Express

常见基本的服务器

- 导入express模块
- 创建express 实例
- 调用 listen(端口号,()=>{}) 启动服务器

```
const express = require('express')
const app = express()
app.listen(80, () => {
    console.log('http://127.0.0.1');
})
```

简单路由

- app.method('url地址',(req,res)=.{})
- req 请求相关的属性方法
- res 响应相关额属性方法
- res.send() 处理好的内容 发送客户端
- req.query对象 获取url携带的参数 查询
    - 客户端： ?name=zs&age=20
    - 服务器：通过req.query.name/.age
- req.params对象 查询动态参数
    - 请求地址：/user/:id :参数名

```
const express = require('express')
const app = express()
app.get('/', (req, res) => {
    res.send({ name: 'zs', age: 18, gender: '男' })
    /*
    {
    "name": "zs",
    "age": 18,
    "gender": "男"
}
    */
})
app.post('/html', (req, res) => {
    res.send('请求成功')
})
//查询 请求携带的参数
app.get('/', (req, res) => {
    res.send(req.query.name)
```

```
})
//动态参数  :参数名
app.get('/:id', (req, res) => {
    res.send({ id: req.params.id, name: 'zs', age: 18, gender: '男' })
})
app.listen(80, () => {
    console.log('http://127.0.0.1');
})

```

静态资源服务器

- app.use(express.static('文件名路径'))
- 存放一些图片 css js 等静态资源
- 存放静态文件的目录名下不会在URL中
- 托管多个静态资源目录 多次调用此函数
    - 如果都有相同文件 则会按顺序查找所需文件

```
app.use(express.static('./assets'))
http://127.0.0.1/logo.png
```

模块化

- 不建议将路由直接挂载到app上 写成单独的模块
    - 创建路由模块对应.js文件
    - 调用 express.Router() 创建路由对象
    - 向路由对象上挂载具体的路由
    - 使用module.exports 向外共享路由对象
    - 使用app.use() 函数注册路由模块
- router.js
- ```
  const express = require('express')
  const router = express.Router()
  router.get('/', (req, res) => {
      res.send('get')
  })
  router.post('/', (req, res) => {
      res.send('get')
  })
  module.exports = router
  ```
- app.js
  ```

- ```
  const express = require('express')
  const app = express()
  //导入 router 模块
  const router = require('./Router/router')
  //使用 router 模块  '/api' 统一请求前缀
  app.use('/api' ,router)
  app.listen(80, () => {
      console.log('http://127.0.0.1');
  })
  ```

中间件

- 原始数据 - 处理环节(中间过程为中间件) - 需要的数据
- 对请求进行预处理
- 客户端 - 请求 - 中间件1 - 中间件2 - 路由处理 - 响应 - 客户端
- 中间件函数多了一个next参数 路由函数 只有req res
- next函数 作用：实现多个中间连续调用 转交给下一个中间件或路由
  - 在当前中间件的业务处理完毕后 必须调用next() 转交给下一个中间件或路由
- 多个中间件共一份req，res 简化代码的书写

```
  //定义中间件    全局生效
      const express = require('express')
const app = express()
const mw = (req, res, next) => {
   console.log('中间函数');
    next()
}
//全局生效
app.use(mw)
//简化形式
app.use((req, res, next) => {
    console.log('中间件函数')
    next()
})
app.listen(80, () => {
    console.log('http://127.0.0.1');
})
```

使用中间件的 作用 多个中间件公用一个req和res 在上游中间件中书写代码
```

```
app.use((req, res, next) => {
   const time = Date.now()
    //为req添加自定义属性和方法
   req.startTime = time
    next()
})
app.get('/', (req, res) => {
    //使用req 自定义属性
    res.send('' + req.startTime)
})
app.post('/', (req, res) => {
   res.send('' + req.startTime)
})
```

定义多个中间件 多次调用app.use() 请求达到服务器会按定义的先后顺序执行

局部生效

- 不适用app.use()定义的中间件
  - 定义中间件 在 路径后，中间件名字，回调函数
- 多个中间件使用 逗号分隔 或数组

```
const mw = function (rea, res, next) {
    console.log('局部中间件');
    next()
}
app.get('/', mw,mw1, (req, res) => {
    res.send('' + req.startTime)
})
```

注意:

- 一定要在路由之前注册中间件
- 请求 可以连续使用多个中间件进行处理
- 不要忘记调用 next()函数
- 调用next()后不要再书写代码
- 多个中间共用一个req和res对象

中间件分类

- 应用级别
  - app.use() 绑定再app上的
- 路由级别
  - router.use() 绑定在express.Router上的
- 错误级别
  - 捕获错误 防止项目异常崩溃 在路由之后
  - (err,req,res,next)

- express内置
  - statoc 托管静态资源
  - json解析json格式的请求数据 req.body 获取请求体里的内容
  - urlencoded 解析url-encoded格式数据 req.body 获取请求体里的内容
  - ```
    app.use(express.json())
    req.body
    app.use(express.urlencoded({extend:false}))
    ```
- 第三方
  - ① 运行 npm install body-parser 安装中间件
  - ② 使用 require 导入中间件
  - ③ 调用 app.use() 注册并使用中间件
  - ```
    require('body-parser')
    app.use(parser.urlencoded({extend:false}))
    ```

自定义

- data 事件 获取数据 end事件 数据接受完毕

接口

```
// 创建基本的服务器
// 导入express
const express = require('express')
// 创建 app 实例
const app = express()
// 导入router
const router = require('./router')
app.use('/api', router)

//启动服务
app.listen(80, () => {
    console.log('http://127.0.0.1');
})

//路由模块
const { query } = require('express')
const express = require('express')
// 创建router实例
const apirouter = express.Router()
// 接口处理
apirouter.method('/get', (req, res) => {
    const data = req.query
    res.send(
        {
```

```
            status: 0,
            msg: '请求成功',
            data: data
        })
})
//post
// 向外暴露apirouter
module.exports = apirouter
```

接口跨域

① 运行 npm install cors 安装中间件

- ② 使用 const cors = require('cors') 导入中间件

③ 在路由之前调用 app.use(cors()) 配置中间件

cors响应头

- Access-Control-Allow-Origin：只允许访问该资源的外域URL　* 表示任何网站访问
- Access-Control-Allow-Headers： 只允许9个请求头
- Access-Control-Allow-Headers：仅支持 get post head 等请求方式 其他需要此方法 声名 *表示所有

数据库

- npm i mysql

配置数据库模块

- 解决因省份原因无法进入数据
  - mysql -u root -p
  - use 数据库名
  - alter user 'root'@'localhost' identified with mysql_native_password by '123456';
  - flush privileges;

```
//导入 mysql 模块
const mysql = require('mysql')
//配置mysql
const db = mysql.createPool({
    host: '127.0.0.1',
    user: 'root',
    password: '123456',
    database: 'ithema'  //数据库名
})
//测试
db.query(`select 1`, (err, results) => {
    if (err) { return console.log(err.message); }
```

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

- 查询

```
const sqlStr = 'select * from emp'
db.query(sqlStr, (err, results) => {
    if (err) { return console.log(err.message); }
    console.log(results);
})
```

- 插入

```
const dept = { id: 23, name: '销售部' }
const sqlStr = 'insert into  dept (id,name) values(?,?)'
db.query(sqlStr, [dept.id, dept.name], (err, results) => {
    if (err) { return console.log(err.message); }
    console.log(results);
})
//便捷方式
const dept = { id: 3, name: '销部', age: 16, status: 1, gender: '男', workno: '2100' }
const sqlStr = 'insert into  users set ?'
db.query(sqlStr, dept, (err, results) => {
    if (err) { return console.log(err.message); }
    if (results.affectedRows === 1) {
        console.log('数据插入成功');
    }
})
```

- 更新

```
const user = { id: 3, name: '张三', }
const sqlStr = 'update users set name=? where id=?'
db.query(sqlStr, [user.name, user.id], (err, results) => {
    if (err) { return console.log(err.message); }
    if (results.affectedRows === 1) {
        console.log('更新数据成功');
    }
})
//便捷 将更新的字段用 ? 代替       下面直接写入对象名  where 后写法不变
const sqlStr = 'update users set  ? where id=?'
 [user, user.id]
```

- 删除

- ```
  const sqlStr = 'delete from users where id=?'
  db.query(sqlStr, 3, (err, results) => {
      if (err) { return console.log(err.message); }
      if (results.affectedRows === 1) {
          console.log('删除数据成功');
      }
  })
  ```

JWT认证

- npm i jsonwebtoken express-jwt
- 在app.js中 const JWT = require('jsonwebtoken');
- const { expressjwt } = require('express-jwt'); 小写
- 定义密钥：const secretKey = 'itheima no1'
- 生成jwt字符串：当访问请求logo时 通过sign(对象信息，加密密钥，配置对象) 将用户信息加密 并返回给客户端
- 
  ```
  // 登录接口
  app.post('/api/login', function(req, res) {
    // ... 省略登录失败情况下的代码
    // 用户登录成功之后，生成 JWT 字符串，通过 token 属性响应给客户端
    res.send({
      status: 200,
      message: '登录成功！',
      // 调用 jwt.sign() 生成 JWT 字符串，三个参数分别是：用户信息对象、加密密钥、配置对象
      token: jwt.sign({ username: userinfo.username }, secretKey, { expiresIn: '30s' })
    })
  })
  ```
- 将JWT 还原为JSON对象
  - 客户端没次访问权限接口的时候，都需要主动通过请求头中的xxx字段，将token字段发送到服务器认证
  - ```
    app.use(expressjwt({ secret: secretKey, algorithms: ['HS256'] }).unless({ path: [/^\/api\//] }))
    ```
- 当express-jwt 中间件配置成功后 我们可以通过在有权限的接口中，使用req.user对象 获取用户信息
- 不要将密码加入到token字符串中

完整

- 导入express模块 创建express实例 启动服务器
- 开启跨域
- 解析post 表单数据 中间件
- jwt 6步

```
const express = require('express') //导入
const app = express()  //创建实例

const jwt1 = require('jsonwebtoken') //导入 生成jwt字符串
const { expressjwt: jwt } = require('express-jwt')  //将字符串转换为 json

// 允许跨域资源共享
const cors = require('cors')
app.use(cors())

const bodyParser = require('body-parser') //解析中间件
app.use(bodyParser.urlencoded({ extended: false })) // 全局生效

const secretKey = 'lihuiliang' //定义密钥

app.use(jwt({ secret: secretKey, algorithms: ['HS256'] }).unless({ path: [/^\/api\//] })) //全局生效  除请求地址为/api外所有需携带 字段验证

app.get('/api/get', (req, res) => {

    const data = req.body
    if (data.id === 'admin' || data.password === '000000') {
        res.send({
            status: 1,
            msg: '登陆成功',
            token: jwt1.sign({ username: data.id }, secretKey, { expiresIn: '3000s' })
        })
    }
    else {
        res.send({
            stasus: 0,
            msg: '登录失败'
        })
    }
})
app.get('/admin', (req, res) => {
    console.log(req)
    res.send({
        status: 200,
        message: '获取用户信息成功！',
        data: req.auth, // 要发送给客户端的用户信息
    })
})
app.use((err, req, res, next) => {

    console.log(err);
```

```
    // 这次错误是由 token 解析失败导致的
    if (err.id === 'UnauthorizedError') {
        return res.send({
            status: 401,
            message: '无效的token',
        })
    }
    res.send({
        status: 500,
        message: '未知的错误',
    })
})

app.listen(80, () => {
    console.log('htpp://127.0.0.1');
})
```

初始化

- 创建项目 npm init -y 装包 创建基本的服务器
- 配置 cors跨域 app.use(cors()) 注意括号
- npm install body-parser
    - const bodyParser = require('body-parser') //解析中间件 app.use(bodyParser.urlencoded({ extended: false })) // 全局生效
    - //配置解析表单数据的中间件 application/x-www-form-urlencoded 格式的数据

      app.use(express.urlencoded({ urlencoded: false }))//内置中间件

      //配置解析application/json格式的数据的中间件

      app.use(express.json())
- 初始化路由 router文件夹 存放请求和处理函数直接的映射关系 和 router_handler文件夹 每个路由对应的处理函数
- req.query url传的数据 用与get 方法 直接获取地址栏传递的参数 动态 参数用 params
- req.body 表单数据 用于 post 方法 得先确认有没有导入'body-parser'

路由模块抽离

- 在handler 文件下 新 建use.js exports.函数名=(req,res)=>{} 向外暴露函数
- router 引入 引入名.函数 即可

代码优化

- 响应数据中间件

- ```
  // 失败只需传递 失败信息
  app.use((req, res, next) => {
      res.cc = (err, status = 1) => {
          res.send({
              status,
              msg: err instanc eof Error ? err.message : err
          })
      }
      next()
  })
  res.cc() ,res.cc('注册失败')
  ```

优化表单验证

- 优化表单验证 永远不要相信前端提交来的任何内容
- npm i @hapi/joi@17.1.0 定义验证规则
- npm i @escook/express-joi 实现自动对表单数据进行验证
- 新建 /schema/user.js 用户信息验证规则模块 初始化代码如下：
  - joi方法： string字符串 alphanum包含字数数字 min最小 max最大 required必选 pattern正则 ref 匹配

- ```
  //在schema / user.js 中
  const joi = require('joi')
  const username = joi.string().alphanum().min(3).max(10).required()
  const password = joi.string().alphanum().pattern(/^(?![a-zA-Z]+$)(?![A-Z0-9]+$)(?![A-Z\W_!@#$%^&*`~()-+=]+$)(?![a-z0-9]+$)(?![a-z\W_!@#$%^&*`~()-+=]+$)(?![0-9\W_!@#$%^&*`~()-+=]+$)[a-zA-Z0-9\W_!@#$%^&*`~()-+=]/)
  exports.userSchema = {
      //三个对象  body  query  params
      body: {
          username,
          password
      }
  }
  //在router 里
  //验证中间件
  const expressjoi = require('@escook/express-joi')
  //导入验证规则
  const { userschema } = require('../schema/user')
  //使用
  router.post('/reguser', expressjoi(userschema), handle.requser)
  //在 api 中
  const joi = require('joi')
  app.use((err,req,res,next)=>{
      if (err instanceof joi.ValidationError) {
          return res.cc(err)
      }
      res.cc(err)

  })
  ```

注册

- 检测数据是否合法(最后一到合法性关口)
- 检测用户名是否被占用
- 对密码进行加密处理 npm i bcryptjs@2.4.3
    - const bcrypt = require('bcryptjs')
    - 加密 bcrypt.hashSync(明文, 10)
    - 解密 bcrypt.compareSync(输入密码，数据库密码)
- 插入新用户
-

```
exports.requser = (req, res) => {
    // 表单数据合法型验证
    const userinfo = req.body
    //验证成功 后 进行数据库查询
    const sqlstr = 'select * from ev_users where username=?'
    db.query(sqlstr, userinfo.username, (err, results) => {
        // 检查数据库 是否报错
        if (err) {
            return res.cc()
        }
        //如果查询结构大于 则 用户名被占用
        if (results.length > 0) {
            return res.cc('用户被占用')
        }
        //用户名可用,继续后续流程
        // 加密 （明文密码,10）
        userinfo.password = bcrypt.hashSync(userinfo.password, 10)
        //插入
        const data = { username: userinfo.username, password: userinfo.password }
        const addSql = 'insert into ev_users set ?'
        db.query(addSql, data, (err, results) => {
            if (err) return res.cc()
            if (results.affectedRows === 1) return res.send({ status: 0, msg: '注册成功'
})
            res.cc('注册失败,稍后重试')
        })
    })
}
```

登录

- 检查表单数据是否合法
- 根据用户名查询数据
- 密码是否正确

- 生成jwt的token字符串
  - 拿到用户信息 安装导入jsonwebtoken包 创建config.js文件 暴露除加密和还原token的jwt字符串
  - 不得包含 头像和密码的值

```
//转jwt字符串
const jwt = require('jsonwebtoken')
//密钥
const jwtSecretKey = 'lihuil'
//登录接口
exports.login = (req, res) => {
    // 获取表单数据
    const userinfo = req.body
    // 查询语句
    const sqlstr = 'select username,password from ev_users where username= ?'
    // 执行sql
    db.query(sqlstr, userinfo.username, (err, results) => {
        //检查mysql 是否报错
        if (err) return res.cc(err)
        //返回数据的长度 不等与 1 则登陆失败
        if (results.length !== 1) return res.cc('登录失败')
        //验证密码 bcrypt.compareSync 解密
        if (bcrypt.compareSync(userinfo.password, results[0].password)) {
            //获取用户信息 进行 token
            const user = { ...results[0], password: ' ', user_pic: ' ' }
            //生成token字符串
            const tokenStr = jwt.sign(user, jwtSecretKey, { expiresIn: '10h' })
            return res.send({
                status: 0,
                msg: '登录成功',
                token: 'Bearer ' + tokenStr
            })
        }
        res.cc('登录失败,用户名或密码错误')
    })
}
```

解析token中间件

- 安装express-jwt 导入包和密钥 app全局注册 和 全局错误中间件
-

```
const { expressjwt } = require('express-jwt')
const { jwtSecretKey } = require('./config')
app.use(expressjwt({ secret: jwtSecretKey, algorithms: ['HS256'] }).unless({ path: [/^\/api\//] }))
app.use((err, req, res, next) => {
    console.log(err);
    // 这次错误是由 token 解析失败导致的
    if (err.id === 'UnauthorizedError') {
        return res.send({

            status: 401,
```

```
            message: '无效的token',
        })
    }
    res.send({
        status: 500,
        message: '未知的错误',
    })
})
```

获取用户信息

- 通过解析token 获取username
- 查询 username 对应的用户信息 并返回
-

```
exports.getUserInfo = (req, res) => {
    const userinfo = req.auth.username
    const sql = 'select id, username,nickname,email,user_pic from ev_users where username=?'
    db.query(sql, userinfo, (err, results) => {
        if (err) return res.cc(err)
        if (results.length !== 1) return res.cc('获取信息失败')
        res.send({
            status: 0,
            msg: '获取个人信息成功',
            data: results[0]
        })
    })
}
```

修改用户信息

- 通过 joi 定义验证规则 通过joi 使用验证规则
- 获取表单信息 和 解析token 获取username
- 通过 username 修改该用户的昵称、邮箱等信息
-

```
//schema
const joi = require('joi')
const id = joi.number().integer().min(1).required()
const nickname = joi.string().required()
const email = joi.string().required()
exports.userinfoSchema = {
    body: {
        id,
        nickname,
        email,
    }

}
```

```
//router
router.post('/my/updateUserInfo', expressjoi(userinfo.userinfoSchema), handler.updateUserInfo)
//router_handler
exports.updateUserInfo = (req, res) => {
    const data = req.body
    console.log(data);
    const username = req.auth.username
    const sql = 'update  ev_users set ? where username=?'
    db.query(sql, [data, username], (err, results) => {
        if (err) return res.cc(err)
        if (results.affectedRows !== 1) return res.cc('更新失败')
        res.send({
            status: 0,
            msg: '更新成功',
        })
    })
}

```

更新密码

- 表单验证 同修改个人信息
- 获取表单信息 和 解析token 获取username
- 通过查询给用户的数据库密码 旧密码相同时 执行更新语句 并对新密码加密

```
exports.updatepassword = (req, res) => {
    //获取客户端传过来的数据
    const data = req.body
    //解析token 获取用户信息
    const username = req.auth.username
    const sql = 'select password from ev_users where username=?'
    // const sql = 'update ev_users set ? where username =?'
    db.query(sql, username, (err, results) => {
        if (err) return res.cc(err)

        if (bcrypt.compareSync(data.oldpassword, results[0].password)) {
            const sql = 'update ev_users set password=? where username =?'
            db.query(sql, [bcrypt.hashSync(data.newpassword, 10), username], (err, results) => {
                if (err) return res.cc(err)
                if (results.affectedRows !== 1) return res.cc('更新密码')
                res.cc('密码更新成功', 0)
            })
        }
    })
}
```

补充方法

文件上传

- 模块 multer 引入muter
-

```
// 在 router
const multer = require('multer')
const stroage = multer.diskStorage({
    destination: function (req, file, cb) {
        cb(null, 'D:/毕设/毕设项目/contestServe/public')
    },
    filename: function (req, file, cb) {
        const name = (file.mimetype).split('/')[1]
        cb(null, file.fieldname + '-' + Date.now() + '.' + name)
    }
})
router.post('/upuserimg', upload.single('file'), userinfo.upimg)
//方法
   const name = 'http://127.0.0.1:3007/api/public/'
    const img = name + req.file.filename
```