JSON 或其他格式的数据序列化为 BSON 格式。

### 反序列化

当从 MongoDB 获取数据时，BSON 数据会被转换回原始的 JSON 格式，便于应用程序进行处理。

## 应用场景

### 高效存储和查询

BSON 格式的紧凑性使其非常适合用于存储和查询大量数据，尤其是在大数据量和高并发的场景下。

### MongoDB 数据库

MongoDB 使用 BSON 格式存储文档，并且在数据传输时使用 BSON 格式（如客户端和服务器之间的通信）。

### 跨语言的兼容性

由于BSON是二进制格式，它能够在多种编程语言之间高效传输数据，同时保留数据类型的丰富性。

# JSON-like 数据

## 基本概念

在 MongoDB 中，JSON-like 数据是指数据以类似于 **JSON (JavaScript Object Notation)** 的格式存储和操作，但实际上采用了 BSON（Binary JSON）格式。

MongoDB 使用的数据格式看起来与 JSON 非常相似，但实际上，存储在 MongoDB 中的数据是 **BSON (Binary JSON)**。BSON 是 JSON 的二进制扩展，它在保留 JSON 可读性的同时，增加了更多的数据类型支持和存储效率。

## 示例

MongoDB 中存储的文档（Document）



注意：MongoDB 数据的实际存储格式是 BSON，但查询和显示时通常会用 JSON-like 格式。

## 优点

### 1、直观易用

开发者可以使用 JSON-like 格式轻松理解和操作 MongoDB 数据。

### 2、灵活性高

文档可以嵌套子文档和数组，适合处理复杂数据结构。



### 3、动态模式

MongoDB 的 JSON-like 文档不需要预定义模式（schema-less），数据字段可以动态变化。

### 4、查询友好

MongoDB 的查询语言也是基于 JSON-like 语法，易于使用。



## 实际用途

### 1、数据库存储

数据存储在 MongoDB 中，以 BSON 格式保存。

### 2、数据交互

通过 MongoDB 驱动程序或 API 操作数据，通常以 JSON-like 格式呈现。

### 3、数据传输

MongoDB 查询结果经常以 JSON-like 格式返回，方便与其他应用集成。

# 优点

## 1、高性能

**读写性能高**：MongoDB 采用了内存映射文件的方式来进行数据存储，读写性能较高，特别是在高负载、大并发场景下。

**水平扩展（Sharding）**：MongoDB 支持水平扩展，能够通过将数据分片存储在多个节点上来提升系统的吞吐量和存储能力。

## 2、灵活的Schema设计

**无固定模式**：MongoDB是一个schema-less数据库，意味着文档的结构不需要提前定义，可以随时对文档的字段进行修改。这种灵活性使得MongoDB适用于处理动态变化的应用程序需求。

**支持嵌套文档**：MongoDB存储的数据是BSON格式，可以存储嵌套的复杂数据结构，适合存储JSON-like数据。这使得数据更具层次感，避免了在关系型数据库中需要做多表关联查询的问题。

## 3、高可扩展性

分片（Sharding）：MongoDB 支持自动分片技术，将数据自动分布到多个节点上，使得它能够处理非常大的数据集和高并发请求。

复制（Replication）：MongoDB 提供主从复制机制，可以通过配置副本集实现数据的高可用性和容错。副本集能自动切换故障节点，确保高可用性。

## 4、易于使用

**简单的查询语言**：MongoDB 提供了简单易懂的查询语言，类似于 JSON 格式的查询，适合开发者进行快速开发。

**灵活的数据模型**：开发者不需要提前创建复杂的关系模型，可以根据实际需求灵活地处理数据结构。

## 5、支持多种索引方式

**索引优化**：MongoDB 支持多种索引类型（如单字段索引、复合索引、全文索引、地理空间索引等），能显著提高查询性能。

## 6、丰富的聚合功能

MongoDB提供了强大的聚合框架，支持像SQL中的GROUP BY、JOIN 等操作，能够进行复杂的数据统计和分析。尽管 MongoDB 不是关系型数据库，但通过其聚合管道，能够实现许多传统数据库中的聚合功能。

## 7、跨平台

MongoDB可以在多个操作系统上运行，包括Windows、Linux和macOS，具有较好的跨平台支持。

# 缺点

## 1、事务支持相对较弱

**事务支持**：MongoDB在早期版本中并不支持ACID事务，这在处理多文档事务时可能会导致一致性问题。虽然MongoDB4.x版本以后增加了多文档事务支持，但其事务机制仍不如传统关系型数据库（如MySQL、PostgreSQL）那样成熟和高效。

**性能开销**：在开启事务时，性能可能会有一定的下降，尤其是在处理复杂事务时。

## 2、数据冗余与一致性问题

**数据冗余**：由于MongoDB是无模式的，设计时可能会出现数据冗余的情况。这会增加存储成本，并且需要额外的机制来确保数据一致性。

**写入一致性**：尽管MongoDB支持副本集以保证数据的高可用性，但在某些高并发的情况下，写入一致性可能会受到影响，尤其是在没有启用写入确认机制时。

## 3、缺乏JOIN操作

MongoDB没有传统关系型数据库中的JOIN操作。这意味着，对于多表关系的数据，开发者必须在应用层自己处理关联，可能需要增加额外的逻辑和计算开销。

在某些复杂的查询场景下，缺乏JOIN功能可能导致性能下降或者增加开发复杂度。

## 4、存储空间效率低

MongoDB使用的是BSON格式存储数据，虽然BSON是二进制格式，但相对于JSON格式，它的存储效率相对较低。因此，MongoDB在某些数据密集型应用中可能会浪费更多的存储空间。

## 5、限制的聚合操作

虽然MongoDB提供了强大的聚合框架，但它仍然比传统的SQL聚合操作要弱一些，特别是在处理复杂的联接查询时，可能会导致性能问题。

聚合操作也不适用于所有场景，某些特定的高性能场景中，可能会发现传统SQL查询更为高效。

## 6、数据备份和恢复复杂

MongoDB的数据备份和恢复相对较为复杂，尤其是在大规模分片环境下，备份和恢复数据的过程可能涉及到很多操作和资源。

如果副本集配置不当，恢复数据时也可能会遇到数据一致性和同步的问题。

## 7、学习曲线较陡

对于没有NoSQL经验的开发者，MongoDB的一些概念和功能（如分片、复制、聚合框架等）可能需要一定的学习成本和时间来掌握。

# 使用场景

## 大规模、高并发的应用

例如社交网络、即时通讯、物联网（IoT）等，这些应用往往具有大量非结构化数据和高并发请求。

## 内容管理系统

因为MongoDB灵活的文档结构非常适合存储动态变化的内容和用户生成的数据。

## 实时分析

MongoDB提供了强大的聚合框架，可以进行复杂的实时数据分析，适合做大数据分析。

## 快速开发和原型设计

开发团队需要快速迭代时，MongoDB 的灵活性和简单查询语言有助于加速开发进程。

# mongodb数据库与javascript的关系

## 基本说明

MongoDB与JavaScript有着紧密的关系，主要体现在数据格式、操作接口以及开发工具上。

## 数据格式上的关系

MongoDB使用的文档格式是**JSON-like数据**，即类JSON格式的数据结构。这种数据格式本身来源于JavaScript的对象表示法（JavaScript Object Notation）

示例：

MongoDB 中存储的文档：



JavaScript中的对象：



**关系**：MongoDB文档的数据格式与JavaScript对象格式几乎一致，开发者可以轻松地将 JavaScript对象存入MongoDB中，或将MongoDB中的数据直接解析为JavaScript对象。

## 操作上的关系

MongoDB提供了原生的 JavaScript 接口，支持通过 JavaScript 脚本直接操作数据库。

### MongoDB Shell

MongoDB 提供了一个交互式的 JavaScript Shell（mongosh），它是基于 JavaScript 的命令行工具，用于管理和操作 MongoDB 数据库。

示例：使用 JavaScript 操作 MongoDB 数据



在 mongosh 中，所有命令都可以看作是 JavaScript 函数调用，因此开发者可以使用 JavaScript 的语法来完成复杂的数据库操作。

## 开发上的关系

MongoDB 提供了官方的 JavaScript 驱动（Node.js Driver），用于在 JavaScript 和 Node.js 环境中操作 MongoDB 数据库。

**示例：使用Node.js操作MongoDB**安装MongoDB驱动：



使用代码操作 MongoDB：



## 使用 JavaScript 实现 MongoDB 聚合操作

MongoDB 的聚合框架（Aggregation Framework）是一种强大的数据处理工具，JavaScript 在实现聚合操作时非常自然。

聚合操作：

db.orders.aggregate([

{ $match: { status: "delivered" } }, // 筛选条件

{ $group: { \_id: "$customerId", totalSpent: { $sum: "$amount" } } }, // 分组并计算总消费

{ $sort: { totalSpent: -1 } } // 按总消费排序

]);

在 JavaScript 应用中，这种操作可以直接转换为数组操作逻辑，开发者可以轻松上手。

## JavaScript与MongoDB的优势

1、一致性：MongoDB 的 JSON-like 数据格式与 JavaScript 对象非常相似，无需数据格式转换。

2、灵活性：开发者可以直接在 MongoDB Shell 或 Node.js 环境中编写 JavaScript 脚本操作 MongoDB，减少学习成本。

3、强大的生态系统：JavaScript 的生态（如 Node.js 和前端框架）与 MongoDB 高度兼容，适合构建全栈 JavaScript 应用。

4、动态模式：MongoDB 不需要预定义模式（Schema-less），与 JavaScript 动态语言特性天然契合。

## 总结

MongoDB 与 JavaScript 的关系主要体现在：

1、数据格式上的一致性（JSON-like 数据与 JavaScript 对象）。

2、操作接口的友好性（MongoDB Shell 和 Node.js 驱动）。

3、开发生态的融合（支持全栈 JavaScript 应用）

# mongodb官方安装文档

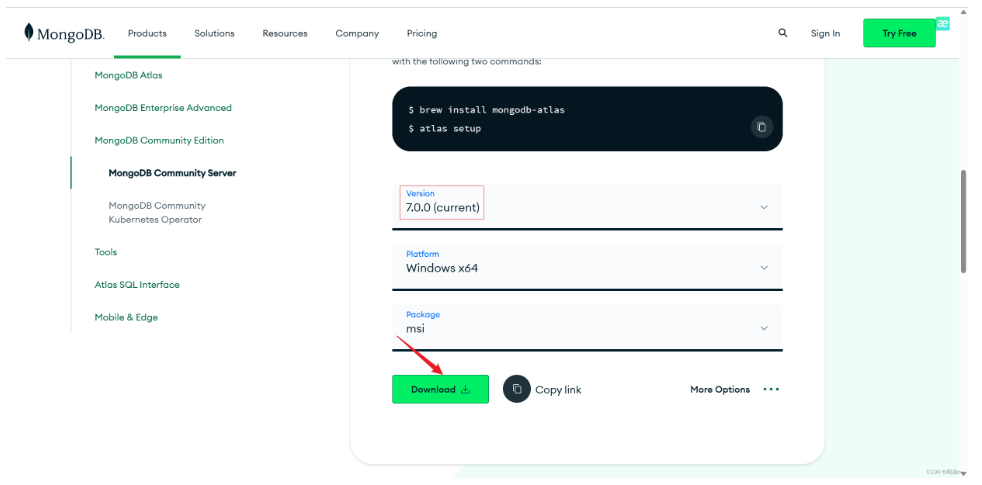
https://www.mongodb.com/zh-cn/docs/manual/installation/

# windows版本安装教程

## 5.0版本安装教程

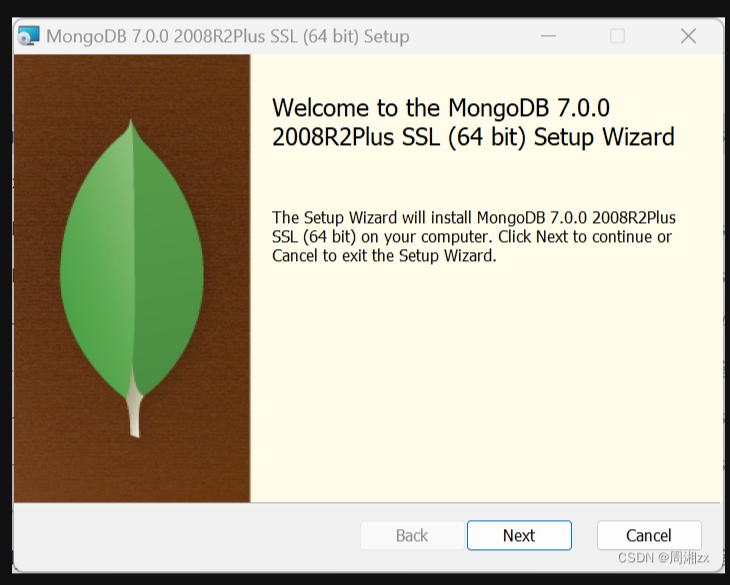
## 7.0版本安装教程

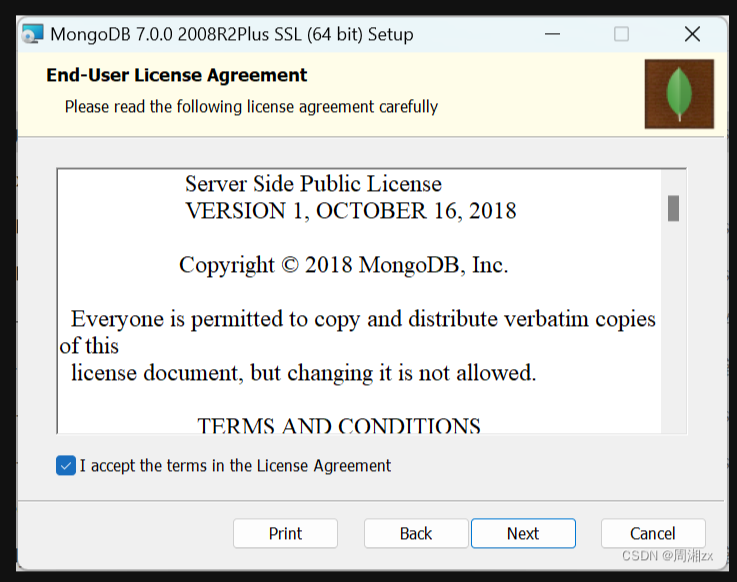
### 1、下载

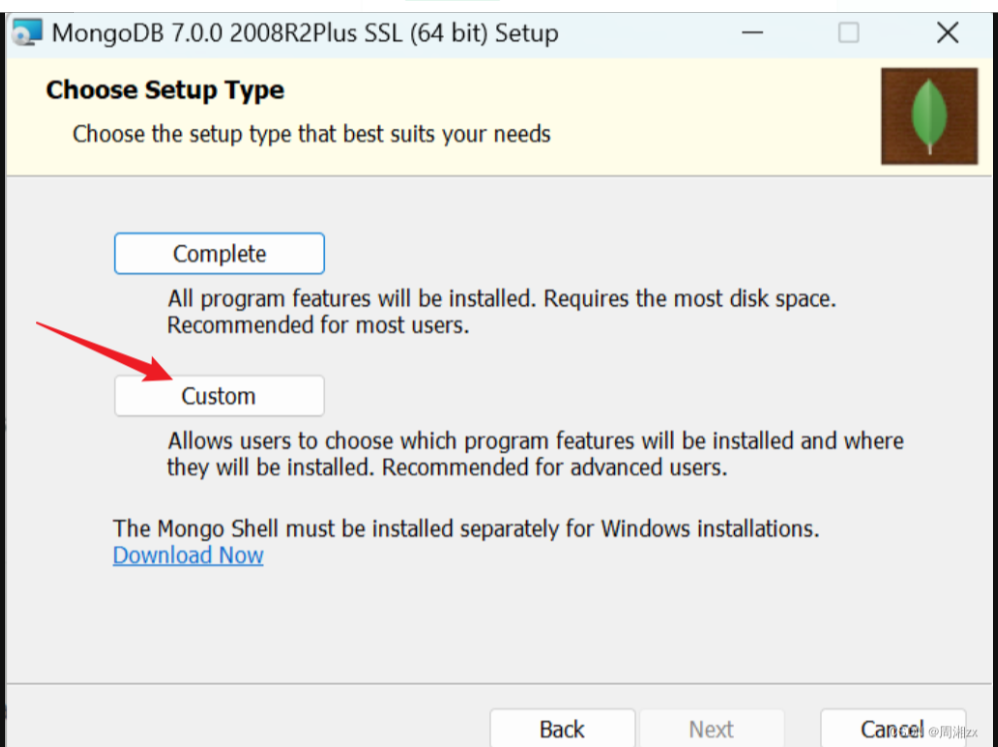


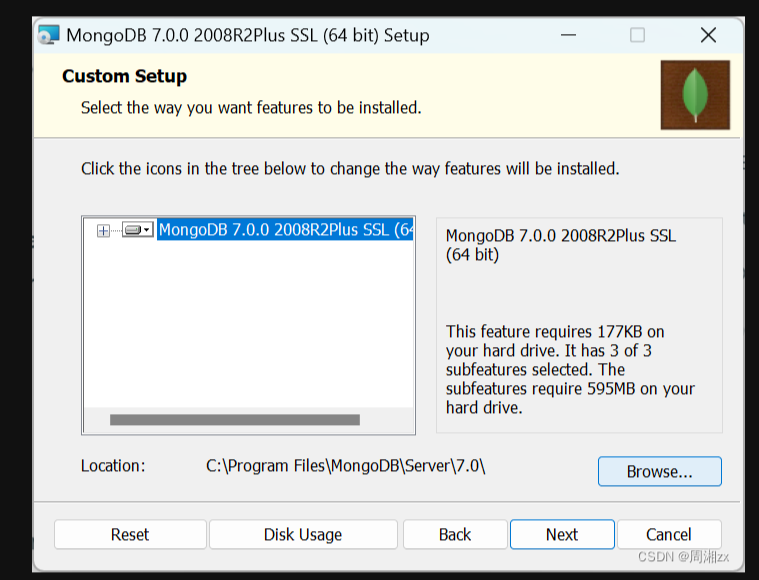
### 2、安装

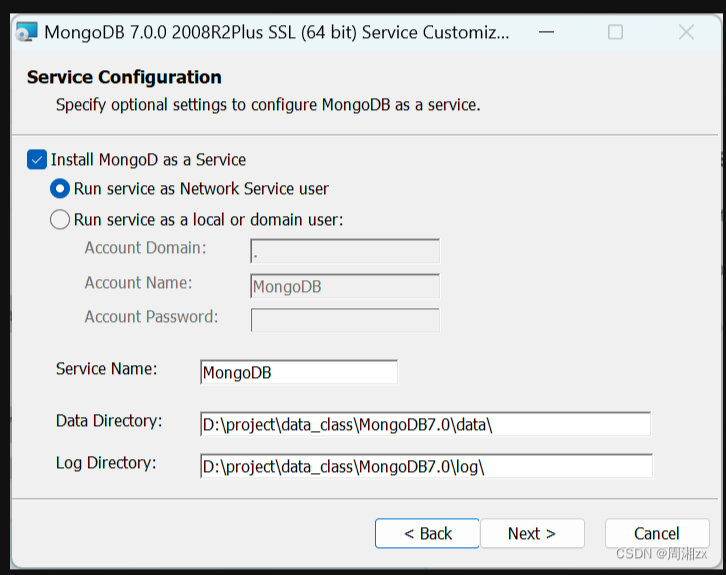
下载后双击打开安装导向,安装步骤如图所示

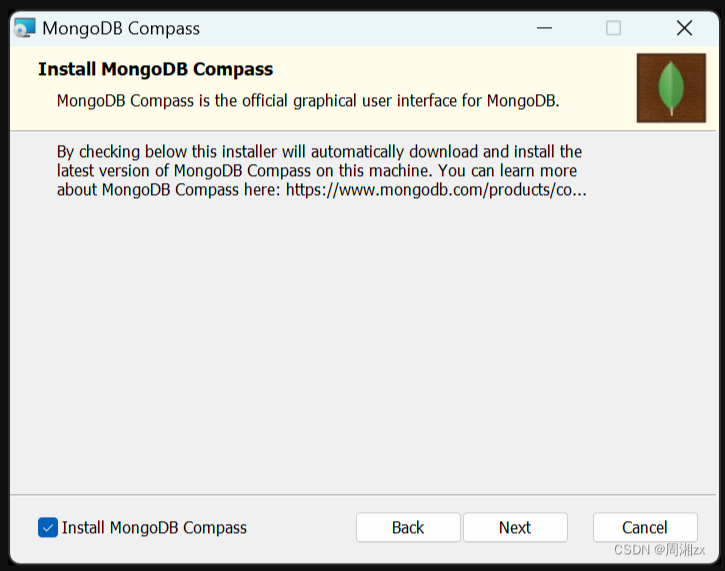


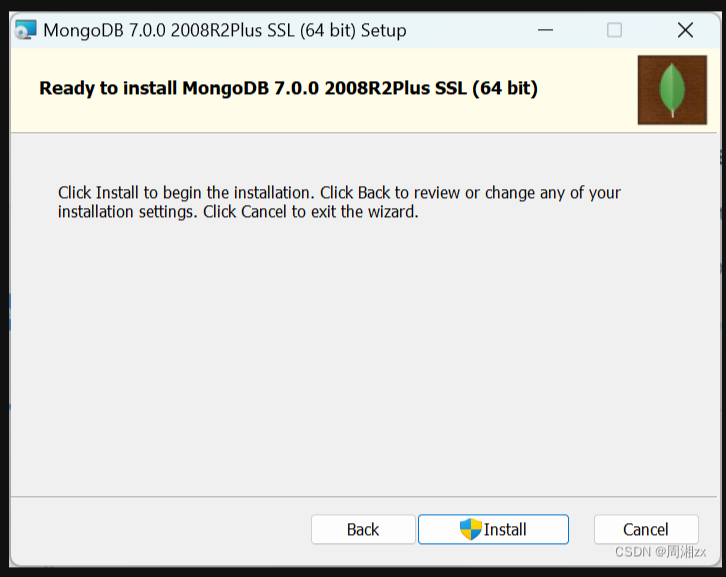


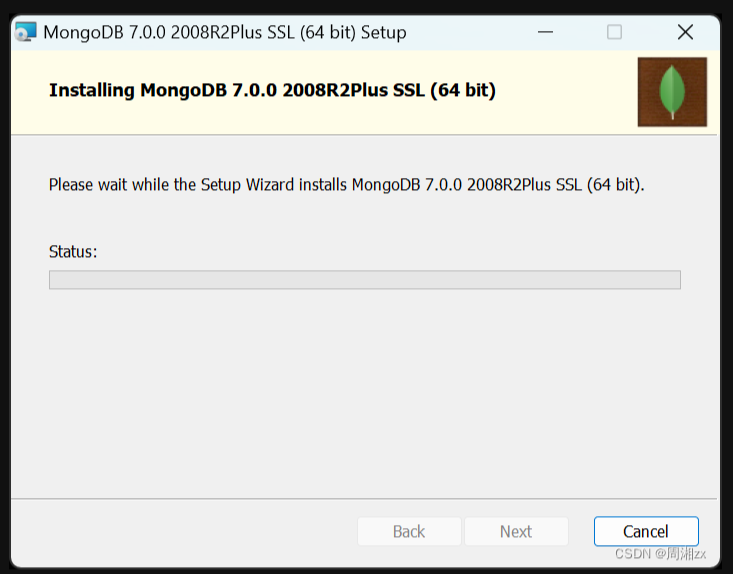


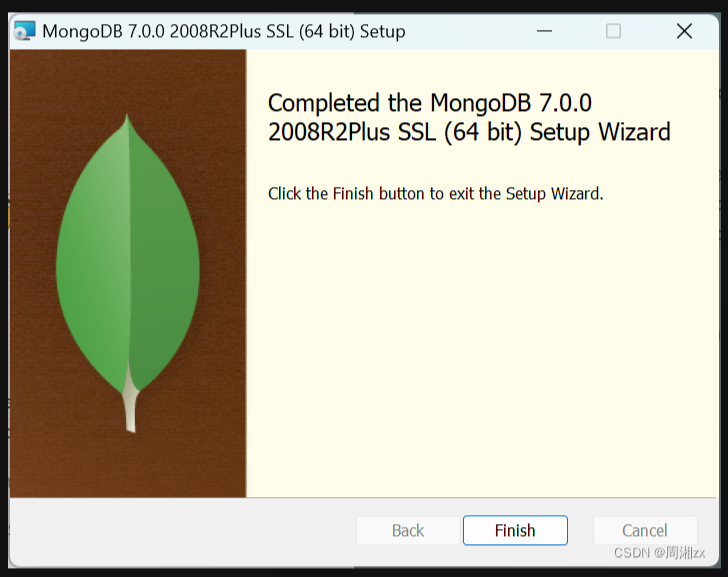










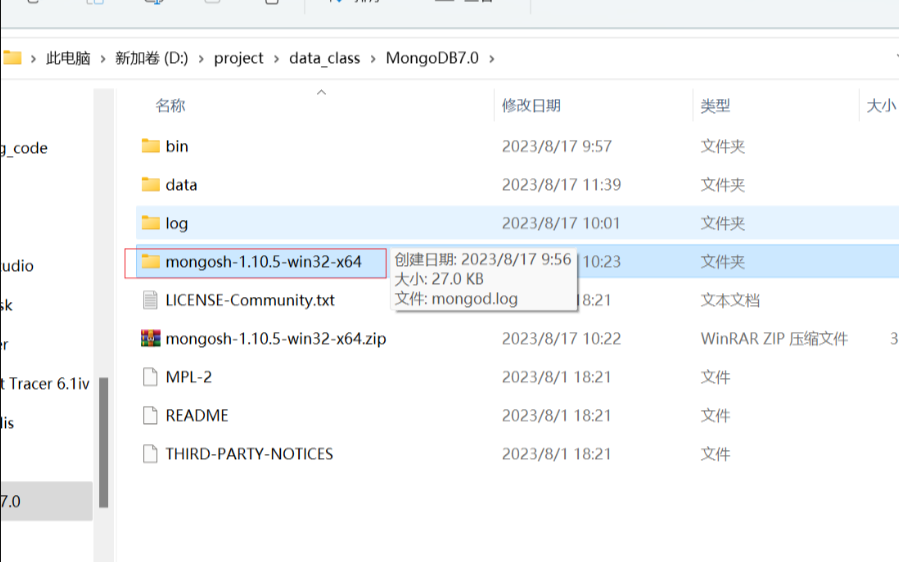


### 3、注意问题

1、安装Mongoshell，**MongoDB6之前shell是直接在里面的6之后需要单独下载**

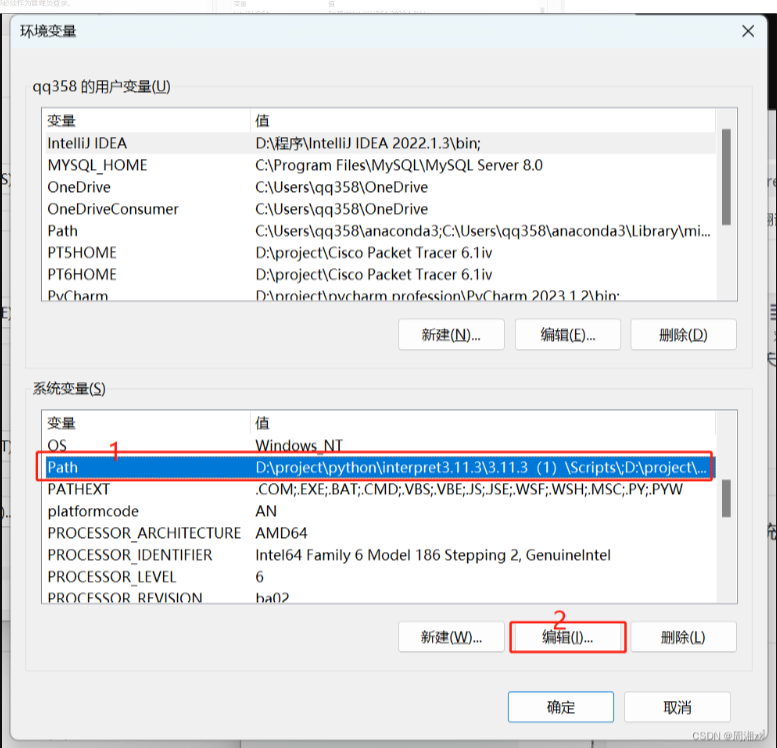
2、mongoDB6没有mong.exe和mongdb.exe，要想通过命令行启动[mongoDB](https://so.csdn.net/so/search?q=mongoDB&spm=1001.2101.3001.7020)需要自己下载一个Mongoshell

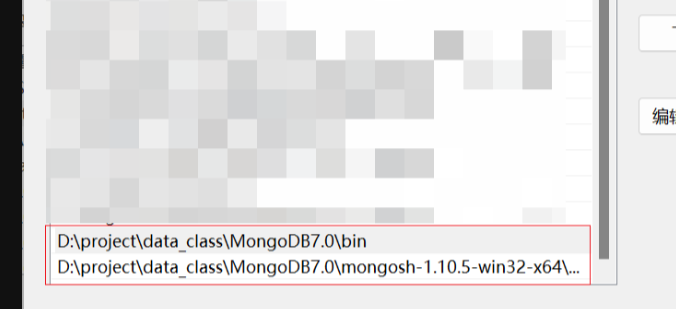
3、下载后Mongoshell解压到mongo的安装路径下



4、配置环境变量

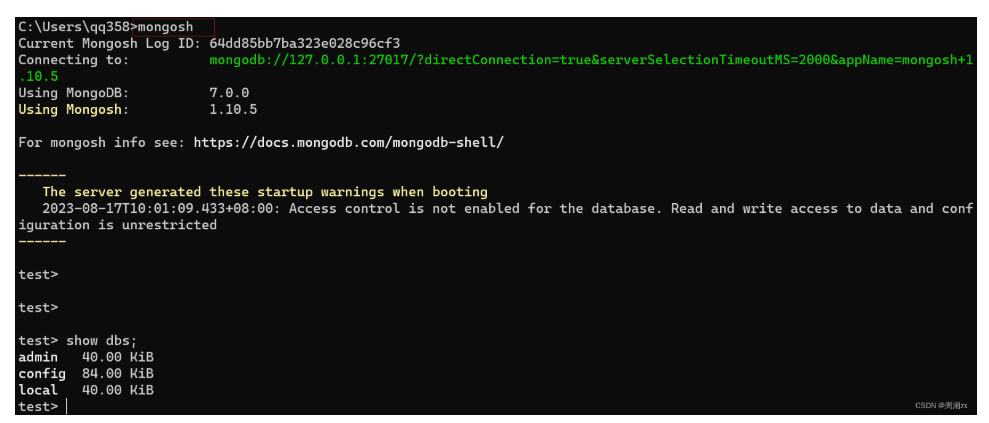
配置两个路径的环境变量





### 4、测试

打开cmd,输入mongosh，即可成功运行mongo



退出mongo使用命令：quit()

## 注意事项

MongoDB6.0以后做出了重大改变，MongoDB 已经不再默认安装 shell 工具，你需要安装一个额外的 shell 工具：[MongoDB Shell](https://www.runoob.com/mongodb/mongodb-shell.html)。

# 判断mongodb服务是否启动成功

服务启动后，访问这个地址http://127.0.0.1:27017/，看到下面的页面表示启动成功。



# 端口号

默认端口号：27017

# 查看mongodb的版本信息

mongod -version



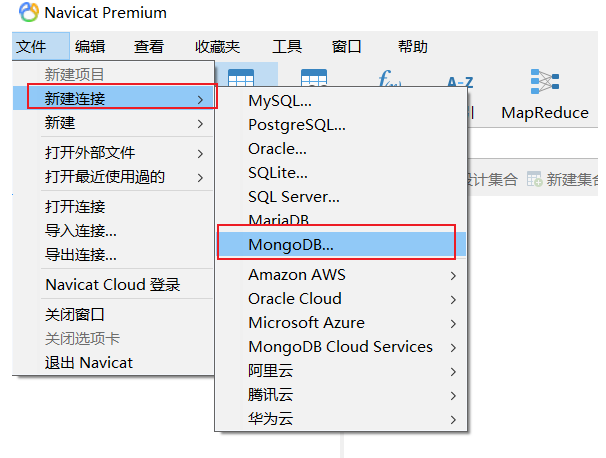
# 推荐的客户端工具



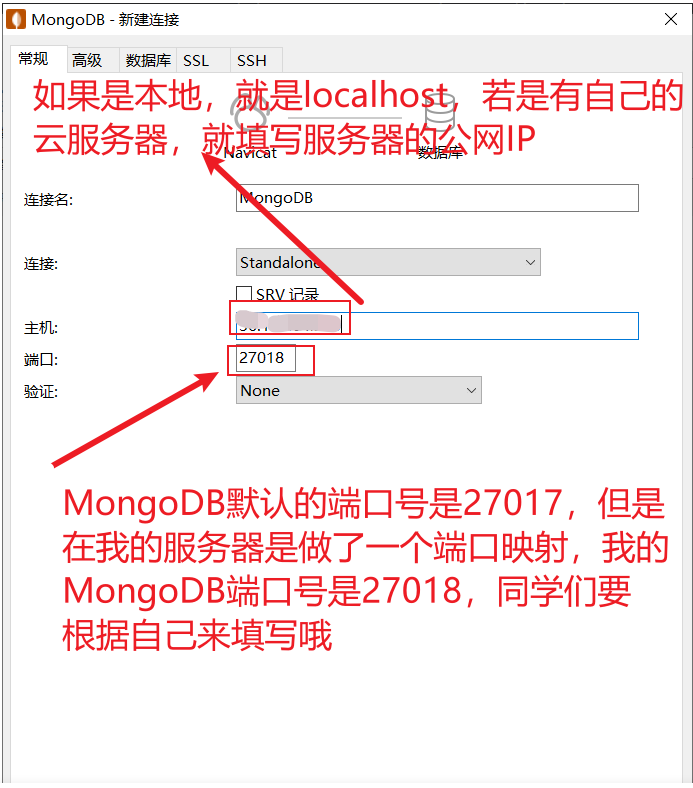
根据你的具体需求选择合适的工具，官方推荐从 MongoDB Compass 入门，复杂场景可考虑 Studio 3T 或 NoSQLBooster。

# 使用navicat连接mongodb数据库

## 1、新建连接

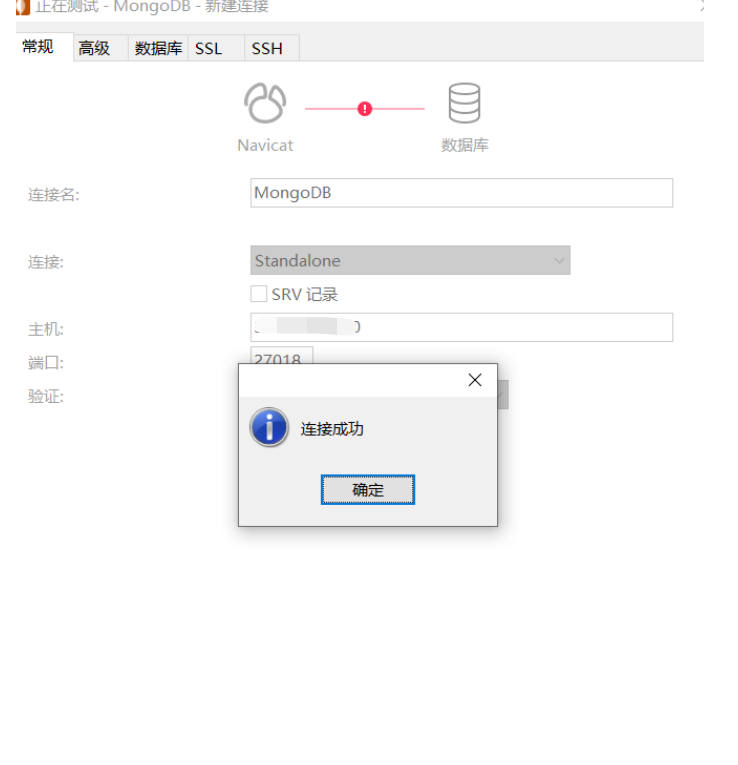


## 2、填写主机和端口



## 3、测试

如果顺利的话，会显示连接成功，然后再点确定即可



# Studio 3T使用教程

## 免费版和收费版的区别

## 快捷键

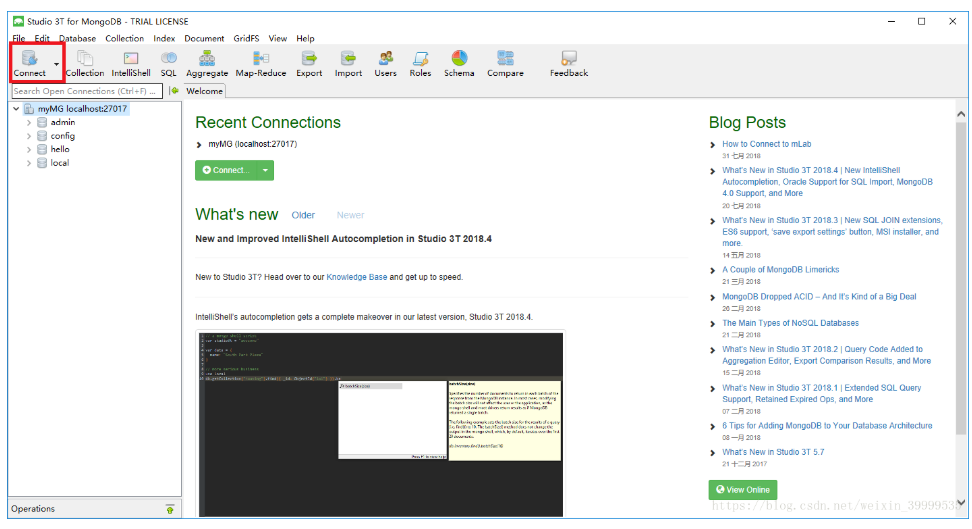
### F9

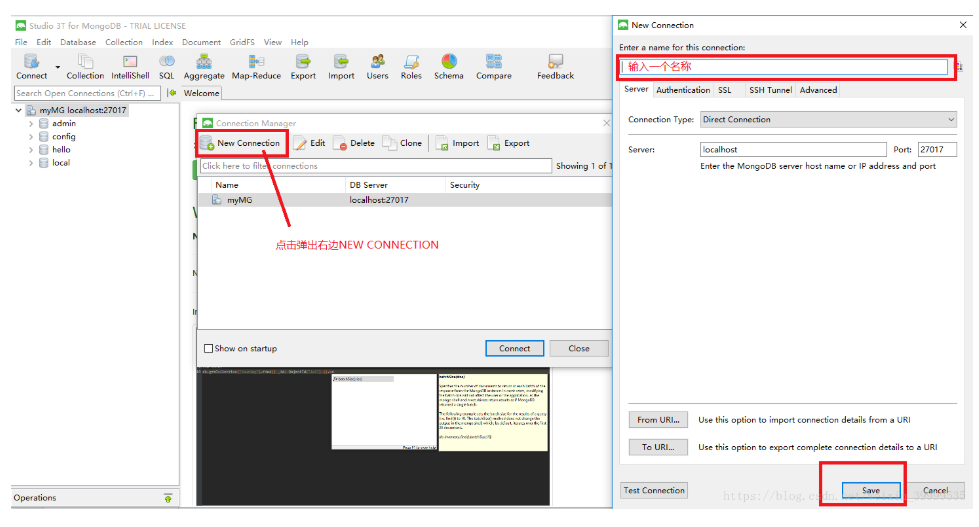
运行选择的sql语句

### F5

运行整个sql脚本

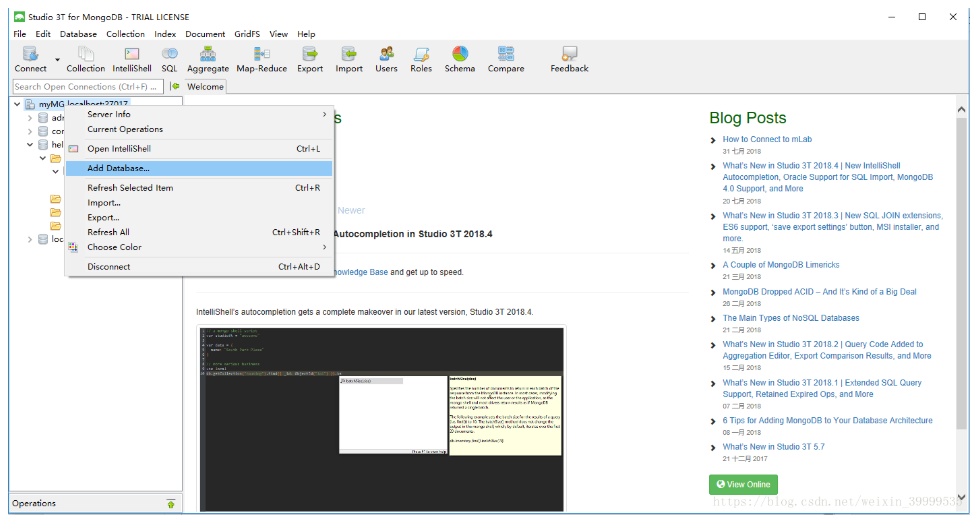
## 新建连接



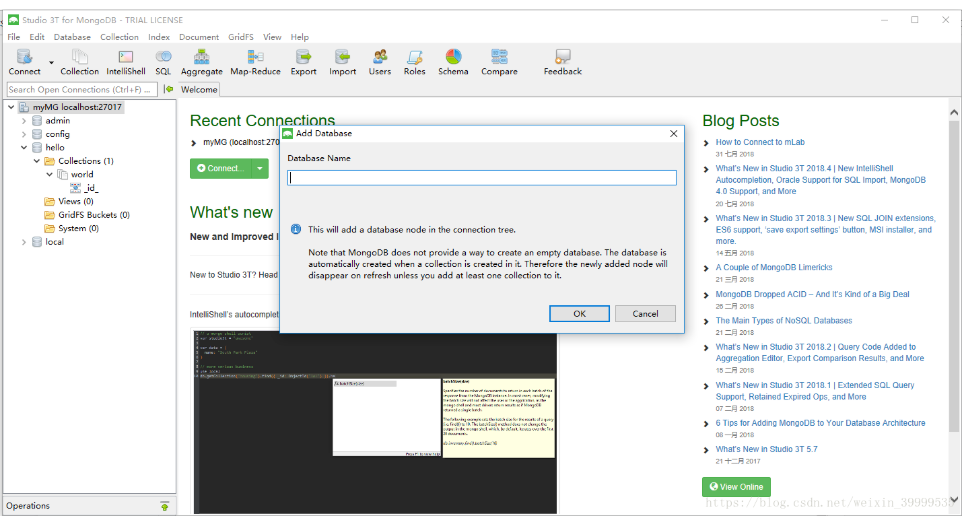


根据提示进行操作，最后点击保存即可。

## 添加数据库

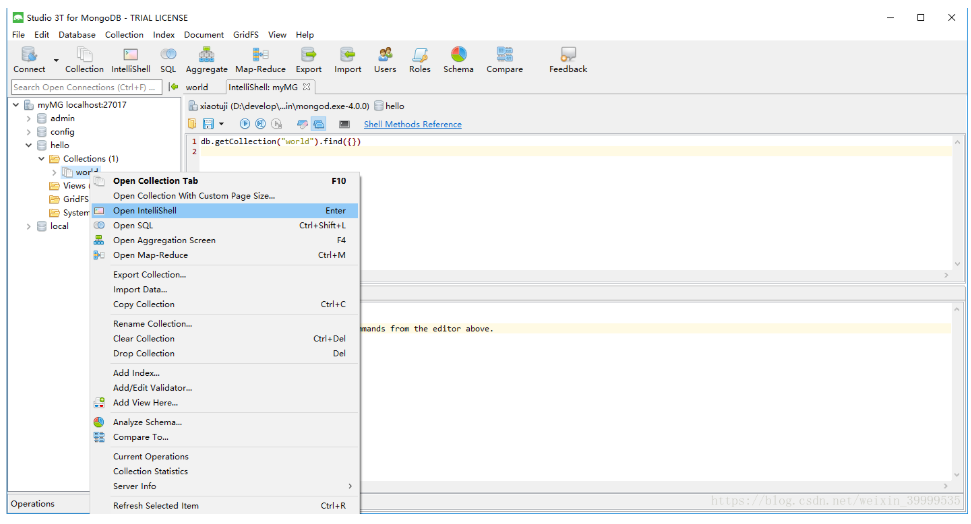


右键新建的连接，选择Add Database新建数据库

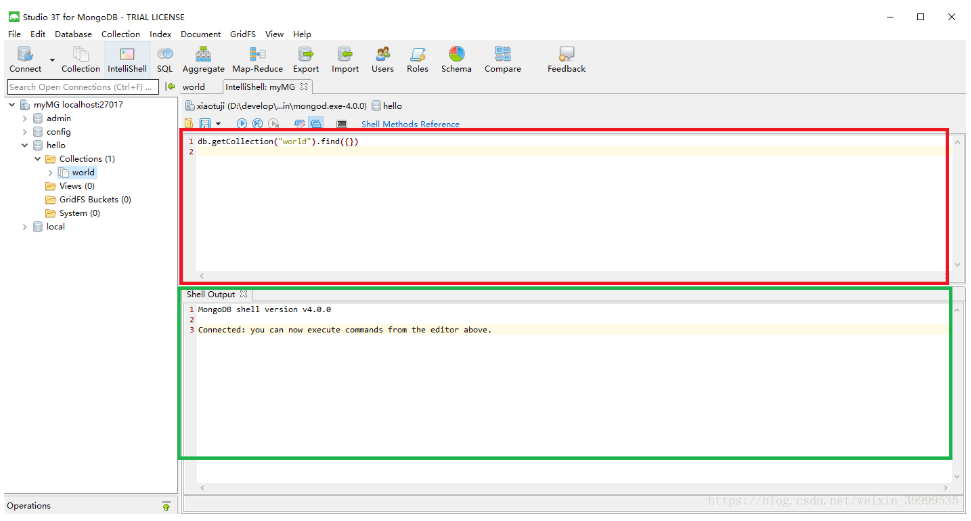


输入数据库名称点击OK

## CRUD操作



首先打开命令行窗口，Open intelliShell



红色框是输入的命令行，绿色框是输出的提示信息，然后在里面写操作语句就可以了。

## 导出数据

### 第一种方法

## 导入数据

# MongoDB Shell

## 基本概念

MongoDB Shell 是 MongoDB 提供的官方交互式界面，允许用户与 MongoDB 数据库进行交互、执行命令和操作数据库。MongoDB Shell 是基于 JavaScript 的，允许用户直接在命令行或者脚本中使用 JavaScript 语言来操作 MongoDB 数据库。

## 安装 MongoDB Shell

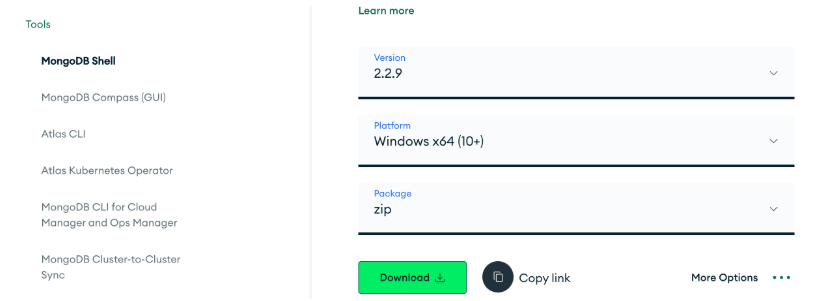
MongoDB Shell 是 MongoDB 数据库安装包的一部分，因此，安装 MongoDB 数据库也会自动安装 MongoDB Shell。

### 在 Windows 上安装 MongoDB Shell

官方安装文档：

https://www.mongodb.com/zh-cn/docs/mongodb-shell/install/

访问 MongoDB 官网下载页面（<https://www.mongodb.com/try/download/shell>），选择适合 Windows 的 MongoDB Shell 版本，并下载安装程序。



运行下载的安装程序，并按照指示进行安装。在安装过程中，确保选择安装 MongoDB Shell（默认情况下会包含在安装中）。可以选择在安装过程中配置环境变量，这样可以在命令行中直接运行 mongosh 命令来启动 MongoDB Shell。

## 使用 MongoDB Shell

### 1、****启动 MongoDB Shell****

在命令行中输入 mongosh 命令，启动 MongoDB Shell，如果 MongoDB 服务器运行在本地默认端口（27017），则可以直接连接。

mongosh

查看版本：

mongosh --version

2.2.9

### ****2、连接到 MongoDB 服务器****

如果 MongoDB 服务器运行在非默认端口或者远程服务器上，可以使用以下命令连接：

mongosh --host <hostname>:<port>

其中 <hostname> 是 MongoDB 服务器的主机名或 IP 地址，<port> 是 MongoDB 服务器的端口号。

### 基本操作命令

#### 查看当前数据库

db

#### 显示数据库列表

show dbs

#### 切换到指定数据库

use <database\_name>

#### 执行查询操作

db.<collection\_name>.find()

#### 插入文档

db.<collection\_name>.insertOne({ ... })

#### 更新文档

db.<collection\_name>.updateOne({ ... })

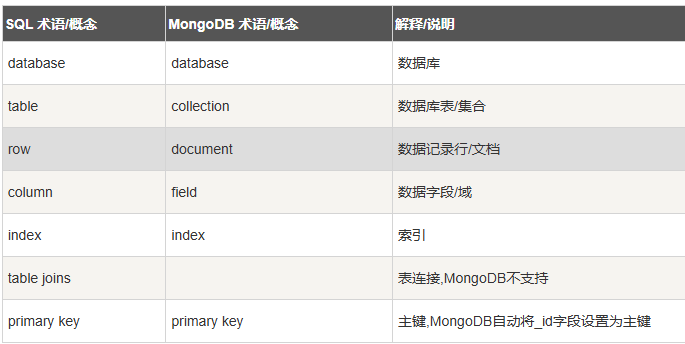
#### 删除文档

db.<collection\_name>.deleteOne({ ... })

#### 退出MongoDB Shell

quit() 或者 exit

# 概念解析



## 文档(document)

MongoDB 的基本数据单元，通常是一个 JSON-like 的结构，可以包含多种数据类型。

## 集合(collection)

类似于关系型数据库中的表，集合是一组文档的容器。在 MongoDB 中，一个集合中的文档不需要有一个固定的模式。

## 数据库(database)

包含一个或多个集合的 MongoDB 实例。一个 MongoDB 中可以建立多个数据库。如果在操作时没有指定数据库，MongoDB 会使用一个名为 test 的默认数据库，该数据库存储在 data 目录中。MongoDB 的单个实例可以容纳多个独立的数据库，每一个都有自己的集合和权限，不同的数据库也放置在不同的文件中。从 MongoDB 4.0 开始，支持在单个数据库上执行多文档事务。

### 特殊数据库

有一些数据库名是保留的，可以直接访问这些有特殊作用的数据库。

#### admin

从权限的角度来看，这是"root"数据库。要是将一个用户添加到这个数据库，这个用户自动继承所有数据库的权限。一些特定的服务器端命令也只能从这个数据库运行，比如列出所有的数据库或者关闭服务器。

#### local

这个数据永远不会被复制，可以用来存储限于本地单台服务器的任意集合。

#### config

当Mongo用于分片设置时，config数据库在内部使用，用于保存分片的相关信息。

## BSON

Binary JSON 的缩写，是 MongoDB 用来存储和传输文档的二进制形式的 JSON。

## 索引（index）

用于优化查询性能的数据结构，可以基于集合中的一个或多个字段创建索引。

## 分片（sharding）

一种分布数据到多个服务器（称为分片）的方法，用于处理大数据集和高吞吐量应用。

## 副本集(replica set)

一组维护相同数据集的 MongoDB 服务器，提供数据的冗余备份和高可用性。

## 主节点(primary)

副本集中负责处理所有写入操作的服务器。

## 从节点

副本集中的服务器，用于读取数据和在主节点故障时接管为主节点。

## MongoDB Shell

MongoDB 提供的命令行界面，用于与 MongoDB 实例交互。

## 聚合框架（Aggregation Framework）

用于执行复杂的数据处理和聚合操作的一系列操作。

## Map-Reduce

一种编程模型，用于处理大量数据集的并行计算。

## GridFS

用于存储和检索大于 BSON 文档大小限制的文件的规范。

## ObjectId

MongoDB 为每个文档自动生成的唯一标识符。

## 事务（Transactions）

从MongoDB 4.0 开始支持，允许一组操作作为一个原子单元执行。

## 操作符（Operators）

用于查询和更新文档的特殊字段

## 连接（Join）

MongoDB 允许在查询中使用 $lookup 操作符来实现类似 SQL 的连接操作。

## TTL（Time-To-Live）

可以为集合中的某些字段设置 TTL，以自动删除旧数据。

## 存储引擎（Storage Engine）

MongoDB 用于数据存储和管理的底层技术，如 WiredTiger 和 MongoDB 的旧存储引擎 MMAPv1。

## MongoDB Compass

MongoDB 的图形界面工具，用于可视化和管理 MongoDB 数据。

## MongoDB Atlas

MongoDB 提供的云服务，允许在云中托管 MongoDB 数据库。

# 默认的存储引擎

## 基本说明

自 MongoDB 3.2 版本起，WiredTiger 是 MongoDB 的默认存储引擎。WiredTiger 提供了高性能、压缩支持和更高的并发处理能力，成为 MongoDB 的核心存储层。

## 支持哪些存储引擎

WiredTiger（默认）

MMAPv1（旧版本的默认存储引擎，在 MongoDB 4.0 后弃用）

In-Memory（主要用于高性能需求的场景）

Ephemeral for Test（用于测试场景）

## 核心特点

### 1、数据压缩

WiredTiger 提供两种主要的压缩方式：

Snappy（默认）：高效的轻量级压缩，适用于大多数场景。

Zlib：压缩率更高，但占用更多 CPU 资源。

压缩可以显著减少存储空间消耗，同时提高数据传输效率。

### 2、文档级并发控制

WiredTiger 支持文档级锁（Document-Level Locking），而非旧版 MMAPv1 的集合级锁（Collection-Level Locking）。这意味着多个线程可以同时操作同一个集合中的不同文档，从而提高并发性能。

### 3、高效的缓存管理

WiredTiger 通过内存映射和自定义缓存分配管理磁盘 IO 和内存：

默认情况下，WiredTiger 将分配系统内存的 50% 或最多 256 MB 作为缓存。

高效的缓存算法减少了磁盘 IO，提高了性能。

### 4、日志记录和崩溃恢复

WiredTiger 提供持久化的日志记录（Write-Ahead Logging, WAL）：

通过将写操作记录到日志文件中，确保在系统崩溃时可以恢复到一致状态。

支持瞬时事务（point-in-time transactions），提高数据可靠性。

### 5、多核利用

WiredTiger 能够充分利用多核处理器，提供更高的性能和吞吐量

## WiredTiger 的配置

### 1、默认配置

当您安装 MongoDB 并不做任何修改时，WiredTiger 是默认的存储引擎，配置示例如下：

storage:

engine: wiredTiger

### 2、自定义参数

可以在 mongod.conf 文件中对 WiredTiger 进行参数配置，例如：

storage:

engine: wiredTiger

wiredTiger:

engineConfig:

cacheSizeGB: 2 # 设置缓存大小为 2 GB

journalCompressor: snappy # 使用 Snappy 压缩日志

collectionConfig:

blockCompressor: zlib # 使用 Zlib 压缩集合

### 3、动态调整

部分参数（如缓存大小）可以动态调整，无需重启服务

db.adminCommand({ setParameter: 1, "wiredTigerConcurrentWriteTransactions": 128 });

## WiredTiger vs MMAPv1对比



## WiredTiger 的局限性

### 1、高并发性能依赖于硬件

需要 SSD 和大内存来充分发挥性能。

### 2、压缩带来的 CPU 开销

压缩虽然节省存储空间，但会消耗额外的 CPU 资源。

### 3、缓存限制

对于内存不足的系统，可能会导致性能下降。

## 适用场景

### 1、高并发写入

如实时日志记录或电商交易系统。

### 2、大量数据存储

需要压缩以节省存储空间。

### 3、数据可靠性要求高

如银行和金融系统。

### 4、需要快速恢复

如容灾备份恢复环境。

# 配置文件

# 数据类型

## String（字符串）

字符串。存储数据常用的数据类型。在 MongoDB 中，UTF-8 编码的字符串才是合法的。

示例：



## Number（数字）

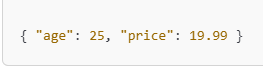
包括以下类型：

int32：32 位整数。

int64：64 位整数。

double：64 位浮点数。

示例：



## Boolean（布尔值）

表示 true 或 false 的布尔值。

示例：



## Object（嵌套文档）

表示嵌套的键值对结构，相当于一个子文档。

应用场景：

存储复杂的嵌套数据，例如地址、配置信息。

示例：



## Array（数组）

表示一个有序的值集合，值可以是任意类型。

应用场景：

存储多个值，例如列表、标签、兴趣。

示例：



## ObjectId（对象ID）

MongoDB 默认的主键 \_id 使用 ObjectId 类型。是一个 12 字节的唯一标识符，分为以下部分：

时间戳（4 字节）。

机器标识符（3 字节）。

进程 ID（2 字节）。

随机值（3 字节）。

应用场景：用于唯一标识文档。



## Date（日期）

用于存储日期和时间，MongoDB 中的日期类型存储为 UTC 时间。

应用场景：存储时间戳、事件时间

示例：



## Null（空值）

表示一个字段没有值或值为 null

应用场景：表示未知或缺失的字段。

示例：



## Binary Data（二进制数据）

存储任意的二进制数据，例如文件、图像

应用场景：存储非结构化数据

示例：



## Regular Expression（正则表达式）

存储正则表达式，用于模式匹配查询

应用场景：搜索文本字段时使用。

示例：

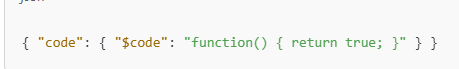


## JavaScript（代码）

存储 JavaScript 代码，可包含作用域（可选）。

应用场景：在查询中使用动态表达式。

示例：

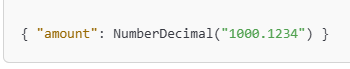


## Decimal128（高精度浮点数）

存储高精度的 128 位十进制数。避免了浮点数的精度丢失问题

应用场景：财务计算、货币相关数据。

示例：



## Min/Max keys

MinKey：表示 BSON 中的最小值。MaxKey：表示 BSON 中的最大值。

应用场景：用于比较查询中的边界值。

示例：



## Timestamp（时间戳）

一个专用的数据类型，表示时间戳，精度为秒。常用于 MongoDB 的内部操作（如 oplog）

应用场景：记录事件时间点，适用于事务和日志操作。

示例：



## Undefined（未定义）

一个废弃的数据类型，表示未定义值。MongoDB 不推荐使用。

示例：



# 用户管理

## 基本概念

进行用户管理涉及用户的创建、分配角色、认证和登录等操作。

## 连接到 MongoDB

首先，打开你的终端并使用 mongosh 命令连接到 MongoDB 服务器

mongosh --host <hostname> --port <port>

## 切换到目标数据库

在 MongoDB 中，用户是针对特定数据库创建的，使用 use 命令切换到你要创建用户的数据库：

use <database\_name>

## 创建用户

使用 db.createUser 命令创建用户并分配角色。

例如，创建一个名为 testuser 的用户，密码为 password123，并赋予 readWrite 和 dbAdmin 角色：

db.createUser({

user: "testuser",

pwd: "password123",

roles: [

{ role: "readWrite", db: "<database\_name>" },

{ role: "dbAdmin", db: "<database\_name>" }

]

})

## 验证用户

创建用户后，你可以使用 db.auth 命令验证用户身份：

db.auth("testuser", "password123")

## 启用身份验证

为了确保只有经过身份验证的用户才能访问 MongoDB，需要启用身份验证。

编辑 MongoDB 配置文件 mongod.conf，并在其中添加以下内容：

security:

authorization: "enabled"

然后重启 MongoDB 服务以应用更改。

## 使用用户身份登录

启用身份验证后，你需要使用创建的用户身份连接到 MongoDB：

mongosh --host <hostname> --port <port> -u "testuser" -p "password123" --authenticationDatabase "<database\_name>"

## 删除用户

使用 db.dropUser 命令删除指定用户。

例如，删除名为 testuser 的用户：

db.dropUser("testuser")

# 连接

## 语法

mongodb://[username:password@]host1[:port1][,...hostN[:portN]][/[defaultauthdb][?options]]

mongodb://：协议头，表示使用 MongoDB。

[username:password@]：（可选）认证信息，包括用户名和密码。

host1[:port1][,...hostN[:portN]]：服务器地址和端口，可以是一个或多个 MongoDB 服务器的地址和端口。

/[defaultauthdb]：（可选）默认认证数据库。

[?options]：（可选）连接选项。

## 示例

### 1、连接到本地 MongoDB 实例（默认端口 27017）

mongodb://localhost

### 2、连接到本地 MongoDB 实例，指定数据库

mongodb://localhost/mydatabase

### 3、使用用户名和密码连接到本地 MongoDB 实例

mongodb://username:password@localhost/mydatabase

### 4、使用java进行连接



# 创建数据库

## 基本概念

在MongoDB中，数据库的创建是一个简单的过程，当你首次向MongoDB中插入数据时，如果数据库不存在，MongoDB会自动创建它。我们只需选择一个数据库名称，并开始向其中插入文档即可。

## 语法

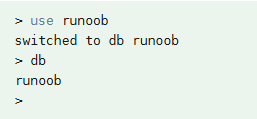
当你使用 use 命令来指定一个数据库时，如果该数据库不存在，MongoDB将自动创建它。

use DATABASE\_NAME

如果数据库不存在，则创建数据库，否则切换到指定数据库。

## 示例

以下实例我们创建了数据库 runoob:



执行 use runoob 命令后，MongoDB 将创建名为 runoob 的新数据库。此时，你可以开始在这个数据库中创建集合和插入文档。

## 默认数据库

MongoDB 中默认的数据库为 test，如果你没有创建新的数据库，集合将存放在 test 数据库中。当您通过 shell 连接到 MongoDB 实例时，如果未使用 use 命令切换到其他数据库，则会默认使用 test 数据库。

例如，在启动 MongoDB 实例并连接到 MongoDB shell 后，如果您开始插入文档而未显式指定数据库，MongoDB 将默认使用 test 数据库。



在这个例子中，如果 test 数据库不存在，则 MongoDB 将自动创建它。

需要注意的是，默认数据库仅在特定情况下才会使用。在实际开发中，您通常会选择自己创建的数据库来存储数据。

**注意:** 在 MongoDB 中，集合只有在内容插入后才会创建，就是说，创建集合(数据表)后要再插入一个文档(记录)，集合才会真正创建。

## 注意事项

1、数据库名不能包含空格、点（.）或美元符号（$）。

2、数据库的创建是自动的，不需要显式创建，除非你需要在创建时指定特定的配置选项。

3、在MongoDB中，只有在数据库中至少有一个集合时，数据库才会在 show dbs 命令的输出中显示。

# 删除数据库

## 语法

# 创建集合

## 基本概念

使用 **createCollection()** 方法来创建集合。

## 语法

db.createCollection(name, options)

参数说明：

name: 要创建的集合名称。

options: 可选参数, 指定有关内存大小及索引的选项。

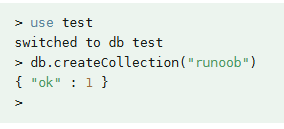
options 可以是如下参数：



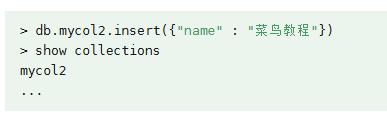
在插入文档时，MongoDB 首先检查固定集合的 size 字段，然后检查 max 字段。

## 示例

在 test 数据库中创建 runoob 集合

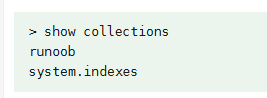


在 MongoDB 中，你不需要创建集合，当你插入一些文档时，MongoDB 会自动创建集合。



# 查看集合

可以使用 show collections 或 show tables 命令



# 更新集合名

## 基本概念

在 MongoDB 中，不能直接通过命令来重命名集合。MongoDB 可以使用 renameCollection 方法来来重命名集合。renameCollection 方法在 MongoDB 的 admin 数据库中运行，可以将一个集合重命名为另一个名称。

## 语法

db.adminCommand({

renameCollection: "sourceDb.sourceCollection",

to: "targetDb.targetCollection",

dropTarget: <boolean>

})

参数说明：

renameCollection：要重命名的集合的完全限定名称（包括数据库名）。

to：目标集合的完全限定名称（包括数据库名）。

dropTarget（可选）：布尔值。如果目标集合已经存在，是否删除目标集合。默认值为 false。

## 示例

假设你要将 test 数据库中的 oldCollection 重命名为 newCollection，可以按以下步骤进行：

1. 确保已连接到 test 数据库

use test

2、运行 renameCollection 命令

db.adminCommand({

renameCollection: "test.oldCollection",

to: "test.newCollection"

});

## 注意事项

1、权限要求：执行 renameCollection 命令需要具有对源数据库和目标数据库的适当权限。通常需要 dbAdmin 或 dbOwner 角色。

2、目标集合不存在：目标集合不能已经存在。如果目标集合存在，则会返回错误。

3、索引和数据：重命名集合会保留所有文档和索引。

## 处理重命名失败的情况

如果重命名过程中发生错误，你可以根据错误消息采取相应的措施。例如，如果目标集合已经存在，可以先删除目标集合（如果确认不需要），然后重新执行重命名操作。

# 删除集合

## 基本概念

MongoDB 中使用 drop() 方法来删除集合。drop() 方法可以永久地从数据库中删除指定的集合及其所有文档，这是一个不可逆的操作，因此需要谨慎使用。

## 语法

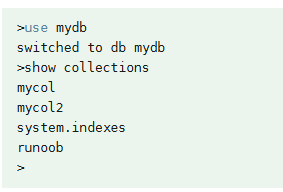
db.collection.drop()

返回值：

如果成功删除选定集合，则 drop() 方法返回 true，否则返回 false。

## 示例

在数据库 mydb 中，我们可以先通过 show collections 命令查看已存在的集合：



接着删除集合 mycol2

>db.mycol2.drop()

true

>

# 插入文档

## 基本概念

文档的数据结构和 JSON 基本一样。所有存储在集合中的数据都是 BSON 格式。

## 语法

db.collection.insertOne()：插入单个文档

db.collection.insertMany()：插入多个文档

db.collection.save()：类似于 insertOne()。如果文档存在，则该文档会被更新；如果文档不存在，则会插入一个新文档。

### insertOne()

用于在集合中插入单个文档

db.collection.insertOne(document, options)

document：要插入的单个文档。

options（可选）：一个可选参数对象，可以包含 writeConcern 和 bypassDocumentValidation 等。

#### 示例

db.myCollection.insertOne({

name: "Alice",

age: 25,

city: "New York"

});

返回结果：

{

"acknowledged": true,

"insertedId": ObjectId("60c72b2f9b1d8b5a5f8e2b2d")

}

### insertMany()

用于在集合中插入多个文档。

db.collection.insertMany(documents, options)

documents：要插入的文档数组。

options（可选）：一个可选参数对象，可以包含 ordered、writeConcern 和 bypassDocumentValidation 等。

#### 示例

db.myCollection.insertMany([

{ name: "Bob", age: 30, city: "Los Angeles" },

{ name: "Charlie", age: 35, city: "Chicago" }

]);

返回结果：

db.myCollection.insertMany([

{ name: "Bob", age: 30, city: "Los Angeles" },

{ name: "Charlie", age: 35, city: "Chicago" }

]);

### db.collection.save()

save() 方法在插入文档时表现得类似于 insertOne()。如果文档包含 \_id 字段且已存在，则该文档会被更新；如果文档不包含 \_id 字段或 \_id 不存在，则会插入一个新文档。

db.collection.save(document, options)

document：要保存的文档。

options（可选）：一个可选参数对象，可以包含 writeConcern 等。

#### 示例

db.myCollection.save({

\_id: ObjectId("60c72b2f9b1d8b5a5f8e2b2d"),

name: "David",

age: 40,

city: "San Francisco"

});

# 更新文档

## 基本概念

更新文档的操作可以使用多种方法实现，常用的方法包括 **updateOne()、updateMany()、replaceOne() 和 findOneAndUpdate()**。

## 更新操作符

### $set

#### 基本概念

$set 是一个常用的更新操作符，用于设置文档中的字段值。如果字段不存在，则会创建该字段；如果字段已存在，则会更新其值。

#### 作用

更新文档中的指定字段值，或者新增字段。

#### 语法

db.collection.updateOne(

<filter>, // 筛选条件

{ $set: { <field>: <value>, ... } }, // 更新操作

<options> // 可选参数

)

#### 示例

##### 1、更新已有字段值

假设有一个集合 users，其中的文档如下：

{ "\_id": 1, "name": "Alice", "age": 25 }

更新 age 字段为 30：

db.users.updateOne(

{ "\_id": 1 }, // 筛选 \_id 为 1 的文档

{ $set: { "age": 30 } } // 使用 $set 更新 age 字段

);

##### 2、新增字段

如果 $set 操作的字段在文档中不存在，则会新增该字段。例如，在上面的文档中新增字段 city：

db.users.updateOne(

{ "\_id": 1 },

{ $set: { "city": "New York" } }

);

执行后，文档变为：

{ "\_id": 1, "name": "Alice", "age": 30, "city": "New York" }

#### 结合数组操作

$set 也可以用来更新数组字段中的某个元素

##### 更新数组中的某个元素

假设有以下文档：

{ "\_id": 1, "items": ["apple", "banana", "cherry"] }

将数组中的第二个元素更新为 "blueberry"：

db.collection.updateOne(

{ "\_id": 1 },

{ $set: { "items.1": "blueberry" } } // 注意数组索引从 0 开始

);

执行后，文档变为：

{ "\_id": 1, "items": ["apple", "blueberry", "cherry"] }

##### 为数组字段新增值（非覆盖）

如果需要新增数组字段而不是覆盖已有数组，可以配合 $set 使用新数组。

db.collection.updateOne(

{ "\_id": 1 },

{ $set: { "tags": ["mongodb", "database"] } }

);

## updateOne()

用于更新匹配过滤器的单个文档

### 语法

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

filter：用于查找文档的查询条件。

update：指定更新操作的文档或更新操作符。

options：可选参数对象，如 upsert、arrayFilters 等。

### 示例

db.myCollection.updateOne(

{ name: "Alice" }, // 过滤条件

{ $set: { age: 26 } }, // 更新操作

{ upsert: false } // 可选参数

);

## updateMany()

用于更新所有匹配过滤器的文档。

### 语法

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

filter：用于查找文档的查询条件。

update：指定更新操作的文档或更新操作符。

options：可选参数对象，如 upsert、arrayFilters 等。

### 示例

db.myCollection.updateMany(

{ age: { $lt: 30 } }, // 过滤条件

{ $set: { status: "active" } }, // 更新操作

{ upsert: false } // 可选参数

);

## replaceOne()

用于替换匹配过滤器的单个文档，新的文档将完全替换旧的文档。

### 语法

db.collection.replaceOne(filter, replacement, options)

filter：用于查找文档的查询条件。

replacement：新的文档，将替换旧的文档。

options：可选参数对象，如 upsert 等。

### 示例

db.myCollection.replaceOne(

{ name: "Bob" }, // 过滤条件

{ name: "Bob", age: 31 } // 新文档

);

## findOneAndUpdate()

用于查找并更新单个文档，可以选择返回更新前或更新后的文档。

### 语法

db.collection.findOneAndUpdate(filter, update, options)

filter：用于查找文档的查询条件。

update：指定更新操作的文档或更新操作符。

options：可选参数对象，如 projection、sort、upsert、returnDocument 等。

### 示例

db.myCollection.findOneAndUpdate(

{ name: "Charlie" }, // 过滤条件

{ $set: { age: 36 } }, // 更新操作

{ returnDocument: "after" } // 可选参数，返回更新后的文档

);

# 删除文档

## 基本概念

常用的删除文档方法包括 deleteOne()、deleteMany() 以及 findOneAndDelete()。

## 使用场景

1、数据清理：删除不再需要的旧数据或无效数据。

2、数据修正：在数据修正过程中删除错误的或重复的文档。

3、自动化任务：在自动化脚本或任务中，根据特定条件删除文档。

## deleteOne()

用于删除匹配过滤器的单个文档。

### 语法

db.collection.deleteOne(filter, options)

filter：用于查找要删除的文档的查询条件。

options（可选）：一个可选参数对象

### 示例

db.myCollection.deleteOne({ name: "Alice" });

返回结果：

{

"acknowledged": true,

"deletedCount": 1

}

## deleteMany()

用于删除所有匹配过滤器的文档。

### 语法

db.collection.deleteMany(filter, options)

filter：用于查找要删除的文档的查询条件。

options（可选）：一个可选参数对象。

### 示例

db.myCollection.deleteMany({ status: "inactive" });

返回结果：

{

"acknowledged": true,

"deletedCount": 1

}

## findOneAndDelete()

用于查找并删除单个文档，并可以选择返回删除的文档。

### 语法

db.collection.findOneAndDelete(filter, options)

filter：用于查找要删除的文档的查询条件。

options：可选参数对象，如 projection、sort 等。

### 示例

db.myCollection.findOneAndDelete(

{ name: "Charlie" },

{ projection: { name: 1, age: 1 } }

);

findOneAndDelete 返回被删除的文档，如果找不到匹配的文档，则返回 null。

## 删除操作的选项

这些删除方法的 options 参数通常可以包含以下选项：

writeConcern：指定写操作的确认级别。

collation：指定比较字符串时使用的排序规则。

projection（仅适用于 findOneAndDelete）：指定返回的字段。

sort（仅适用于 findOneAndDelete）：指定排序顺序以确定要删除的文档。

# 查询文档

## 基本概念

MongoDB 查询文档使用 find()、findOne() 方法。

## 语法

db.collection.find(query, projection)

query（查询条件）：用于查找文档的查询条件。默认为 {}，即匹配所有文档。

projection（投影条件）（可选）：指定返回结果中包含或排除的字段。

投影规则

**显示字段**：指定要返回的字段，将其值设为 1。

示例：{ name: 1, age: 1 }，返回 name 和 age 字段。

**隐藏字段**：指定不返回的字段，将其值设为 0。

示例：{ \_id: 0 }，不返回 \_id 字段。

投影的限制

显示字段和隐藏字段不能混用，除非 \_id 字段。

正确：{ name: 1, age: 1, \_id: 0 }

错误：{ name: 1, age: 0 }（会报错）

## findOne()

findOne() 方法用于查找集合中的单个文档。如果找到多个匹配的文档，它只返回第一个。

### 语法

db.collection.findOne(query, projection)

query：用于查找文档的查询条件。默认为 {}，即匹配所有文档。

projection（可选）：指定返回结果中包含或排除的字段。

查找单个文档：

db.myCollection.findOne({name:"Alice"});

查找单个文档，并只返回指定字段

db.myCollection.findOne(

{ name: "Alice" },

{ name: 1, age: 1, \_id: 0 }

);

## 示例

### 1、查找所有文档

db.myCollection.find();

### 2、按条件查找文档

db.myCollection.find({ age: { $gt: 25 } });

### 3、按条件查找文档，并只返回指定字段

db.myCollection.find(

{ age: { $gt: 25 } },

{ name: 1, age: 1, \_id: 0 }

);

### 4、使用易读的方式查询数据

db.col.find().pretty()

pretty() 方法以格式化的方式来显示所有文档。



## 高级查找方法

### 1、使用比较操作符

查找年龄大于 25 的文档

db.myCollection.find({ age: { $gt: 25 } });

### 2、使用逻辑操作符

查找年龄大于 25 且城市为 "New York" 的文档:

db.myCollection.find({

$and: [

{ age: { $gt: 25 } },

{ city: "New York" }

]

});

# 条件操作符

## 基本概念

条件操作符用于在查询文档时进行过滤、比较和逻辑操作。

## 比较操作符



### $eq（等于）

{ field: { $eq: value } }

### $ne（不等于）

{ field: { $ne: value } }

### $gt（大于）

{ field: { $gt: value } }

### $gte（大于等于）

{ field: { $gte: value } }

### $lt（小于）

db.collection.find({ age: { $lt: 25 } })

### $lte（小于等于）

{ field: { $lte: value } }

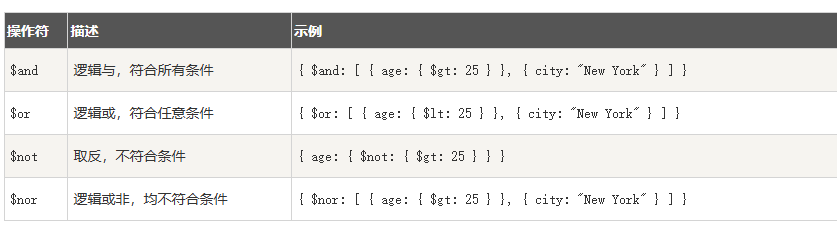
### $in（在数组中）

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

### $nin（不在数组中）

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

## 逻辑操作符



### $and（逻辑与，符合所有条件）

逻辑与，符合所有条件

语法：

{ $and: [ { condition1 }, { condition2 } ] }

查找年龄大于 25 且小于 35 的人

db.collection.find({$and:[{age:{$gt:25}},{age:{$lt:35}}]})

### $or（逻辑或，符合任意条件）

语法：

{ $or: [ { condition1 }, { condition2 } ] }

查找年龄大于 25 或小于 18 的人

db.collection.find({ $or: [{ age: { $gt: 25 } }, { age: { $lt: 18 } }] })

### $not（取反，不符合条件）

语法：

{ field: { $not: { condition } } }

查找年龄不大于 25 的人:

db.collection.find({ age: { $not: { $gt: 25 } } })

### $nor（逻辑或非，均不符合条件）

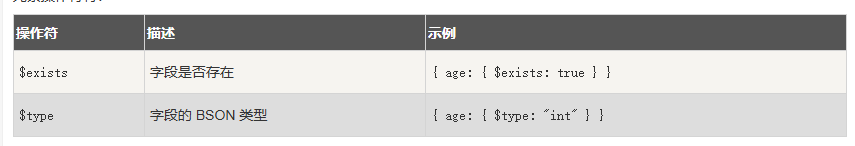
语法：

{ $nor: [ { condition1 }, { condition2 } ] }

查找年龄不大于 25 且不小于 18 的人:

db.collection.find({ $nor: [{ age: { $gt: 25 } }, { age: { $lt: 18 } }] })

## 元素操作符



### $exists (字段是否存在)

语法：

{ field: { $exists: boolean } }

查找包含 age 字段的文档:

db.collection.find({ age: { $exists: true } })

### $type (字段类型)

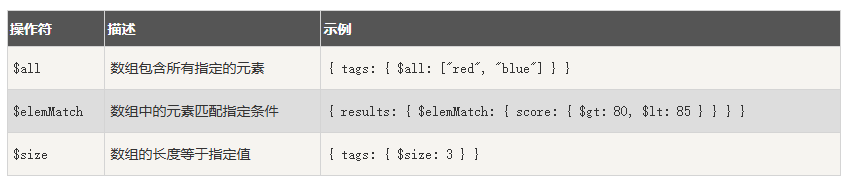
语法格式：

{ field: { $type: "type" } }

查找 age 字段为整数类型的文档

db.collection.find({ age: { $type: "int" } })

## 数组操作符



### $all（数组包含所有指定的元素）

语法：

{ field: { $all: [value1, value2, ...] } }

查找 tags 数组中包含 "mongodb" 和 "database" 的文档:

db.collection.find({ tags: { $all: ["mongodb", "database"] } })

### $elemMatch（数组中至少有一个元素符合条件）

语法：

{ field: { $elemMatch: { condition } } }

查找 items 数组中 quantity 大于 10 的元素:

db.collection.find({ items: { $elemMatch: { quantity: { $gt: 10 } } } })

### $size（数组的大小）

语法：

{ field: { $size: value } }

查找 tags 数组大小为 3 的文档:

db.collection.find({ tags: { $size: 3 } })

## 其他操作符



### $regex（正则表达式匹配）

语法：

{ field: { $regex: "pattern" } }

查找 name 以 "John" 开头的文档

db.collection.find({ name: { $regex: /^John/ } })

### $text（进行文本搜索）

### $where（使用 JavaScript 表达式进行条件过滤）

### $near（查找接近指定点的文档）

### $geoWithin（查找在指定地理区域内的文档）

# $type 操作符

## 基本概念

在 MongoDB 中，$type 操作符用于查询具有指定类型的字段的文档。MongoDB 的 $type 操作符用于查询字段的 BSON 数据类型。它允许您指定一个或多个类型，并返回匹配这些类型的文档。

## 语法

db.collection.find({ field: { $type: <type> } })

field：要检查类型的字段。

type：指定的 BSON 类型，可以是类型的数字代码或类型名称的字符串。

## 示例

### 1、查找字段类型为字符串的文档

db.myCollection.find({ fieldName: { $type: "string" } })

### 2、查找字段类型为数字的文档，例如，查找 age 字段类型为整数的文档：

db.myCollection.find({ age: { $type: "int" } })

# Limit与Skip方法

## 基本概念

如果你需要在 MongoDB 中读取指定数量的数据记录，可以使用 MongoDB 的 limit() 方法，如果想跳过指定数量的文档读取数据可以使用 skip() 方法。limit() 方法用于限制查询结果返回的文档数量，而 skip() 方法用于跳过指定数量的文档。这两个方法通常一起使用，可以用来实现分页查询或在大型数据集上进行分批处理。

## limit()方法

### 基本概念

用于限制查询结果返回的文档数量

### 语法

db.collection.find().limit(<limit>)

<limit>：返回的文档数量

### 示例

// 返回前 10 个文档

db.myCollection.find().limit(10);

## skip() 方法

### 基本概念

我们除了可以使用 limit() 方法来读取指定数量的数据外，还可以使用 skip() 方法来跳过指定数量的数据，skip() 方法同样接受一个数字参数作为跳过的记录条数。skip() 方法用于跳过指定数量的文档，从而实现分页或分批查询。

### 语法

db.collection.find().skip(<skip>)

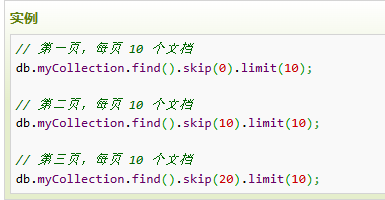
<skip>：要跳过的文档数量。

### 示例

// 跳过前 10 个文档，返回接下来的 10 个文档

db.myCollection.find().skip(10).limit(10);

分页查询：



**注:**skip()方法默认参数为 0 。

## 注意事项

1、skip() 和 limit() 方法通常用于配合使用，以实现分页查询。但是在大型数据集上使用 skip() 可能会导致性能问题，因为 MongoDB 在执行查询时需要扫描并跳过指定数量的文档，因此建议仅在需要时才使用 skip() 方法，尽量避免在大型数据集上连续使用。

2、当结合 skip() 和 limit() 时，skip() 应该在 limit() 之前使用，以避免意外行为。

3、通过使用 limit() 和 skip() 方法，您可以实现在 MongoDB 中对查询结果进行分页处理，提高查询的灵活性和性能。

# 排序

## 基本概念

在 MongoDB 中使用 sort() 方法对数据进行排序，sort() 方法可以通过参数指定排序的字段，并使用 1 和 -1 来指定排序的方式，其中 1 为升序排列，而 -1 是用于降序排列。

## 语法

db.collection.find().sort({ field1: 1, field2: -1 })

{ field1: 1, field2: -1 }：指定要排序的字段及排序顺序。1 表示升序，-1 表示降序。

## 示例

### 1、按 age 字段升序排序

// 按 age 字段升序排序

db.myCollection.find().sort({ age: 1 });

### 2、按 createdAt 字段降序排序

// 按 createdAt 字段降序排序

db.myCollection.find().sort({ createdAt: -1 });

### 3、可以指定多个字段进行排序，MongoDB 将按照指定的字段顺序依次排序。

// 先按 age 字段升序排序，再按 createdAt 字段降序排序

db.myCollection.find().sort({ age: 1, createdAt: -1 });

## 注意事项

1、MongoDB 在执行排序时会对查询结果进行排序，因此可能会影响性能，特别是在大型数据集上排序操作可能会较慢。

2、如果排序字段上有索引，排序操作可能会更高效。在执行频繁的排序操作时，可以考虑创建适当的索引以提高性能。

3、通过使用 sort() 方法，您可以方便地对 MongoDB 查询结果进行排序，以满足不同排序需求，并根据需要按升序或降序排列文档。

4、skip(), limilt(), sort()三个放在一起执行的时候，执行的顺序是先 sort(), 然后是 skip()，最后是显示的 limit()。

# 索引

## 基本概念

索引通常能够极大的提高查询的效率，如果没有索引，MongoDB在读取数据时必须扫描集合中的每个文件并选取那些符合查询条件的记录。索引是特殊的数据结构，索引存储在一个易于遍历读取的数据集合中，索引是对数据库表中一列或多列的值进行排序的一种结构。

## 常见的类型

### 单字段索引

基于单个字段的索引。

### 复合索引

基于多个字段组合的索引。

### 文本索引

用于支持全文搜索。

### 地理空间索引

用于地理空间数据的查询。

### 哈希索引

用于对字段值进行哈希处理的索引。

## 创建索引

MongoDB 使用 createIndex() 方法来创建索引。

### 语法

db.collection.createIndex( keys, options )

db：数据库的引用。

collection：集合的名称。

keys：一个对象，指定了字段名和索引的排序方向（1 表示升序，-1 表示降序）。

options：一个可选参数，可以包含索引的额外选项。

options 参数是一个对象，可以包含多种配置选项，以下是一些常用的选项：

unique：如果设置为 true，则创建唯一索引，确保索引字段的值在集合中是唯一的。

background：如果设置为 true，则索引创建过程在后台运行，不影响其他数据库操作。

name：指定索引的名称，如果不指定，MongoDB 会根据索引的字段自动生成一个名称。

sparse：如果设置为 true，创建稀疏索引，只索引那些包含索引字段的文档。

expireAfterSeconds：设置索引字段的过期时间，MongoDB 将自动删除过期的文档。

v：索引版本，通常不需要手动设置。

weights：为文本索引指定权重。

### 示例

1、创建 age 字段的升序索引

db.myCollection.createIndex({ age: 1 })

2、创建 name 字段的文本索引

db.myCollection.createIndex({ name: "text" });

## 创建哈希索引

从 MongoDB 3.2 版本开始，可以使用哈希索引对字段进行哈希，以支持大范围的数值查找。

db.collection.createIndex( { field: "hashed" } )

## 查看索引

使用 getIndexes() 方法可以查看集合中的所有索引

db.collection.getIndexes()

## 删除索引

使用 dropIndex() 或 dropIndexes() 方法可以删除索引

### 示例

#### 1、删除指定的索引

db.collection.dropIndex( "indexName" )

#### 2、删除所有索引

db.collection.dropIndexes()

## 索引策略

### 1、查询频率

优先考虑那些经常用于查询的字段。

### 2、字段基数

字段值的基数越高（即唯一值越多），索引的效果越好。

### 3、索引大小

索引的大小会影响数据库的内存占用和查询性能。

## 索引优化

### 1、选择合适的索引类型

根据查询需求选择合适的索引类型

### 2、创建复合索引

对于经常一起使用的字段，考虑创建复合索引以提高查询效率。

### 3、监控索引性能

定期监控索引的使用情况，根据实际需求调整索引。

## 注意事项

1、索引虽然可以提高查询性能，但也会增加写操作的开销。因此，在创建索引时需要权衡查询性能和写入性能。

2、索引会占用额外的存储空间，特别是对于大型数据集，需要考虑索引的存储成本。

3、通过合理地设计和使用索引，可以大大提高 MongoDB 数据库的查询性能和响应速度，从而更好地支持应用程序的需求。

# 聚合

# 关系

# 数据库引用