# NodeJS后台框架

# Express+Sequelize

## 搭建项目

安装Expressjs + Express generator

npm install express –g 全局安装Express

npm install -g express-generator 全局安装Express Generator Express项目生成工具

npm install --save sequelize 添加Sequelizejs

npm install --save pg pg-hstore // postgreSql

npm install --save mysql2 // mysql 或 mariadb

npm install --save sqlite3

npm install --save tedious // MSSQL

pm2

npm i pm2 –g

可以利用它来简化很多node应用管理的繁琐任务，如性能jiank监控，自动重启，负载均衡等

pm2 start bin\www --name=文件夹名 --watch

调用express相关指令创建项目，具体可以参考package.json的bin文件

## <https://demopark.github.io/sequelize-docs-Zh-CN/>

## sequelize.js+mysql2

### 建立连接

const Sequelize = require(‘sequelize’)

1. 自己配置参数

const sequelize = new Sequelize('database', 'username', 'password', {

host: 'localhost',

dialect: /\* 'mysql' | 'mariadb' | 'postgres' | 'mssql' 之一 \*/

pool: { //规定数据库连接池

max: 5,

min: 0,

acquire: 30000,

idle: 10000

}

define: {//规定数据库的默认参数

// `timestamps` 字段指定是否将创建 `createdAt` 和 `updatedAt` 字段.

// 该值默认为 true, 但是当前设定为 false

timestamps: false

}

});

2、传递连接 URI

const sequelize = new Sequelize('postgres://user:pass@example.com:5432/dbname');

### 测试连接

sequelize

.authenticate()

.then(() => {

console.log('Connection has been established successfully.');

})

.catch(err => {

console.error('Unable to connect to the database:', err);

});

### 关闭连接

sequelize.close()

### 表建模

const User = sequelize.define(‘表名', {

字段: {

type: Sequelize.STRING,//类型

allowNull: false//是否为空

values: ['active', 'pending', 'deleted']//枚举

unique: true // unique属性用来创建一个唯一约束.

primaryKey: true// primaryKey用于定义主键.

autoIncrement: true// autoIncrement可用于创建自增的整数列

// 这可以创建一个外键:

bar\_id: {

type: Sequelize.INTEGER,

references: {

// 这是引用另一个模型

model: Bar,

// 这是引用模型的列名称

key: 'id',

}

comment: '这是一个包含注释的列名'

}

}, 字段: {

type: Sequelize.STRING

// allowNull 默认为 true

// 时间戳

createdAt: Sequelize.DATE,

updatedAt: Sequelize.DATE,

validate: {

is: ["^[a-z]+$",'i'], // 只允许字母

is: /^[a-z]+$/i, // 与上一个示例相同,使用了真正的正则表达式

not: ["[a-z]",'i'], // 不允许字母

isEmail: true, // 检查邮件格式 (foo@bar.com)

isUrl:{ // 检查连接格式 (<http://foo.com>)

msg:这是自定义的提示信息

}

isIP: true, // 检查 IPv4 (129.89.23.1) 或 IPv6 格式

isIPv4: true, // 检查 IPv4 (129.89.23.1) 格式

isIPv6: true, // 检查 IPv6 格式

isAlpha: true, // 只允许字母

isAlphanumeric: true, // 只允许使用字母数字

isNumeric: true, // 只允许数字

isInt: true, // 检查是否为有效整数

isFloat: true, // 检查是否为有效浮点数

isDecimal: true, // 检查是否为任意数字

isLowercase: true, // 检查是否为小写

isUppercase: true, // 检查是否为大写

notNull: true, // 不允许为空

isNull: true, // 只允许为空

notEmpty: true, // 不允许空字符串

equals: 'specific value', // 只允许一个特定值

contains: 'foo', // 检查是否包含特定的子字符串

notIn: [['foo', 'bar']], // 检查是否值不是其中之一

isIn: [['foo', 'bar']], // 检查是否值是其中之一

notContains: 'bar', // 不允许包含特定的子字符串

len: [2,10], // 只允许长度在2到10之间的值

isUUID: 4, // 只允许uuids

isDate: true, // 只允许日期字符串

isAfter: "2011-11-05", // 只允许在特定日期之后的日期字符串

isBefore: "2011-11-05", // 只允许在特定日期之前的日期字符串

max: 23, // 只允许值 <= 23

min: 23, // 只允许值 >= 23

isCreditCard: true, // 检查有效的信用卡号码

// 自定义验证器的示例:

isEven(value) {

if (parseInt(value) % 2 !== 0) {

throw new Error('Only even values are allowed!');

}

}

isGreaterThanOtherField(value) {

if (parseInt(value) <= parseInt(this.otherField)) {

throw new Error('Bar must be greater than otherField.');

}

}

}

// 模型的名称. 该模型将以此名称存储在`sequelize.models`中.

// 在这种情况下,默认为类名,即Bar.

// 这将控制自动生成的foreignKey和关联命名的名称

modelName: 'bar',

// 不添加时间戳属性 (updatedAt, createdAt)

timestamps: false,

// 不删除数据库条目,但将新添加的属性deletedAt设置为当前日期(删除完成时).

// paranoid 只有在启用时间戳时才能工作

paranoid: true,

// 将自动设置所有属性的字段参数为下划线命名方式.

// 不会覆盖已经定义的字段选项

underscored: true,

// 禁用修改表名; 默认情况下,sequelize将自动将所有传递的模型名称(define的第一个参数)转换为复数. 如果你不想这样,请设置以下内容

freezeTableName: true,

// 定义表的名称

tableName: 'my\_very\_custom\_table\_name',

// 启用乐观锁定. 启用时,sequelize将向模型添加版本计数属性,

// 并在保存过时的实例时引发OptimisticLockingError错误.

// 设置为true或具有要用于启用的属性名称的字符串.

version: true,

}, {

});

### 模型与数据库同步

单表同步

// 注意:如果表已经存在,使用`force:true`将删除该表

User.sync({ force: true }).then(() => {

// 现在数据库中的 `users` 表对应于模型定义

return User.create({

firstName: 'John',

lastName: 'Hancock'

});

});

// 创建表:

Project.sync()

Task.sync()

// 强制创建!

Project.sync({force: true}) // 这将先丢弃表,然后重新创建它

// 删除表:

Project.drop()

Task.drop()

// 事件处理:

Project.[sync|drop]().then(() => {

// 好吧...一切都很好！

}).catch(error => {

// oooh,你输入了错误的数据库凭据？

})

全表同步

sequelize.sync()

// 同步所有尚未在数据库中的模型

sequelize.sync()

// 强制同步所有模型

sequelize.sync({force: true})

// 删除所有表

sequelize.drop()

// 广播处理:

sequelize.[sync|drop]().then(() => {

// woot woot

}).catch(error => {

// whooops

})

### 增删改查

#### 查

// 查找所有用户

User.findAll().then(users => {

console.log("All users:", JSON.stringify(users, null, 4));

});

Project.findByPk(123)//查主键

Project.findOne({ where: {title: 'aProject'} })//差一个

User.findOrCreate({where: {username: 'sdepold'}, defaults: {job: 'Technical Lead JavaScript'}})//搜索，找不到就创建

Project;//搜索并统计

.findAndCountAll({

where: {

title: {

[Op.like]: 'foo%'

}

},

include: [

{ model: Profile, required: true}

],

offset: 10,

limit: 2

})

.then(result => {

console.log(result.count);

console.log(result.rows);

})

Project.findAll({

where: {

id: {

[Op.and]: {a: 5}, // 且 (a = 5)

[Op.or]: [{a: 5}, {a: 6}], // (a = 5 或 a = 6)

[Op.gt]: 6, // id > 6

[Op.gte]: 6, // id >= 6

[Op.lt]: 10, // id < 10

[Op.lte]: 10, // id <= 10

[Op.ne]: 20, // id != 20

[Op.between]: [6, 10], // 在 6 和 10 之间

[Op.notBetween]: [11, 15], // 不在 11 和 15 之间

[Op.in]: [1, 2], // 在 [1, 2] 之中

[Op.notIn]: [1, 2], // 不在 [1, 2] 之中

[Op.like]: '%hat', // 包含 '%hat'

[Op.notLike]: '%hat', // 不包含 '%hat'

[Op.iLike]: '%hat', // 包含 '%hat' (不区分大小写) (仅限 PG)

[Op.notILike]: '%hat', // 不包含 '%hat' (仅限 PG)

[Op.overlap]: [1, 2], // && [1, 2] (PG数组重叠运算符)

[Op.contains]: [1, 2], // @> [1, 2] (PG数组包含运算符)

[Op.contained]: [1, 2], // <@ [1, 2] (PG数组包含于运算符)

[Op.any]: [2,3], // 任何数组[2, 3]::INTEGER (仅限 PG)

},

status: {

[Op.not]: false, // status 不为 FALSE

}

}

})

// 跳过前10个元素,并获取2个

Project.findAll({ offset: 10, limit: 2 })

Project.findAll({ where: { ... }, raw: true })//原始查询

Project.count()//计算数据库中元素的出现次数

Project.max('age')

Project.min('age')

Project.sum('age') 特定属性的值求和

Task.findAll({ include: [ User ] })关联数据

Model.findAll({//只找foo,bar as baz

attributes: ['foo', ['bar', ‘baz’]]

});

Model.findAll({

attributes: [[sequelize.fn('COUNT', sequelize.col('hats')), 'no\_hats']]

attributes: { exclude: ['baz'] }//排除baz

});

SELECT COUNT(hats) AS no\_hats ...

#### 增

// 创建新用户

User.create({ firstName: "Jane", lastName: "Doe" }).then(jane => {

console.log("Jane's auto-generated ID:", jane.id);

});

// 删除所有名为“Jane”的人

User.destroy({

where: {

firstName: "Jane"

}

}).then(() => {

console.log("Done");

});

// 将所有没有姓氏的人改为“Doe”

User.update({ lastName: "Doe" }, {

where: {

lastName: null

}

}).then(() => {

console.log("Done");

});

### 数据类型

Sequelize.STRING // VARCHAR(255)

Sequelize.STRING(1234) // VARCHAR(1234)

Sequelize.STRING.BINARY // VARCHAR BINARY

Sequelize.TEXT // TEXT

Sequelize.TEXT('tiny') // TINYTEXT

Sequelize.CITEXT // CITEXT 仅 PostgreSQL 和 SQLite.

Sequelize.INTEGER // INTEGER

Sequelize.BIGINT // BIGINT

Sequelize.BIGINT(11) // BIGINT(11)

Sequelize.FLOAT // FLOAT

Sequelize.FLOAT(11) // FLOAT(11)

Sequelize.FLOAT(11, 10) // FLOAT(11,10)

Sequelize.REAL // REAL 仅 PostgreSQL.

Sequelize.REAL(11) // REAL(11) 仅 PostgreSQL.

Sequelize.REAL(11, 12) // REAL(11,12) 仅 PostgreSQL.

Sequelize.DOUBLE // DOUBLE

Sequelize.DOUBLE(11) // DOUBLE(11)

Sequelize.DOUBLE(11, 10) // DOUBLE(11,10)

Sequelize.DECIMAL // DECIMAL

Sequelize.DECIMAL(10, 2) // DECIMAL(10,2)

Sequelize.DATE // mysql / sqlite 为 DATETIME, postgres 为带时区的 TIMESTAMP

Sequelize.DATE(6) // DATETIME(6) 适用 mysql 5.6.4+. 小数秒支持最多6位精度

Sequelize.DATEONLY // DATE 不带时间.

Sequelize.BOOLEAN // TINYINT(1)

Sequelize.ENUM('value 1', 'value 2') // 一个允许值为'value 1'和'value 2'的ENUM

Sequelize.ARRAY(Sequelize.TEXT) // 定义一个数组. 仅 PostgreSQL.

Sequelize.ARRAY(Sequelize.ENUM) // 定义一个ENUM数组. 仅 PostgreSQL.

Sequelize.JSON // JSON 列. 仅 PostgreSQL, SQLite 和 MySQL.

Sequelize.JSONB // JSONB 列. 仅 PostgreSQL.

Sequelize.BLOB // BLOB (PostgreSQL 为 bytea)

Sequelize.BLOB('tiny') // TINYBLOB (PostgreSQL 为 bytea. 其余参数是 medium 和 long)

Sequelize.UUID // PostgreSQL 和 SQLite 的 UUID 数据类型,MySQL 的 CHAR(36) BINARY(使用defaultValue:Sequelize.UUIDV1 或 Sequelize.UUIDV4 来让 sequelize 自动生成 id).

Sequelize.CIDR // PostgreSQL 的 CIDR 数据类型

Sequelize.INET // PostgreSQL 的 INET 数据类型

Sequelize.MACADDR // PostgreSQL 的 MACADDR 数据类型

Sequelize.RANGE(Sequelize.INTEGER) // 定义 int4range 范围. 仅 PostgreSQL.

Sequelize.RANGE(Sequelize.BIGINT) // 定义 int8range 范围. 仅 PostgreSQL.

Sequelize.RANGE(Sequelize.DATE) // 定义 tstzrange 范围. 仅 PostgreSQL.

Sequelize.RANGE(Sequelize.DATEONLY) // 定义 daterange 范围. 仅 PostgreSQL.

Sequelize.RANGE(Sequelize.DECIMAL) // 定义 numrange 范围. 仅 PostgreSQL.

Sequelize.ARRAY(Sequelize.RANGE(Sequelize.DATE)) // 定义 tstzrange 范围的数组. 仅 PostgreSQL.

Sequelize.GEOMETRY // Spatial 列. 仅 PostgreSQL (带有 PostGIS) 或 MySQL.

Sequelize.GEOMETRY('POINT') // 带有 geometry 类型的 spatial 列. 仅 PostgreSQL (带有 PostGIS) 或 MySQL.

Sequelize.GEOMETRY('POINT', 4326) // 具有 geometry 类型和 SRID 的 spatial 列. 仅 PostgreSQL (带有 PostGIS) 或 MySQL.

,integer,bigint,float 和 double 还支持 unsigned 和 zerofill 属性

Sequelize.INTEGER.UNSIGNED // INTEGER UNSIGNED

Sequelize.INTEGER(11).UNSIGNED // INTEGER(11) UNSIGNED

Sequelize.INTEGER(11).ZEROFILL // INTEGER(11) ZEROFILL

Sequelize.INTEGER(11).ZEROFILL.UNSIGNED // INTEGER(11) UNSIGNED ZEROFILL

Sequelize.INTEGER(11).UNSIGNED.ZEROFILL // INTEGER(11) UNSIGNED ZEROFILL

可以给值规定范围，注意几点

// 空的范围:

Timeline.create({ range: [] }); // range = 'empty'

// 无边界范围:

Timeline.create({ range: [null, null] }); // range = '[,)'

// range = '[,"2016-01-01 00:00:00+00:00")'

Timeline.create({ range: [null, new Date(Date.UTC(2016, 0, 1))] });

// 无穷范围:

// range = '[-infinity,"2016-01-01 00:00:00+00:00")'

Timeline.create({ range: [-Infinity, new Date(Date.UTC(2016, 0, 1))] });

### 模型导入

// 在你的服务器文件中 - 例如 app.js

const Project = sequelize.import(\_\_dirname + "/path/to/models/project")

// 模型已经在 /path/to/models/project.js 中定义好

// 你可能会注意到,DataTypes与上述相同

module.exports = (sequelize, DataTypes) => {

class Project extends sequelize.Model { }

Project.init({

name: DataTypes.STRING,

description: DataTypes.TEXT

}, { sequelize });

return Project;

}

### requlize生命周期hooks函数

(1)

beforeBulkCreate(instances, options)

beforeBulkDestroy(options)

beforeBulkUpdate(options)

(2)

beforeValidate(instance, options)

(-)

validate

(3)

afterValidate(instance, options)

- or -

validationFailed(instance, options, error)

(4)

beforeCreate(instance, options)

beforeDestroy(instance, options)

beforeUpdate(instance, options)

beforeSave(instance, options)

beforeUpsert(values, options)

(-)

create

destroy

update

(5)

afterCreate(instance, options)

afterDestroy(instance, options)

afterUpdate(instance, options)

afterSave(instance, options)

afterUpsert(created, options)

(6)

afterBulkCreate(instances, options)

afterBulkDestroy(options)

afterBulkUpdate(options)

(7) 在获取数据库连接之前和之后立即执行的 hook:

beforeConnect(config)

afterConnect(connection, config)

#### hook的使用

#### 当前作用域

// 方法1 通过 .init() 方法

class User extends Model {}

User.init({

username: DataTypes.STRING,

mood: {

type: DataTypes.ENUM,

values: ['happy', 'sad', 'neutral']

}

}, {

hooks: {

beforeValidate: (user, options) => {

user.mood = 'happy';

},

afterValidate: (user, options) => {

user.username = 'Toni';

}

},

sequelize

});

// 方法2 通过 .addHook() 方法

User.addHook('beforeValidate', (user, options) => {

user.mood = 'happy';

});

User.addHook('afterValidate', 'someCustomName', (user, options) => {

return Promise.reject(new Error("I'm afraid I can't let you do that!"));

});

// 方法3 通过直接方法

User.beforeCreate((user, options) => {

return hashPassword(user.password).then(hashedPw => {

user.password = hashedPw;

});

});

User.afterValidate('myHookAfter', (user, options) => {

user.username = 'Toni';

});

Book.removeHook('afterCreate', 'notifyUsers');//移除hook

#### 全局

const sequelize = new Sequelize(..., {

define: {

hooks: {

beforeCreate: () => {

// 做些什么

}

}

}

});

sequelize.addHook('beforeCreate', () => {

// 做些什么

});

定义模型hook

class User extends Model {}

User.init({}, { sequelize });

class Project extends Model {}

Project.init({}, {

hooks: {

beforeCreate: () => {

// 做些其它什么

}

},

sequelize

});

User.create() // 运行全局 hook

Project.create() // 运行其自身的 hook (因为全局 hook 被覆盖)

先运行本地hook再运行全局的hook

### 更改查询定义

const Op = Sequelize.Op;

const operatorsAliases = {

$eq: Op.eq,

$ne: Op.ne,

$gte: Op.gte,

$gt: Op.gt,

$lte: Op.lte,

$lt: Op.lt,

$not: Op.not,

$in: Op.in,

$notIn: Op.notIn,

$is: Op.is,

$like: Op.like,

$notLike: Op.notLike,

$iLike: Op.iLike,

$notILike: Op.notILike,

$regexp: Op.regexp,

$notRegexp: Op.notRegexp,

$iRegexp: Op.iRegexp,

$notIRegexp: Op.notIRegexp,

$between: Op.between,

$notBetween: Op.notBetween,

$overlap: Op.overlap,

$contains: Op.contains,

$contained: Op.contained,

$adjacent: Op.adjacent,

$strictLeft: Op.strictLeft,

$strictRight: Op.strictRight,

$noExtendRight: Op.noExtendRight,

$noExtendLeft: Op.noExtendLeft,

$and: Op.and,

$or: Op.or,

$any: Op.any,

$all: Op.all,

$values: Op.values,

$col: Op.col

};

const connection = new Sequelize(db, user, pass, { operatorsAliases });

json查询

// ISJSON - 测试一个字符串是否包含有效的 JSON

User.findAll({

where: sequelize.where(sequelize.fn('ISJSON', sequelize.col('userDetails')), 1)

})

// JSON\_VALUE - 从 JSON 字符串提取标量值

User.findAll({

attributes: [[ sequelize.fn('JSON\_VALUE', sequelize.col('userDetails'), '$.address.Line1'), 'address line 1']]

})

// JSON\_VALUE - 从 JSON 字符串中查询标量值

User.findAll({

where: sequelize.where(sequelize.fn('JSON\_VALUE', sequelize.col('userDetails'), '$.address.Line1'), '14, Foo Street')

})

// JSON\_QUERY - 提取一个对象或数组

User.findAll({

attributes: [[ sequelize.fn('JSON\_QUERY', sequelize.col('userDetails'), '$.address'), 'full address']]

})