MongoDB

MongoDB属于非关系型数据库，它是由C++编写的分布式文档数据库。内部使用类似于Json的bson二进制格式。

中文手册

https://www.w3cschool.cn/mongodb/

安装

https://www.mongodb.com/try/download/community

自行下载对应操作系统的MongoDB，并运行它。

windows可以下载官方zip，解压即可使用。

组件	文件名
Server	mongod.exe
Router	mongos.exe, Query Router, Sharding Cluster
Client	mongo.exe
MonitoringTools	mongostat.exe,mongotop.exe
ImportExportTools	mongodump.exe, mongorestore.exe, mongoexport.exe, mongoimport.exe
MiscellaneousTools	bsondump.exe, mongofiles.exe, mongooplog.exe, mongoperf.exe

运行

```
$ cd  /o/mongodb3.6/bin
$ ./mongod.exe
2019-08-02T03:26:13.234-0700 I STORAGE  [initandlisten] exception in initAndListen: NonExistentPath: Data directory O:\data\db\ not found., terminating
启动服务出错，原因在于找不到数据目录。默认是/data/db
windows下在当前盘符根目录下创建目录即可`o:/data/db`
```

选项说明

- --bind_ip ip 逗号分隔IP地址。默认localhost
- --bind_ip_all 绑定所有本地IP地址
- --port port 端口，默认27017
- --dbpath path 数据路径，缺省为`\data\db\`。windows下缺省就是当前盘符的根目录
- --logpath path 指定日志文件，替代stdout，说明默认是控制台打印日志
- -f file 指定配置文件，yaml格式
- 注册windows服务
  - --install 注册windows服务
  - --serviceName name 服务名称
  - --serviceDisplayName name 服务显示名

配置文件

mongodb配置使用YAML格式

- 嵌套使用缩进完成，不支持Tab等制表符，支持空格
  - 缩进空格数不限制，只要同一级元素对齐就行
- 冒号后要有空格
- 大小写敏感
- #表示注释
- 字符串不需要引号，有特殊字符串时可以使用引号
- 布尔
  - true、True、TRUE、yes、YES都是真
  - false、False、FALSE、no、NO都是假
- null、Null、~波浪线都是空，不指定值默认也是空

Yaml参考 https://www.w3cschool.cn/iqmrhf/dotvpozt.html

配置 http://mongoing.com/docs/reference/configuration-options.html

```
systemLog:
    destination: file
    path: 'o:/mongodb3.6/logs/mongod.log'
    logAppend:  true
storage:
    dbPath: "o:/mongodb3.6/db"
net:
    bindIp: "127.0.0.1"
    port: 27017
```

选项

- systemLog
  - destination，缺省是输出日志到std，file表示输出到文件
  - path，日志文件路径。文件目录必须存在
  - logAppend，true表示在已存在的日志文件追加。默认false，每次启动服务，重新创建新的日志。
- storage
  - dbPath，必须指定mongodb的数据目录，**目录必须存在**
- net
  - bindIp，缺省绑定到127.0.0.1
  - port，端口，缺省为27017，客户端连接用

Windows下注册为服务的命令如下，使用了配置文件：

`$ mongod.exe -f "o:/mongodb3.6/bin/mongod.yml" --serviceName mongod --serviceDisplayName mongo --install`

注意，注册服务得需要管理员权限。

```
storage:
    dbPath: "o:/mongodb3.6/db"
net:
    bindIp: "127.0.0.1"
    port: 27017
```

没有配置日志，信息将显示在控制台中

```
$ pwd
/o/mongodb3.6
$ mongod.exe -f ./mongod.yml
```

客户端

客户端连接

```
$ bin/mongo.exe
MongoDB shell version v3.6.13
help 打开帮助
show dbs      查看当前有哪些库
use blog      有就切换过去，没有就创建后切换过去。刚创建的并不在数据库列表中，需要写入数据后才能看到
db            查看当前数据库
db.users.insert({user:"tom", age:20}) db指代当前数据库；users集合名
```

也可以使用官方的可视化工具Compass。https://www.mongodb.com/products/compass

驱动

驱动 https://www.mongodb.com/docs/drivers/

Go驱动 https://www.mongodb.com/docs/drivers/go/current/

驱动安装

```
$ go get go.mongodb.org/mongo-driver/mongo
```

连接字符串

https://www.mongodb.com/docs/manual/reference/connection-string/#examples

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

mongodb://wayne:wayne@mongodb0.example.com:27017
```

连接例子 https://www.mongodb.com/docs/drivers/go/current/fundamentals/connection/#connection-example

快速入门 https://www.mongodb.com/docs/drivers/go/current/quick-start/

```
package main

import (
    "context"
    "fmt"
    "log"
    "time"

    "go.mongodb.org/mongo-driver/mongo"
    "go.mongodb.org/mongo-driver/mongo/options"
)

var client *mongo.Client
var db *mongo.Database
var users *mongo.Collection

func init() {
    url := "mongodb://127.0.0.1:27017//"
    opts := options.Client()
    opts.ApplyURI(url).SetConnectTimeout(5 * time.Second)

    var err error
    client, err = mongo.Connect(context.TODO(), opts) // context.TODO() 空上下文
    if err != nil {
        log.Fatal(err)
    }

    err = client.Ping(context.TODO(), nil)
```

```
    if err != nil {
        log.Fatal(err)
    }
    fmt.Println("~~~~~~~~~~~~~~~~~~~~~~~~~~")

    // 不能用:=
    db = client.Database("test")    // 库
    users = db.Collection("users") // 集合，相当于表
}
```

```
// 断开连接放到其他函数里
defer func() {
    if err := c.Disconnect(context.TODO()); err != nil {
        log.Fatal(err)
    }
}()
fmt.Println("~~~~~~~~~~~~~~~~~~~~~~~~~~")
```

基本概念

MongoDB中可以创建使用多个库，但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。

RDBMS	MongoDB
Database	Database
Table	Collection
Row	Document
Column	Field
Join	Embedded Document嵌入文档或Reference引用
Primary Key	主键 (MongoDB提供了key为 _id)

Go Driver使用，官方博客 https://www.mongodb.com/blog/post/mongodb-go-driver-tutorial

数据封装

```go
type User struct {
    ID   primitive.ObjectID `bson:"_id,omitempty"`
    Name string
    Age  int
}

func (u User) String() string {
    return fmt.Sprintf("<%s: %s,%d>", u.ID, u.Name, u.Age)
}
```

Tag参考 https://www.mongodb.com/docs/drivers/go/upcoming/fundamentals/bson/#struct-tags

User结构体中ID一定要使用omitempty，新增时结构体ID不设置则为零值，提交时不会提交ID，数据库自动生成_id

ObjectId有12字节组成，参考 bson/primitive/objectid.go/NewObjectID()函数

- 4字节时间戳
- 5字节进程唯一值
- 3字节随机数，每次加1

插入数据

操作参考 https://www.mongodb.com/docs/drivers/go/current/usage-examples/

```go
// 插入一条
func insertOne() {
    tom := User{Name: "tom", Age: 33}
    insertResult, err := users.InsertOne(context.TODO(), tom)
    if err != nil {
        log.Fatal(err)
    }
    fmt.Println(insertResult.InsertedID)
}

// 插入多条
func insertMany() {
    jerry := User{Name: "jerry", Age: 20}
    ben := User{Name: "ben", Age: 16}
    insertManyResult, err := users.InsertMany(context.TODO(), []interface{}{jerry, ben})
    if err != nil {
        log.Fatal(err)
    }
    fmt.Println("~~~~~~~~~~~~~~~~~~~~~~~~~~")
    fmt.Println(insertManyResult.InsertedIDs...)
}
```

BSON

https://www.mongodb.com/docs/drivers/go/upcoming/fundamentals/bson/

MOngoDB的Go库提供的构建BSON的数据类型分为4种

- `D`: An ordered representation of a BSON document (slice)，表示有序的，切片且元素是二元的
- `M`: An unordered representation of a BSON document (map)，表示无序的，map且元素是kv对
- `A`: An ordered representation of a BSON array
- `E`: A single element inside a D type

具体使用看下面的例子

查询

单条查询

```go
// 找一条
func findOne() {
    // 条件
    // filter := bson.D{{"name", "tom"}} // slice
    // filter := bson.D{{"name", bson.D{{"$eq", "tom"}}}}
    filter := bson.M{"name": "tom"} // map
    // filter := bson.M{"name": bson.M{"$ne": "jerry"}}
    // filter := bson.D{{}} // 没有条件全部都符合
    var u User
    err := users.FindOne(context.TODO(), filter).Decode(&u)
    if err != nil {
        if err == mongo.ErrNoDocuments {
            // 说明没有任何匹配文档
            log.Println("没有匹配的文档")
            return
        }
        log.Fatal(err)
    }
    fmt.Println(u)
}
```

多条查询

```go
// 查多条，遍历结果
func findMany1() {
    filter := bson.M{} // 无条件，全部符合
    cursor, err := users.Find(context.TODO(), filter)
    if err != nil {
        log.Fatal(err)
    }
    var results []*User
    for cursor.Next(context.TODO()) {
        var u User
        err = cursor.Decode(&u)
        if err != nil {
```

```
            log.Fatal(err)
        }
        results = append(results, &u) // 装入容器
    }
    if err := cursor.Err(); err != nil {
        log.Fatal(err)
    }
    cursor.Close(context.TODO()) // 关闭游标
    fmt.Println(results)
}

// 查多条，成批装入容器
func findMany2() {
    filter := bson.M{} // 无条件，全部符合
    var results []*User
    cursor, err := users.Find(context.TODO(), filter)
    if err != nil {
        log.Fatal(err)
    }
    err = cursor.All(context.TODO(), &results)
    if err != nil {
        log.Fatal(err)
    }
    cursor.Close(context.TODO()) // 关闭游标
    for i, r := range results {
        fmt.Println(i, r)
    }
}
```

查询条件

改造上面的findMany2函数，可以使用下面表格中不同filter

```
func findByFilter(filter interface{}) {
    var results []*User
    cursor, err := users.Find(context.TODO(), filter)
    if err != nil {
        log.Fatal(err)
    }
    err = cursor.All(context.TODO(), &results)
    if err != nil {
        log.Fatal(err)
    }
    cursor.Close(context.TODO()) // 关闭游标
    fmt.Println(results)
}
```

比较符号	含义	filter示例
$lt	小于	`bson.M{"age": bson.M{"$lt": 20}}`
$gt	大于	`bson.M{"age": bson.M{"$gt": 20}}`
$lte	小于等于	`bson.M{"age": bson.M{"$lte": 20}}`
$gte	大于等于	`bson.M{"age": bson.M{"$gte": 20}}`
$ne	不等于	`bson.M{"age": bson.M{"$ne": 20}}`
$eq	等于，可以不用这个符号	`bson.M{"age": bson.M{"$eq": 20}}` `bson.M{"age": 20}`
$in	在范围内	`bson.M{"age": bson.M{"$in": []int{16, 33}}}`
$nin	不在范围内	`bson.M{"age": bson.M{"$nin": []int{16, 33}}}`

https://www.mongodb.com/docs/manual/reference/operator/query/and/

逻辑符号	含义	filter示例
$and	与	`bson.M{"$and": []bson.M{{"name": "tom"}, {"age": 33}}}` `bson.M{"$and": []bson.M{{"name": "tom"}, {"age": bson.M{"$gt": 40}}}}`
$or	或	`bson.M{"$or": []bson.M{{"name": "tom"}, {"age": bson.M{"$lt": 20}}}}`
$not	非	`bson.M{"age": bson.M{"$not": bson.M{"$gte": 20}}}`

`bson.M{"age": bson.M{"$gte": 20}}` 取反为 `bson.M{"age": bson.M{"$not": bson.M{"$gte": 20}}}`

元素	含义	示例
$exists	文档中是否有这个字段	`bson.M{"Name": bson.M{"$exists": true}}`
$type	字段是否是指定的类型	`bson.M{"age": bson.M{"$type": 16}}`

`bson.M{"name": bson.M{"$exists": true}}` 标识所有具有Name字段的文档，注意Name和name不一样。

常用类型，参考 https://docs.mongodb.com/manual/reference/operator/query/type/#op._S_type

- 字符串类型编码为2，别名为string
- 整型编码为16，别名为int
- 长整型编码为18，别名为long

改造函数findByFilter为findAll，如下

```go
func findAll(filter interface{}, opt *options.FindOptions) {
    var results []*User
    cursor, err := users.Find(context.TODO(), filter, opt)
    if err != nil {
        log.Fatal(err)
    }
    err = cursor.All(context.TODO(), &results)
    if err != nil {
        log.Fatal(err)
    }
    cursor.Close(context.TODO()) // 关闭游标
    fmt.Println(results)
}
```

投影

```go
filter := bson.M{"age": bson.M{"$gt": 18}}
opt := options.Find()
opt.SetProjection(bson.M{"name": false, "age": false}) // name、age字段不投影，都显示为零值
findAll(filter, opt)
```

```go
opt.SetProjection(bson.M{"name": true}) // name投影，age字段零值显示
```

排序

```go
opt.SetSort(bson.M{"age": 1}) // 升序
opt.SetSort(bson.M{"age": -1}) // 降序
```

分页

```go
opt.SetSkip(1)  // offset
opt.SetLimit(1) // limit
```

更新

更新操作符	含义	示例
$inc	对给定字段数字值增减	`bson.M{"$inc": bson.M{"age": -5}}`
$set	设置字段值，如果字段不存在则创建	`bson.M{"$set": bson.M{"gender": "M"}}`
$unset	移除字段	`{'$unset':{'Name':""}}`

```go
// 更新一个
func updateOne() {
    filter := bson.M{"age": bson.M{"$exists": true}} // 所有有age字段的文档
    update := bson.M{"$inc": bson.M{"age": -5}}      // age字段减5
    ur, err := users.UpdateOne(context.TODO(), filter, update)
    if err != nil {
        log.Fatal(err)
    }
    fmt.Println(ur.MatchedCount, ur.ModifiedCount)
}
```

```go
// 更新多个
func updateMany() {
    filter := bson.M{"age": bson.M{"$exists": true}} // 所有有age字段的文档
    update := bson.M{"$set": bson.M{"gender": "M"}}  // 为符合条件的文档设置gender字段
    users.UpdateMany(context.TODO(), filter, update)
}
```

```go
update := bson.M{"$unset": bson.M{"gender": ""}} // 为符合条件的文档移除gender字段
```

```go
// 找到一批更新第一个，ReplaceOne更新除ID以外所有字段
filter := bson.M{"age": bson.M{"$exists": true}} // 所有有age字段的文档
replacement := User{Name: "Sam", Age: 48}
ur, err := users.ReplaceOne(context.TODO(), filter, replacement)
if err != nil {
    log.Fatal(err)
}
fmt.Println(ur.MatchedCount, ur.ModifiedCount)
```

删除

```go
// 删除一个
func deleteOne() {
    filter := bson.M{} // 没有条件，匹配所有文档
    dr, err := users.DeleteOne(context.TODO(), filter)
    if err != nil {
        log.Fatal(err)
    }
    fmt.Println(dr.DeletedCount)
}

// 删除多个
func deleteMany() {
    filter := bson.M{} // 没有条件，匹配所有文档
    dr, err := users.DeleteMany(context.TODO(), filter)
    if err != nil {
        log.Fatal(err)
    }
```

```
    fmt.Println(dr.DeletedCount)
}
```

`users.DeleteMany(context.TODO(), bson.M{})` 删除所有文档，**危险**！