数据持久化 - MySQL

回顾

- http协议
- websocket
- socket.io

课堂目标

- 掌握node.js中实现持久化的多种方法
- 掌握mysql下载、安装和配置
- 掌握node.js中原生mysql驱动模块的应用
- 掌握node.js中的ORM模块Sequelize的应用
- 掌握Sequelize的应用案例

资源

- MySQL相关：
  - MySQL：下载
  - node驱动：文档
  - Sequelize：文档、api
- mongodb相关：
  - MongoDB：下载
  - node驱动：文档
  - mongoose：文档
- redis相关：
  - redis：下载
  - node_redis：文档

node.js中实现持久化的多种方法

- 文件系统 fs
- 数据库
  - 关系型数据库-mysql
  - 文档型数据库-mongodb
  - 键值对数据库-redis

文件系统数据库

```
// fsdb.js
// 实现一个文件系统读写数据库
const fs = require("fs");

function get(key) {
  fs.readFile("./db.json", (err, data) => {
    const json = JSON.parse(data);
    console.log(json[key]);
  });
}
function set(key, value) {
  fs.readFile("./db.json", (err, data) => {
    // 可能是空文件，则设置为空对象
    const json = data ? JSON.parse(data) : {};
    json[key] = value; // 设置值
    // 重新写入文件
    fs.writeFile("./db.json", JSON.stringify(json), err => {
      if (err) {
        console.log(err);
      }
      console.log("写入成功！");
    });
  });
}

// 命令行接口部分
const readline = require("readline");
const rl = readline.createInterface({
  input: process.stdin,
  output: process.stdout
});

rl.on("line", function(input) {
  const [op, key, value] = input.split(" ");

  if (op === 'get') {
    get(key)
  } else if (op === 'set') {
    set(key, value)
  } else if(op === 'quit'){
    rl.close();
  }else {
    console.log('没有该操作');
  }
});

rl.on("close", function() {
  console.log("程序结束");
  process.exit(0);
});
```

MySQL安装、配置

mac

windows

> 菜鸟教程 http://www.runoob.com/mysql/mysql-tutorial.html

node.js原生驱动

- 安装mysql模块：`npm i mysql --save`
- mysql模块基本使用

```
// mysql.js
const mysql = require("mysql");
// 连接配置
const cfg = {
  host: "localhost",
  user: "root",
  password: "example", // 修改为你的密码
  database: "kaikeba" // 请确保数据库存在
};
// 创建连接对象
const conn = mysql.createConnection(cfg);

// 连接
conn.connect(err => {
  if (err) {
    throw err;
  } else {
    console.log("连接成功! ");
  }
});

// 查询 conn.query()
// 创建表
const CREATE_SQL = `CREATE TABLE IF NOT EXISTS test (
                    id INT NOT NULL AUTO_INCREMENT,
                    message VARCHAR(45) NULL,
                    PRIMARY KEY (id))`;
const INSERT_SQL = `INSERT INTO test(message) VALUES(?)`;
const SELECT_SQL = `SELECT * FROM test`;
conn.query(CREATE_SQL, err => {
  if (err) {
    throw err;
  }
  // 插入数据
  conn.query(INSERT_SQL, "hello,world", (err, result) => {
    if (err) {
      throw err;
    }
    console.log(result);
    conn.query(SELECT_SQL, (err, results) => {
        console.log(results);
        conn.end(); // 若query语句有嵌套，则end需在此执行
    })
  });
});
```

ES2017写法

```
// mysql2.js
(async () => {
    // get the client
    const mysql = require('mysql2/promise');
    // 连接配置
    const cfg = {
        host: "localhost",
        user: "root",
        password: "example", // 修改为你的密码
        database: "kaikeba" // 请确保数据库存在
    };
    // create the connection
    const connection = await mysql.createConnection(cfg);


    // 查询 conn.query()
    // 创建表
    const CREATE_SQL = `CREATE TABLE IF NOT EXISTS test (
                    id INT NOT NULL AUTO_INCREMENT,
                    message VARCHAR(45) NULL,
                    PRIMARY KEY (id))`;
    const INSERT_SQL = `INSERT INTO test(message) VALUES(?)`;
    const SELECT_SQL = `SELECT * FROM test`;

    // query database
    let ret = await connection.execute(CREATE_SQL);
    console.log('create:', ret)
    ret = await connection.execute(INSERT_SQL, ['abc']);
    console.log('insert:', ret)
    const [rows, fields] = await connection.execute(SELECT_SQL);
    console.log('select:', rows)

})()
```

Node.js ORM - Sequelize

- 概述：基于Promise的ORM(Object Relation Mapping)，是一种数据库中间件 支持多种数据库、事务、关联等

  > 中间件是介于应用系统和系统软件之间的一类软件，它使用系统软件所提供的基础服务（功能），衔接网络上应用系统的各个部分或不同的应用，能够达到资源共享、功能共享的目的。目前，它并没有很严格的定义，但是普遍接受IDC的定义：中间件是一种独立的系统软件服务程序，分布式应用软件借助这种软件在不同的技术之间共享资源，中间件位于客户机服务器的操作系统之上，管理计算资源和网络通信。从这个意义上可以用一个等式来表示中间件：中间件=平台+通信，这也就限定了只有用于分布式系统中才能叫中间件，同时也把它与支撑软件和实用软件区分开来。

- 安装：`npm i sequelize mysql2 -S`
- 基本使用：

```
(async () => {
    const Sequelize = require("sequelize");

    // 建立连接
    const sequelize = new Sequelize("kaikeba", "root", "example", {
        host: "localhost",
        dialect: "mysql",
        operatorsAliases: false // 仍可通过传入 operators map 至
operatorsAliases 的方式来使用字符串运算符，但会返回弃用警告
    });

    // 定义模型
    const Fruit = sequelize.define("Fruit", {
        name: { type: Sequelize.STRING(20), allowNull: false },
        price: { type: Sequelize.FLOAT, allowNull: false },
        stock: { type: Sequelize.INTEGER, defaultValue: 0 }
    });

    // 同步数据库，force: true则会删除已存在表
    let ret = await Fruit.sync()
    console.log('sync',ret)
    ret = await Fruit.create({
        name: "香蕉",
        price: 3.5
    })
    console.log('create',ret)
    ret = await Fruit.findAll()

    await Fruit.update(
        { price: 4 },
        { where: { name:'香蕉'} }
    )

    console.log('findAll',JSON.stringify(ret))
    const Op = Sequelize.Op;
    ret = await Fruit.findAll({
        // where: { price: { [Op.lt]:4 }, stock: { [Op.gte]: 100 } }
        where: { price: { [Op.lt]: 4, [Op.gt]: 2 } }
    })
    console.log('findAll', JSON.stringify(ret, '', '\t'))


})()
```

- 强制同步：创建表之前先删除已存在的表

```
Fruit.sync({force: true})
```

- 避免自动生成时间戳字段

```
const Fruit = sequelize.define("Fruit", {}, {
  timestamps: false
});
```

- 指定表名：`freezeTableName: true`或`tableName:'xxx'`

  > 设置前者则以modelName作为表名；设置后者则按其值作为表名。
  >
  > 蛇形命名 underscored: true,
  >
  > 默认驼峰命名

- UUID-主键

```
id: {
            type: Sequelize.DataTypes.UUID,
            defaultValue: Sequelize.DataTypes.UUIDV1,
            primaryKey: true
        },
```

- Getters & Setters：可用于定义伪属性或映射到数据库字段的保护属性

```
// 定义为属性的一部分
name: {
      type: Sequelize.STRING,
      allowNull: false,
      get() {
        const fname = this.getDataValue("name");
        const price = this.getDataValue("price");
        const stock = this.getDataValue("stock");
        return `${fname}(价格：￥${price} 库存：${stock}kg)`;
      }
}

// 定义为模型选项
// options中
{
    getterMethods:{
      amount(){
        return this.getDataValue("stock") + "kg";
      }
    },
    setterMethods:{
      amount(val){
        const idx = val.indexOf('kg');
        const v = val.slice(0, idx);
        this.setDataValue('stock', v);
      }
    }
}

// 通过模型实例触发setterMethods
Fruit.findAll().then(fruits => {
    console.log(JSON.stringify(fruits));
    // 修改amount，触发setterMethods
    fruits[0].amount = '150kg';
    fruits[0].save();
});
```

- 校验：可以通过校验功能验证模型字段格式、内容，校验会在`create`、`update`和`save`时自动运行

```
price: {
    validate: {
        isFloat: { msg: "价格字段请输入数字" },
        min: { args: [0], msg: "价格字段必须大于0" }
    }
},
stock: {
    validate: {
        isNumeric: { msg: "库存字段请输入数字" }
    }
}
```

- 模型扩展：可添加模型实例方法或类方法扩展模型

```
// 添加类级别方法
Fruit.classify = function(name) {
  const tropicFruits = ['香蕉', '芒果', '椰子']; // 热带水果
  return tropicFruits.includes(name) ? '热带水果':'其他水果';
};

// 添加实例级别方法
Fruit.prototype.totalPrice = function(count) {
  return (this.price * count).toFixed(2);
};

// 使用类方法
['香蕉','草莓'].forEach(f => console.log(f+'是'+Fruit.classify(f)));

// 使用实例方法
Fruit.findAll().then(fruits => {
    const [f1] = fruits;
    console.log(`买5kg${f1.name}需要￥${f1.totalPrice(5)}`);
});
```

- 数据查询

```
// 通过id查询(不支持了)
Fruit.findById(1).then(fruit => {
    // fruit是一个Fruit实例，若没有则为null
    console.log(fruit.get());
});


// 通过属性查询
Fruit.findOne({ where: { name: "香蕉" } }).then(fruit => {
    // fruit是首个匹配项，若没有则为null
    console.log(fruit.get());
});

// 指定查询字段
Fruit.findOne({ attributes: ['name'] }).then(fruit => {
    // fruit是首个匹配项，若没有则为null
    console.log(fruit.get());
```

```
});

// 获取数据和总条数
Fruit.findAndCountAll().then(result => {
    console.log(result.count);
    console.log(result.rows.length);
});

// 查询操作符
const Op = Sequelize.Op;
Fruit.findAll({
    // where: { price: { [Op.lt]:4 }, stock: { [Op.gte]: 100 } }
    where: { price: { [Op.lt]:4,[Op.gt]:2 }}
}).then(fruits => {
    console.log(fruits.length);
});

// 或语句
Fruit.findAll({
    // where: { [Op.or]:[{price: { [Op.lt]:4 }}, {stock: { [Op.gte]: 100 }}]
}
    where: { price: { [Op.or]:[{[Op.gt]:3 }, {[Op.lt]:2 }]}}
}).then(fruits => {
    console.log(fruits[0].get());
});

// 分页
Fruit.findAll({
    offset: 0,
    limit: 2,
})

// 排序
Fruit.findAll({
    order: [['price', 'DESC']],
})

// 聚合
  Fruit.max("price").then(max => {
    console.log("max", max);
  });
  Fruit.sum("price").then(sum => {
    console.log("sum", sum);
  });
```

- 更新

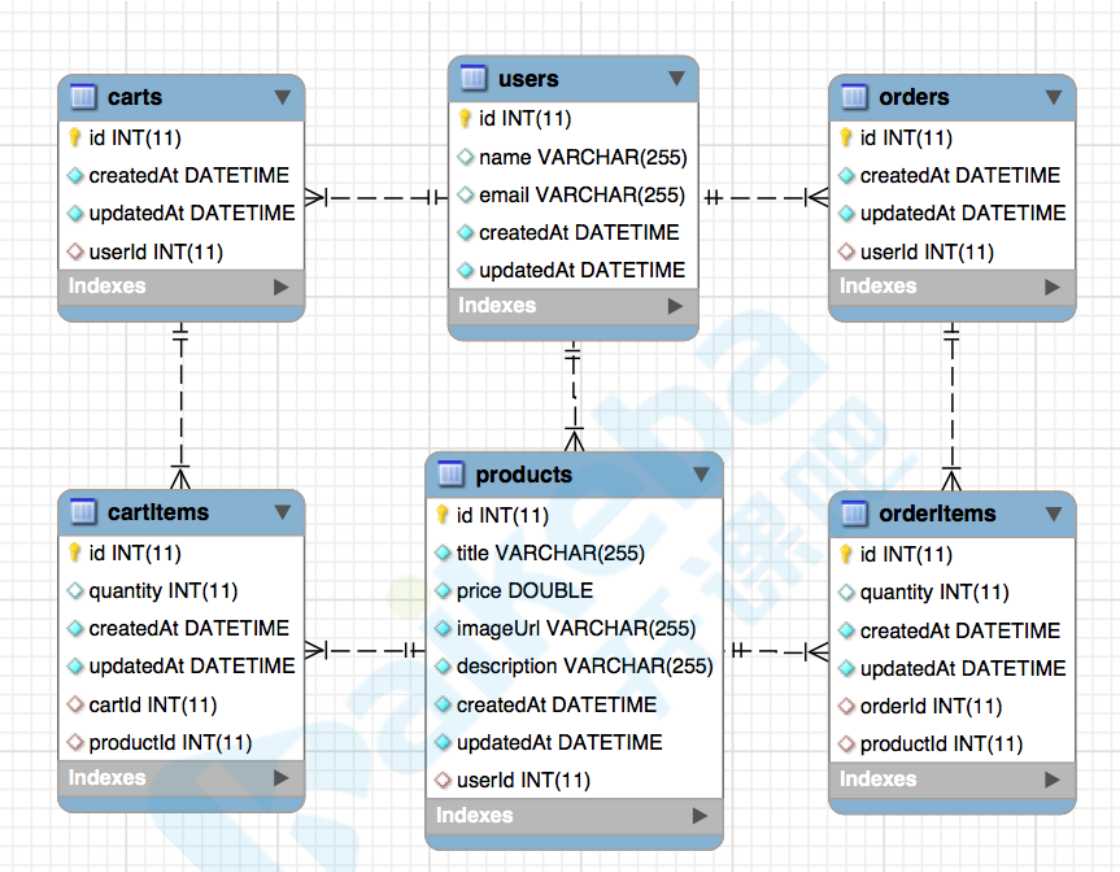
```
Fruit.findById(1).then(fruit => {
    // 方式1
    fruit.price = 4;
    fruit.save().then(()=>console.log('update!!!!'));
});
// 方式2
Fruit.update({price:4}, {where:{id:1}}).then(r => {
    console.log(r);
    console.log('update!!!!')
})
```

- 删除

```
// 方式1
Fruit.findOne({ where: { id: 1 } }).then(r => r.destroy());

// 方式2
Fruit.destroy({ where: { id: 1 } }).then(r => console.log(r));
```

实体关系图和与域模型 ERD



- 初始化数据库

```
// 初始化数据库
const sequelize = require('./util/database');
const Product = require('./models/product');
const User = require('./models/user');
const Cart = require('./models/cart');
const CartItem = require('./models/cart-item');
const Order = require('./models/order');
const OrderItem = require('./models/order-item');


Product.belongsTo(User, {
    constraints: true,
    onDelete: 'CASCADE'
});
User.hasMany(Product);
User.hasOne(Cart);
Cart.belongsTo(User);
Cart.belongsToMany(Product, {
    through: CartItem
```

```
});
Product.belongsToMany(Cart, {
    through: CartItem
});
Order.belongsTo(User);
User.hasMany(Order);
Order.belongsToMany(Product, {
    through: OrderItem
});
Product.belongsToMany(Order, {
    through: OrderItem
});
```

- 同步数据

```
// 同步数据
sequelize.sync().then(
    async result => {
        let user = await User.findByPk(1)
        if (!user) {
            user = await User.create({
                name: 'Sourav',
                email: 'sourav.dey9@gmail.com'
            })
            await user.createCart();
        }
        app.listen(3000, () => console.log("Listening to port 3000"));
    })
```

- 中间件鉴权

```
app.use(async (ctx, next) => {
    const user = await User.findByPk(1)
    ctx.user = user;
    await next();
});
```

- 功能实现

```
const router = require('koa-router')()
/**
 * 查询产品
 */
router.get('/admin/products', async (ctx, next) => {
    // const products = await ctx.user.getProducts()
    const products = await Product.findAll()
    ctx.body = { prods: products }
})

/**
 * 创建产品
 */
router.post('/admin/product', async ctx => {
```

```
    const body = ctx.request.body
    const res = await ctx.user.createProduct(body)
    ctx.body = { success: true }
})

/**
 * 删除产品
 */
router.delete('/admin/product/:id', async (ctx, next) => {
    const id = ctx.params.id
    const res = await Product.destroy({
        where: {
            id
        }
    })
    ctx.body = { success: true }
})

/**
 * 查询购物车
 */
router.get('/cart', async ctx => {
    const cart = await ctx.user.getCart()
    const products = await cart.getProducts()
    ctx.body = { products }
})

/**
 * 添加购物车
 */
router.post('/cart', async ctx => {
    const { body } = ctx.request
    const prodId = body.id
    let newQty = 1
    const cart = await ctx.user.getCart()
    const products = await cart.getProducts({
        where: {
            id: prodId
        }
    })

    let product
    if (products.length > 0) {
        product = products[0]
    }
    if (product) {
        const oldQty = product.cartItem.quantity
        newQty = oldQty + 1
    } else {
        product = await Product.findByPk(prodId)
    }
    await cart.addProduct(product, {
        through: {
            quantity: newQty
        }
    })
    ctx.body = { success: true }
})
```

```
/**
 * 添加订单
 */
router.post('/orders', async ctx => {
    const cart = await ctx.user.getCart()
    const products = await cart.getProducts()
    const order = await ctx.user.createOrder()
    const result = await order.addProduct(
        products.map(p => {
            p.orderItem = {
                quantity: p.cartItem.quantity
            }
            return p
        })
    )
    await cart.setProducts(null)
    ctx.body = { success: true }
})

/**
 * 删除购物车
 */
router.delete('/cartItem/:id', async ctx => {
    const id = ctx.params.id
    const cart = await ctx.user.getCart()
    const products = await cart.getProducts({
        where: { id }
    })
    const product = products[0]
    await product.cartItem.destroy()
    ctx.body = { success: true }
})

/**
 * 查询订单
 */
router.get('/orders', async ctx => {
    const orders = await ctx.user.getOrders({ include: ['products'], order:
[['id', 'DESC']] })
    ctx.body = { orders }
})


app.use(router.routes())
```

> Restful服务
>
> 实践指南 http://www.ruanyifeng.com/blog/2014/05/restful_api.html
>
> 原理 http://www.ruanyifeng.com/blog/2011/09/restful.html

> TODO List范例
>
> https://github.com/BayliSade/TodoList

关于新版本的警告问题

https://segmentfault.com/a/1190000011583806

购物车相关接口实现