MongoDB 讲师:梁开权(逍遥)

课程目标

- 了解MongoDB数据库结构
- 掌握基本的CRUD和高级查询操作
- 掌握Spring Data MongoDB的使用

MongoDB介绍

简介

MongoDB是一个基于分布式文件存储的数据库。由C++语言编写。旨在为WEB应用提供可扩展的高性能数据存储解决方案。 MongoDB是一个介于关系数据库和非关系数据库之间的产品，是非关系数据库当中功能最丰富，最像关系数据库的。它支持的数据结构非常松散，是类似json的bson格式，因此可以存储比较复杂的数据类型。Mongo最大的特点是它支持的查询语言非常强大，其语法有点类似于面向对象的查询语言，几乎可以实现类似关系数据库单表查询的绝大部分功能，而且还支持对数据建立索引。

特点

- 高性能
- 易部署
- 易使用
- 非常方便的存储数据

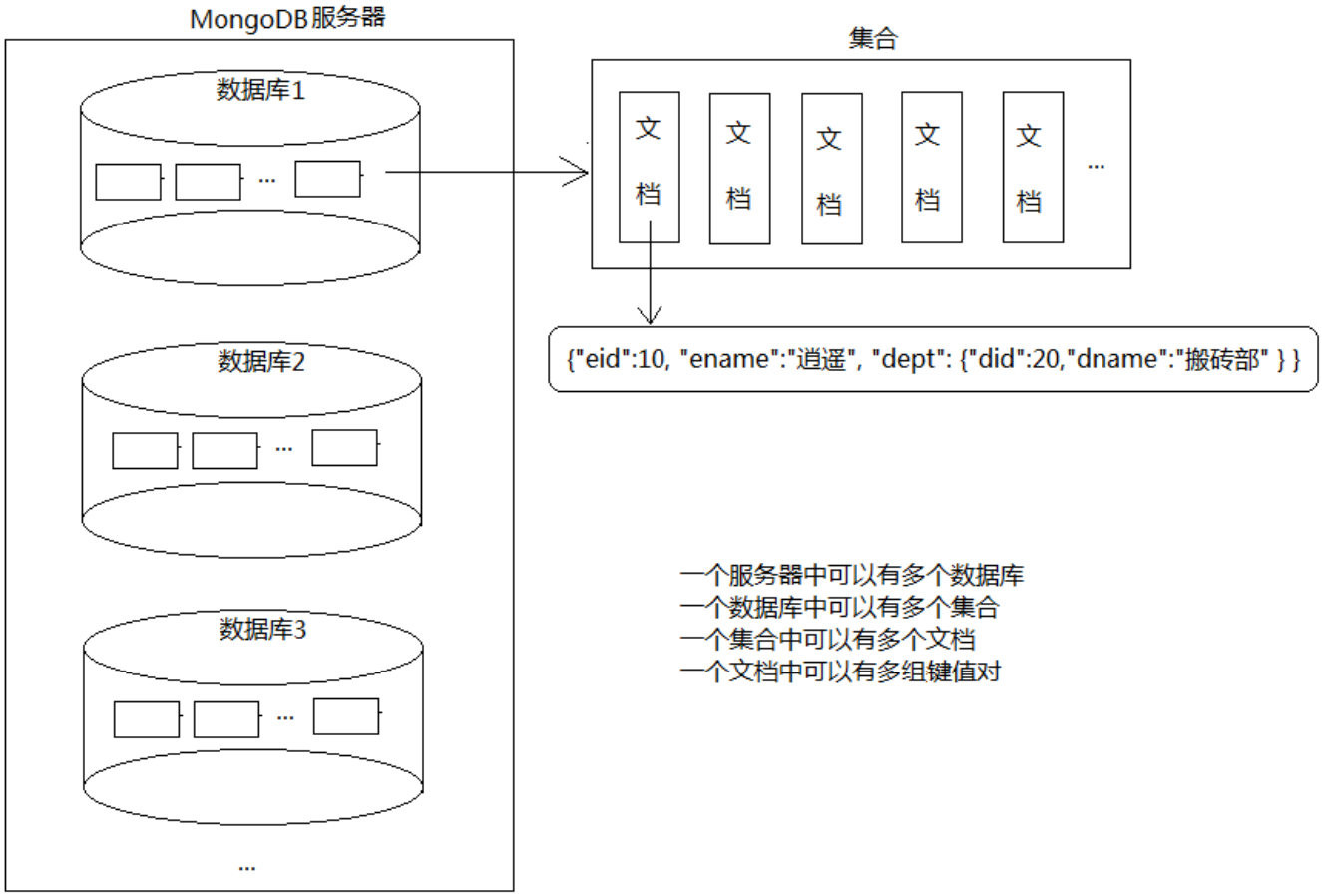
对比MySQL

	Mongodb	Mysql
数据库模型	非关系型	关系型
存储方式	虚拟内存+持久化	不同的引擎有不同的存储方式
查询语句	独特的Mongodb查询方式	传统Sql语句
架构特点	可以通过副本集，以及分片来实现高可用	常见有单点，M-S，MHA，MMM，Cluster等架构方式
数据处理方式	基于内存，将热数据存在物理内存中，从而达到高速读写	不同的引擎拥有其自己的特点
成熟度	新兴数据库，成熟度较低	拥有较为成熟的体系，成熟度较高
广泛度	Nosql数据库中，mongodb是较为完善的DB之一，使用人群也在不断增长	开源数据库的份额在不断增加，mysql的份额也在持续增长

目前环境下，只要对事务要求不高的业务都能被MongoDB所取代，属于及其热门的NoSQL数据库

数据库结构

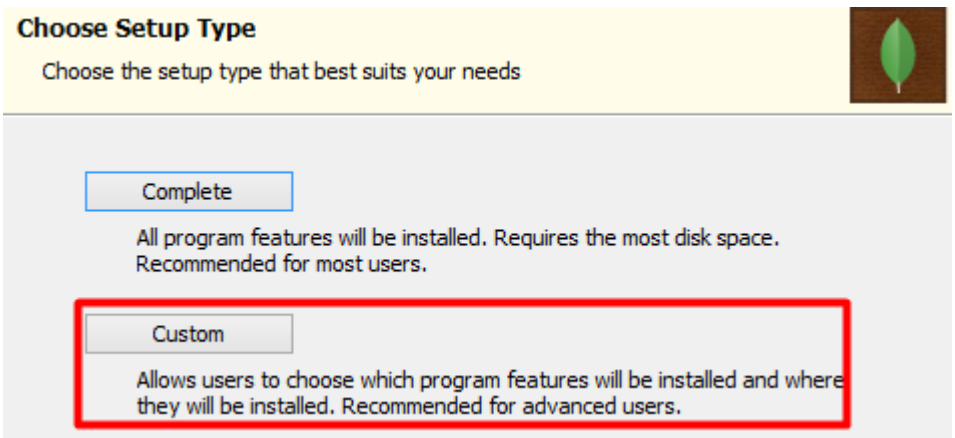
MongoDB属于NoSQL数据库，自然也是没有表相关概念的，该数据库存储使用的是集合，集合中存储的是文档(树状结构数据)

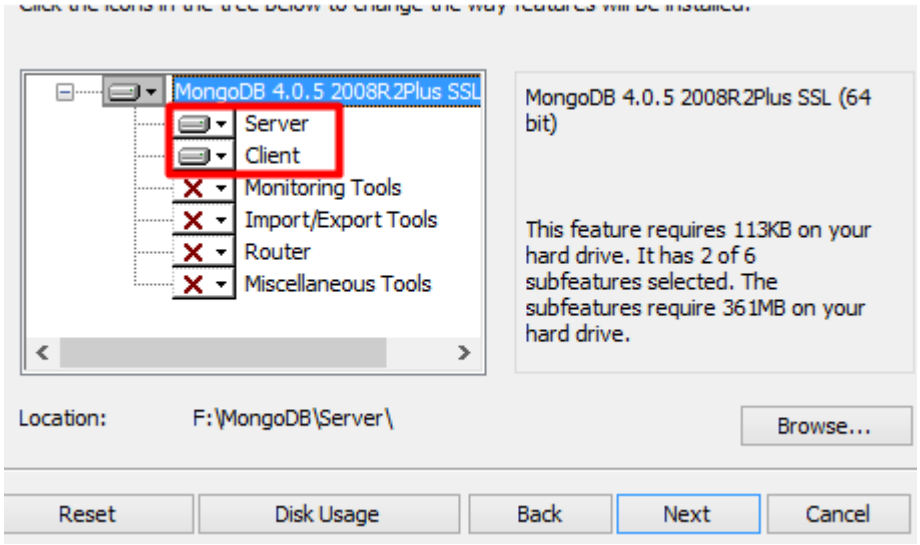


安装和连接

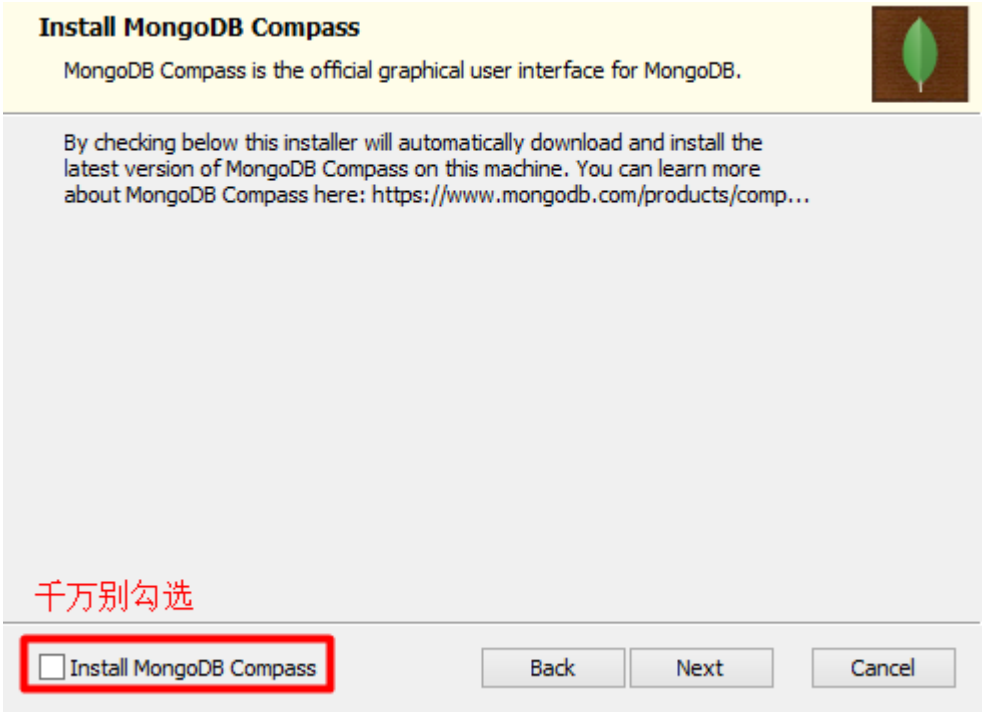
安装客户端和服务器

根据安装包的提示操作，选择自定义安装，其中只安装服务端和客户端即可





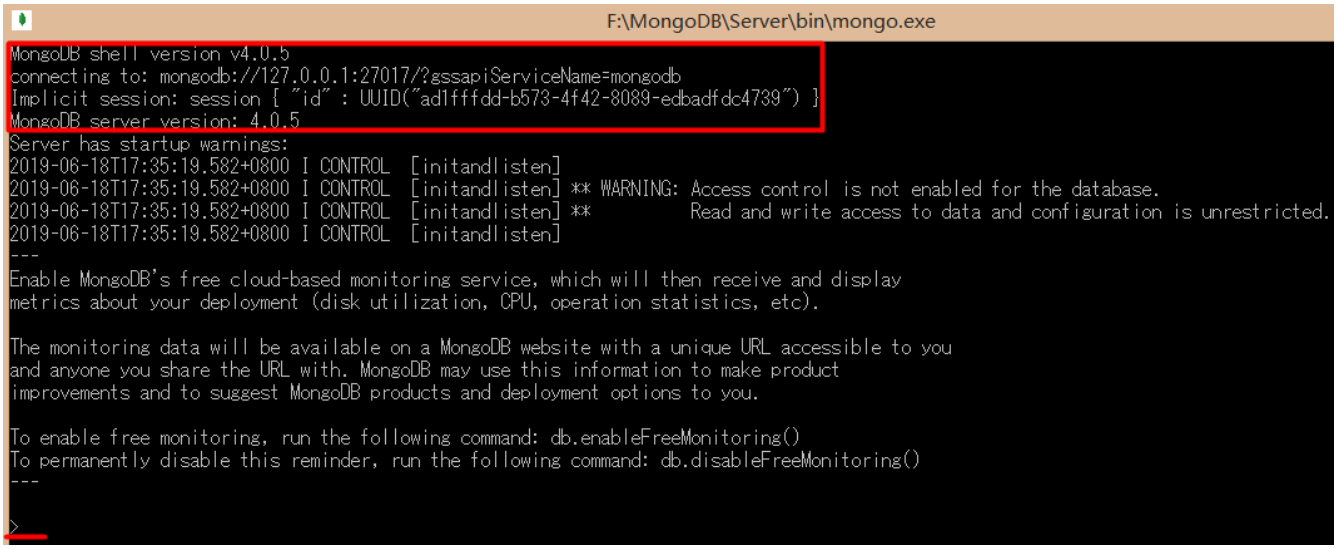
最后千万不要勾选



连接服务器

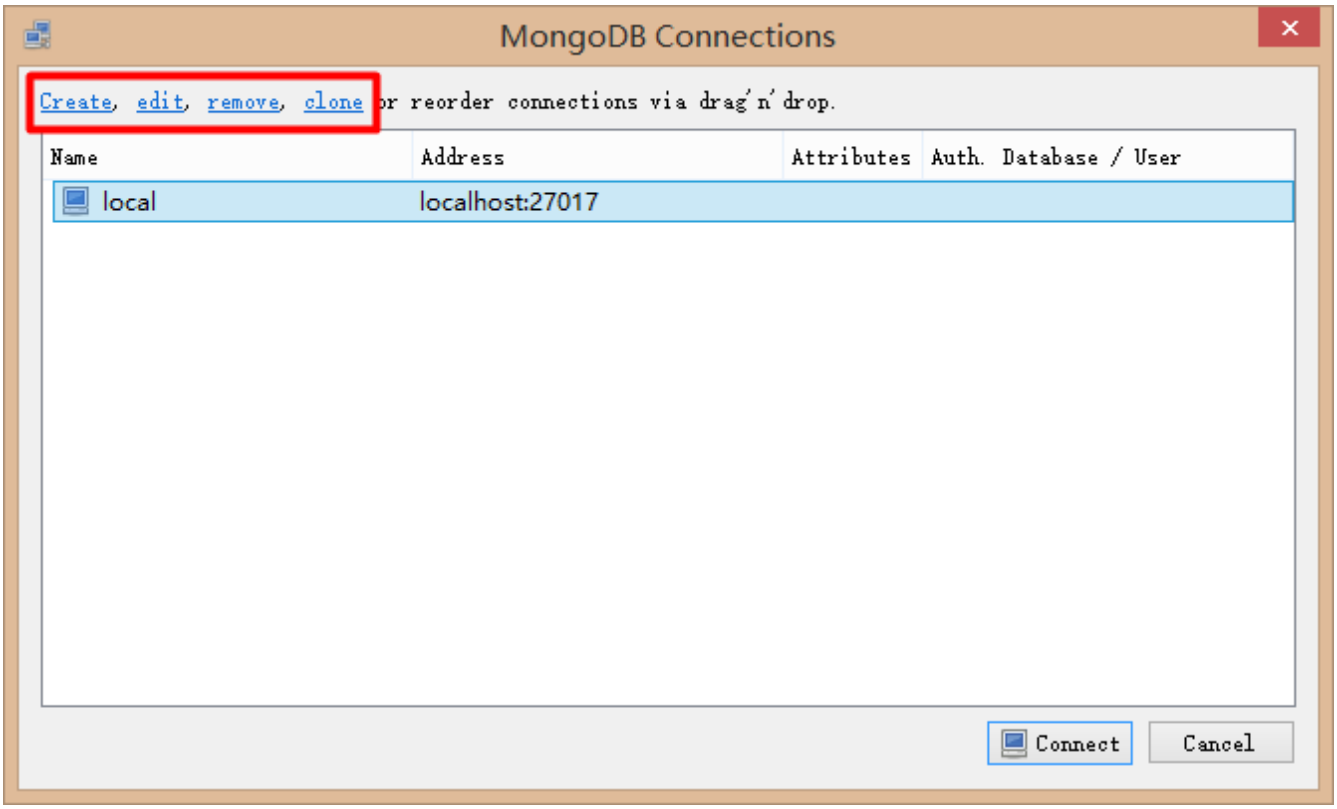
安装好后，默认会启动MongoDB服务器，可以在服务列表中找到**MongoDB Server**查看是否处于运行状态，已经处于运行状态则直接进入**安装目录/Server/bin**,运行mongo.exe，出现以下画面表示安装成功了

补充：MongoDB默认端口27017



robo3t客户端

图形界面工具，简单易用，傻瓜式安装



基本操作

MongoDB的操作有很多，需要掌握更多技能的同学可以查询在线的文档，这里主要是讲这么去学习这个工具

在线文档教程: https://www.runoob.com/mongodb/mongodb-tutorial.html

语法

```
show dbs: 查询所有数据库
use 数据库名: 创建并且选中数据库，数据库已经存在则直接选中
db: 查询当前选择的数据库
db.dropDatabase(): 删除当前选中的数据库
show collections: 查询当前库中的集合
db.createCollection("集合名"): 创建集合
db.集合名.drop(): 删除集合

注意: db.集合名 == db.getCollection("集合名")
```

新增操作

数据类型

在MongoDB中支持以下的数据类型

```
String（字符串）: mongodb中的字符串是UTF-8有效的
Integer（整数）: 存储数值。整数可以是32位或64位，具体取决于您的服务器
Boolean（布尔）: 存储布尔(true/false)值
Double（双精度）: 存储浮点值
Arrays（数组）: 将数组或列表或多个值存储到一个键中
Timestamp（时间戳）: 存储时间戳
Object（对象）: 嵌入式文档
Null （空值）: 存储Null值
Symbol（符号）: 与字符串相同，用于具有特定符号类型的语言
Date（日期）: 以UNIX时间格式存储当前日期或时间
Object ID（对象ID） : 存储文档ID
Binary data（二进制数据）: 存储二进制数据
Code（代码）: 将JavaScript代码存储到文档中
Regular expression（正则表达式）: 存储正则表达式
```

语法

往集合中新增文档，当集合不存在时会自动先创建集合，再往集合中添加文档，但是不要依赖自动创建集合的特性

```
db.集合名.insert( 文档 ) : 往集合中插入一个文档
db.集合名.find(): 查询集合中所有文档
```

当操作成功时，集合会给文档生成一个**_id**字段，该字段就是文档的主键，也能在插入数据时自己指定该字段的值，但是不建议这样做

实例

```
//插入一个员工对象
db.users.insert({id: NumberLong(1), name: "逍遥", age: NumberInt(18)})
db.users.insert({id: 2, name: "bunny", age: 20})

注意：直接写数字默认是Duble类型，其实在使用时是不影响的，仅仅只是类型问题
```

更新操作

修改集合中已经存在的文档

语法

```
db.集合名.update(
  <query>,
  <update>,
  {
        upsert: <boolean>,
        multi: <boolean>,
        writeConcern: <document>
  }
)

简化方法：
更新1个：db.集合名.updateOne( ... )
更新所有：db.集合名.updateMany( ... )
```

参数说明：

- **query** : update的查询条件，类似sql update查询内where后面的。
- **update** : update的对象和一些更新的操作符（如$，$inc...）等，也可以理解为sql update查询内set后面的
- **upsert** : 可选，这个参数的意思是，如果不存在update的记录，是否插入objNew，true为插入，默认是false，不插入。
- **multi** : 可选，mongodb 默认是false，只更新找到的第一条记录，如果这个参数为true，就把按条件查出来多条记录全部更新。
- **writeConcern** :可选，抛出异常的级别

实例

```
//把一个带有name=逍遥的文档，修改其eage值为30
db.employees.update({name: "逍遥"}, {$set: {age: 30}})
db.employees.updateOne({name: "逍遥"}, {$set: {age: 30}})

//修改所有name=逍遥的文档，修改其name=bunny，age=20
db.employees.update({name: "逍遥"}, {$set: {name: "bunny", age: 30}}, {multi:true})
db.employees.updateMany({name: "逍遥"}, {$set: {name: "bunny", age: 30}})
```

删除操作

删除集合中的文档

语法

```
db.集合名.remove(
  <query>,
  {
        justOne: <boolean>,
        writeConcern: <document>
  }
)

简化方法：
删除1个：db.集合名.removeOne( ... )
删除所有：db.集合名.removeMany( ... )
```

参数说明：

- **query**：（可选）删除的文档的条件。
- **justOne**：（可选）如果设为 true 或 1，则只删除一个文档，如果不设置该参数，或使用默认值 false，则删除所有匹配条件的文档。
- **writeConcern**：（可选）抛出异常的级别

实例

```
//删除_id=xxx的文档
db.users.remove({_id: ObjectId("xxx")})
db.users.removeOne({_id: ObjectId("xxx")})

//删除1个带有name=bunny的文档
db.users.removeOne({name: "bunny"}, {justOne: true})

//删除当前数据库中所有文档
db.users.removeMany()

//删除所有带有name=bunny的文档
db.users.removeMany({name: "bunny"})
```

查询操作

语法

```
db.集合名.find(query, projection)
db.集合名.find(...).pretty()
```

- **query** ：可选，使用查询操作符指定查询条件
- **projection** ：可选，使用投影操作符指定返回的键。查询时返回文档中所有键值， 只需省略该参数即可（默认省略）。
- 调用pretty()可以格式化查询的内容

实例

```
//查询所有文档
db.users.find()
//查询所有文档，并且格式化打印
db.users.find().pretty()
```

排序查询

```
db.users.find().sort({字段: 1}) -> 按照字段升序排列
db.users.find().sort({字段: -1}) -> 按照字段降序排列
```

分页查询

```
sikp(num): 跳过num个文档，相当于start
limit(num): 限制显示num个文档，相当于pageSize
如：
db.users.find().sikp(num).limit(num)
```

```
阶段练习：
1：按照年龄降序排列，查询第2页，每页显示3个
```

高级查询

高级查询在数据库中是经常使用的，在MongoDB中也同样是提供了高级查询的功能

等值

```
find({字段: 值})
```

比较查询

```
语法 -> find({字段: {比较操作符: 值， ...}})
```

比较操作符

- (>) 大于 - $gt
- (<) 小于 - $lt
- (>=) 大于等于 - $gte
- (<=) 小于等于 - $lte

- (!=) 不等 - $ne
- 集合运算 - $in 如:{name: {$in: ["xiaoyao"，"bunny"]}}
- 判断存在 - $exists 如:{name: {$exists:true}}

逻辑查询

```
语法 -> find({逻辑操作符: [条件1， 条件2， ...]})
```

逻辑操作符

- (&&) 与 - $and
- (||) 或 - $or
- (!) 非 - $not

模糊查询

MongoDB的模糊查询使用的是正则表达式的语法 如:{name: {$regex: /^.*keyword.*$/}}

实际上MongoDB也是不擅长执行模糊查询的，在实际开发中也是不使用的，**该功能了解即可**

```
阶段练习：
1：查询所有name=逍遥的文档
2：查询所有name=bunny或者age<30的文档
3：查询所有name含有zhang并且30<=age<=32的文档
```

设置用户

以下操作必须在cmd命令行中操作，执行以下命令

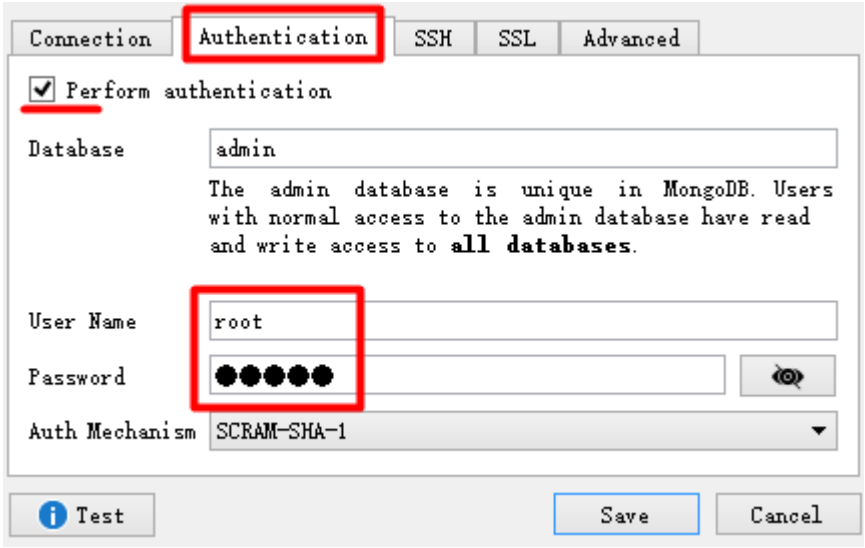
```
//1.选中admin数据库
use admin
//2.往里面添加一个超级管理员账号
db.createUser({user:"root"， pwd: "admin"， roles:["root"]})
//user:账号    pwd:密码    roles:角色->root超级管理员
```

修改MongoDB的配置文件：**安装目录/Server/bin/mongod.cfg**

```
#约在29行位置，配置开启权限认证
security:
  authorization: enabled
```

完成上面配置后重启服务器

在登录时就必须要输入密码，否则不能执行任何的命令



Spring Boot Data MongoDB

准备环境

```
<parent>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-parent</artifactId>
    <version>2.1.3.RELEASE</version>
</parent>

<properties>
    <java.version>1.8</java.version>
</properties>

<dependencies>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-test</artifactId>
        <scope>test</scope>
    </dependency>

    <dependency>
        <groupId>org.projectlombok</groupId>
        <artifactId>lombok</artifactId>
        <scope>provided</scope>
    </dependency>

    <!--spring boot data mongodb-->
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-data-mongodb</artifactId>
    </dependency>
</dependencies>
```

```
<build>
    <plugins>
        <!-- SpringBoot打包插件 -->
        <plugin>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-maven-plugin</artifactId>
        </plugin>
    </plugins>
</build>
```

配置连接参数

```
# application.properties
# 配置数据库连接
#格式: mongodb://账号:密码@ip:端口/数据库?认证数据库
spring.data.mongodb.uri=mongodb://root:admin@localhost/admin?authSource=admin

# 配置MongoTemplate的执行日志
logging.level.org.springframework.data.mongodb.core=debug
```

Java代码

```
//注入MongoTemplate
@Autowired
private MongoTemplate mongoTemplate;
```

domain中的对象的主键使用 ObjectId _id 字段来表示

```
/**
 * 文档的id使用ObjectId类型来封装，并且贴上@Id注解
 */
@AllArgsConstructor
@NoArgsConstructor
@Setter@Getter@ToString
public class User implements Serializable {
    @Id //文档的ID
    private ObjectId _id;

    private Long id;
    private String name;
    private Integer age;
}
```

条件限定

Query对象用于封装查询条件，配合Criteria一起使用，来完成各种条件的描述

```
//一个Criteria对象可以理解为是一个限定条件
Criteria.where(String key).is(Object val); //设置一个等值条件
Criteria.orOperator(Criteria ...); //设置一组或的逻辑条件

//模糊查询（了解）
Criteria.where(String key).regex(String regex); //使用正则表达式匹配查询

注意：Criteria对象可以由其静态方法和构造器获取
```

Criteria封装了所有对条件的描述，常见的有以下方法

> lt / lte / gt / gte / ne / ...

最后通过Query对象的addCriteria把条件封装到Query对象中

```
Query对象.addCriteria(Criteria criteria); //添加查询条件
Query对象.skip(start).limit(pageSize); //分页查询
```

MongoTemplate的API方法

```
//往集合中添加文档，返回的对象中包含数据库生成的_id
T mongoTemplate.insert(T t, String collectionName);

//更新集合中第一个找到的文档
mongoTemplate.updateFirst(Query query, Update update, String collectionName);
//更新集合中所有找到的文档
mongoTemplate.updateMulti(Query query, Update update, String collectionName);

//删除集合中找到的所有的文档
mongoTemplate.remove(Query query, String collectionName);

//根据id查询一个集合中的文档
T mongoTemplate.findById(Object id, Class<T> type, String collectionName);
//查询集合中所有文档
List<T> mongoTemplate.findAll(Class<T> type, String collectionName);
//根据条件查询集合中的文档
List<User> mongoTemplate.find(Query query, Class<T> type, String collectionName);
```

实例代码

```
// 插入一个文档
@Test
public void testInsert() throws Exception {
    User user = new User(null, 10L, "逍遥", 18);
    user = mongoTemplate.insert(user, "users");
    System.err.println(user);
}

// 更新一个文档
@Test
public void testUpdate() throws Exception {
```

```
    Query query = new Query();
    // {_id: ObjectId("5cf1eb15ae4ec63818ff9b35")}
    query.addCriteria(Criteria.where("_id").is("5cf1eb15ae4ec63818ff9b35"));
    // {name: {$set: "张全蛋"}, age: {$set: 28}}
    Update update = Update.update("name", "张全蛋").set("age", 28);
    // 更新1个匹配文档
    UpdateResult result = mongoTemplate.updateFirst(query, update, "users");
    // 更新所有匹配文档
    // UpdateResult result = mongoTemplate.updateMulti(query, update, "users");
    System.err.println(result.getModifiedCount());
}

// 删除一个文档
@Test
public void testDelete() throws Exception {
    Query query = new Query();
    // {_id: ObjectId("5cf1f05aae4ec616a41898a7")}
    query.addCriteria(Criteria.where("_id").is("5cf1f05aae4ec616a41898a7"));
    // 删除所有匹配文档
    DeleteResult result = mongoTemplate.remove(query, "users");
    System.err.println(result.getDeletedCount());
}

// 查询一个文档
@Test
public void testGet() throws Exception {
    // 根据_id查询一个文档
    User user = mongoTemplate.findById("5cf1ea73ae4ec600dc754ada", User.class,"users");
    System.err.println(user);
}

// 查询所有文档
@Test
public void testList() throws Exception {
    // 查询所有文档
    List<User> list = mongoTemplate.findAll(User.class, "users");
    list.forEach(System.err::println);
}

// 分页查询文档，显示第2页，每页显示3个，按照id升序排列
@Test
public void testQuery1() throws Exception {
    // 创建查询对象
    Query query = new Query();
    // 设置分页信息
    query.skip(3).limit(3);
    // 设置排序规则
    query.with(new Sort(Sort.Direction.ASC,"id"));

    List<User> list = mongoTemplate.find(query, User.class, "users");
    list.forEach(System.err::println);
}
```

```
// 查询所有name为bunny的文档
@Test
public void testQuery2() throws Exception {
    // 构建限制条件 {"name": "bunny"}
    Criteria criteria = Criteria.where("name").is("bunny");
    // 创建查询对象
    Query query = new Query();
    // 添加限制条件
    query.addCriteria(criteria);

    List<User> list = mongoTemplate.find(query, User.class, "users");
    list.forEach(System.err::println);
}

// 查询所有name为bunny或者age<30的文档
@Test
public void testQuery3() throws Exception {
    // 构建限制条件 { "$or": [{"name": "bunny"}, {"age": {"$lt": 30}}] }
    Criteria criteria = new Criteria().orOperator(
        Criteria.where("name").is("bunny"),
        Criteria.where("age").lt(30)
    );
    // 创建查询对象
    Query query = new Query();
    // 添加限制条件
    query.addCriteria(criteria);

    List<User> list = mongoTemplate.find(query, User.class, "users");
    list.forEach(System.err::println);
}

// 查询所有name含有wang并且30<=age<=32的文档
@Test
public void testQuery4() throws Exception {
    // 构建限制条件 { "$and" : [{"name": {"$regex": "^.*wang.*$"} }, {"age": {"$gte": 30,
"$lte": 32 } }] }
    Criteria criteria = new Criteria().andOperator(
        Criteria.where("name").regex("^.*wang.*$"),
        Criteria.where("age").gte(30).lte(32)
    );
    // 创建查询对象
    Query query = new Query();
    // 添加限制条件
    query.addCriteria(criteria);

    List<User> list = mongoTemplate.find(query, User.class, "users");
    list.forEach(System.err::println);
}
```