link: null
title: 珠峰架构师成长计划
description: Model是由通过Schema构造而成
除了具有Schema定义的数据库骨架以外，还可以操作数据库
如何通过Schema来创建Model呢，如下:
keywords: null
author: null
date: null
publisher: 珠峰架构师成长计划
stats: paragraph=104 sentences=134, words=1039

1. MongoDB简介

- Mongodb是一个开源的NoSQL数据库，相比MySQL那样的关系型数据库，它更显得 轻巧、灵活，非常适合在数据规模很大、事务性不强的场合下使用。
- 同时它也是一个对象数据库，没有表和行的概念，也没有固定的模式和结构，所有的数据都是以文档的形式存存储(文档，就是一个关联数组式的对象，它的内部由属性组成，一个属性对应的值可能是一个数、字符串、日期、数组、甚至是一个嵌套文档)，数据格式就是JSON。

2. Mongoose是什么？

- Mongoose是MongoDB的一个对象模型工具
- 同时它也是针对MongoDB操作的一个对象模型库,封装了MongoDB对文档的的一些增删改查等常用方法
- 让NodeJS操作Mongodb数据库变得更加灵活简单
- Mongoose因为封装了MongoDB对文档操作的常用方法，可以高效处理mongodb,还提供了类似Schema的功能，如hook、plugin、virtual、populate等机制。
- 官网mongoosejs (http://mongoosejs.com/)

3. 使用 mongoose

```
$ cnpm install mongoose -S
```

```
let mongoose = require("mongoose");
let db = mongoose.createConnection("mongodb://user:pass@ip:port/database",{ useNewUrlParser: true,useUnifiedTopology: true});
```

- user 用户名
- pass 密码
- ip IP地址
- port 端口号
- database 数据库

```
let mongoose = require('mongoose');
let connection = mongoose.createConnection("mongodb://127.0.0.1/zfpx",{ useNewUrlParser: true,useUnifiedTopology: true});
connection.on('error', function (error) {
    console.log('数据库连接失败: ' + error);
});
connection.on('open', function (error) {
    console.log('数据库连接成功');
});
```

- Schema是数据库集合的模型骨架
- 定义了集合中的字段的名称和类型以及默认值等信息
- NodeJS中的基本数据类型都属于 Schema.Type
- 另外Mongoose还定义了自己的类型
- 基本属性类型有:
    - 字符串(String)
    - 日期型(Date)
    - 数值型(Number)
    - 布尔型(Boolean)
    - null
    - 数组([])
    - 内嵌文档

```
var personSchema = new Schema({
     name:String,
     binary:Buffer,
     living:Boolean,
     birthday:Date,
     age:Number,
     _id:Schema.Types.ObjectId,
     _fk:Schema.Types.ObjectId,
     array:[],
     arrOfString:[String],
     arrOfNumber:[Number],
     arrOfDate:[Date],
     arrOfBuffer:[Buffer],
     arrOfBoolean:[Boolean],
     arrOfObjectId:[Schema.Types.ObjectId]
     nested:{ name:String}
   });

   let p = new Person();
   p.name= 'zfpx';
   p.age = 25;
   p.birthday = new Date();
   p.married = false;
   p.mixed= {any:{other:'other'}};
   p._otherId = new mongoose.Types.ObjectId;
   p.hobby.push('smoking');
   p.ofString.push('string');
   p.ofNumber.pop(3);
   p.ofDates.addToSet(new Date);
   p.ofBuffer.pop();
   p.ofMixed = ['anything',3,{name:'zfpx'}];
   p.nested.name = 'zfpx';
```

Model是由通过Schema构造而成 除了具有Schema定义的数据库骨架以外，还可以操作数据库 如何通过Schema来创建Model呢，如下:

```
var mongoose = require('mongoose');
var connection = mongoose.createConnection("mongodb://127.0.0.1/zfpx",{ useNewUrlParser: true,useUnifiedTopology: true});
connection.on('error', function (error) {
    console.log('数据库连接失败: ' + error);
});
connection.on('open', function (error) {
    console.log('数据库连接成功');
});
let PersonSchema = new mongoose.Schema({
      name:String,
      age:Number,
});

var PersonModel = connection.model("Person", PersonSchema);

var PersonModel = connection.model('Person');
```

> 拥有了Model，我们也就拥有了操作数据库的能力 在数据库中的集合名称等于 模型名转小写再转复数,比如 Person>person>people,Child>child>children

- 通过Model创建的实体，它也可以操作数据库
- 使用Model创建Entity，如下示例

```
let personEntity = new PersonModel({
     name : "zhufeng",
     age  : 6
});
console.log(personEntity);
```

> Schema生成Model，Model创造Entity，Model和Entity都可对数据库操作,但Model比Entity可以实现的功能更多

```
let mongoose = require("mongoose");
let conn = mongoose.createConnection("mongodb://127.0.0.1/zfpx",{ useNewUrlParser: true,useUnifiedTopology: true});
let PersonSchema = new mongoose.Schema({
    name: {type: String},
    age: {type: Number, default: 0}
});
let PersonModel = conn.model("Person", PersonSchema);

let PersonEntity = new PersonModel({
    name: "zfpx",
    age: 6
});

PersonEntity.save(function (error, doc) {
    if (error) {
        console.log("error :" + error);
    } else {

        console.log(doc);
    }
});
```

- 存储在mongodb集合中的每个文档都有一个默认的主键_id
- 这个主键名称是固定的，它可以是mongodb支持的任何数据类型，默认是ObjectId 该类型的值由系统自己生成，从某种意义上几乎不会重复
- ObjectId使用12字节的存储空间，是一个由24个16进制数字组成的字符串（每个字节可以存储两个16进制数字）

```
5d9c70b3 f88966 4f24 d9caa5
```

部分 值 含义 4字节 5d9c70b3 时间戳是自 1970 年 1 月 1 日（08:00:00 GMT）至当前时间的总秒数，它也被称为 Unix 时间戳，单位为秒 3字节 f88966 所在主机的唯一标识符,通常是机器主机名的散列值(hash),可以确保不同主机生成不同的

不产生冲突 2字节 4f24 产生ObjectId的进程的进程标识符(PID) 3字节 d9caa5 由一个随机数开始的计数器生成的值

```
let ts = parseInt('5d9c70b3', 16);;
console.log(ts);
let date = new Date(ts*1000);
console.log(date.toLocaleString());

console.log(parseInt('4f24',16));
console.log(parseInt('d9caa5',16))
```

> 前9个字节保证了同一秒钟不同机器不同进程产生的ObjectId是唯一的,最后3个字节是一个自动增加的计数器，确保相同进程同一秒产生的ObjectId也是不一样的,一秒钟最多允许每个进程拥有256的3次方(16777216)个不同的ObjectId 每一个文档都有一个特殊的键_id，这个键在文档所属的集合中是唯一的。

```
Model.find(查询条件,callback);
```

```
Model.find({},function(error,docs){

});

Model.find({ "age": 6 }, function (error, docs) {
  if(error){
    console.log("error :" + error);
  }else{
    console.log(docs);
  }
});
```

```
Model.create(文档数据, callback))
```

```
 PersonModel.create({ name:"zfpx", age:7}, function(error,doc){
    if(error) {
        console.log(error);
    } else {
        console.log(doc);
    }
});
`
```

```
Entity.save(callback))
```

```
var PersonEntity = new PersonModel({name:"zfpx",age: 9});

PersonEntity.save(function(error,doc) {
   if(error) {
      console.log(error);
   } else {
      console.log(doc);
   }
});
```

```
Model.update(查询条件,更新对象,callback);
```

```
var conditions = {name : 'zfpx'};
  var update = {$set : { age : 100 }};
  PersonModel.update(conditions, update, function(error){
      if(error) {
          console.log(error);
      } else {
          console.log('Update success!');
        }
    });
```

请注意如果匹配到多条记录，默认只更新一条，如果要更新匹配到的所有记录的话需要加一个参数 {multi:true}

```
Model.remove(查询条件,callback);
```

```
var conditions = { name: 'zfpx' };
PersonModel.remove(conditions, function(error){
    if(error) {
          console.log(error);
    } else {
        console.log('Delete success!');
    }
});
```

```
        PersonModel.create([
                          { name:"zfpx1", age:1 },
                          { name:"zfpx2", age:2 },
                          { name:"zfpx3", age:3 },
                          { name:"zfpx4", age:4 },
                          { name:"zfpx5", age:5 },
                          { name:"zfpx6", age:6},
                          { name:"zfpx7", age:7 },
                          { name:"zfpx8", age:8 },
                          { name:"zfpx9", age:9},
                          { name:"zfpx10",age:10 }
                         ], function(error,docs) {
            if(error) {
                console.log(error);
            } else {
                console.log('save ok');
            }
        });
```

```
find(Conditions,field,callback)
```

```

Model.find({},{name:1, age:1, _id:0}, function(err,docs){

})
```

我们只需要把显示的属性设置为大于零的数就可以，当然1是最好理解的，_id是默认返回，如果不要显示加上("_id":0)

与find相同，但只返回单个文档，也就说当查询到即一个符合条件的数据时，将停止继续查询，并返回查询结果 语法

```
findOne(Conditions,callback)
```

```
TestModel.findOne({ age: 6}, function (err, doc){

});
```

与findOne相同，但它只接收文档的_id作为参数，返回单个文档 语法

```
findById(_id, callback)
```

```
PersonModel.findById(person._id, function (err, doc){

});
```

查询时我们经常会碰到要根据某些字段进行条件筛选查询，比如说Number类型，怎么办呢，我们就可以使用$gt(>)、$lt(

```
Model.find({"age":{"$gt":6}},function(error,docs){

});

Model.find({"age":{"$lt":6}},function(error,docs){

});

Model.find({"age":{"$gt":6,"$lt":9}},function(error,docs){

});
```

$ne(!=)操作符的含义相当于不等于、不包含，查询时我们可通过它进行条件判定，具体使用方法如下：

```
Model.find({ age:{ $ne:6}},function(error,docs){

});
```

和$ne操作符相反，$in相当于包含、等于，查询时查找包含于指定字段条件的数据

```
Model.find({ age:{ $in: 6}},function(error,docs){

});

Model.find({ age:{$in:[6,7]}},function(error,docs){

});
```

可以查询多个键值的任意给定值，只要满足其中一个就可返回，用于存在多个条件判定的情况下使用，如下示例：

```
Model.find({"$or":[{"name":"zfpx"},{"age":6}]},function(error,docs){

});
```

$exists操作符，可用于判断某些关键字段是否存在来进行条件查询。如下示例：

```
Model.find({name: {$exists: true}},function(error,docs){

});

Model.find({email: {$exists: false}},function(error,docs){

});
```

可以限制结果的数量,跳过部分结果,根据任意键对结果进行各种排序

所有这些选项都要在查询被发送到服务器之前指定

在查询操作中,有时数据量会很大,这时我们就需要对返回结果的数量进行限制 那么我们就可以使用limit函数，通过它来限制结果数量。 语法

```
find(Conditions,fields,options,callback);
```

```
Model.find({},null,{limit:20},function(err,docs){
        console.log(docs);
});
```

如果匹配的结果不到20个，则返回匹配数量的结果，也就是说limit函数指定的是上限而非下限

skip函数的功能是略过指定数量的匹配结果，返回余下的查询结果 如下示例：

```
find(Conditions,fields,options,callback);
```

```
Model.find({},null,{skip:4},function(err,docs){
        console.log(docs);
});
```

如果查询结果数量中少于4个的话，则不会返回任何结果。

sort函数可以将查询结果数据进行排序操作 该函数的参数是一个或多个键/值对 键代表要排序的键名,值代表排序的方向,1是升序,-1是降序 语法

```
find(Conditions,fields,options,callback)
```

```
 Model.find({},null,{sort:{age:-1}},function(err,docs){

});
```

sort函数可根据用户自定义条件有选择性的来进行排序显示数据结果。

```
Model('User').find({})
  .sort({createAt:-1})
  .skip((pageNum-1)*pageSize)
  .limit(pageSize)
  .populate('user')
  .exec(function(err,docs){
     console.log(docs);
  });
```

```
var mongoose = require('mongoose');

mongoose.connect('mongodb://localhost:27017/201606blog');

var CourseSchema = new mongoose.Schema({
    name:String
});
var CourseModel = mongoose.model('Course',CourseSchema);
var PersonSchema = new mongoose.Schema({
    name:{
        type:String,
        required:true
    },

    course:{
        type:mongoose.Schema.Types.ObjectId,
        ref:'Course'
    }
});
var PersonModel = mongoose.model('Person',PersonSchema);
CourseModel.create({name:'node.js'},function(err,course){
    PersonModel.create({name:'zfpx',course:course._id},function(err,doc){
        console.log(doc);
        PersonModel.findById(doc._id).populate('course').exec(function(err,doc){
            console.log(doc);
        })
    })
});
```