Koa2服务框架搭建（通用模板）

*writed by **lvxin** at 2020-05-18*

> 基于nodejs的Koa框架搭建的服务端项目，借鉴Java服务端项目的框架的特点，结合了公司的实际项目，整理归纳的通用模板。

前言

大部分程序员都会有一颗成为全站程序员的心。

作为前端程序员，用nodejs作为进入后端的敲门砖，应该是最适合的选择了：

- 1. 前端开发中js知识的储备，减少了学习成本；
- 2. nodejs良好的发展势头和活跃的社区；
- 3. 有多种成熟的框架的选择；
- 4. 。。。

书写目的

在使用Koa2提供服务端接口的时候，会有很多重复的工作，比如请求的校验，返回的数据格式，以及异常的处理等等都是一些通用的东西，需要不断的重复书写。

这些内容如果不通过封装统一处理，不仅繁琐，还会出现不同的参与者书写不同的形式，甚至同一个开发者前后出现不一致的风格。

所以通过借鉴Java项目中的良好的架构，总结和整理了基于koa2框架的nodejs服务端模板。

在后续更多开发者参与nodejs开发后，希望大家提供更好更丰富的内容。

nodejs简介

Node.js是基于Chrome JavaScript运行时建立的一个平台，实际上它是对Google Chrome V8引擎进行了封装，它主要用于创建快速的、可扩展的网络应用。Node.js采用事件驱动和非阻塞I/O模型，使其变得轻量和高效，非常适合构建运行在分布式设备的数据密集型的实时应用。

运行于浏览器的JavaScript，浏览器就是JavaScript代码的解析器，而Node.js则是服务器端

JavaScript的代码解析器，存在于服务器端的JavaScript代码由Node.js来解析和运行。

JavaScript解析器只是JavaScript代码运行的一种环境，浏览器是JavaScript运行的一种环境，浏览器为JavaScript提供了操作DOM对象和window对象等的接口。Node.js也是JavaScript运行的一种环境，Node.js为JavaScript提供了操作文件、创建HTTP服务、 创建TCP/UDP服务等的接口，所以Node.js可以完成其他后台语言（Python、PHP等）能完成的工作。

koa2框架简介

关于中间件栈

每个中间件默认接受两个参数

- 第一个参数是 Context对象
- 第二个参数是next函数，对于一个中间件函数来讲，next函数很关键
    - 无next调用
        - 不执行后续的一系列中间件
    - 有next调用，表示接着往下执行
        - 调用next函数之前，执行当前中间件逻辑
        - 调用next函数之后，该函数暂停，把执行权转交给下一个中间件，下一个中间件开始执行

多个中间件通过app.use(middleWare)加载后，形成一个“中间件栈”，执行顺序以中间件里的next函数调用为界限：

```JavaScript
// app.js
const koa = require('koa');
const app = new koa();

const one = (ctx, next) => {
  console.log('one next之前')
  next()
  console.log('one next之后')
}

const two = (ctx, next) => {
  console.log('two next之前')
  next()
  console.log('two next之后')
}
const three = (ctx, next) => {
  console.log('three next之前')
  next()
}
const four = (ctx, next) => {
  next()
  console.log('four next之后')
}
const five = (ctx, next) => {
  console.log('five next之前')
  next()
  console.log('five next之后')
}

app.use(one)
app.use(two)
app.use(three)
app.use(four)
app.use(five)
app.use(ctx => {
  console.log('返回结果');
  ctx.body = 'hello'
})

app.listen(3000)
```

打印顺序：

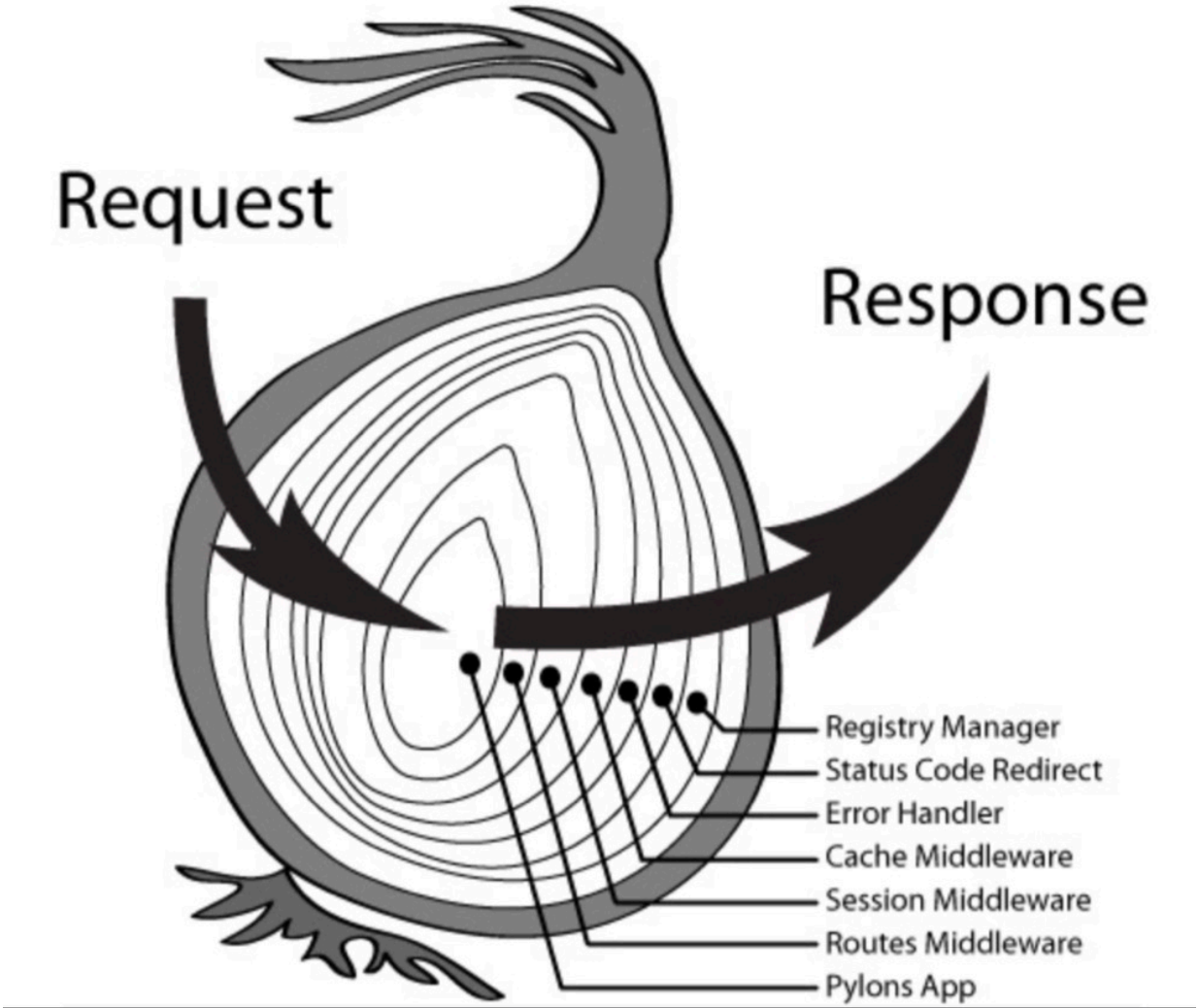
```
one next之前
two next之前
three next之前
five next之前
返回结果
five next之后
four next之后
two next之后
one next之后
```

1. 最外层的中间件首先执行。
2. 调用next函数，把执行权交给下一个中间件。
3. ...
4. 最内层的中间件最后执行。
5. 执行结束后，把执行权交回上一层的中间件。
6. ...
7. 最外层的中间件收回执行权之后，执行next函数后面的代码。

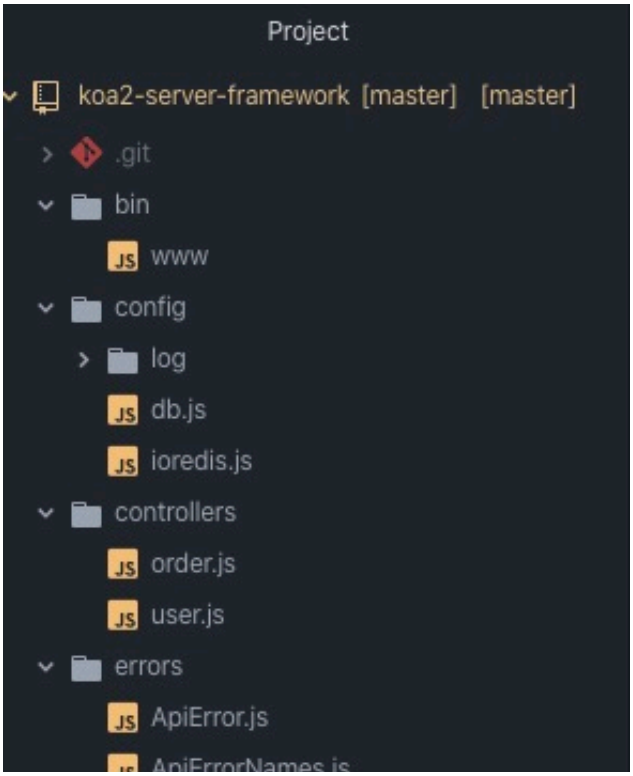
中间件流程控制简单描述就是：中间件的执行是以next为界限，先执行本层中next以前的部分，遇到next后执行下一层，一层层下去。当最后一层中间件执行完毕后，再返回上一层执行next以后的部分，一层层回来。所以如果某一层没有next，说明到该层方法就执行完毕了，就开始返回上一层执行上一层的next之后的部分了。

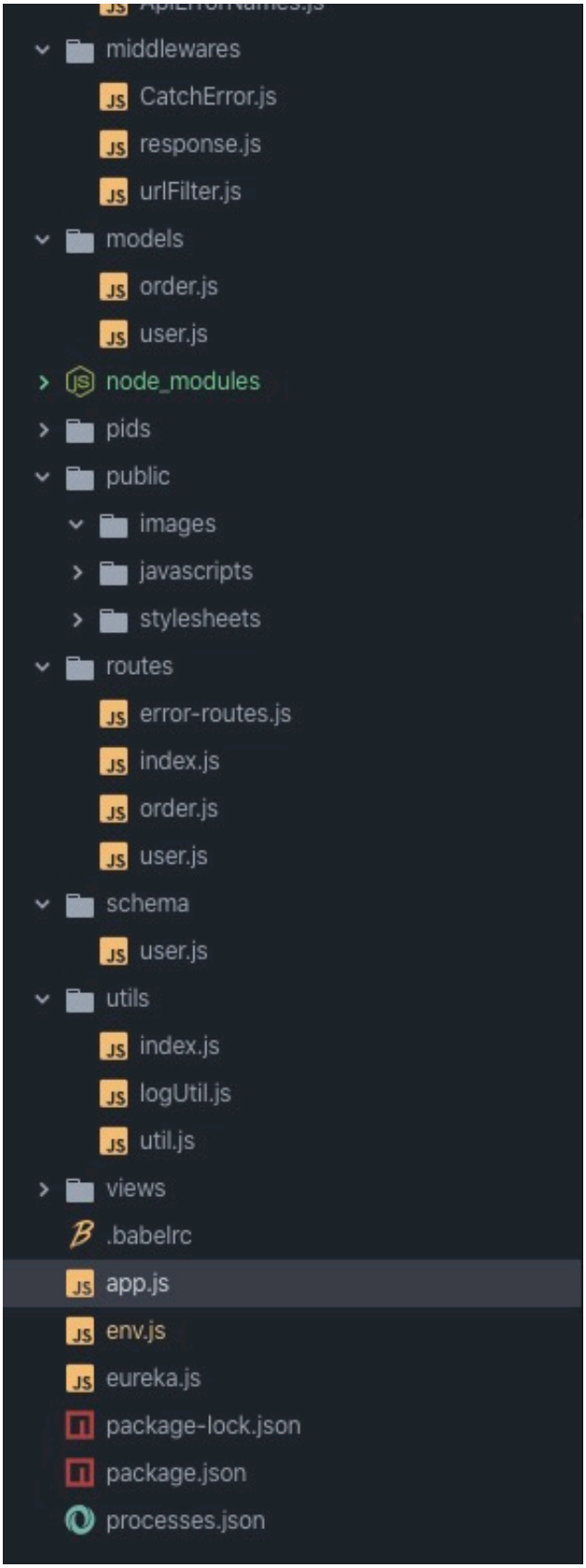
如果中间件内部没有调用next函数，那么执行权就不会传递下去，而是向上返回了。 试试如果把five函数离得next去掉，则浏览器不会正常显示返回内容(Not Found)了。

这也就是koa所谓的洋葱模型，从外面一层层的深入，再一层层的穿出来。


Request
Response
Registry Manager
Status Code Redirect
Error Handler
Cache Middleware
Session Middleware
Routes Middleware
Pylons App

项目结构


Project
koa2-server-framework [master] [master]
.git
bin
www
config
log
db.js
ioredis.js
controllers
order.js
user.js
errors
ApiError.js



我们的app.js

```JavaScript
// app.js
import Koa from 'koa'
```

```
import koaRouter from 'koa-router'
import views from 'koa-views'
import json from 'koa-json'
import onerror from 'koa-onerror'
import bodyparser from 'koa-bodyparser'
import logger from 'koa-logger'
import KoaBody from 'koa-body'
import path from 'path'
import './eureka'
import mainRoutes from './routes';
import errorRoutes from './routes/error-routes'
import urlFilter from './middlewares/urlFilter'
import response from './middlewares/response'
import catchError from './middlewares/catchError'

// const util = './utils/util'
const app = new Koa()
const router = koaRouter()

// error handler
onerror(app)

// middlewares
app.use(bodyparser({
  enableTypes: ['json', 'form', 'text']
}))
app.use(json())
app.use(logger())
app.use(require('koa-static')(__dirname + '/public'))

app.use(views(__dirname + '/views', {
  extension: 'ejs'
}))

// logger
app.use(async (ctx, next) => {
  const start = new Date()
  console.log('start logger')
  await next()
  const ms = new Date() - start
  console.log(`${ctx.method} ${ctx.url} - ${ms}ms`)
})

app.use(require('koa-static')(__dirname + '/public'))
app.use(catchError)
// 响应请求处理
app.use(response)

```

```
// 登录token验证
app.use(urlFilter('/auth/'))
app.use(mainRoutes.routes())
app.use(mainRoutes.allowedMethods())
app.use(errorRoutes())

// error-handling
app.on('error', (err, ctx) => {
  console.error('server error', err, ctx)
})

module.exports = app
```

解决问题

- 不同运行环境配置

JavaScript

```
// package.js
"scripts": {
    "start": "node bin/www nodemon index.js --exec babel-node",
    "dev": "cross-env NODE_ENV=dev PORT=3001 ./node_modules/.bin/no
    "beta": "cross-env NODE_ENV=beta PORT=3001 ./node_modules/.bin/
    "prd": "cross-env NODE_ENV=production PORT=3001 pm2 start bin/w
    "test": "echo \"Error: no test specified\" && exit 1",
    "lint": "eslint --ext .js"
  },
```

- 路由配置

JavaScript

```
// ./routes/index.js
import koaRouter from 'koa-router'
import User from './user'
import Order from './order'

const router = koaRouter()

router.prefix('/api')
  .use('/user', User.routes())
  .use('/order', Order.routes())

module.exports = router
```

- 网络请求的封装

JavaScript

```
// ./middlewares/uriFilter.js

// const VerifyToken = require('./tokenVerify')
import ApiError from '../errors/ApiError'
import ApiErrorNames from '../errors/ApiErrorNames'
import {
  newRedis,
  RedisGet
} from '../config/ioredis'

const urlFilter = (pattern) => {
  return async (ctx, next) => {
    let reg = new RegExp(pattern)
    // 先去执行路由
    if (reg.test(ctx.originalUrl)) {
      // 登录token验证
      const token = ctx.request.headers['x-access-token']
      // VerifyToken.verify(token)
      if (token) {
        console.log('**********', token, reg.test(ctx.originalUrl),
        const uuaToken = `uua-shiro-cache:shiro-activeSessionCache:
        var checkeToken = await RedisGet(uuaToken)
        if (checkeToken == null) {
          throw new ApiError(ApiErrorNames.USER_LOGIN_EXPIRED);
        }
      } else {
        throw new ApiError(ApiErrorNames.USER_UN_LOGIN)
      }
    }
    await next()

  }
}

module.exports = urlFilter
```

- 网络请求返回内容的封装

JavaScript

```
// ./middlewares/response.js
/*
 * 网络请求返回数据封装
 *
 * @ author lvxin
 * @ use 统一响应请求中间件
 * @ error-data 返回错误时，可携带的数据
 * @ error-msg  自定义的错误提示信息
 * @ error-status 错误返回码
 * @ error-errdata 可返回服务器生成的错误
 * @ success-data  请求成功时响应的数据
 * @ success-msg  请求成功时响应的提示信息
 * @ 调用ctx.error()   响应错误
 * @ 调用ctx.success()  响应成功
 */
import ApiError from '../errors/ApiError'

module.exports = async (ctx, next) => {
  try {
    ctx.error = ({
      status,
      code,
      msg,
      data
    }) => {
      ctx.status = status || 200;
      ctx.body = {
        code: code || '-1',
        msg: msg || '请求处理失败',
        data: data || null
      };
    }
    ctx.success = ({
      data,
      msg
    }) => {
      ctx.body = {
        code: 1,
        msg: msg || '请求处理成功',
        data: data
      };
    }
    await next()
  } catch (error) {
    throw error
  }
}
```

- 异常捕获封装

JavaScript

```
// ./middlewares/response.js
/*
 * 网络请求返回数据封装
 *
 * @ author lvxin
 * @ use 统一响应请求中间件
 * @ error-data 返回错误时，可携带的数据
 * @ error-msg  自定义的错误提示信息
 * @ error-status 错误返回码
 * @ error-errdata 可返回服务器生成的错误
 * @ success-data  请求成功时响应的数据
 * @ success-msg  请求成功时响应的提示信息
 * @ 调用ctx.error()   响应错误
 * @ 调用ctx.success()  响应成功
 */
import ApiError from '../errors/ApiError'

module.exports = async (ctx, next) => {
  try {
    ctx.error = ({
      status,
      code,
      msg,
      data
    }) => {
      ctx.status = status || 200;
      ctx.body = {
        code: code || '-1',
        msg: msg || '请求处理失败',
        data: data || null
      };
    }
    ctx.success = ({
      data,
      msg
    }) => {
      ctx.body = {
        code: 1,
        msg: msg || '请求处理成功',
        data: data
      };
    }
    await next()
  } catch (error) {
    throw error
  }
}
```

异常错误封装

```JavaScript
import ApiErrorNames from './ApiErrorNames'

// 自定义API异常
class ApiError extends Error {
  constructor(errorName) {
    super()
    var errorInfo = ApiErrorNames.getErrorInfo(errorName)

    this.name = errorName
    this.code = errorInfo.code
    this.message = errorInfo.message
  }
}

module.exports = ApiError
```

异常定义

JavaScript

```
// API 错误名称
var ApiErrorNames = {}

ApiErrorNames.UNKNOW_ERROR = 'unknow_error'
ApiErrorNames.USER_NOT_EXIST = 'user_not_exist'
ApiErrorNames.USER_LOGIN_EXPIRED = 'user_login_expired'
ApiErrorNames.USER_UN_LOGIN = 'user_un_login'

// API 错误名称对应的错误信息
const errorMap = new Map()

errorMap.set(ApiErrorNames.UNKNOW_ERROR, {
  code: -1,
  message: '未知错误'
})
errorMap.set(ApiErrorNames.USER_NOT_EXIST, {
  code: 101,
  message: '用户不存在'
})
errorMap.set(ApiErrorNames.USER_UN_LOGIN, {
  code: '-200',
  message: '用户登录'
})
errorMap.set(ApiErrorNames.USER_LOGIN_EXPIRED, {
  code: '-201',
  message: '用户登录过期'
})

// 根据错误名称获取错误信息
ApiErrorNames.getErrorInfo = (errorName) => {
  let errorInfo = null

  if (errorName) {
    errorInfo = errorMap.get(errorName)
  }

  // 如果没有对应的错误信息，默认‘未知错误’
  if (!errorInfo) {
    errorName = ApiErrorNames.UNKNOW_ERROR
    errorInfo = errorMap.get(errorName)
  }

  return errorInfo
}

module.exports = ApiErrorNames
```

- 数据库操作

```JavaScript
// ./config/db.js
import Sequelize from 'sequelize'
import database from '../env'

console.log('process.env.NODE_ENV=db---' + process.env.NODE_ENV)
console.log(database[process.env.NODE_ENV], '******database')

let configInfo = database[process.env.NODE_ENV]
// // 预发环境
const config = {
  // 数据库
  database: 'healthy_life_management',
  // 用户名
  username: configInfo.username,
  // 密码
  password: configInfo.password,
  // 使用哪个数据库程序
  dialect: 'mysql',
  // 地址
  host: configInfo.dbHost,
  // 端口
  port: configInfo.dbPort,
  // 连接池
  pool: {
    max: 5,
    min: 0,
    acquire: 30000,
    idle: 10000
  },
  define: {
    timestamps: false // 取消Sequelize自动给数据表加入时间戳（createdAt
  },
  timezone: '+08:00' // Mysql时区与Node时区不一致问题
}
console.log(config, 'config')
const HealthyDB = new Sequelize(config)

module.exports = {
  HealthyDB // 将HealthyDB暴露出接口，方便Model调用
}
```

- 生成db models

```
sequelize-auto -o "./schema" -d todolist -h 127.0.0.1 -u root -p 33
```

参数解释：

-o 参数后面的是输出的文件夹目录，

-d 参数后面的是数据库名，

-h 参数后面是数据库地址，

-u 参数后面是数据库用户名，

-p 参数后面是端口号，

-x 参数后面是数据库密码，这个要根据自己的数据库密码来！

-e 参数后面指定数据库为mysql

-t 参数后面指定表名

结束语

JUST DO IT.