Node.js基础

Nodejs是什么

nodejs是一个异步的事件驱动的javaScript运行时

https://nodejs.org/en/

node.js特性其实是js的特性

- 非阻塞I/O
- 事件驱动

node历史-为性能而生

并发处理

- 多进程-CApache
- 多线程-java
- 异步IO-js
- 协程-lua openresty go deno-go TS

下一代Node deno

https://studygolang.com/articles/13101

与前端不同

- js核心语法不变
- 前端BOM DOM
- 后端fs http buffer event os
- 运行node程序

```
1 //01-run.js
2 console.log('hello word')
3 console.log('222')
```

运行：node 01-run.js

每次修改js文件需要重新执行才能生效，安装nodemon可以监视文件改动，自动重启：

npm i nodemon -g

- 调试node程序：Debug -Start Debugging

  https://nodejs.org/en/

eventloop的区别（https://www.processon.com/view/link/5e70b1c2e4b011fcce9b89b5#map）

模块(module)

- 使用模块(module)
  - node内建模块

```
//02-module.js
const os=require('os');
const mem=os.freemem()/os.totalmem()*100;
console.log("内存占用率"+mem.toFixed(2)+"%")
```

  - 第三方模块

    https://www.npmjs.com/

```
//同级cpu占用率，先安装
npm i download-git-repo -S
```

```
const download=require("download-git-repo");
const repo='https://github.com/su37josephxia/vue-template';
const desc='../test';
download('','../test',err=>{
    console.log(err,'Eorr','Sucess')
})
```

  - 完善代码

```
const download=require("download-git-repo");
const ora=require('ora')
const process=ora(`火箭下载...项目`)
process.start();
const repo='https://github.com/su37josephxia/vue-template';
const desc='../test';
download('','../test',err=>{
        // console.log(err,'Eorr','Sucess')
        if(err){
            process.fail();
        }else{
            process.succeed();
        }
})
```

  - promisefy

    如何让异步任务串行化

```
const repo = 'github:su37josephxia/vue-template'
const desc = '../test';
clone(repo, desc)
async function clone(repo, desc) {
    const { promisify } = require('uitl');
    const download = require("download-git-repo");
    const process = ora(`火箭下载...项目`)
    process.start();
    try {
        await download(repo,desc);
```

```
    } catch (error) {
        process.fail();
    }
    process.succeed();
}
```

- 自定义模块：代码分割、复用手段

```
//download.js
module.exports.clone=async function clone(repo, desc) {
    const { promisify } = require('uitl');
    const download = require("download-git-repo");
    const process = ora(`火箭下载...项目`)
    process.start();
    try {
        await download(repo,desc);
    } catch (error) {
        process.fail();
    }
    process.succeed();
}

//module.js
const {clone}=reuire('./download')
clone();
```

导出内容可以是导出对象的属性

```
//download.js
module.exports.clone=async function()

//module.js
const {clone}=reuire('./download')
clone();
```

核心API

- fs-文件系统

```
const fs=require('fs');

//同步调用
const data=fs.readFileSync('./run.js');
console.log(data.toString())

//异步调用
fs.readFile('./run.js',(err,data)=>{
    if(err){ throw err}
    console.log(data.toString())
})
console.log('其他操作');

// //fs常常搭配path api使用
const path=require('path');
```

```
fs.readFile(path.resolve(path.resolve(__dirname,'./run.js')),
(err,data)=>{
    if(err) throw err
    console.log(data.toString())
})


//promisify
const {promisify}=require('util');
const readFile=promisify(fs.readFile);
readFile('./run.js').then(data=>console.log(data.toString()));

//fs promises api node v10
const fsp=require('fs').promises;
fsp.readFile('./run.js').then(data=>console.log(data.toString()))
.catch(err=>console.log(err))

//async/await
(async ()=>{
    const fs=require('fs');
    const {promisify}=require('util');
    const readFile=promisify(fs.readFile);
    const data=await readFile('./index.html');
    console.log('data',data.toString())
})()

//引用方式
Buffer.from(data).toString('utf-8')
```

读取数据类型为Buffer

- Buffer-用于在TCP流、文件系统操作、以及其他上下文中与八位字节流进行交互。八位字节组成的数组，可以有效的在js中存储二进制数据

```
//创建一个长度为10字节以0填充的Buffer
const buf=Buffer.alloc(10);
console.log(buf)

//创建一个Buffer包含ascii
//ascii查询http://ascii.911cha.com/
const buf2=Buffer.from('a');
console.log(buf2,buf2.toString())

//创建Buffer包含UTF-8字节
//UTF-8:一种变长的编码方案，使用1-6个字节来存储
//UTF-32:一种固定长度的编码方案，不管字符编号的大小，始终使用4个字节来存储
//UTF-16:介于UTF-8和UTF-32之间，使用2个或者4个字节来存储，长度及固定又可变
const buf3=Buffer.from('中');
console.log(buf3)

//写入Buffer数据
buf.write('hello');
console.log(buf);

//读取Buffer数据
console.log(buf3.toString());
```

```
//合并Buffer
const buf4=Buffer.concat(buf2,buf3);
console.log(buf4.toString())
```

Buffer类似数组，所以很多数组方法它都有GBK转码iconv-lite

- http：用于创建web服务的模块

  创建一个http服务器

```
const http=require('http');
const fs=require('fs');
const server=http.createServer((req,res)=>{
     console.log('this is a request');
     response.end('a response from server');
});

server.listen(3000)
```

```
//打印原型链
function getPrototypeChain(obj){
    var protoChain=[];
    while(obj==Object.getPrototypeOf(obj)){
        //返回给定对象的原型。如果没有继承属性，则返回null
        protoChain.push(obj);
    }
    protoChain.push(null);
    return protoChain;
}
```

显示一个首页

```
const http=require('http');
const fs=require('fs');
const server=http.createServer((req,res)=>{
    // console.log('11');
    // response.end('hello node');
    const {url,method}=req;
    console.log(url,method)
    if(url==='/'&&method==='GET'){
        fs.readFile('./index.html',(err,data)=>{
            console.log(err)
            if(err){
                res.writeHead(500,{
                    'Content-Type':'text/plain;charset=utf-8'
                });
                res.end(`500服务器端错误`);
                return;
            }
            res.statusCode=200;
            res.setHeader('Content-Type','text/html');
            res.end(data)
        })
    }else if(url==='/users'&&method==='GET'){
        res.writeHead(200,{'Content-Type':'application/json'});
        res.end(JSON.stringify([{name:'tom',age:20}]))
    }
```

```
    else{
        res.statusCode=404;
        res.setHeader('Content-Type','text/plain;charset=utf-8');
        res.end('404,页面没有找到')
    }
});

server.listen(3000)
```

- sream-是用于与node中流数据交互交互的接口

```
//stream.js
const fs=require('fs');

//创建输入输出流
const  rs=fs.createReadStream('./fs.js');
const ws=fs.createWriteStream('./fs2.js');
rs.pipe(ws)

//二进制友好,图片操作
const rs2=fs.createReadStream('./img.jpg');
const ws2=fs.createWriteStream('./img2.jpg');

rs2.pipe(ws2)

//图片响应请求，http.js
 const {url,method,headers}=req;
 else if(method==='GET'&&headers.accept.indexOf('image/*')!==-1){
        fs.createReadStream('.'+url).pipe(res)
 }
```

Accept代表发送端（客户端）希望接收的数据类型。比如：Accept:text/xml;代表客户端希望接受的数据类型为xml类型

Content-Type代表发送端（客户端|服务器）发送的实体数据的数据类型。比如：Content-Type:text/html;代表发送端发送的数据格式是html

二者合起来，Accept:text/xml;Content-Type:text/html;即代表希望接收的数据类型是xml格式，本次请求发送的数据的数据格式是html

异步编程- 如何控制好异步过程

参考资料料 阮阮一一峰 Javascript异步编程的4种方方法（http://www.ruanyifeng.com/blog/2012/12/asynchronous%EF%BC%BFjavascript.html）

- JS的执行行行环境是单线程（Single thread）
- I/O处理理需要回调函数异步处理理 (异步I/O)
- 前端异步IO可以消除UI阻塞，提高用户体验
- 而放在后端则可以提高CPU和内存里利用率

串联异步处理

异步操作队列列化，按照期望的顺序执行。

Callback

回调地域太可怕

```
const logTime = name => {
  console.log(`Log....${name} ` + new Date().toLocaleTimeString());
};
exports.callback = () => {
  setTimeout(() => {
    logTime("callback 1");
    setTimeout(() => {
      logTime("callback 2");
    }, 100);
  }, 100);
};

```

测试代码

```
test('callback', done => {
callback()
// 延时4s结束
setTimeout(done, 1000)
})
```

Promise

Promise对象用于异步操作，它表示一个尚未完成且预计在未来完成的异步操作。说白了了就是一个异步执行的状态机，异步执行的承诺

```
const promise = (name, delay = 100) =>
  new Promise(resolve => {
    setTimeout(() => {
      logTime(name);
      resolve();
    }, delay);
  });
exports.promise = () => {
  promise("Promise1")
    .then(promise("Promise2"))
    .then(promise("Promise3"))
    .then(promise("Promise4"));
};
```

Gennerator

ES6 新引入了 Generator 函数，可以通过 yield 关键字，把函数的执行流挂起，为改变执行流程提供了可能，从而为异步编程提供解决方方案。

- function -> function* 称为Gennerator函数
- 函数内部有 yield 表达式

```
function* func() {
  console.log("one");

```

```
  yield "1";
  console.log("two");
  yield "2";
  console.log("three");
  return "3";
}
const f = func();
f.next();
// one
// {value: "1", done: false}
f.next();
// two
// {value: "2", done: false}
f.next();
// three
// {value: "3", done: true}
f.next();
// {value: undefined, done: true}
// 或者通过迭代器器
for (const [key, value] of func()) {
  console.log(`${key}: ${value}`);
}
```

逻辑代码

```
const co = function(gen, name) {
  const it = gen(name);
  const ret = it.next();
  ret.value.then(function(res) {
    it.next(res);
  });
};
exports.generator = () => {
  const generator = function*(name) {
    yield promise(name + 1);
    yield promise(name + 2);
    yield promise(name + 3);
    yield promise(name + 4);
  };
  const co = generator => {
    if ((it = generator.next().value)) {
      it.then(res => {
        co(generator);
      });
    } else {
      return;
    }
  };
  co(generator("Co-Generator"));
};
```

async/await

async/await是es7推出的一套关于异步的终极解决方案

- 任何一个await语句后面的 Promise 对象变为reject状态，那么整个async函数都会中断执行。
- async函数返回的 Promise 对象，必须等到内部所有await命令后面的 Promise 对象执行完，才会发生生状态改变，除非遇到return语句或者抛出错误。也就是说，只有async函数内部的异步操作执行完，才会执行then方法指定的回调函数。

```
exports.asyncAwait = async () => {
  await promise('Async/Await1')
  await promise('Async/Await2')
  await promise('Async/Await3')
  await promise('Async/Await4')
  }
```

事件监听方式处理

采用事件驱动模式。任务的执行不不取决于代码的顺序，而取决于某个事件是否发生。

```
  exports.event = async () => {
    const asyncFun = name => event => {
    setTimeout(() => {
    logTime(name)
    event.emit('end')
    }, 100)
    return event
    }
    const ary = [
    asyncFun('event1'),
    asyncFun('event2'),
    asyncFun('event3')
  ]
  const { EventEmitter } =require('events')
  const event = new EventEmitter()
  let i = 0
  event.on('end', () => i < ary.length &&
  ary[i++](event))
  event.emit('end')
  }
```

eventEmmiter

```
const promise = (name, delay = 100) =>
  new Promise(resolve => {
    setTimeout(() => {
      logTime(name);
      resolve();
    }, delay);
  });
exports.promise = () => {
  promise("Promise1")
    .then(promise("Promise2"))
    .then(promise("Promise3"))
    .then(promise("Promise4"));
```

```
};

```

扩展阅读eventEimitter源码解析 / 订阅发布机制

```
class EventEmitter {
  constructor() {
    this.handler = {};
  }
  on(eventName, callback) {
    if (!this.handles) {
      this.handles = {};
    }
    if (!this.handles[eventName]) {
      this.handles[eventName] = [];
    }
    this.handles[eventName].push(callback);
  }
  emit(eventName, ...arg) {
    if (this.handles[eventName]) {
      for (var i = 0; i < this.handles[eventName].length; i++) {
        this.handles[eventName][i](...arg);
      }
    }
  }
}
const event = new EventEmitter();
event.on("some_event", num => {
  console.log("some_event 事件触发:" + num);
});
let num = 0;
setInterval(() => {
  event.emit("some_event", num++);
}, 1000);

```

CLI工具

创建工程

```
mkdir vue-auto-router-cli
cd vue-auto-router-cli
npm init -y
npm i commander download-git-repo ora handlebars figlet clear chalk open -S
```

```
# bin/kkb.js
#指定脚本解释器为node
#!/usr/bin/env node
console.log('cli.....')
# package.json
"bin": {
 "kkb": "./bin/kkb.js"
},
# 将npm 模块链接到对应的运行项目中去
npm link
# 删除的情况
ls /usr/local/bin/
rm /usr/local/bin/kkb
```

定制命令行界面

kkb.js文件

```
#!/usr/bin/env node
const program = require('commander')
program.version(require('../package').version)
program
 .command('init <name>')
 .description('init project')
 .action(name => {
console.log('init ' + name)
})
program.parse(process.argv)
```

打印欢迎界面

/lib/init.js

```
const {promisify} = require('util')
const figlet = promisify(require('figlet'))
const clear = require('clear')
const chalk = require('chalk')
const log = content => console.log(chalk.green(content))
module.exports = async name => {
  // 打印欢迎画面
  clear()
  const data = await figlet('KKB Welcome')
  log(data)
}
```

```
// bin/kkb.js
program
 .command('init <name>')
 .description('init project')
 .action(require('../lib/init'))
```

克隆脚手架

/lib/download.js

```
const {promisify} = require('util')
module.exports.clone = async function(repo,desc) {
  const download = promisify(require('download-git-repo'))
  const ora = require('ora')
  const process = ora(`下载.....${repo}`)
  process.start()
  await download(repo, desc)
  process.succeed()
}
```

/lib/init.js

```
const {clone} = require('./download')
module.exports.init = async name => {
  // console.log('init ' + name)
  log('🚀创建项目:' + name)
  // 从github克隆项目到指定文件夹
  await clone('github:su37josephxia/vue-template', name)
}
```

安装依赖

```
// promisiy化spawn
// 对接输出流
const spawn = async (...args) => {
  const { spawn } = require('child_process');
  return new Promise(resolve => {
    const proc = spawn(...args)
    proc.stdout.pipe(process.stdout)
    proc.stderr.pipe(process.stderr)
    proc.on('close', () => {
      resolve()
    })
  })
}
module.exports.init = async name => {

  // ....
  log('安装依赖')
  await spawn('cnpm', ['install'], { cwd: `./${name}` })
  log(chalk.green(`
👌安装完成:
To get Start:
===========================
  cd ${name}
  npm run serve
===========================
      `))
}
```

启动项目

```
const open = require("open")
module.exports.init = async name => {
// ...
// 打开浏览器
  open(`http://localhost:8080`);
  await spawn('npm', ['run', 'serve'], { cwd: `./${name}` })
}
```

约定路由功能

- loader 文件扫描
- 代码模板渲染 hbs Mustache风格模板

/lib/refresh.js

```
const fs = require('fs')
const handlebars = require('handlebars')
const chalk = require('chalk')
module.exports = async () => {
  // 获取页面列表
  const list =
    fs.readdirSync('./src/views')
     .filter(v => v !== 'Home.vue')
     .map(v => ({
        name: v.replace('.vue', '').toLowerCase(),
        file: v
     }))
  // 生成路由定义
  compile({
    list
 }, './src/router.js', './template/router.js.hbs')
  // 生成菜单
  compile({
    list
 }, './src/App.vue', './template/App.vue.hbs')
  /**
  * 编译模板文件
  * @param meta 数据定义
  * @param filePath 目标文件路径
  * @param templatePath 模板文件路径
  */
  function compile(meta, filePath, templatePath) {
    if (fs.existsSync(templatePath)) {
      const content = fs.readFileSync(templatePath).toString();
      const result = handlebars.compile(content)(meta);
      fs.writeFileSync(filePath, result);
   }
    console.log(chalk.green(`🚀${filePath} 创建成功`))
 }
}
```

/bin/kkb

```
program
 .command('refresh')
 .description('refresh routers...')
 .action(require('../lib/refresh'))
```

发布npm

```
#!/usr/bin/env bash
npm config get registry # 检查仓库镜像库
npm config set registry=http://registry.npmjs.org
echo '请进行登录相关操作：'
npm login # 登陆
echo "-------publishing-------"
npm publish # 发布
npm config set registry=https://registry.npm.taobao.org # 设置为淘宝镜像
echo "发布完成"
exit
```

Koa2源码解读

知识点

koa

- 概述：Koa是一个新的web框架，致力于成为web应用和API开发领域中的一个更小，更富有表现力、更健壮的基石
- 特点：
  - 轻量，无捆绑
  - 中间件架构
  - 优雅的API设计
  - 增强的错误处理
- 安装：npm i koa -S
- 中间件机制、请求、响应处理

```
const Koa=require('koa');
const app=new Koa();
app.use((ctx,next)=>{
    ctx.body=[{
        name:'tom'
    }]
    next();
})

app.use((ctx,next)=>{
    console.log('url',ctx.url);
    if(ctx.url==='/html'){
        ctx.type='text/html;charset=utf-8';
        ctx.body=`<b>我的名字是:${ctx.body[0].name}</b>`
    }
})

app.listen(3000);
```

```
//搞个小路由
const router={};
router['/html']=ctx=>{
    ctx.type='text/html;charset=utf-8';
    ctx.body=`<b>我的名字是:${ctx.body[0].name}</b>`
};
router[ctx.url](ctx)
```

常见的中间件操作

- 静态服务

```
app.use(require('koa-static')(__dirname+'/'));
```

- 路由

```
const router=require('koa-router')();
router.get('/string',async(ctx,next)=>{
    ctx.body='koa2 string';
})
router.get('/json',async(ctx,next)=>{
    ctx.body={
        title:'koa2 json'
    }
})

app.use(router.routes())

```

- 日志

```
app.use(async(ctx,next)=>{
    const start=new Date().getTime();
    console.log(`start:${ctx.url}`);
    await next();
    const end=new Date().getTime();
    console.log(`请求${ctx.url},耗时${parseInt(end-start)}ms`)
})
```

```
const Koa=require('koa');
const app=new Koa();
app.use(require('koa-static')(__dirname+'/'));

const router=require('koa-router')();
router.get('/string',async(ctx,next)=>{
    ctx.body='koa2 string';
})
router.get('/json',async(ctx,next)=>{
    ctx.body={
        title:'koa2 json'
    }
})


app.use(async(ctx,next)=>{
```

```
    const start=new Date().getTime();
    console.log(`start:${ctx.url}`);
    await next();
    const end=new Date().getTime();
    console.log(`请求${ctx.url},耗时${parseInt(end-start)}ms`)
})

app.use(router.routes())


app.listen(3000)
```

注意：ctx.request是context经过封装的请求对象，ctx.req是context提供的node.js原生HTTP请求对象，同理ctx.response是context经过封装的响应对象，ctx.res是context提供的node.js原生HTTP请求对象。

具体koa2 API文档可见 https://github.com/koajs/koa/blob/master/docs/api/context.md#ctxreq

Koa原理

- 一个基于nodejs的入门级http服务，类似下面的代码

```
const http=require('http');
const server=http.createServer((req,res)=>{
    res.writeHead(200);
    res.end('hi kaikeba')
})
server.listen(3000,()=>{
    console.log('监听3000端口')
})
```

- koa的目标是用更简单化、流程化、模块化的方式实现回调部分

```
//创建kkb.js
const http=require('http');
class KKB{
    listen(...args){
        const server=http.createServer((req,res)=>{
            this.callback(req,res)
        })
        server.listen(...args);
    }
    use(callback){
        this.callback=callback;
    }
}

module.exports=KKB;

//调用app.js
const KKB=require('./kkb');
const app=new KKB();
```

```
app.use((req,res)=>{
    res.writeHead(200);
    res.end('hi kaikeba');
})
app.listen(3000,()=>{
    console.log('监听端口3000')
})
```

context

- koa为了能够简化API，引入上下文context概念，将原始请求对象req和响应对象res封装并挂载到context上，并且在context上设置getter和setter,从而简化操作

```
//app.js
app.use(ctx=>{
    ctx.body='hehe'
})
```

- 封装request、response和context

  https://github.com/koajs/koa/blob/master/lib/response.js

```
//request.js
module.exports={
    get url(){
        return this.req.url;
    },
    get method(){
        return  this.req.method.toLowerCase();
    },
}

//response.js
module.exports={
    get body(){
      return this._body;
    },
    set body(val){
        this._body=val;
    }
}

//context.js
module.exports={
    get url(){
        return this.request.url;
    },
    get body(){
        return this.response.body;
    },
    set body(val){
        this.response.body=val
    },
    get method(){
        return this.request.method;
    }
```

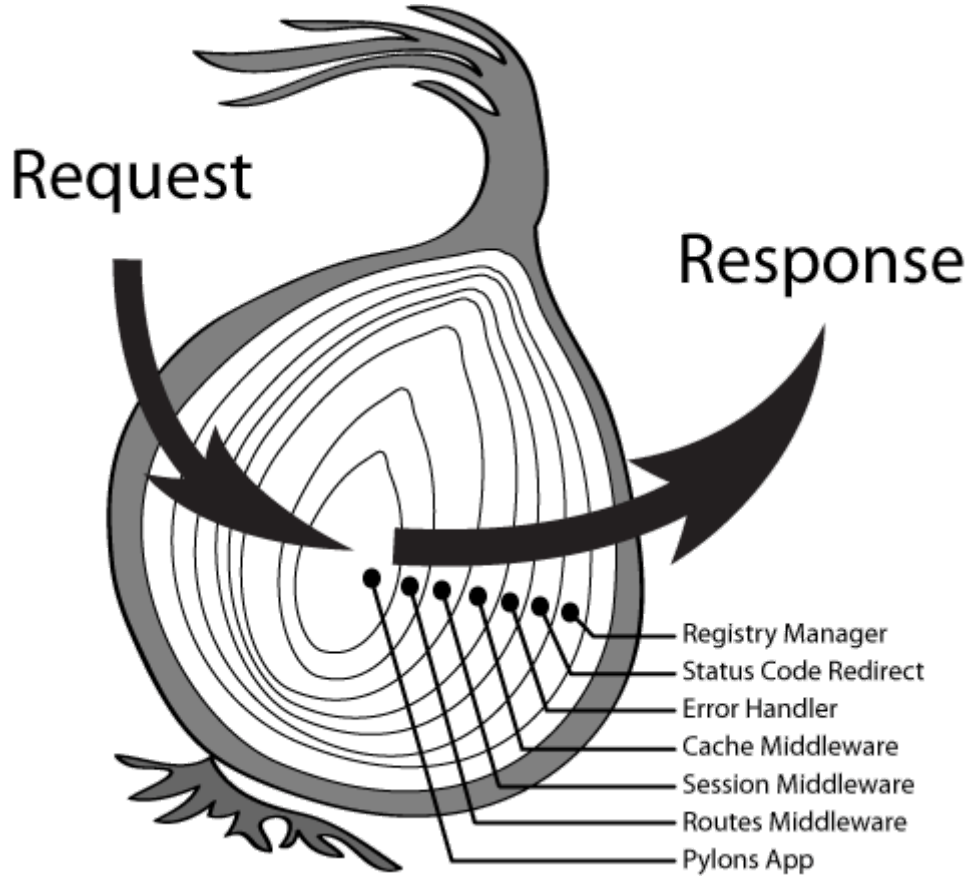
```
}
```

```
//kkb2.js
const http=require('http');
const context=require('./context')
const request=require('./request')
const response=require('./response')
class KKB{
    listen(...args){
        const server=http.createServer((req,res)=>{
            //创建上下文
            let ctx=this.createContext(req,res);
            this.callback(ctx);
            //响应
            res.end(ctx.body);
        })
        server.listen(...args);
    }

    //构建上下文，把res和req都挂载到ctx之上，并且在ctx.req和ctx.request.req同时保存
    createContext(req,res){
        const ctx=Object.create(context);
        ctx.request=Object.create(request);
        ctx.response=Object.create(response);

        ctx.req=ctx.request.req=req;
        ctx.res=ctx.response.res=res;
        return ctx;
    }

    use(callback){
        this.callback=callback;
    }
}

module.exports=KKB;
```

中间件

- Koa中间件机制：Koa中间件机制就是函数组合的概念，将一组需要顺序执行的函数复合为一个函数。外层函数的参数实际是内层函数的返回值。洋葱圈模型可以形象表示这种机制，是源码中的精髓和难点。

  中间件洋葱图：


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

中间件执行顺序图：

```
var koa = require('koa');
var app = koa();

app.use(function* responseTime(next) {
  var start = new Date;
  yield next;
  var ms = new Date - start;
  this.set('X-Response-Time', ms + 'ms');
});

app.use(function* logger(next) {
  var start = new Date;
  yield next;
  var used = new Date - start;
  console.log('%s %s %s %sms',
    this.method,
    this.originalUrl,
    this.status, used);
});

app.use(function* contentLength(next) {
  yield next;
  if (!this.body) return;
  this.set('Content-Length', this.body.length);
});

app.use(function* body(next) {
  yield next;
  if (this.path !== '/') return;
  this.body = 'Hello World';
});

app.listen(3000);
```

- 知识储备：函数组合

```
const add=(x,y)=>x+y;
const square=z=>z*z;
const fn=(x,y)=>square(add(x,y));
console.log(fn(1,2))
```

上面就算是俩次函数组合调用，我们可以把它合并成一个函数

```
const compose=(fn1,fn2)=>(...args)=>fn2(fn1(...args));
const fn=compose(add,square);
```

多个函数组合：中间件的数目是不固定的，我们可以采用数组来模拟

```
const compose=(...[first,...other])=>(...args)=>{
    let ret=first(...args);
    other.forEach(fn=>{
        ret=fn(ret);
    })
    return ret;
}
const fn=compose(add,square);
console.log(fn(1,2))
```

- 异步中间件：上面的函数都是同步的，挨个遍历执行即可，如果是异步的函数呢，是一个promise，我们要支持async+await的中间件，所以我们要等异步结束后，在执行下一个中间件

```
function composeAsync(middlewares){
  return function(){
      return dispatch(0);
      //执行第0个
      function dispatch(i){
          let fn=middlewares[i];
          if(!fn){
              return Promise.resolve();
          }
          return Promise.resolve(fn(function next(){
              //promise完成后，在执行下一个
              return dispatch(i+1)
          }))
      }
  }
}

async function fn1(next){
    console.log("fn1");
    await next();
    console.log("end fn1")
}

async function fn2(next){
    console.log("fn2");
    await delay();
    await next();
    console.log("end fn2");
}

function fn3(next){
    console.log("fn3")
}

function delay(){
    return new Promise((resolve,reject)=>{
        setTimeout(()=>{
            resolve();
        },2000)
    })
}

const middlewares=[fn1,fn2,fn3];
```

```
const finalFn=composeAsync(middlewares);
finalFn();
```

```
[nodemon] to restart at any time, enter `rs`
[nodemon] watching: *.*
[nodemon] starting `node .\square.js`
fn1
fn2
fn3
end fn2
end fn1
[nodemon] clean exit - waiting for changes before
```

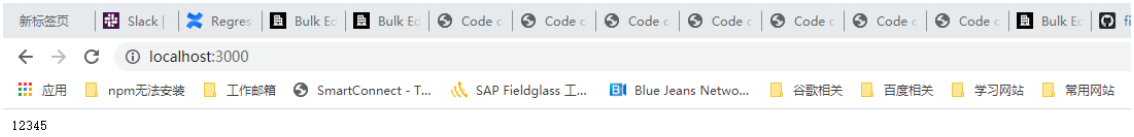
- compose用在koa中，kkb3.js

```
const http=require('http')
const context=require('./context')
const request=require('./request')
const response=require('./response')
class KKB{
    //初始化中间件数组
    constructor(){
        this.middlewares=[];
    }
    listen(...args){
        const server=http.createServer(async (req,res)=>{
            //创建上下文
            let ctx=this.createContext(req,res);
            //中间件合成
            const fn=this.compose(this.middlewares);
            //执行合并函数并传入上下文
            await fn(ctx);
            //响应
            res.end(ctx.body);
        })
        server.listen(...args);
    }

    use(middleware){
        //将中间件加到数组里
        this.middlewares.push(middleware)
    }

    //合成函数
    compose(middlewares){
        return function(ctx){
            //传入上下文
            return dispatch(0);
            function dispatch(i){
                let fn=middlewares[i];
                if(!fn){
                    return Promise.resolve();
                }
```

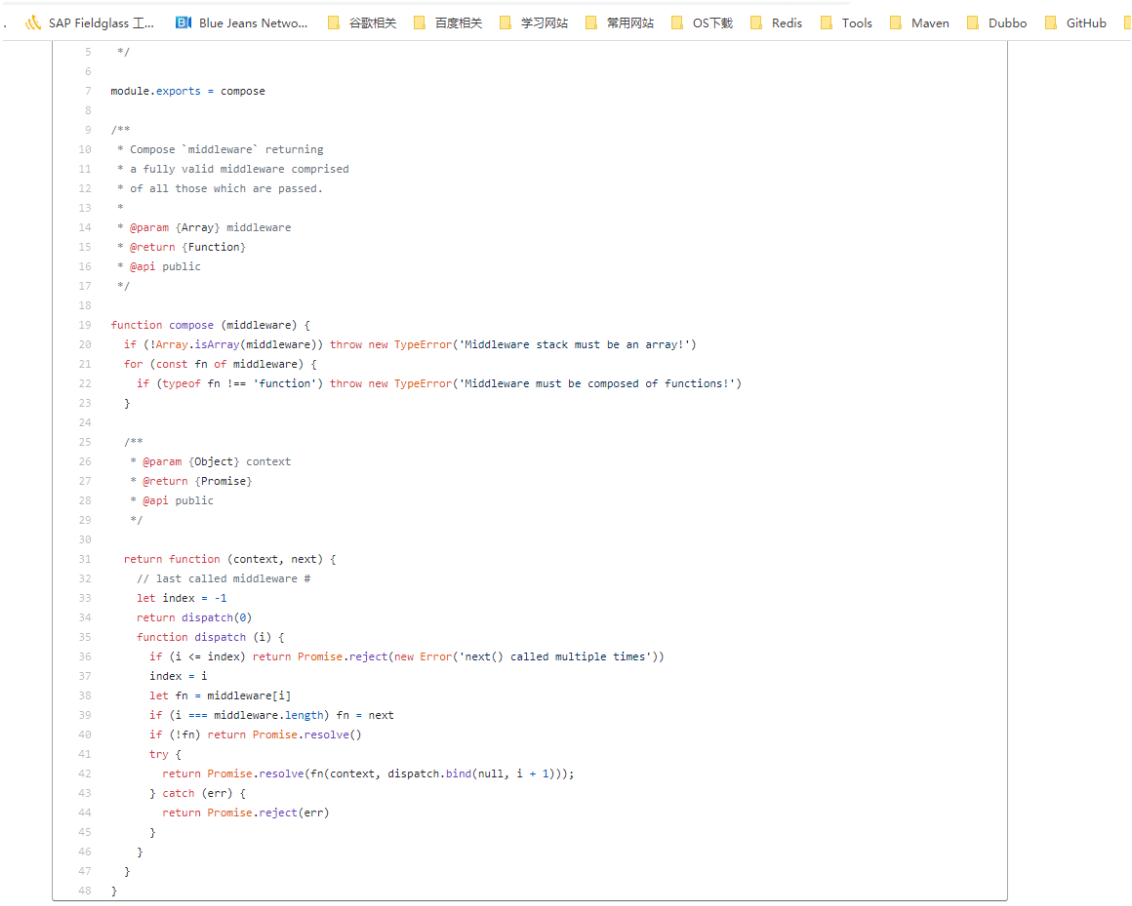
```
                return Promise.resolve(
                    fn(ctx,function next(){
                        //将上下文传入中间件,mid(ctx,next)
                        return dispatch(i+1);
                    }
                ))
            };
        }
    }

    //构建上下文，把res和req都挂载到ctx之上，并且在ctx.req和ctx.request.req同时保存
    createContext(req,res){
        let ctx=Object.create(context);
        ctx.request=Object.create(request);
        ctx.response=Object.create(response);

        ctx.req=ctx.request.req=req;
        ctx.res=ctx.response.res=res;
        return ctx;
    }
}

module.exports=KKB;
```

使用app.js

```
const KKB=require('./kkb3');
const app=new KKB();

const delay=()=>Promise.resolve(resolve=>setTimeout(()=>resolve(),2000))

app.use(async(ctx,next)=>{
    ctx.body='1';
    await next();
    ctx.body+='5';
})

app.use(async(ctx,next)=>{
    ctx.body+='2';
    await delay();
    await next();
    ctx.body+='4';
})

app.use(async(ctx,next)=>{
    ctx.body+='3'
})

app.listen(3000,()=>{
    console.log('监听端口3000')
})


```



Koa-compose的源码(https://github.com/koajs/compose/blob/master/index.js)



常见koa中间件的实现

- koa中间件的规范：
  - 一个async函数
  - 接收ctx和next两个参数
  - 任务结束需要执行next

```
const mid=async (ctx,next)=>{
    //来到中间件，洋葱圈左边
    next()//进入其他中间件
    //再次来到中间件，洋葱圈右边
}
```

- 中间件常见任务：
  - 请求拦截

    - 路由
    - 日志
    - 静态文件服务
- 路由router

  将来可能的用法

```
const Koa=require('./kkb3')
const Router=require('./router2');
const app=new Koa();
const router=new Router();

router.get('/index',async ctx=>{ctx.body='index page'});
router.get('/post',async ctx=>{ctx.body='post page'})
router.get('/list',async ctx=>{ctx.body='list page'})
router.post('/index',async ctx=>{ctx.body='post index page'})

//路由实例输出父中间件 router.routes
app.use(router.routes());
```

  routes()的返回值是一个中间件，由于需要用到method，所以需要挂载method到ctx之上

```
class Router{
    constructor(){
        this.stack=[];
    }

    register(path,methods,middleware){
        let route={path,methods,middleware};
        this.stack.push(route);
    }

    //现在只支持get和post，其他的同理
    get(path,middleware){
        this.register(path,'get',middleware)
    }
    post(path,middleware){
        this.register(path,'post',middleware)
    }

    routes(){
        let stock=this.stack;
        return async function(ctx,next){
            let currentPath=ctx.url;
            let route;
            for(let i=0;i<stock.length;i++){
                let item=stock[i];

 if(currentPath===item.path&&item.methods.indexOf(ctx.method)>=0){
                    //判断path和method
                    route=item.middleware;
                    break;
                }
            }

            if(typeof route==='function'){
                route(ctx,next);
```

```
                return;
            }
            await next();
        }
    }

}
module.exports=Router;
```

- 静态文件服务koa-static
  - 配置绝对资源目录地址，默认为static
  - 获取文件或者目录信息
  - 静态文件获取
  - 返回

```
const fs=require('fs');
const path=require('path');

module.exports=(dirPath='./public')=>{
    return async (ctx,next)=>{
        if(ctx.url.indexOf('/public')===0){
            //public开头，读取文件
            const url=path.resolve(__dirname,dirPath);
            const fileBaseName=path.basename(url);
            const filePath=url+ctx.url.replace('/public','');
            try {
                stats=fs.statSync(filePath);
                //
console.log(stats,fs.statSync(filePath).isDirectory());
                if(stats.isDirectory()){
                    const dir=fs.readdirSync(filePath);
                    const ret=['<div style="padding-left:20px">'];
                    console.log(ret,dir)
                    dir.forEach(filename=>{
                        console.log(filename,"+++++++++");
                        //简单认为不带小数点的格式，就是文件夹，实际应该用
stateSync
                        if(filename.indexOf(".")>-1){
                            ret.push(
                                `<p><a style="color:black"
href="${ctx.url}/${filename}">${filename}</a></p>`
                            )
                        }else{
                            //文件
                            ret.push(
                                `<p><a
href="${ctx.url}/${filename}">${filename}</a></p>`
                            )
                        }

                    })
                    ret.push("</div>");
                    console.log(ret,ret.join(""))
                        ctx.body=ret.join("");
                }else{
```

```
                    console.log("文件")
                    const content=fs.readFileSync(filePath);
                    ctx.body=content;
                }
            } catch (error) {
                //报错了 文件不存在
                ctx.body="404,not found"
            }
        }else {
            //否则不是静态资源，直接去下一个中间件
            await next();
        }
    }
}
```

使用

```
const static=require('./static');
app.use(static(__dirname+'/public'))
```

- 请求拦截：黑名单中存在的ip访问将被拒绝

```
//iptable.js
module.exports=async function(ctx,next){
    const {req,res}=ctx;
    const blackList=['127.0.0.1'];
    const ip=getClientIP(req);

    if(blackList.includes(ip)){
        //出现在黑名单终将被拒绝
        ctx.body="not allowed";
    }else {
        await next();
    }
}


function getClientIP(req){
    return (
        req.headers['x-forwarded-for']||//判断是否有反向代理IP
        req.connection.remoteAddress||//判断connection的远程IP
        req.socket.remoteAddress||//判断后端的socket的IP
        req.connection.socket.remoteAddress
    )
}

//app3.js
app.use(require('./iptable'));
app.listen(3000,'0.0.0.0',()=>{
    console.log("监听端口3000")
})
```

网络编程http https http2 websocket

TCP协议 - 实现一个即时通讯IM

- Socket实现

原理：Net模块提供一个异步API能够创建基于流的TCP服务器，客户端与服务器建立连接后，服务器可以获得一个全双工Socket对象，服务器可以保存Socket对象列表，在接收某客户端消息时，推送给其他客户端。

```
//socket.js
const net=require('net');
const chatServer=net.createServer();
const clinetList=[];
chatServer.on('connection',clinet=>{
  clinet.write('hi friend \n');
  clinetList.push(clinet);
  clinet.on('data',data=>{
    console.log('receive:',data.write.toString());
    clinetList.forEach(v=>{
      v.write(data)
    })
  })
})


chatServer.listen(3000)
```

通过Telnet连接服务器

```
telnet localhost 9000
```

http协议

```
//观察http协议
curl -v http://www.baidu.com
```

- http协议详解（http://typora-app/HTTP%E5%8D%8F%E8%AE%AE%E8%AF%A6%E8%A7%A3.md）
- 创建接口，api.js

```
const http=require("http");
const fs=require("fs");
http.createServer((req,res)=>{
    const {method,url}=req;
```

```
    if(method==='GET'&&url==='/'){
        fs.readFile('./index.html',(err,data)=>{
            res.setHeader('Content-Type',"text/html");
            res.end(data)
        })
    }else if(method==='GET'&&url==="/api/users"){
        res.setHeader('Contnet-Type',"application/json");
        res.end(JSON.stringify([{name:"tom",age:20}]))
    }
}).listen(3000);
```

- 请求接口

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <meta http-equiv="X-UA-Compatible" content="ie=edge">
    <title>Document</title>
</head>
<body>
    <script src="https://unpkg.com/axios@0.19.2/dist/axios.min.js"></script>
    <script>
     (async ()=>{
         const res=await axios.get("/api/users");
         console.log('data',res.data);
         document.writeln(`Response:${JSON.stringify(res.data)}`)
     })()
    </script>
</body>
</html>
```

- 埋点更容易

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <meta http-equiv="X-UA-Compatible" content="ie=edge">
    <title>Document</title>
</head>
<body>
    <script src="https://unpkg.com/axios@0.19.2/dist/axios.min.js"></script>
    <!-- <script>
     (async ()=>{
         const res=await axios.get("/api/users");
         console.log('data',res.data);

         document.writeln(`Response:${JSON.stringify(res.data)}`)
     })()
```

```
    </script> -->
    <script>
     const img=new Image();
         img.src="/api/users?abc=123"
    </script>
</body>
</html>
```

协议 端口 host

- 跨域：浏览器同源政策引起的接口调用问题

```
//api.js
//这个是后端
const http=require("http");
const fs=require("fs");
const api=  http.createServer((req,res)=>{
    const {method,url}=req;
    console.log(url,method)
    if(method==='GET'&&url==='/'){
        fs.readFile('./index.html',(err,data)=>{
            res.setHeader('Content-Type',"text/html");
            res.end(data)
        })
    }else if(method==='GET'&&url==="/api/users"){
        res.setHeader('Contnet-Type',"application/json");
        res.end(JSON.stringify([{name:"tom",age:20}]))
    }
})
// api.listen(4000)

module.exports=api;
```

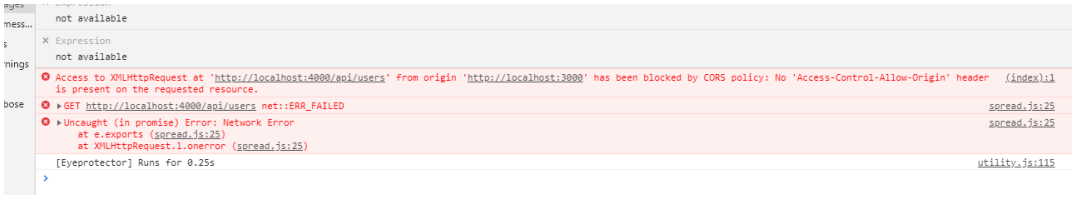
```
//proxy.js
const express=require('express');
const app=express();
//静态页面,前端监听的是3000端口
app.use(express.static(__dirname+'/'));
module.exports=app;
```

```
//index.js
const api=require('./api');
const proxy=require('./proxy');
api.listen(4000);
proxy.listen(3000)
```

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-
scale=1.0">
    <meta http-equiv="X-UA-Compatible" content="ie=edge">
    <title>Document</title>
</head>
```

```
<body>
    <script src="https://unpkg.com/axios@0.19.2/dist/axios.min.js">
</script>
    <script>
     (async ()=>{
         //通过baseURL方式
         axios.defaults.baseURL="http://localhost:4000"//请求后端的端口
         const res=await axios.get("/api/users");
         console.log('data',res.data);

         document.writeln(`Response:${JSON.stringify(res.data)}`)
     })()
    </script>
    <!-- <script>
     const img=new Image();
         img.src="/api/users?abc=123"
    </script> -->
</body>
</html>
```

- 浏览器抛出跨域错误



- 常用解决方案

1. JSONP(JSON with Padding)，前端+后端方案，绕过跨域

   前端构造script标签请求指定URL(由script标签发出的GET请求不受同源策略限制)，服务器返回一个函数执行语句，该函数名称通常由查询参callback的值决定，函数的参数为服务器返回的json数据。该函数在前端执行后即可获得数据。

```
<!DOCTYPE html>
<html>

<body>

<h2>Request With a Callback Function</h2>
<p>The PHP file returns a call to the function you send as a callback.
</p>

<button onclick="clickButton()">Click me!</button>

<p id="demo"></p>

<script>
function clickButton() {
  var s = document.createElement("script");
  s.src = "demo_jsonp2.php?callback=myDisplayFunction";
  document.body.appendChild(s);
}

function myDisplayFunction(myObj) {
  document.getElementById("demo").innerHTML = myObj.name;
```

```
}
</script>

</body>
</html>

```

2. 代理服务器

   请求同源服务器，通过该服务器转发请求至目标服务器，得到结果再转发给前端

   前端开发中测试服务器的代理功能就是采用的该解决方案，但是最终发布上线时如果web应用和接口服务器不在一起仍会跨域

3. CORS(Cross Origin Resource Share)-跨域资源共享，后端方案，解决跨域

预检请求(https://www.jianshu.com/p/b55086cbd9af)

> 原理：cors是w3c规范，真正意义上解决跨域问题。它需要服务器对请求进行检查并对响应头做相应处理，从而允许跨域请求

1. 简单请求

简单请求不会触发CORS预检请求，“简属于
单请求”术语并不属于Fetch(其中定义了CORS)规范。
若满足所有下述条件，则该请求可视为“简单请求”：

- 使用下列方法之一：
  - `GET`
  - `HEAD`
  - `POST`
    - `Content-Type`：(仅当POST方法的Content-Type值等于下列之一才算做简单需求)
      - `text/plain`
      - `multipart/form-data`
      - `application/x-www-form-urlencoded`

> **注意:** WebKit Nightly 和 Safari Technology Preview 为Accept
> , Accept-Language
> , 和 Content-Language
> 首部字段的值添加了额外的限制。如果这些首部字段的值是“非标准”的，WebKit/Safari 就不会将这些请求视为“简单请求”。WebKit/Safari 并没有在文档中列出哪些值是“非标准”的，不过我们可以在这里找到相关讨论：Require preflight for non-standard CORS-safelisted request headers Accept, Accept-Language, and Content-Language, Allow commas in Accept, Accept-Language, and Content-Language request headers for simple CORS, and Switch to a blacklist model for restricted Accept headers in simple CORS requests。其它浏览器并不支持这些额外的限制，因为它们不属于规范的一部分。

具体实现：

- 响应简单请求：动词为get/post/head,没有自定义请求头，Content-Type是application/x-www-form-urlencoded,multipart/form-data或text/plain之一，通过添加以下响应头解决：

```
//api.js
const http=require("http");
const fs=require("fs");
const api=  http.createServer((req,res)=>{
    const {method,url}=req;
    console.log(url,method)
    if(method==='GET'&&url==='/'){
        fs.readFile('./index.html',(err,data)=>{
            res.setHeader('Content-Type',"text/html");
            res.end(data)
        })
    }else if(method==='GET'&&url==="/api/users"){
        res.setHeader('Contnet-Type',"application/json");
        res.setHeader('Access-Control-Allow-Origin','http://localhost:3000')//访问控制允许源
        res.end(JSON.stringify([{name:"tom",age:20}]))
    }

})
// api.listen(3000)

module.exports=api;
```

该案例中可以通过添加自定义的x-token请求头使请求变为preflight请求

```
//index.html
 axios.defaults.baseURL="http://localhost:4000"
 const res=await axios.get("/api/users",{
           headers:{'X-Token':'zyl'}
  });
```

- 响应preflight请求，需要响应浏览器发出的options请求(预检请求)，并根据情况设置响应头：

```
else if(method==='OPTIONS'&&url==='/api/users'){
        res.writeHead(200,{
            'Access-Control-Allow-Origin':'http://localhost:3000',
            "Access-Control-Allow-Headers":'X-Token,Content-Type',
            'Access-Control-Allow-Methods':'PUT'
        })
        res.end();
    }
```

此时服务器需要允许x-token.

如果请求为post,还传递了参数：

```
//index.html
 const res=await axios.post("/api/users",{foo:'bar'},{
            headers:{'X-Token':'zyl'}
});

//index.js
else if((method==='GET'||method==='POST')&&url==="/api/users"){
        res.setHeader('Contnet-Type',"application/json");
        res.setHeader('Access-Control-Allow-
Origin','http://localhost:3000')
        res.end(JSON.stringify([{name:"tom",age:20}]))
}
```

则服务器还需要允许Content-Type请求头

- 如果要携带cookie信息，则请求变为credential请求:

```
//index.js
//预检options中和/api/users接口中均需添加
res.setHeader('Access-Control-Allow-Credentials','true')
//设置cookie
res.setHeader('Set-Cookie','cookie1=zyl12')


//index.html
//ajax服务
axios.defaults.withCredentials=true;

```

application/x-www-from-urlencoded,会将表单内的数据转换为键值对，比如,name=java&age = 23。Content-Type是application/json时，客户端可以通过body发送json字符串；如果是application/xml时可以发送xml字符串。

Proxy代理模式

```
//proxy.js
const express=require('express');
const {createProxyMiddleware}=require('http-proxy-middleware')
const app=express();
//静态页面
app.use(express.static(__dirname+'/'));
app.use('/api',createProxyMiddleware({target:'http://localhost:4000',changeOr
igin:false}))
module.exports=app;
```

```
//index.html
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <meta http-equiv="X-UA-Compatible" content="ie=edge">
    <title>Document</title>
</head>
<body>
```

```
    <script src="https://unpkg.com/axios@0.19.2/dist/axios.min.js"></script>
    <script>
     (async ()=>{

         //ajax服务
         axios.defaults.withCredentials=true;
         //通过baseURL方式
         //axios.defaults.baseURL="http://localhost:4000"
         const res=await axios.post("/api/users",{foo:'bar'},{
            headers:{'X-Token':'zyl'}
         });
        //  const res=await axios.get("/api/users",{foo:'bar'},{
        //     headers:{'X-Token':'zyl'}
        //  });
         console.log('data',res.data);

         document.writeln(`Response:${JSON.stringify(res.data)}`)
     })()
    </script>
    <!-- <script>
     const img=new Image();
         img.src="/api/users?abc=123"
    </script> -->
</body>
</html>
```

对比一下nginx与webpack devserver

```
module.exports={
    devServer:{
        disableHostCheck:true,
        compress:true,
        port:5000,
        proxy:{
            '/api':{
                target:'http://localhost:4000',
                changeOrigin:true,
            }
        }
    }
}
```

nginx

```
server{
    listen 80;
    #server_name www.josephxia.com;
    location / {
        root /var/www/html
        index index.html index.htm;
        try_files $uri $uri/ /index.html
    }
    location /api {
        proxy_pass http://127.0.0.1:3000;
        proxy_redirect off;
        proxy_set_header Host $host;
```

```
        proxy_set_header X-Real-IP $remote_addr;
        proxy_set_header X-Forward-For $proxy_add_x_forwarded_for;
    }
}
```

BodyParser

- application/x-www-form-urlencoded

```
<form action="/api/save" method="post">
        <input type="text" name="abc" value="123">
        <input type="submit" value="save">
    </form>
```

```
//api.js
else if(method==="POST"&&url==="/api/save"){
        let reqData=[];
        let size=0;
        req.on('data',data=>{
            console.log('>>>req on',data);
            reqData.push(data);
            size+=data.lenght;
        })
        req.on('end',function(){
            console.log('end');
            const data=Buffer.concat(reqData,size);
            console.log('data:',size,data.toString());
            res.end(`formdata:${data.toString()}`)
        })

    }
```

- application/json

```
await axios.post('/api/save',{
    a:1,
    b:2
})
```

```
//模拟application/x-www-form-urlencoded
    await axios.post('/api/save','a=1&b=3',{
        headers:{
            'Content-Type':'application/x-www-form-urlencoded',
        }
    })
```

上传文件

```
//Stream pipe
request.pipe(fis)
response.end();
```

```
        const {pathname}=require('url').parse(req.url);
```

```
        const fileName=req.headers['file-name']?req.headers['file-name']:null
        const outputFile=path.resolve(__dirname,fileName);
       //Buffer connect
        let reqData=[];
        let size=0;
        req.on('data',data=>{
            console.log('>>>req on',data);
            reqData.push(data);
            size+=data.length;
            console.log(data.length)
        })
        req.on('end',function(){
            console.log('end....');
            const data=Buffer.concat(reqData,size);
            console.log('data:',size,data.toString());
             fs.writeFileSync(outputFile,data)
            res.end(`formdata:${data.toString()}`)
        })
```

```
       //流事件写入
        const fis=fs.createWriteStream(outputFile);
        req.on('data',data=>{
            console.log('data:',data);
            fis.write(data);
        })
        req.on('end',()=>{
            fis.end();
            res.end();
        })
```

实战一个爬虫

原理：服务端模拟客户端发送请求到目标服务器获取页面内容并解析，获取其中关注部分的数据

```
//spider.js
const originRequest=require('request');
const cheerio=require('cheerio');
const iconv=require('iconv-lite');

function request(url,callback){
    const options={
        url,
        encoding:null
    };

    originRequest(url,options,callback);
}


for(let i=100553;i<100563;i++){
    const url=`https://www.dy2018.com/i/${i}.html`;
    request(url,(err,res,body)=>{
        const html=iconv.decode(body,'gb2312');
        const $=cheerio.load(html);
        console.log($(".title_all h1").text());
```

```
    })
}
```

实现一个即时通讯IM

- socket实现

  原理：Net模块提供一个异步API能创建基于流的TCP服务器，客户端与服务器建立连接后，服务器可以获得一个全双工socket对象，服务器可以保存socket对象列表，在接收某客户端消息时，推送给其他客户端。

```
//socket.js
const net=require('net');
const chatServer=net.createServer();
const clientList=[];
chatServer.on('connection',client=>{
    client.write('hi\n');
    clientList.push(client);
    client.on('data',data=>{
        console.log('receive:',data.toString());
        clientList.forEach(v=>{
            v.write(data);
        })
    })
})

chatServer.listen(9000)
```

  通过Telnet连接服务器

```
telnet localhost 9000
```

- Http实现

  原理：客户端通过ajax方式发送数据给http服务器，服务器缓存信息，其他客户通过轮询方式查询最新数据更新列表。

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <meta http-equiv="X-UA-Compatible" content="ie=edge">
    <title>Document</title>
</head>
<body>
    <script src="https://cdn.jsdelivr.net/npm/vue/dist/vue.js"></script>
    <script src="https://unpkg.com/axios/dist/axios.min.js"></script>
    <div id="app">
            <input v-model="message">
            <button v-on:click="send">发送</button>
            <button v-on:click="clear">清空</button>
            <div v-for="item in list">{{item}}</div>
    </div>
```

```
    <script>
        const host="http://localhost:3000";
        let app=new Vue({
            el:"#app",
            data:{
                list:[],
                message:'Hello Vue!'
            },
            methods: {
               async  send() {
                    let res=await axios.post(host+'/send',{
                        message:this.message
                    })
                    this.list=res.data;
                },
                async clear(){
                    let res=await axios.post(host+'/clear');
                    this.list=res.data;
                }
            },
            mounted() {
                setInterval(async ()=>{
                        const res=await axios.get(host+'/list');
                        this.list=res.data;
                },1000)
            },
        })
    </script>
</body>
</html>
```

```
const express=require('express');
const path=require('path');
const app=express();
const bodyParser=require('body-parser');

app.use(bodyParser.json());

const list=['ccc','ddd'];

app.get('/',(req,res)=>{
    res.sendFile(path.resolve('./index.html'))
})

app.get('/list',(req,res)=>{
    res.end(JSON.stringify(list))
})

app.post('/send',(req,res)=>{
    list.push(req.body.message);
    res.end(JSON.stringify(list))
})

app.post('/clear',(req,res)=>{
    list.length=0;
    res.end(JSON.stringify(list))
```

```
})

app.listen(3000)
```

- socket.io实现
  - 安装：npm i socket.io -S
  - 两部分：nodejs模块，客户端js

```
//服务端: chat-socketio.js
const app=require('express')();
const http=require('http').Server(app);
const io=require('socket.io')(http);

app.get('/',(req,res)=>{
    res.sendFile(__dirname+'/index.html')
})

io.on('connection',(socket)=>{
    console.log('a user connected');

    //响应某用户发送信息
    socket.on('chat message',(msg)=>{
        console.log('chat message:'+msg);
        //广播给所有人
        io.emit('chat message',msg);
        //广播给除了发送者外所有人
        // socket.broadcast.emit('chat message',msg)
    })

    socket.on('disconnect',()=>{
        console.log('user disconnected')
    })
})

http.listen(3000,()=>{
    cobnsole.log('listening on *:3000')
})
```

```
//客户端: index.html
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-
scale=1.0">
    <meta http-equiv="X-UA-Compatible" content="ie=edge">
    <title>Document</title>
</head>
<style>
 *{
     margin: 0;
     padding: 0;
     box-sizing: border-box;
 }
 body{
```

```
    font-size: 13px Helvetica,Arial;
}

form{
    background: #000;
    padding: 3px;
    position: fixed;
    bottom: 0;
    width:100%;
}

 form input{
    border: 0;
    padding: 10px;
    width:90%;
    margin-right:0.5%;
 }

 form button{
    width:9%;
    background: rgb(130,224,255);
    border: none;
    padding:10px;
 }

 #messages {
    list-style-type:none;
    margin: 0;
    padding:0;
 }
 #messages li{
    padding:5px 10px;
 }

 #messages li:nth-child(odd){
    background: #eee;
 }

</style>
<body>
    <ul id="messsges"></ul>
    <form>
        <input id='m' autocomplete="off">
        <button>send</button>
    </form>
    <script src="https://cdnjs.cloudflare.com/ajax/libs/socket.io/2.2.0/socket.io.js"></script>
    <script src="http://libs.baidu.com/jquery/2.1.1/jquery.min.js"></script>
    <script>
    $(function(){
        const socket=io();
        $("form").submit(e=>{
            e.preventDefault();
            socket.emit("chat message",$("#m").val());
            $("#m").val("");
            return false
```

```
        })

        socket.on("chat message",msg=>{
            $("#messages").append($("<li>").text(msg))
        })
    })
    </script>
</body>
</html>
```

socket.io库特点

- 源于HTML5标准
- 支持优雅降级
    - WebSocket
    - WebSocket over Flash
    - XHR Polling
    - XHR Multipart Streaming
    - Forever Iframe
    - JSONP Polling

Https

- 创建证书

```
#创建私钥
openssl genrsa -out privatekey.pem 1024
#创建证书签名请求
openssl req -new -key privatekey.pem -out certrequest.csr
#获取证书,线上证书需要经过证书授权中心签名的文件；下面只创建一个学习使用证书
openssl x509 -req -in certrequest.csr -signkey privatekey.pem -out
certificate.pem
#创建pfx文件
openssl pkcs12 -export -in certificate.pem -inkey privatekey.pem -out
certificate.pfx

```

Http2

- 多路复用-雪碧图、多域名CDN、接口合并
    - 官方演示-https://http2.akamai.com/demo
    - 多路复用允许同时通过单一的HTTP/2连接发起多重的请求-响应消息；而HTTP/1.1协议中，浏览器客户端在同一时间，针对同一域名下的请求有一定数量限制。超过限制数目的请求会被阻塞
    - 首部压缩
        - http/1.x的header由于cookie和user agent很容易膨胀，而且每次都要重复发送。http/2使用encoder来减少需要传输的header大小，通讯双方各自cache一份header fields表，既避免了重复header的传输，又减少了需要传输的大小。高效的压缩算法可以很大的压缩header，减少发送包的数量从而降低延迟.
    - 服务端推送
        - 在HTTP/2中，服务器可以对客户端的一个请求发送多个响应。举个例子，如果一个请求请求的是index.html,服务器很可能会同时响应index.html、logo.jpg以及css和js文件，

因为它知道客户端会用到这些东西。这相当于在一个HTML文档内集合了所有的资源

HTTP与HTTPS的区别

HTTPS和HTTP的区别主要如下：

1、https协议需要到ca申请证书，一般免费证书较少，因而需要一定费用。

2、http是超文本传输协议，信息是明文传输，https则是具有安全性的ssl加密传输协议。

3、http和https使用的是完全不同的连接方式，用的端口也不一样，前者是80，后者是443。

4、http的连接很简单，是无状态的；HTTPS协议是由SSL+HTTP协议构建的可进行加密传输、身份认证的网络协议，比http协议安全。

数据持久化-mysql

资源

- mysql相关
  - MySQL:下载（https://dev.mysql.com/downloads/）
  - node驱动：文档(https://github.com/mysqljs/mysql)
  - Sequelize:文档（https://github.com/sequelize/sequelize）、api(https://sequelize.org/)
- mongodb相关：
  - MongoDB：下载(https://www.mongodb.com/download-center/community)
  - node驱动：文档(https://github.com/mongodb/node-mongodb-native)
  - mongoose:文档(https://mongoosejs.com/docs/guide.html)
- redis相关:
  - redis:下载(https://redis.io/download)
  - node_redis:文档(https://github.com/NodeRedis/node-redis)

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

```
//fs/index.js
const fs=require('fs');
function get(key){
    fs.readFile('./db.json',(err,data)=>{
        const json=JSON.parse(data);
        console.log(json);
    })
}

function set(key,value){
    fs.readFile('./db.json',(err,data)=>{
        //判断是否为空
        const json=data?JSON.parse(data):{};
```

```
        json[key]=value;
        //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
const readline=require('readline');
const r1=readline.createInterface({
    input:process.stdin,
    output:process.stdout
})

r1.on('line',function(input){
    const [op,kay,value]=input.split(' ');
    if(op==='get'){
        get(key)
    }else if(op==='set'){
        set(key,value)
    }else if(op==='quit'){
        r1.close();
    }else{
        console.log('没有该操作')
    }
})

r1.on('close',()=>{
    console.log('程序结束');
    process.exit(0);
})
```

Mysql安装和配置

mac

windows(http://typora-app/doc/mysql%E5%AE%89%E8%A3%85%E6%96%87%E6%A1%A3_windows.md)

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

node.js原生驱动

- 安装mysql模块:npm i mysql -S
- mysql模块基本使用

```
//mysql.js
const mysql = require('mysql');
//连接配置
const cfg = {
    host: 'localhost',
    user: 'root',
    password: 'yw@910714',//yw@910714
```

```
    database: 'zyl-kaikeba'
}

//创建连接对象
const conn = mysql.createConnection(cfg);

//连接
conn.connect(err => {
    if (err) {
        console.log(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 () => {
    const mysql = require('mysql2/promise');
    //连接设置
    const cfg = {
        host: 'localhost',
        user: 'root',
        password: 'yw@910714',//yw@910714
```

```
        database: 'zyl-kaikeba'
    }

    //创建连接
    const connection = await mysql.createConnection(cfg);

    //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`;

    //查询数据库
    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(https://sequelize.org/master/manual/index.html)

- 概述：基于Promise的ORM（Object Relation Mapping）,支持多种数据库、事务、关联等
- 安装：npm i sequelize mysql2 -S
- 基本使用

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

    //建立连接
    const sequelize=new Sequelize('zyl-kaikeba','root','yw@910714',{
        host:'localhost',
        dialect:'mysql',
        operatorsAliases:false
    })

    //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:'芒果',
        price:5
    })

    console.log('create',ret);

    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,[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或者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.UUID,
           primaryKey:true,
       },
```

- Getters&Setters: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
//定义模型
    const Fruit=sequelize.define('Fruit',{
        id:{
            type:Sequelize.DataTypes.UUID,
            defaultValue:Sequelize.DataTypes.UUID,
            primaryKey:true,
        },
```

```
        name:{type:Sequelize.STRING(20),
            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)`
            }
        },
        price:{type:Sequelize.FLOAT,allowNull:false},
        stock:{type:Sequelize.INTEGER,defaultValue:0}
    },{
        // tableName:'TBL_FRUIT'
        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)
            }
        }
    })

```

- 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:{type:Sequelize.FLOAT,allowNull:false,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:{type:Sequelize.INTEGER,defaultValue:0,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

```
    Fruit.findAll().then(fruits=>{
        console.log(JSON.stringify(fruits));
        //修改amount,触发setterMehods
        fruits[0].amount='150kg';
        fruits[0].save();
    })

    Fruit.classify=(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=(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)));
```

- 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);
    })


    //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)
   })
```

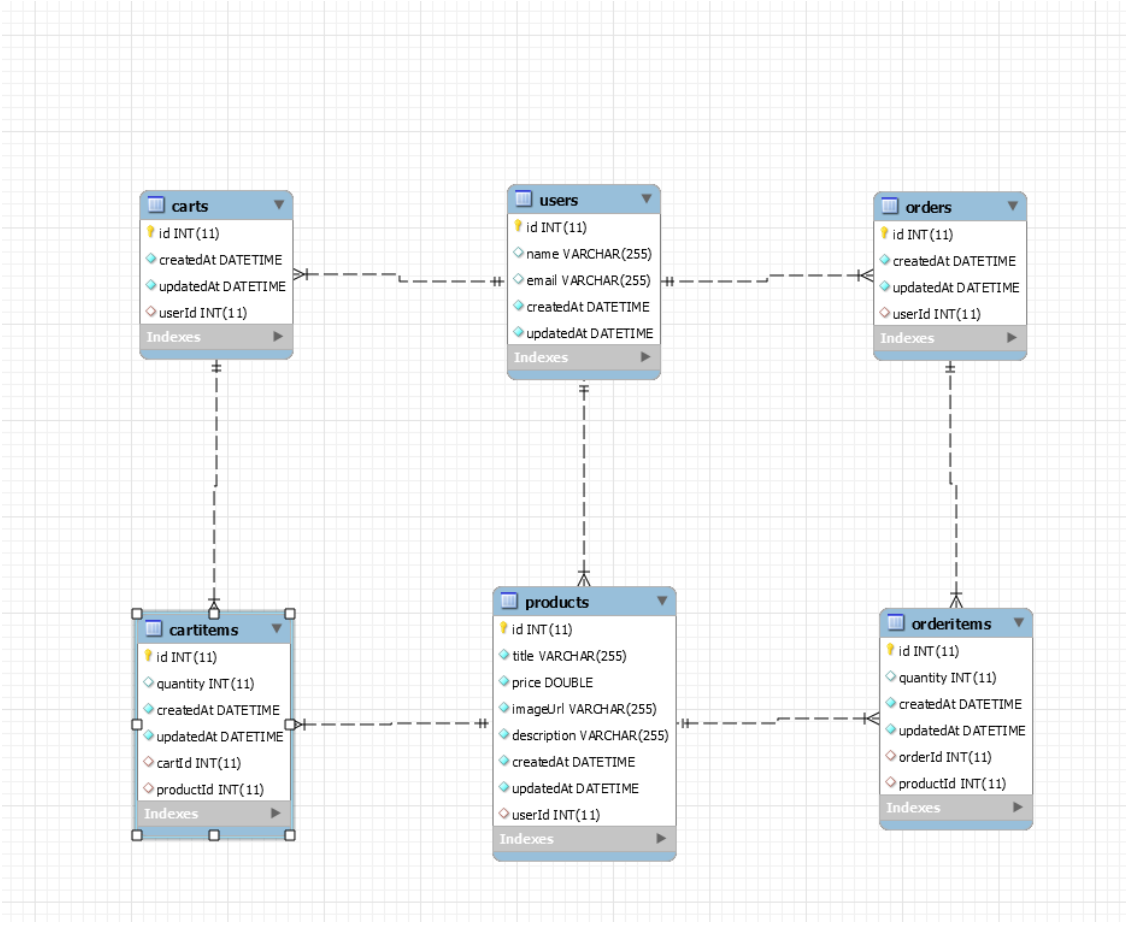
- 更新

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

- 删除

```
//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))
```

实例关系图与域模型ERD




```
const Koa = require('koa')
const app = new Koa()
const bodyParser = require('koa-bodyparser')
app.use(require('koa-static')(__dirname + '/'))
app.use(bodyParser())

// 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');


// 加载用户 - 代替鉴权
app.use(async (ctx, next) => {
    const user = await User.findByPk(1)
    ctx.user = user;
    await next();
});

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

router.post('/admin/product', async ctx => {
    const body = ctx.request.body
    const res = await ctx.user.createProduct(body)
    ctx.body = { success: true }
})

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

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.body
    console.log('ctx.body', ctx.request.body)
    const prodId = body.id;
    let fetchedCart;
    let newQty = 1;

    // 获取购物车
    const cart = await ctx.user.getCart()
    console.log('cart', cart)
    fetchedCart = cart;
    const products = await cart.getProducts({
        where: {
```

```
            id: prodId
        }
    });

    let product;
    // 判断购物车数量
    if (products.length > 0) {
        product = products[0];
    }
    if (product) {
        const oldQty = product.cartItem.quantity;
        newQty = oldQty + 1;
        console.log("newQty", newQty);
    } else {
        product = await Product.findByPk(prodId);
    }

    await fetchedCart.addProduct(product, {
        through: {
            quantity: newQty
        }
    });
    ctx.body = { success: true }
})

router.post('/orders', async ctx => {
    let fetchedCart;
    const cart = await ctx.user.getCart();
    fetchedCart = cart;
    const products = await cart.getProducts();
    const order = await ctx.user.createOrder();
    const result = await order.addProducts(
        products.map(p => {
            p.orderItem = {
                quantity: p.cartItem.quantity
            };
            return p;
        })
    );
    await fetchedCart.setProducts(null);
    ctx.body = { success: true }
})
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 }
})
router.get('/orders', async ctx => {
    const orders = await ctx.user.getOrders({ include: ['products'], order:
[['id', 'DESC']] })
    ctx.body = { orders }
})

```

```
125
126 app.use(router.routes())
127
128 // app.use('/admin', adminRoutes.routes);
129 // app.use(shopRoutes);
130
131
132 Product.belongsTo(User, {
133     constraints: true,
134     onDelete: 'CASCADE'
135 });
136 User.hasMany(Product);
137 User.hasOne(Cart);
138 Cart.belongsTo(User);
139 Cart.belongsToMany(Product, {
140     through: CartItem
141 });
142 Product.belongsToMany(Cart, {
143     through: CartItem
144 });
145 Order.belongsTo(User);
146 User.hasMany(Order);
147 Order.belongsToMany(Product, {
148     through: OrderItem
149 });
150 Product.belongsToMany(Order, {
151     through: OrderItem
152 });
153
154 sequelize.sync().then(
155     async result => {
156         let user = await User.findByPk(1)
157         if (!user) {
158             user = await User.create({
159                 name: 'Sourav',
160                 email: 'sourav.dey9@gmail.com'
161             })
162             await user.createCart();
163         }
164         app.listen(3000, () => console.log("Listening to port 3000"));
165     })
166
```

Restfull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

数据持久化-mongodb

资源

- mongodb相关：
  - MongoDB：下载(https://www.mongodb.com/download-center/community)
  - node驱动：文档(https://github.com/mongodb/node-mongodb-native)
  - mongoose:文档(https://mongoosejs.com/docs/guide.html)

- redis相关:
  - redis:下载(https://redis.io/download)
  - node_redis:文档(https://github.com/NodeRedis/node-redis)
- 可视化工具:Robo3T(https://robomongo.org/)

mongodb安装、配置

- 下载安装（https://www.runoob.com/mongodb/mongodb-window-install.html）
- 配置环境变量
- 创建dbpath文件夹
- 启动:

```
mongo
//默认连接
```

- 测试

```
//查询所有db数据库
show dbs

//切换/创建数据库，当创建一个集合(table)的时候会自动创建当前数据库
use test

//插入一条数据库
db.fruits.save({name:'apple',price:5})

//条件查询
db.fruits.find({price:5});

//得到当前db的所有聚集集合
db.getCollectionNames();

//查询
db.fruits.find()
```

mongo命令行操作(https://docs.mongodb.com/manual/reference/method/)

参考资料

菜鸟教程（https://www.runoob.com/mongodb/mongodb-create-database.html）

官网：https://docs.mongodb.com/manual/reference/method/

mongodb原生驱动

http://mongodb.github.io/node-mongodb-native/3.1/quick-start/quick-start/

官网API

- 安装mongodb模块：npm i mongodb -S
- 连接mongodb

```
//helloword.js
(async ()=>{
    const {MongoClient:MongoDB}=require('mongodb');
```

```
    //创建客户端
    const client=new MongoDB('mongodb://localhost:27017',
            {
                useNewUrlParser:true
            }
    );


    let ret;
    //创建连接
    ret=await client.connect();
    // console.log('ret',ret)


    const db=client.db('test');
    const fruits=db.collection('fruits');

    //添加文档
    ret=await fruits.insertOne({
        name:'芒果'
    })
    console.log('插入成功',JSON.stringify(ret))

    //查询文档
    ret=await fruits.findOne();
    console.log('查询文档',ret);

    //更新文档
    //更新的操作符$set
    ret=await fruits.updateOne({name:'芒果'},{
        $set:{name:'apple'}
    })

    console.log('更新文档',JSON.stringify(ret.result))

    //删除文档
    ret=await fruits.deleteOne({name:'apple'});
    await fruits.deleteMany();
    client.close();

})()
```

- 案例：瓜果超市
  - 提取数据库配置，./models/conf.js

```
//models/conf.js
module.exports={
    url:'mongodb://localhost:27017',
    dbName:'test'
}
```

  - 封装数据库连接，./models/db.js

```
const conf=require('./conf');
const EventEmitter=require('events').EventEmitter;
```

```
3
4  //客户端
5  const MongoClient=require('mongodb').MongoClient;
6
7  class Mongodb{
8      constructor(conf){
9          //保存conf
10         this.conf=conf;
11
12         this.emiter=new EventEmitter();
13         //连接
14         this.client=new MongoClient(conf.url,
   {useNewUrlParser:true});
15         this.client.connect(err=>{
16             if(err) throw err;
17             console.log('连接成功');
18             this.emiter.emit('connect');
19         })
20     }
21
22     col(colName,dbName=conf.dbName){
23         return this.client.db(dbName).collection(colName)
24     }
25
26     once(event,cb){
27         this.emiter.once(event,cb)
28     }
29 }
30
31 module.exports=new Mongodb(conf)
```

- eventEmitter

```
1  //eventEmitter.js
2  const EventEmitter=require('events').EventEmitter;
3  const event=new EventEmitter();
4  event.on('some_event',num=>{
5      console.log('some_event事件触发: '+num);
6  })
7  let num=0;
8  setInterval(()=>{
9      event.emit('some_event',num++)
10 },1000)
```

- 添加测试数据，./initData.js

```
1  const  mongodb=require('./models/db');
2  mongodb.once('connect',async()=>{
3      const col=mongodb.col('fruits');
4      //删除已存在
5      await col.deleteMany();
6
7      const data=new Array(100).fill().map((v,i)=>{
8          return {name:'xxx'+i,price:i,category:Math.random()>0.5?'蔬
   菜':'水果'}
9      })
```

```
    //插入
    await col.insertMany(data);
    console.log('插入测试数据成功')
})
```

```
//index.html
<!DOCTYPE html>
<html lang="en">

<head>
    <meta charset="UTF-8" />
    <meta name="viewport" content="width=device-width, initial-scale=1.0" />
    <meta http-equiv="X-UA-Compatible" content="ie=edge" />

    <script src="https://cdn.jsdelivr.net/npm/vue/dist/vue.js"></script>
    <script src="https://unpkg.com/element-ui/lib/index.js"></script>
    <script src="https://unpkg.com/axios/dist/axios.min.js"></script>
    <link rel="stylesheet" href="https://unpkg.com/element-ui/lib/theme-chalk/index.css" />
    <title>瓜果超市</title>
</head>

<body>
    <div id="app">
        <el-input placeholder="请输入内容" v-model="search" class="input-with-select"  @change="changeHandler">
            <el-button slot="append" icon="el-icon-search"></el-button>
        </el-input>
        <el-radio-group v-model="category" @change="getData">
            <el-radio-button v-for="v in categorys" :label="v" :key="v">{{v}}</el-radio-button>
        </el-radio-group>
        <el-table :data="fruits" style="width: 100%">
            <el-table-column prop="name" label="名称" width="180">
            </el-table-column>
            <el-table-column prop="price" label="价格" width="180">
            </el-table-column>
            <el-table-column prop="category" label="种类">
            </el-table-column>
        </el-table>
        <el-pagination layout="prev, pager, next" @current-change="currentChange" :total="total">
        </el-pagination>
    </div>
    <script>
        var app = new Vue({
            el: "#app",
            data: {
                page: 1,
                total: 0,
                fruits: [],
```

```
                categorys: [],
                category: [],
                search:''
            },
            created() {
                this.getData()

                this.getCategory()
            },
            methods: {
                async currentChange(page) {
                    this.page = page;
                    await this.getData()
                },
                async changeHandler(val){
                    console.log('search...',val)
                    this.search = val
                    await this.getData()
                },
                async getData() {
                    const res = await axios.get(`/api/list?page=${this.page}&category=${this.category}&keyword=${this.search}`)
                    const data = res.data.data
                    this.fruits = data.fruits
                    this.total = data.pagination.total
                },
                async getCategory() {
                    const res = await axios.get(`/api/category`)
                    this.categorys = res.data.data
                    console.log('category', this.categorys)
                }
            }
        });
    </script>
</body>

</html>
```

- 接口编写,index.js

```
const express = require('express');
const app = express();
const path = require('path');
const mongo = require('./models/db');

app.get('/', (req, res) => {
    res.sendFile(path.resolve('./index.html'))
})

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

    //分页查询
    const { page } = req.query;
    try {
        const col = mongo.col('fruits');
        const total = await col.find().count();
```

```
        const fruits = await col
            .find()
            .skip((page - 1) * 5)
            .limit(5)
            .toArray();

        res.json({
            ok: 1,
            data: {
                fruits,
                pagination:
                {
                    total, page
                }
            }
        })
    } catch (error) {
        console.log(err)
    }
})

app.listen(3000)
```

- 添加类别搜索功能

```
const express = require('express');
const app = express();
const path = require('path');
const mongo = require('./models/db');

app.get('/', (req, res) => {
    res.sendFile(path.resolve('./index.html'))
})

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

    //分页查询
    const { page,category,keyword } = req.query;
    //构造条件
    const condition={};
    if(category){
        condition.category=category;
    }

    if(keyword){
        condition.name={$regex:new RegExp(keyword)}
    }

    try {
        const col = mongo.col('fruits');
        const total = await col.find().count();
        const fruits = await col
            .find(condition)
            .skip((page - 1) * 5)
            .limit(5)
            .toArray();

```

```
        res.json({
            ok: 1,
            data: {
                fruits,
                pagination:
                {
                    total, page
                }
            }
        })
    } catch (error) {
        console.log(err)
    }
})


app.get('/api/category',async (req,res)=>{
    const col=mongo.col('fruits');
    const data=await col.distinct('category');
    res.json({ok:1,data})
})

app.listen(3000)
```

- 操作符

  操作符文档: https://docs.mongodb.com/manual/reference/operator/query/

  - 查询操作符:提供多种方式定位数据库数据

```
//比较$eq,$gt,$gte,$in等
const col=mongo.col('fruits');
    await col.find({price:{$gt:10}}).toArray();

    //逻辑$and,$not,$nor,$or
    //price>0或者price<5
    await col.find({$or:[{price:{$gt:10}},{price:{$lt:10}}]})
    //price>0且price<5
    await col.find({$nor:[{price:{$gt:10}},{price:{$lt:10}}]})

    //元素$exists,$type
    await col.insertOne({name:'芒果',price:20,stack:true})
    await col.find({stack:{$exists:true}})

   //模拟$regex,$text,$expr
   await col.find({name:{$regex:/芒/}})
   await col.createIndex({name:'text'})//验证文本搜索需首先对字段加索引
   await col.find({$text:{$search:'芒果'}})//按词搜索，单独字查询不出结果

   //数组$all,#elementMatch,$size
   col.insertOne({tags:['热带','甜']})//插入带标签数据
   // $all:查询指定字段包含所有指定内容的文档
   await col.find({tags:{$all:['热带','甜']}})
   //$elemMatch:指定字段数组中至少有一个满足所有查询规则
   col.insertOne({hisPrice:[20,25,30]})//数据准备
   col.find({hisPrice:{$elemMatch:{$gt:24,$lt:26}}})//历史价位有没有出现在24-26之间

```

```
    //地理空间$geoIntersects,$geowithin,$near,$nearSphere
    //创建stations集合
    const stations=db.collection('stations');
    //添加测试数组，执行一次即可
      await stations.insertMany([
          {name:'天安门东',loc:[116.407851,39.911408]},
          {name:'天安门西',loc:[116.398056,39.913723]},
          {name:'王府井',loc:[116.417809,39.91435]}
      ])

      await stations.createIndex({loc:'2dsphere'})

      r=await stations.find({
          loc:{
              $nearSphere:{
                  $geometry:{
                      type:'Point',
                      coordinates:[116.403847,39.915526]
                  },
                  $maxDistance:1000
              }
          }
      }).toArray();
      console.log('天安门附近地铁站',r)

```

- 更新操作符:可以修改数据库数据或添加附加数据

```
/字段相关:$set,$unset,$setOnInsert,$rename,$inc,$min,$max,$mul
    //更新多个字段
    await col.updateOne(
        {name:'芒果'},
        {$set:{price:19.8,category:'热带水果'}}
    )
    //更新内嵌字段
    // {$set:{...,area:{city:'三亚'}}}

    //数组相关:$,$[],$addToSet,$pull,$pop,$push,$pullAll
    //$push用于新增
     insertOne({tags:['热带','甜']})//添加tags数组字段
     col.updateMany({name:'芒果'},{$push:{tags:'上火'}})
     //$push,$pushAll用于删除符合条件项，$pop删除首项-1或尾项1
     col.updateMany({name:'芒果'},{$push:{tags:1}})
     col.updateMany({name:'芒果'},{$pop:{tags:1}})

     //$,$[]用于修改
     col.updateMany({name:'芒果',tags:'甜'},{$set:{'tags.$':'香甜'}})

     //修改器，常结合数组操作符使用：$each,$position,$slice,$sort
    //  $push:{tags:{$each:['上火','真香'],$slice:-3}}
```

- 聚合操作符：使用aggregate方法，使文档顺序通过管道阶段从而得到最终结果

```
//聚合管道阶段:$group,$count,$sort,$skip,$limit,$project等
    //分页查询
    r=await col.aggregate([{$sort:{price:-1}},{$skip:0},
{$limit:2}]).toArray();
    //投射：只选择name,price并排除_id
    col.aggregate([{$sort:{price:-1}},{$project:{
        name:1,price:1,_id:0
    }}]).toArray();

    //聚合管道操作：$add,$avg,$sum等
    //按name字段分组，统计组内price总和
    col.aggregate([{$group:{_id:'$name',total:
       {$sum:'price'}}}]).toArray();
```

ODM-Mongoose

- 安装：npm i mongoose -S
- 基本使用：

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

//连接
mongoose.connect('mongodb://localhost:27017/test',{
    useNewUrlParser:true
})

const conn=mongoose.connection;

conn.on('error',()=>{
    console.error('连接数据库失败')
})

conn.once('open',async ()=>{
    //定义一个Schema
    const Schema=mongoose.Schema({
        category:String,
        name:String
    })


    //编译模型
    const Model=mongoose.model('fruit',Schema);

    try {
       //创建数据
    let r=await Model.create({
        category:'温带水果',
        name:'苹果'
    })

    console.log('插入数据',r)

    //查询，find返回Query,实现了then和catch,可以当Promise使用
    //如果需要返回Promise,调用其exec()
    r=await Model.find({name:'苹果'})
    console.log('查询结果',ret);
```

```
38
39      //更新updateOne返回Query
40      r=await Model.updateOne({
41              name:'苹果'
42      },{$set:{name:'芒果'}})
43      console.log('更新结果',r)
44
45      //删除,deleteOne返回Query
46      r=await Model.deleteOne({name:'苹果'});
47      console.log('删除结果',r)
48      } catch (error) {
49         console.log(error)
50      }
51
52
53  })
```

- Schema
  - 字段定义

```
1   //定义一个Schema
2       const Schema = mongoose.Schema({
3           title: { type: String, required: [true, '标题为必填项'] },//定义校验规则
4           author: String,
5           body: String,
6           comments: [{ body: String, date: Date }],//定义对象数组
7           date: { type: Date, default: Date.now },//指定默认值
8           hidden: Boolean,
9           meta: {
10              //定义对象
11              votes: Number,
12              favs: Number
13          }
14      });
15
16      //定义多个索引
17      Schema.index({ title: 1, author: 1, date: -1 })
18
19      //编译模型
20      const Model = mongoose.model('blog', Schema);
21          const blog = new Model({
22              title: 'nodejs持久化',
23              author: 'jerry',
24              body: '...'
25          })
26          const r = await blog.save();
27          console.log('新增blog', r)
```

可选字段类型:

- String
- Number
- Date
- Buffer
- Boolean

- Mixed
- ObjectId
- Array

避免创建索引警告

```
mongoose.connect('mongodb://localhost:27017/test', {
    useNewUrlParser: true
})
```

- 定义实例方法:抽象出常用方法便于复用

```
      //定义实例方法
        Schema.methods.findByAuthor=()=>{
            return this.model('blog').find({author:this.author}).exec();
        }
        //获取模型实例
          const Model = mongoose.model('blog', Schema);
        const blog = new Model({
          ...
        })

        //调用实例
        r=await blog.findByAuthor();
        console.log('findByAuthor',r)
```

- 静态方法

```
      //定义实例方法
        Schema.statics.findByAuthor=(author)=>{
            return this.model('blog').find({author}).exec();
        }
        //获取模型实例
          const Model = mongoose.model('blog', Schema);
        const blog = new Model({
          ...
        })

        //调用实例
        r=await blog.findByAuthor('jerry');
        console.log('findByAuthor',r)
```

- 虚拟属性

```
    //虚拟属性
        Schema.virtual('commentsCount').get(()=>{
            return this.comments.length
        })

        await Model.findOne({author:'jerry'});
        console.log('blog留言数',r.commentsCount)
```

通用接口实现

- config.js

```
module.exports={
    db:{
        url:'mongodb://localhost:27017/test',
        options:{ useNewUrlParser: true}
    }
}
```

- 用户模型，./model/user.js

```
module.exports={
    schema:{
        mobile:{
            type:String,
            required:true
        },
        realName:{
            type:String,
            required:true
        }
    }
}
```

- /framework/loader.js编写

```
const path=require('path');
const fs=require('fs');
const mongoose=require('mongoose');

const loader=(dir,cb)=>{
    //获取绝对路径
    const url=path.resolve(__dirname,dir);
    const files=fs.readdirSync(url);
    files.forEach(filename=>{
        //去掉后缀
        filename=filename.replace('.js','');
        //导入文件
        const file=require(url+'/'+filename);

        //处理逻辑
        cb(filename,file)
    })
}


const loaderModel=config=>app=>{
    mongoose.connect(config.db.url,config.db.options)
    const conn=mongoose.connection;
    conn.on('err',()=>console.log('连接数据失败'));
   app.$model={};
   loader('../model',(filename,{schema})=>{
    console.log(filename,schema)
    app.$model[filename]=mongoose.model(filename,schema)
```

```
    })
}

moule.exports={
    loaderModel
}
```

- /framework/router.js

```
const router=require('koa-router')();
const {init,get,create,update,del}=require('./api')

router.get('/api/:list',init,get)
router.post('/api/:list',init,create)
router.put('/api/:list/:id',init,update)
router.delete('/api/:list/:id',init,del)

module.exports=router.routes();
```

- Api编写，/framework/api.js

```
module.exports={
    async init(ctx,next){
        const model=ctx.app.$model[ctx.params.list];
        if(model){
            ctx.list=model;
            await next()
        }else{
            ctx.body='no this model'
        }
    },
    async get(ctx){
        ctx.body=await ctx.list.find({})
    },

    async create(ctx){
        const res=await ctx.list.create(ctx.reqest.body)
        ctx.body=res;
    },

    async update(ctx){
        const res=await
ctx.list.updateOne({_id:ctx.params.id},ctx.request.body);
        ctx.body=res;
    },

    async del(){
        const res=await ctx.list.deleteOne({_id:ctx.params.id});
        ctx.body=res;
    }


}
```

- index.js

```
const Koa=require('koa');
const app=new Koa();
const config=require('./config');
const {loaderModel}=require('./framework/loader')
loaderModel(config)(app)
const bodyParser=require('koa-bodyparser');

app.use(bodyParser());
app.use(require('koa-static')(__dirname+'/'))
const restful=require('./framework/router')
app.use(restful)

app.listen(3000)
```

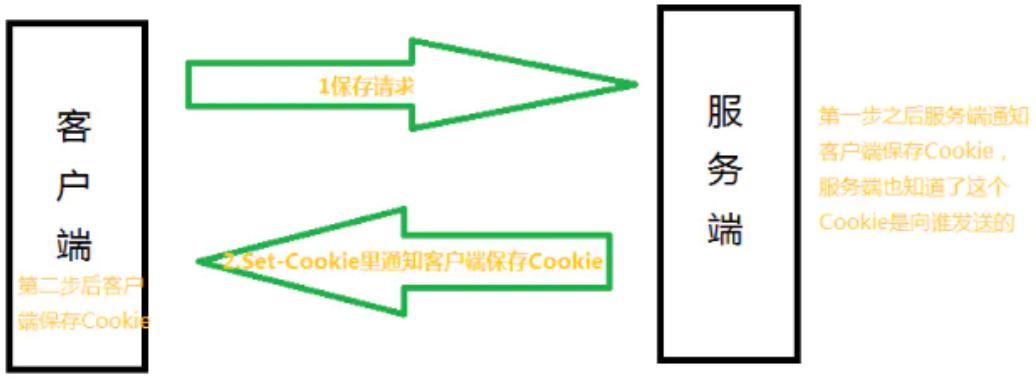
实战-鉴权

session-cookie方式

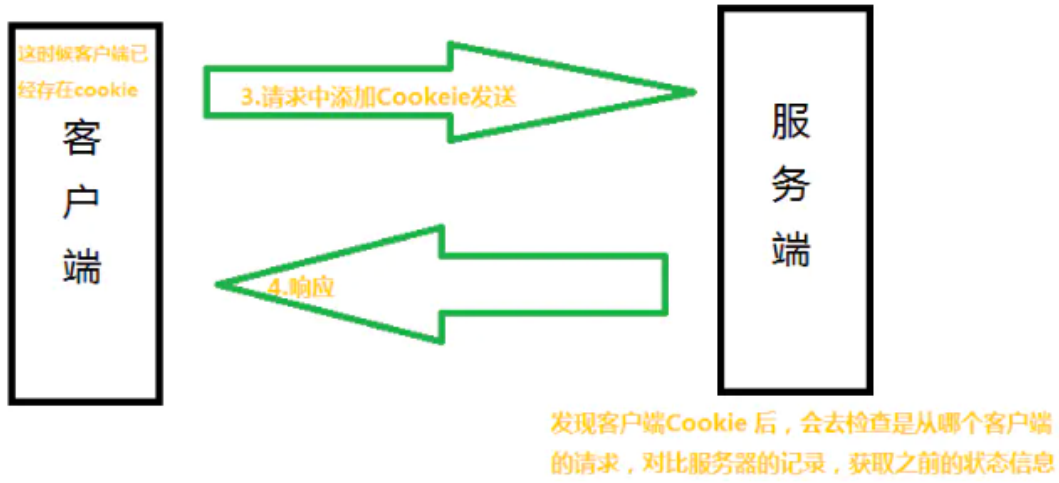
cookie原理解析

```
//cookie.js
const http=require('http');
const app=http.createServer((req,res)=>{
    if(req.url==='/favicon.ico'){
        res.end('');
        return
    }

    //观察cookie存在
    console.log('cookie',req.headers.cookie);
    //设置cookie
    res.setHeader('Set-Cookie','cookie1=abc;');
    res.end('hello cookie')
})

app.listen(3000)
```

- Header Set-Cookie负责设置cookie
- 请求传递Cookie
- 原理



客户端保存了Cookie之后的发起请求



session的原理解析

```
//session.js
const http=require('http');
const session={};
const app=http.createServer((req,res)=>{
    //观察cookie存在
    console.log('cookie',req.headers.cookie);
    const sessionKey='sid';
    const cookie=req.headers.cookie;
    if(cookie&&cookie.indexOf(sessionKey)>-1){
        res.end('Come Back');
        //简略写法未必具有通用型
        const pattern=new RegExp(`${sessionKey}=([^;]+);?\s*`);
        const sid=pattern.exec(cookie)[1];
        console.log('session:',sid,session,session[sid]);
    }else{
        const sid=(Math.random()*99999999).toFixed();
        //设置cookie
        res.setHeader('Set-Cookie',`${sessionKey}=${sid}`);
        session[sid]={name:'zyl'};
        res.end('Hello')
    }
```

```
    res.end('hello cookie')
})

app.listen(3000)
```

- 原理

实现原理：

1.服务器在接受客户端首次访问时在服务器端创建session,然后保存session(我们可以将session保存在内存中，也可以保存在redis中，推荐使用后者)，然后给这个session生成一个唯一的标识字符串，然后在响应头中种下这个唯一标识字符串。

2.签名。这一步通过秘钥对sid进行签名处理，避免客户端修改sid.(非必需步骤)

3.浏览器中收到请求响应的时候会解析响应头，然后将sid保存在本地cookie中，浏览器在下次http请求的请求头中会带上该域名下的cookie信息

4.服务器在接受客户端请求时会去解析请求头cookie中的sid，然后根据这个sid去找服务器端保存的该客户端的session，然后判断该请求是否合法

哈希Hash-SHA MD5

把一个不定长摘要定长结果

摘要 zhengyali->x4sdffferkff -防篡改

雪崩效应

摘要

对称DES

非对称-RSA

- koa中的session使用：npm i koa-session -S

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

```
5
6
7  //签名key keys作用 用来对cookie进行签名
8  app.keys=['some secret'];
9
10 //配置项
11 const SESS_CONFIG={
12     key:'kkb:sess',//cookie键名
13     maxAge:24*60*60*1000,//有效期，默认一天
14     httpOnly:true,//仅服务器修改
15     signed:true,//签名cookie，防篡改
16 }
17
18 //注册
19 app.use(session(SESS_CONFIG,app));
20
21 //测试
22 app.use(ctx=>{
23     if(ctx.path==='/favicon.ico'){
24         return;
25     }
26
27     //获取
28    let n=ctx.session.count||0;
29    //设置
30    ctx.session.count=++n;
31    ctx.body=`第${n}次访问`;
32
33 })
34
35 app.listen(3000);
```

使用redis存储session

HMAC

```
1  //redis.js
2  const redis=require('redis');
3
4  const client=redis.createClient(6379,'localhost');
5  client.set('hello','This is a value');
6
7  client.get('hello',(err,v)=>{
8      console.log('redis get',v)
9  })
```

- 安装:npm i koa-redis -S
- 配置使用

```
1  //koa-redis.js
2  const redisStore=require('koa-redis');
3  const redis=require('redis');
4  const redisClient=redis.createClient(6379,'localhost');
5  const session=require('koa-session');
6  const koa=require('koa');
7  const app=new koa();
```

```
8
9  const wrapper=require('co-redis');
10 const client=wrapper(redisClient);
11
12 //签名key keys作用 用来对cookie进行签名
13 app.keys=['some secret'];
14
15 app.use(session({
16     key:'kkb:sess',
17     maxAge:24*60*60*1000,//有效期，默认一天
18     httpOnly:true,//仅服务器修改
19     signed:true,//签名cookie,就是为了防篡改
20     store:redisStore({client})//此处可以不必指定client
21 },app))
22
23
24
25 app.use(async(ctx,next)=>{
26     const keys=await client.keys('*');
27     keys.forEach(async key => {
28         console.log(await client.get(key))
29     });
30     await next();
31 })
32
33 //测试
34 app.use(ctx=>{
35     if(ctx.path==='/favicon.ico'){
36         return;
37     }
38
39     //获取
40    let n=ctx.session.count||0;
41    //设置
42    ctx.session.count=++n;
43    ctx.body=`第${n}次访问`;
44
45 })
46
47 app.listen(3000);
```

- session-cookie方式

```
1  //index.html
2  <!DOCTYPE html>
3  <html lang="en">
4  <head>
5      <meta charset="UTF-8">
6      <meta name="viewport" content="width=device-width, initial-
   scale=1.0">
7      <meta http-equiv="X-UA-Compatible" content="ie=edge">
8      <title>Document</title>
9      <script src="https://cdn.jsdelivr.net/npm/vue/dist/vue.js"></script>
10     <script src="https://unpkg.com/axios@0.19.2/dist/axios.min.js">
   </script>
11 </head>
12 <body>
```

```
    <div id="app">
        <div>
            <input v-model="username">
            <input v-model="password">
        </div>
        <div>
            <button @click="login">Login</button>
            <button @click="logout">Logout</button>
            <button @click="getUser">GetUsers</button>
        </div>
        <div>
            <button
onclick="document.getElementById('log').innerHTML=''">
                clear log
            </button>
        </div>
        <h6 id="log"></h6>
    </div>
    <script>
      axios.defaults.withCredentials=true;
      axios.interceptors.response.use(response=>{
          console.log(response)

 document.getElementById('log').append(JSON.stringify(response.data));
          return response;
      })
     const app=new Vue({
         el:'#app',
         data:{
             username:'test',
             password:'test'
         },
         methods: {
             async login(){
                 await axios.post('/users/login',{
                     username:this.username,
                     password:this.password
                 })
             },
             async logout(){
                 await axios.post('/users/logout')
             },
             async getUser(){
                 await axios.get('/users/getUser')
             }
         },
     })
    </script>
</body>
</html>
```

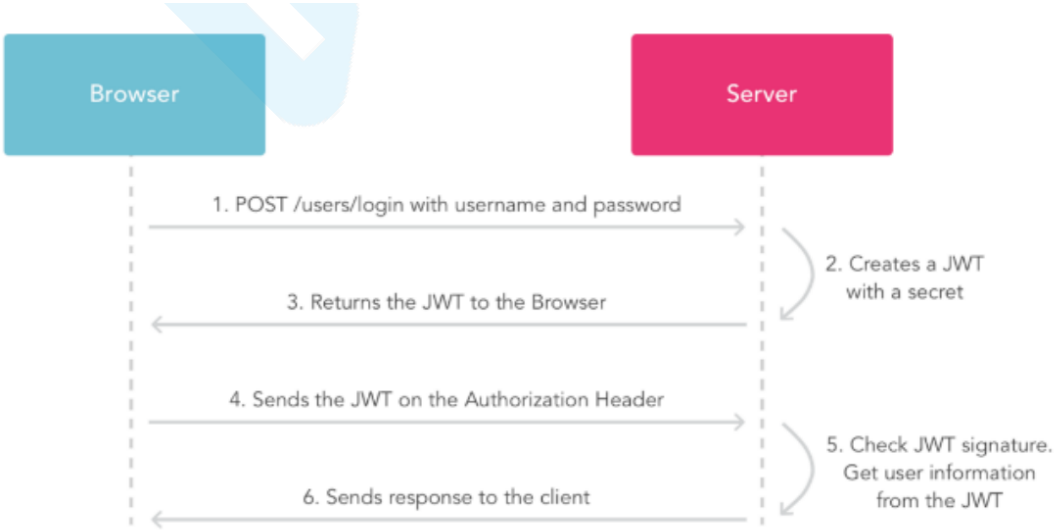
```
//login.js
const Koa=require('koa');
const router=require('koa-router')();
const session=require('koa-session');
const cors=require('koa2-cors');
const bodyParser=require('koa-bodyparser');
```

```
const static=require('koa-static');
const app=new Koa();

//配置session的中间件
app.use(cors({
    credentials:true
}))

app.keys=['some secret'];

app.use(static(__dirname+'/'));
app.use(bodyParser());
app.use(session(app));

app.use((ctx,next)=>{
    if(ctx.url.indexOf('login')>-1){
        next();
    }else{
        console.log('session',ctx.session.userinfo);
        if(!ctx.session.userinfo){
            ctx.body={
                message:'登录失败'
            }
        }else{
            next();
        }
    }
})


router.post('/users/login',async ctx=>{
    const {body}=ctx.request;
    console.log('body',body);
     //设置session
     ctx.session.userinfo=body.username;
     ctx.body={
         message:'登录成功'
     }
})

router.post('/users/logout',async ctx=>{
    //设置session
    delete ctx.session.userinfo;
    ctx.body={
        message:'登出系统'
    }
})

router.get('/users/getUser',async ctx=>{
    console.log(ctx.session)
    ctx.body={
        message:'获取数据成功',
        userinfo:ctx.session.userinfo
    }
})

app.use(router.routes());

```

```
app.use(router.allowedMethods());
app.listen(3000);
```

Token验证

- session不足
- 服务器有状态
- 不灵活如果APP该怎么办？跨域怎么办?
- 原理



1.客户端使用用户名跟密码请求登录

2.服务端收到请求，去验证用户名和密码

3.验证成功后，服务端会签发一个令牌(Token),再把这个Token发送给客户端

4.客户端收到Token以后可以把它存储起来，比如放在cookie里或者Local Storage里

5.客户端每次向服务端请求资源的时候需要带着服务端签发的Token

6.服务端收到请求，然后去验证客户端请求里面带着的Token,如果验证成功，就向客户端返回请求的数据

- 案例：令牌认证
  - 登录页：index.html

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <meta http-equiv="X-UA-Compatible" content="ie=edge">
    <title>Document</title>
    <script src="https://cdn.jsdelivr.net/npm/vue/dist/vue.js"></script>
    <script src="https://unpkg.com/axios@0.19.2/dist/axios.min.js"></script>
</head>
<body>
    <div id="app">
```

```
        <div>
            <input v-model="username">
            <input v-model="password">
        </div>
        <div>
                <button @click="login">Login</button>
                <button @click="logout">Logout</button>
                <button @click="getUser">GetUsers</button>
        </div>
        <div>
            <button @click="logs=[]">clear log</button>
        </div>
        <!--日志-->
        <ul>
            <li v-for="(log,idx) in logs" :key="idx">
                {{log}}
            </li>
        </ul>
    </div>
    <script>
        axios.interceptors.request.use(config=>{
            const token=window.localStorage.getItem("token");
            if(token){
                //判断是否存在token，如果存在的话，则每个http header都加上token
                //  Bearer是JWT的认证头部信息
 config.headers.common['Authorization']="Bearer"+token;
            }
            return config;
        },err=>{
            return Promise.reject(err)
        })

        axios.interceptors.response.use(response=>{
            app.logs.push(JSON.stringify(response.data));
            return response;
        },err=>{
            app.logs.push(JSON.stringify(response.data));
            return Promise.reject(err)
        })

        const app=new Vue({
            el:'#app',
            data:{
                username:'test',
                password:'test',
                logs:[]
            },
            methods: {
               async login() {
                    const res=await axios.post('/users/login-token',{
                        username:this.username,
                        password:this.password
                    });
                    localStorage.setItem('token',res.data.token);
                },
```

```
                async logout(){
                    localStorage.removeItem('token')
                },
                async getUser(){
                    await axios.get('/users/getUser-token')
                }

            },
        })
    </script>
</body>
</html>
```

- 登录接口
  - 安装依赖:npm i jsonwebtoken koa-jwt v-S
  - 接口编写，index.js

```
const Koa=require('koa');
const router=require('koa-router')();

const jwt=require('jsonwebtoken');
const jwtAuth=require('koa-jwt');
const secret="it's a secret";
const cors=require('koa2-cors');

const bodyParser=require('koa-bodyparser');
const static=require('koa-static');
const app=new Koa();
app.keys=['some secret'];

app.use(static(__dirname+'/'));
app.use(bodyParser());

router.post('/users/login-token',async ctx=>{
    const {body}=ctx.request;
    //登录逻辑，略
    //设置session
    const userinfo=body.username;
    ctx.body={
        message:'登录成功',
        user:userinfo,
        //生成token返回给客户端
        token:jwt.sign({
            data:userinfo,
            //设置token过期时间，一小时，秒为单位
            exp:Math.floor(Date.now()/1000)+60*60
        },
        secret
        )
    }
})

router.get("/users/getUser-token",jwtAuth({secret}),async ctx=>{
    //验证通过，state.user
    console.log(ctx.state.user);
```

```
39
40      //获取session
41      ctx.body={
42          message:'获取数据成功',
43          userinfo:ctx.state.user.data
44      }
45  })
46
47  app.use(router.routes());
48  app.use(router.allowedMethods());
49  app.listen(3000)
50
```

JWT（JSON WEB TOKEN）原理解析

1.Bearer Token包含三个组成部分：令牌头、payload、哈希

```
1  eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJkYXRhIjp7InVzZXJuYW1lIjoiYWJjIiwi
   cGFzc3dvcmQiOiIxMTExMTEifSwiZXhwIjoxNTU3MTU1NzMwLCJpYXQiOjE1NTcxNTIxMzB9.
   pjGaxzX2srG_MEZizzmFEy7JM3t8tjkiu3yULgzFwUk
```

2.签名：默认使用base64对payload编码，使用hs256算法对令牌头、payload和密钥进行签名生成哈希

3.验证：默认使用hs256算法对hs256算法对令牌中数据签名并将结果和令牌中哈希比对

```
1  //jsonwebtoken.js
2  const jsonwebtoken=require('jsonwebtoken');
3  const secret='12345678';
4  const opt={
5      secret:'jwt_secret',
6      key:'user'
7  }
8
9  const user={
10     username:'abc',
11     password:'111111',
12 }
13
14 const token=jsonwebtoken.sign({
15     data:user,
16     //设置token过期时间
17     exp:Math.floor(Date.now()/1000)+(60*60),
18 },secret)
19
20 console.log('生成token:'+token)
21 //eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJkYXRhIjp7InVzZXJuYW1lIjoiYWJjI
   iwicGFzc3dvcmQiOiIxMTExMTEifSwiZXhwIjoxNTg3NTMyODY2LCJpYXQiOjE1ODc1MjkyN
   jZ9.cJOzgS1_Pssj-VEhPO-dlHVgyBwY7_HYjkRdIu-rCeE
22
23 console.log('解码: ',jsonwebtoken.verify(token,secret,opt));
24
25 //解码:  { data: { username: 'abc', password: '111111' },
26 //exp: 1587532938,
27 //iat: 1587529338 }
28
```

HMAC SHA256 HMAC(Hash Message Authentication Code)散列消息鉴别码，基于密钥的Hash算法的认证协议。消息鉴别码实现鉴别的原理是用公开函数和密钥产生一个固定长度的值作为认证标识，用这个标识鉴别信息的完整性。使用一个密钥生成一个固定大小的小数据块，即MAC，并将其加入到消息中，然后传输。接收方利用与发送方共享的密钥进行鉴别认证等。

BASE64按照RFC2045的定义，Base64被定义为：Base64内容传送编码被设计用来把任意序列的8位字节描述为一种不易别人直接识别的形式。(The Base64 Contnt-Transfer-Encoding is designed to reprsent arbitrary sequence of ocets in a form that need not be humanly readable).常见于邮件、http加密，截取http信息，你会发现登录操作的用户名、密码字段通过BASE64编码的

Beare

Beare作为一种认证类型（基于OAuth2.0）,使用"Bearer"关键词进行定义

参看文档：

jsonwebtoken(https://www.npmjs.com/package/jsonwebtoken)、koa-jwt(https://www.npmjs.com/package/koa-jwt)

阮一峰JWT解释

http://www.ruanyifeng.com/blog/2018/07/json_web_token-tutorial.html

OAuth（开放授权）

- 概述：三方登录主要基于OAuth2.0.OAuth协议为用户资源的授权提供了一个安全的、开放而简易的标准。与以往的授权方式不同之处是OAuth的授权不会使第三方触及到用户的账号信息（如用户名和密码），即第三方无需使用用户的用户名与密码就可以申请获得该用户资源的授权，因此OAuth是安全的。
- OAuth登录
  - 登录页面index.html

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <meta http-equiv="X-UA-Compatible" content="ie=edge">
    <title>Document</title>
    <script src="https://cdn.jsdelivr.net/npm/vue/dist/vue.js"></script>
    <script src="https://unpkg.com/axios@0.19.2/dist/axios.min.js"></script>
</head>
<body>
    <div id="app">
        <a href="/github/login">login with github</a>
    </div>
</body>
</html>
```

  - 登录接口index.js

```
const Koa=require('koa');
const router=require('koa-router')();
```

```
const static=require('koa-static');
const app=new Koa();
const axios=require('axios');
const querystring=require('querystring');

app.use(static(__dirname+'/'));
const config={
    client_id:'de7e36f329969336b594',
    client_secret:'bb87d25fde8232c69b3985ecda7ee51844c663ee'
}

router.get('/github/login',async ctx=>{
    const dataStr=(new Date()).valueOf();
    //重定向认证接口，并配置参数
    let path="https://github.com/login/oauth/authorize";
    path+=`?client_id=${config.client_id}`;

    //转发到授权服务器
    ctx.redirect(path)
})

router.get('/oauth/github/callback',async ctx=>{
    console.log('callback..');
    const code=ctx.query.code;
    const params={
        client_id:config.client_id,
        client_secret:config.client_secret,
        code:code
    }
    let res=await
axios.post('https://github.com/login/oauth/access_token',params);
    const access_token=querystring.parse(res.data).access_token
    res=await axios.get(`https://api.github.com/user?
access_token=${access_token}`);

    console.log('userAcess:',res.data);
    ctx.body=`
    <h1>hello ${res.data.login}</h1>
    <img src="${res.data.avatar_url}"/>
    `
})

app.use(router.routes());
app.use(router.allowedMethods());
app.listen(3000)

```

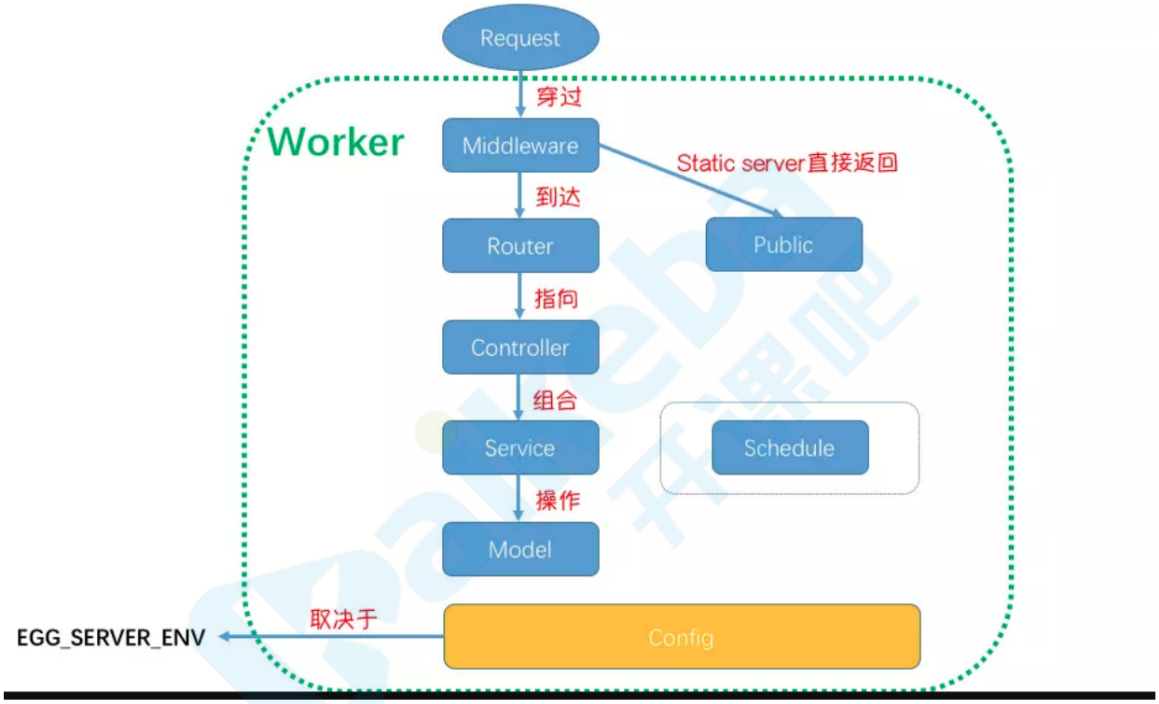
- 单点登录

```
cd passport
node app.js
cd  ../system
PORT=8081 SERVER_NAME=a node app.js
PORT=8082 SERVRT_NAME=b node app.js
```

基于Koa定制自己企业级MVC框架

egg.js体验

- 架构



- 创建项目

```
//创建项目
npm i egg-init -g
egg-init egg --type=simple
cd egg
npm i

//启动项目
npm run dev
localhost:7001
```

- 项目浏览结构
  - public
  - Router->Controller->Service->Model
  - Schedule
- 创建一个路由，router.js

```
  router.get('/', controller.home.index);
```

- 创建一个控制器,user.js

```
'use strict';
const Controller = require('egg').Controller;
class UserController extends Controller {
  async index() {
    const { ctx } = this;
   ctx.body = 'hi,user';
  }
}

module.exports = UserController;

```

- 创建一个服务，./app/service/user.js

```
'use strict';

const { Service } = require('egg');
class UserService extends Service {
  async getAll() {
    return [{
      name: 'zyl',
    }];
  }
}

module.exports = UserService;
```

- 使用服务,./app/contoller/user.js

```
'use strict';
const Controller = require('egg').Controller;
class UserController extends Controller {
  async index() {
    const { ctx } = this;
    // ctx.body = 'hi,user';
    ctx.body = await ctx.service.user.getAll();
  }
}

module.exports = UserController;
```

- 创建模型层：以mysql_sequelize为例演示数据持久化
  - 安装：npm i egg-sequelize mysql2 -S
  - 在config/plugin.js中引入egg-sequelize插件

```
squelize: {
    enable: true,
    package: 'egg-sequlize',
  },
```

  - 在congfig/config.default.js中编写sequelize配置

```
//const userConfig中
sequelize: {
      dialect: 'mysql',
      host: '127.0.0.1',
      port: 3306,
      username: 'root',
      password: 'example',
      database: 'kaikeba',
    },
```

- 编写User模型，./app/model/user.js

```
'use strict';
module.exports = app => {
  const { STRING } = app.Sequelize;
  const User = app.model.define('user',
    { name: STRING(30) },
    { timestamps: false }
  );

  // 数据库同步
  User.sync({ force: true });
  return User;
};
```

- 服务中或者控制器中调用：ctx.model.User或app.model.User

```
'use strict';

const { Service } = require('egg');
class UserService extends Service {
  async getAll() {
    return await this.ctx.model.User.findAll();
  }
}

module.exports = UserService;

//或者控制器
ctx.body=await this.ctx.model.User.findAll();


//需要同步数据库
<https://eggjs.org/zh-cn/tutorials/sequelize.html/>
//添加测试数据
const User=this.ctx.model.User
await User.sync({force:true})
await User.create({
    name:'zyl'
})
```

实现MVC分层架构

- 目标是创建约定大于配置、开发效率高、可维护性强的项目架构
- 路由处理
  - 规范：
    - 所有路由，都要放在routes文件夹中
    - 若导出路由对象，使用动词+空格+路径作为key,值是操作方法
    - 若导出函数，则函数返回第二条约定格式的对象
  - 路由定义
    - 新建routes/index.js，默认index.js没有前缀

```
module.exports={
    'get /':async ctx=>{
        ctx.body='首页'
    },

    'get /detail':async ctx=>{
        ctx.body='详情页'
    }
}
```

    - 新建routes/user.js路由前缀是/user

```
module.exports={
    // /user/
    'get /':async ctx=>{
        ctx.body='用户首页'
    },
    // /user/detail
    'get /detail':async ctx=>{
        ctx.body='用户详情页'
    }
}
```

    - 路由加载器，新建kkb-loader.js

```
const fs=require('fs');
const path=require('path');

const Router=require('koa-router');

//读取目录
function load(dir,cb){
    //获取绝对路径
    const url=path.resolve(__dirname,dir);
    //读取路径下的文件
    const files=fs.readFileSync(url);

    //遍历路由文件，将路由配置解析到路由器中
    files.forEach(fileName=>{
        //去掉后缀
        fileName=fileName.replace('.js','');
        //导入文件
```

```
        const file=require(url+'/'+fileName);
        //处理逻辑
        cb(fileName,file);

    })
}

function initRouter(){
    const router=new Router();
    load('routes',(fileName,routes)=>{
        //若是index,则无前缀，别的文件前缀就是文件名
        const prefix=fileName==='index'?'':`/${fileName}`;

        //遍历路由并添加到路由器
        Object.keys(routes).forEach(route=>{
            const [method,path]=route.split(' ');

            console.log(`正在映射地址:${method.toLocaleLowerCase()}${prefix}${path}`)
            router[method](prefix+path,routes[key]);
        })
    });

    return router;
}

module.exports={initRouter}
```

- 测试，引入kkb-loader.js

```
const Koa=require('koa');
const app=new Koa();
const {initRouter}=require('./kkb-loader');
app.use(initRouter().routes());

app.listen(3000)
```

- 封装，创建kkb.js

```
const koa=require('koa');
const {initRouter}=require('./kkb-loader');

class kkb{
     constructor(config){
                this.$app=new koa(config);
                this.$router=initRouter();
                this.$app.use(this.$router.routes())
     }
     start(port){
         this.$app.listen(port,()=>{
             console.log('服务器启动成功,端口',port)
         })
     }
}

module.exports=kkb;
```

- 修改app.js

```
const kkb=require('./kkb');
const app=new kkb();
app.start(3000)
```

- 控制器：抽取route中业务逻辑至controller
  - 约定：controller文件夹下面存放业务逻辑代码，框架自动加载并集中暴露
  - 新建controller/home.js

```
module.exports={
  index:async ctx=>{
    ctx.body='首页'
  },
  detail:ctx=>{
     ctx.body='详情页面'
  }
}
```

  - 修改路由声明，routes/index.js

```
//需要传递kkb实例并访问其$ctrl中暴露的控制器
module.exports=(app)=>({
    'get /': app.$ctrl.home.index,
    'get /detail': app.$ctrl.home.detail
})
```

- 加载控制器，更新kkb-loader.js

```
function initConroller(){
    const controllers={};
    //读取目录
    load('controller',(filename,controller)=>{
        controllers[filename]=controller;
    })

    return controllers;
}
```

- 初始化控制器，kkb.js

```
const koa=require('koa');
const {initRouter,initConroller,initService}=require('./kkb-loader');

class kkb{
     constructor(config){
                this.$app=new koa(config);
                this.$ctrl=initConroller()//加载ctrl
                this.$router=initRouter(this);
                this.$app.use(this.$router.routes())
     }
     start(port){
         this.$app.listen(port,()=>{
             console.log('服务器启动成功,端口',port)
```

```
        })
    }
}

module.exports=kkb;
```

- 修改路由初始化逻辑，能够处理函数形式的声明，kkb-loader.js

```
function initRouter(app) {
    const router = new Router();
    load('routes', (fileName, routes) => {
        //若是index,则无前缀，别的文件前缀就是文件名
        const prefix = fileName === 'index' ? '' : `/${fileName}`;
        //路由类型判断
        routes=typeof routes=='function'?routes(app):routes
        //遍历路由并添加到路由器
        Object.keys(routes).forEach(key => {
            const [method, path] = key.split(' ');
            console.log(`正在映射地址:${method.toLocaleLowerCase()}${prefix}${path}`)

             router[method](prefix+path,routes[key]);

        })
    });

    return router;
}
```

- 服务：抽离通用逻辑至service文件夹，利于复用
  - 新建service/user.js

```
const delay=(data,tick)=>new Promise(resolve=>{
    setTimeout(()=>{
resolve(data)
    },tick)
})

module.exports={
    getName(){
        return delay('jerry',1000)
    },
    getAge(){
        return 20;
    }
}
```

  - 加载service

```
//kkb-loader.js
function initService(){
    const services={};
    //读取目录
    load('service',(filename,service)=>{
        services[filename]=service;
```

```
    })

    return services;
}



module.exports = { initService }

//kkb.js
 this.$service=initService();

```

- 更新路由

```
//routes/user.js
module.exports={
    'get /':async app=>{
        const name=await app.$service.user.getName();
        app.ctx.body="用户"+name
    },

    'get /detail':async app=>{
        app.ctx.body='用户年龄'+app.$service.user.getAge();
    }
}


//routes/index.js
module.exports=(app)=>({
    'get /': app.$ctrl.home.index,
    'get /detail': app.$ctrl.home.detail
})

//conroller/home.js
module.exports=app=>({
    index:async ctx=>{
        const name=await app.$service.user.getName();
        app.ctx.body="ctr user"+name
    },

    detail:async ctx=>{
        app.ctx.body='详情页'
    }
})
```

- 数据库集成
  - 集成sequelize:npm i sequelize mysql2 -S
  - 约定:
    - config/config.js中存放项目配置项
    - key表示对应配置目标
    - model中存放数据模型
  - 配置sequelize连接配置项，index.js

```
//config/index.js
module.exports={
    db:{
        dialect:'mysql',
        host:'localhost',
        database:'zyl-kaikeba',
        username:'root',
        password:'yw@910714'
    }
}
```

- 新增loadConfig,kkb-loader.js

```
const Sequelize=require('sequelize');
function loadConfig(app){
    load("config",(filename,conf)=>{
        if(conf.db){
            app.$db=new Sequelize(conf.db)

        }
    })
}


//kkb.js
loadConfig(this)
```

- 新建数据库模型,model/user.js

```
const {STRING} =require('sequelize');
module.exports={
    schema:{
        name:STRING(30)
    },
    options:{
        timestamps:false
    }
}
```

- loaderModel和loaderConfig初始化，kkb-loader.js

```

const Sequelize=require('sequelize');
function loadConfig(app){
    load("config",(filename,conf)=>{
        if(conf.db){
            app.$db=new Sequelize(conf.db)


            //加载模型
            app.$model={};
            load("model",(filename,{schema,options})=>{

 app.$model[filename]=app.$db.define(filename,schema,options)
            })
```

```
            app.$db.sync();

        }
    })
}
```

- 在controller中使用$db,home.js

```
module.exports=app=>({
    index:async ctx=>{
        const name=await app.$model.user.findAll();
        app.ctx.body=name
    },

    detail:async ctx=>{
        app.ctx.body='详情页'
    }
})
```

- 在service中使用$db,user.js

```
//service/user.js
module.exports=(app)=>({
    getName(){
        return app.$model.user.findAll();
    },
    getAge(){
        return 20;
    }
})

//修改kkb-loader.js
function initService(app){
    const services={};
    //读取目录
    load('service',(filename,service)=>{
        services[filename]=service(app);
    })

    return services;
}

//修改kkb.js
 this.$service=initService(this);

```

- 中间件
  - 规定koa中间件放入middleware文件夹
  - 编写一个请求记录中间件，./middleware/logger.js

```
module.exports=async (ctx,next)=>{
   console.log(ctx.method+" "+ctx.path);
   const start=new Date();
   await next();
   const duration=new Date()-start;
   console.log(ctx.method+" "+ctx.path+" "+duration);
}
```

- 配置中间件，./config/index.js

```
module.exports={
    db:{
        dialect:'mysql',
        host:'localhost',
        database:'zyl-kaikeba',
        username:'root',
        password:'yw@910714'
    },
    middleware:['logger']
}
```

- 加载中间件，kkb-loader.js

```
const Sequelize=require('sequelize');
function loadConfig(app){
    load("config",(filename,conf)=>{
        if(conf.db){
            app.$db=new Sequelize(conf.db)


            //加载模型
            app.$model={};
            load("model",(filename,{schema,options})=>{

 app.$model[filename]=app.$db.define(filename,schema,options)
            })
            app.$db.sync();

            if(conf.middleware){
                conf.middleware.forEach(mid=>{
                    const
midPath=path.resolve(__dirname,"middleware",mid);
                    app.$app.use(require(midPath))
                })
            }
        }
    })
}
```

- 调用,kkb.js

```
const koa=require('koa');
const
{initRouter,initConroller,initService,loadConfig,initSchedule}=requ
ire('./kkb-loader');
```

```
3
4  class kkb{
5       constructor(config){
6                  this.$app=new koa(config);
7                  loadConfig(this)
8                  this.$service=initService(this);
9                  this.$ctrl=initConroller(this)//加载ctrl
10                 this.$router=initRouter(this);
11                 this.$app.use(this.$router.routes())
12                 initSchedule()
13      }
14      start(port){
15          this.$app.listen(port,()=>{
16              console.log('服务器启动成功,端口',port)
17          })
18      }
19 }
20
21 module.exports=kkb;
```

- 定时任务
  - 使用node-schedule来管理定时任务

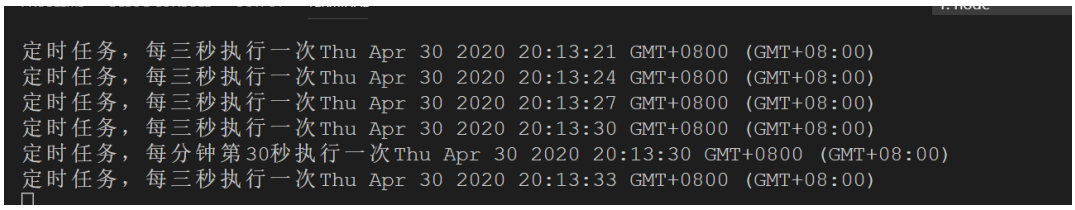
```
1  npm i node-schedule -S
```

  - 约定：schedule目录，存放定时任务，使用crontab格式来启动定时

```
1  //log.js
2  module.exports={
3      interval:'*/3 * * * * *',
4      handler(){
5          console.log('定时任务，每三秒执行一次'+new Date())
6      }
7  }
8
9
10 //user.js
11 module.exports={
12     interval:'30 * * * * *',
13     handler(){
14         console.log('定时任务，每分钟第30秒执行一次'+new Date())
15     }
16 }
```

  - 新增loadSchedule函数,kkb-loader.js

```
const schedule=require('node-schedule');
function initSchedule(){
    load('schedule',(filename,scheduleConfig)=>{

 schedule.scheduleJob(scheduleConfig.interval,scheduleConfig.handle
r);

    })
}

//kkb.js
 initSchedule()
```



egg-实战

https://eggjs.org/zh-cn/

1.创建项目

```
//创建项目
npm i egg-init -g
egg-init egg --type=simple
cd egg
npm i

//启动项目
npm run dev
localhost:7001
```

2.添加swagger-doc

-添加Controller方法

```
'use strict';

const { Controller } = require('egg');

/**
 * @Controller 用户管理
 */
class UserContoller extends Controller {
  // eslint-disable-next-line no-useless-constructor
  constructor(ctx) {
    super(ctx);
  }

  /**
   * @summary 创建用户
   * @description 创建用户，记录用户账号/密码/类型
```

```
    * @router post /api/user
    * @request body createUserRequest *body
    * @response 200 baseResponse 创建成功
    */
  async create() {
    const { ctx,service } = this;
    ctx.body = 'user ctrl';

  }
}

module.exports = UserContoller;

```

- contract

```
//app.contract/index.js
'use strict';

module.exports = {
  baseRequest: {
    id: {
      type: 'string',
      description: 'id唯一键',
      required: true,
      example: '1',
    },
  },
  baseResponse: {
    code: { type: 'integer', required: true, example: 0 },
    data: { type: 'string', example: '请求成功' },
    errorMessage: { type: 'string', example: '请求成功' },
  },
};


//app.contract/user.js
'use strict';

module.exports = {
  createUserRequest: {
    mobile: { type: 'string', required: true, description: '手机号', example: '17324418996', format: /^1[34578]\d{9}$/ },
    password: { type: 'string', required: true, description: '密码', example: '111' },
    realName: { type: 'string', required: true, description: '姓名', example: 'Tom' },
  },
};

```

- 添加SwaggerDoc功能

```
npm i egg-swagger-doc-feat -S
```

```
//config/plugin
  swaggerdoc: {
    enable: true,
    package: 'egg-swagger-doc-feat',
  }
```

```
  config.swaggerdoc = {
    dirScanner: './app/controller',
    apiInfo: {
      title: 'zyl接口',
      description: 'zyl接口 swagger-ui for egg',
      version: '1.0.0',
    },
    schemes: [ 'http', 'https' ],
    consumes: [ 'application/json' ],
    produces: [ 'application/json' ],
    enableSecurity: false,
    // enableValidate :true
    routerMap: true, // 自动注册路由
    enable: true,
  };
```

http://localhost:7001/swagger-ui.html

http://localhost:7001/swagger-doc

- 增加异常处理中间件

```
// middleware/error_handler.js
'use strict';
module.exports = (option, app) => {
  return async (ctx, next) => {
    try {
      await next();
    } catch (err) {
      app.emit('error', err, this);
      const status = err.status || 500;
      const error = status === 500 && app.config.env === 'prod' ? 'Internal Server Error' : err.message;
      ctx.body = {
        code: status,
        error,
      };
      if (status === 422) {
        ctx.body.detail = err.errors;
      }

      ctx.status = 200;
    }
  };
};

```

```
//config.default.js
config.middleware = [ 'errorHandler' ];
```

- helper方法实现统一响应格式

  helper函数用来提供一些实用的utility函数

它的作用在于我们可以将一些常用的动作抽离在helper.js里面成为一个独立的函数，这样可以用javascript来写复杂的逻辑，避免逻辑分散各处。另一个好处是helper这样一个简单的函数，可以让我们更容易编写测试用例

框架内置了一些常用的helper函数。我们可以编写自定义的helper函数

```
//contoller/user.js
    const res = { abc: 123 };
    //设置相应内容和响应状态码
    ctx.helper.success({ ctx, res });
```

```
//extend/helper.js
'use strict';
const moment = require('moment');

// 格式化时间
exports.formatTime = time => moment(time).format('YYYY-MM-DD HH:mm:ss');


// 统一成功应答
exports.success = ({ ctx, res = null, msg = '请求成功' }) => {
  ctx.body = {
    code: 0,
    data: res,
    msg,
  };
  ctx.status = 200;
};

```

- Vliadate检查

```
npm i egg-validate -S
```

```
//config/plugin.js
  validate: {
    enable: true,
    package: 'egg-validate',
  }
```

```
  async create() {
    const { ctx,service } = this;
    // ctx.body = 'user ctrl';
    // 校验参数
    ctx.validate(ctx.rule.createUserRequest);
    const res = { abc: 123 };
    ctx.helper.success({ ctx, res });

  }
```

- 添加model层

```
npm i egg-mongoose -S
```

```
  mongoose: {
    enable: true,
    package: 'egg-mongoose',
  },
```

```
//config.default.js
  config.mongoose = {
    url: 'mongodb://127.0.0.1:27017/egg_x',
    options: {
      autoReconnect: true,
      reconnectTries: Number.MAX_VALUE,
      bufferMaxEntries: 0,
    },
  };
```

```
//model/user.js
'use strict';
module.exports = app => {
  const mongoose = app.mongoose;
  const UserSchema = new mongoose.Schema({
    mobile: { type: 'string', required: true, unique: true },
    password: { type: 'string', required: true },
    realName: { type: 'string', required: true },
    avatar: { type: String, default:
'https://1.gravatar.com/avatar/a3e54af3cb6e157e496ae430aed4f4a3?
s=96&d=mm' },
    extra: { type: mongoose.Schema.Types.Mixed },
    createdAt: { type: Date, default: Date.now },
  });
  return mongoose.model('User', UserSchema);
}
;

```

- 添加service层

```
npm i egg-bcrypt -S
```

```
  bcrypt: {
    enable: true,
    package: 'egg-bcrypt',
  },
```

```
//service/user.js
'use strict';

const Service = require('egg').Service;

class UserService extends Service {

  /**
```

```
    * 创建用户
    * @param {*} payload
    */

  async create(payload) {
    const { ctx } = this;
    payload.password = await this.ctx.genHash(payload.password);
    return ctx.model.User.create(payload);
  }
}

module.exports = UserService;


```

- conroller调用

```
  async create() {
    const { ctx, service } = this;
    // ctx.body = 'user ctrl';
    // 校验参数
    ctx.validate(ctx.rule.createUserRequest);
    // const res = { abc: 123 };
    // ctx.helper.success({ ctx, res });

    // 组装参数
    const payload = ctx.request.body || {};
    // 调用Servic进行业务处理
    const res = await service.user.create(payload);
    // 设置响应内容和响应状态码
    ctx.helper.success({ ctx, res });

  }
```

通过生命周期初始化数据

https://eggjs.org/en/basics/app-start.html#mobileAside

```
//app.js
'use strict';
/**
 * 全局定义
 * @param app
 */

class AppBootHook {
  constructor(app) {
    this.app = app;
    app.root_path = __dirname;
  }

  configwillLoad() {
    // ready to call configDidLoad
    // config,plugin files are referred
    // this is the last chance to modify the config
  }
```

```
  configDidLoad() {
    // config,plugin files have been loaded
  }

  async didLoad() {
    // All files have loaded,start plugin here
  }

  async willReady() {
  // All plugins have started,can do some thing before app ready
  }

  async didReady() {
    // worker is ready,can do some things
    // don't need to block the app boot
    console.log('----init Data------');
    const ctx = await this.app.createAnonymousContext();
    await ctx.model.User.remove();
    await ctx.service.user.create({
      mobile: '17324418996',
      password: '111',
      realName: 'zyl',
    });
  }

  async serverDidReady() {
  //
  }

  async beforeClose() {
    // do some thing befroe app close
  }
}

module.exports = AppBootHook;

```

用户鉴权模块

注册jwt模块

```
npm i egg-jwt -S
```

```
//plugin.js
  jwt: {
    enable: true,
    package: 'egg-jwt',
  }


 //config.default.js
   config.jwt = {
    secret: 'Great4-M',
    enable: true, // default is false
    match: /^\/api/, // optional
  };
```

- service层

```
//service/actionToken.js
'use strict';

const { Servicde } = require('egg');
class ActionTokenService extends Servicde {
  async apply(_id) {
    const { ctx } = this;
    return ctx.app.jwt.sign({
      data: {
        _id,
      },
      exp: Math.floor(Date.now() / 1000) + (60 * 60 * 24 * 7),
    }, ctx.app.config.jwt.secret);
  }
}

module.exports = ActionTokenService;

```

```
//service/userAccess.js
'use strict';

const { Servicde } = require('egg');
class UserAccessService extends Servicde {
  async login(payload) {
    const { ctx, service } = this;
    const user = await service.user.findByMobile(payload.mobile);
    if (!user) {
      ctx.throw(404, 'user not found');
    }
    const verifyPsw = await ctx.compare(payload.password,
user.password);
    if (!verifyPsw) {
      ctx.throw(404, 'user password is error');
    }

    // 生成Token令牌
    return { token: await service.actionToken.apply(user._id) };
  }

```

```
  async logout() {
    //
  }
  async current() {
    const { ctx, service } = this;
    // ctx.state.user可以提取到jwt编码的data
    const _id = ctx.state.user.data._id;
    const user = await service.user.find(_id);
    if (!user) {
      ctx.throw(404, 'user is not found');
    }
    user.password = 'How old are you?';
    return user;

  }

}

module.exports = UserAccessService;

```

- Contract层

```
//app/contract/userAccess.js
'use strict';
module.exports = {
  loginRequest: {
    mobile: {
      type: 'string', required: true, description: '手机号', example: '17324418996', format: /^1[345678]\d{9}$/,
    },
    password: {
      type: 'string', required: true, description: '密码', example: '111',
    },
  },
}
;

```

- Controller层

```
//controller/userAccess.js
'use strict';

const { Controller } = require('egg');

/**
 * @Controller 用户权限管理
 */
class UserAcessContoller extends Controller {
  // eslint-disable-next-line no-useless-constructor
  constructor(ctx) {
    super(ctx);
  }

```

```
  /**
     * @summary 用户登录
     * @description 用户登录
     * @router post /auth/jwt/login
     * @request body loginRequest *body
     * @response 200 baseResponse 创建成功
     */
  async login() {
    const { ctx, service } = this;
    // ctx.body = 'user ctrl';
    // 校验参数
    ctx.validate(ctx.rule.loginRequest);

    // 组装参数
    const payload = ctx.request.body || {};
    // 调用Servic进行业务处理
    const res = await service.userAccess.login(payload);
    // 设置响应内容和响应状态码
    ctx.helper.success({ ctx, res });

  }


  /**
     * @summary 用户登出
     * @description 用户登出
     * @router post /auth/jwt/logout
     * @request body loginRequest *body
     * @response 200 baseResponse 创建成功
     */
  async logout() {
    const { ctx, service } = this;
    await service.userAccess.logout;
    // 设置响应内容和响应状态码
    ctx.helper.success({ ctx });

  }
}

module.exports = UserAcessContoller;

```

文件上传

```
npm i await-stream-ready stream-wormhole image-downloader -S
```

- controller

```
//app/controller/upload.js
'use strict';

const fs = require('fs');
const path = require('path');
const awaitWriteStream = require('await-stream-ready');
const sendToWormhole = require('stream-wormhole');
const download = require('image-downloader');
```

```

const { Controller } = require('egg');

/**
 * @Controller 用户权限管理
 */
class UploadContoller extends Controller {
  // eslint-disable-next-line no-useless-constructor
  constructor(ctx) {
    super(ctx);
  }

  /**
     * @summary 上传单个文件
     * @description 上传单个文件
     * @router post /api/upload/single
     */
  async create() {
    const { ctx, service } = this;
    // ctx.body = 'user ctrl';
    // 要通过ctx.getFileStream便捷的获取到用户上传的文件，需要满足俩个条件：
    // 只支持上传一个文件
    // 上传文件必须在所有其他的fileds后面，否则在拿到文件流时可能还获取不到fileds
    const stream = await ctx.getFileStream();
    // 所有表单字段都能通过`stream.fields`获取到
    const filename = path.basename(stream.filename);// 文件名称
    const extname = path.extname(stream.filename).toLowerCase();// 文件扩展名
    const uuid = (Math.random() * 999999).toFixed();
    // 组装参数stream
    const target = path.join(this.config.baseDir, 'app/public/uploads', `${uuid}${extname}`);
    const writeStream = fs.createWriteStream(target);
    try {
      await awaitWriteStream(stream.pipe(writeStream));
    } catch (error) {
      // 必须将上传的文件流消耗掉，要不然浏览器相应会卡死
      await sendToWormhole(stream);
      throw error;
    }
    ctx.helper.success({ ctx });

  }
}

module.exports = UploadContoller;

```

```
//upload.html
<!DOCTYPE html>
<html lang="en">

<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <meta http-equiv="X-UA-Compatible" content="ie=edge">
```

```
    <title>上传文件</title>
    <link rel="stylesheet" href="https://unpkg.com/element-ui/lib/theme-chalk/index.css">
    <style>
        .avatar-uploader .el-upload {
            border: 1px dashed #d9d9d9;
            border-radius: 6px;
            cursor: pointer;
            position: relative;
            overflow: hidden;
            width: 178px;
            height: 178px;
        }

        .avatar-uploader .el-upload:hover {
            border-color: #409EFF;
        }

        .avatar-uploader-icon {
            font-size: 28px;
            color: #8c939d;
            width: 178px;
            height: 178px;
            line-height: 178px;
            text-align: center;
        }

        .avatar {
            width: 178px;
            height: 178px;
            display: block;
        }

        .image-preview {
            width: 178px;
            height: 178px;
            position: relative;
            border: 1px dashed #d9d9d9;
            border-radius: 6px;
            float: left;
        }

        .image-preview .image-preview-wrapper {
            position: relative;
            width: 100%;
            height: 100%;
        }

        .image-preview .image-preview-wrapper img {
            width: 100%;
            height: 100%;
        }

        .image-preview .image-preview-action {
            position: absolute;
            width: 100%;
            height: 100%;
            left: 0;
```

```
            top: 0;
            cursor: default;
            text-align: center;
            color: #fff;
            opacity: 0;
            font-size: 20px;
            background-color: rgba(0, 0, 0, .5);
            transition: opacity .3s;
            cursor: pointer;
            text-align: center;
            line-height: 200px;
        }

        .image-preview .image-preview-action .el-icon-delete {
            font-size: 32px;
        }

        .image-preview:hover .image-preview-action {
            opacity: 1;
        }
    </style>
</head>

<body>
    <div id="app">

        <el-upload v-show="imageUrl.length < 1" class="avatar-uploader" :action="serverUrl" :show-file-list="false"
            multiple :before-upload="beforeUpload" :headers="token" :on-success="handleSuccess"
            :on-progress="uploadProcess">
            <i v-show="imageUrl =='' && imgFlag == false" class="el-icon-plus avatar-uploader-icon"></i>
            <el-progress v-show="imgFlag == true" type="circle" :percentage="percent" style="margin-top: 20px">
            </el-progress>
        </el-upload>

        <div class="image-preview" v-show="imageUrl.length > 1">
            <div class="image-preview-wrapper">
                <img :src="imageUrl">
                <div class="image-preview-action">
                    <i @click="handleRemove" class="el-icon-delete"></i>
                </div>
            </div>
        </div>

    </div>
</body>
<script src="https://unpkg.com/vue/dist/vue.js"></script>
<script src="https://unpkg.com/element-ui/lib/index.js"></script>
<script>
    new Vue({
        el: '#app',
        data() {
            return {
                serverUrl: "/api/upload/single", // 后台上传接口
```

```
                token: {},
                imgFlag: false,
                percent: 0,
                imageUrl: '',
            };
        },
        mounted() {},
        methods: {
            handleRemove(file, fileList) {
                this.imageUrl = '';
            },
            beforeUpload(file) {
                const isLt10M = file.size / 1024 / 1024 < 10;
                if (
                    ['image/jpeg',
                        'image/gif',
                        'image/png',
                        'image/bmp'
                    ].indexOf(file.type) == -1) {
                    this.$message.error('请上传正确的图片格式');
                    return false;
                }
                if (!isLt10M) {
                    this.$message.error('上传图片不能超过10MB哦!');
                    return false;
                }
                // 设置认证信息
                const token = window.localStorage.getItem("token")
                this.token["Authorization"] = "Bearer " + token
            },
            handleSuccess(res, file) {
                this.imgFlag = false;
                this.percent = 0;
                if (res) {
                    this.imageUrl = URL.createObjectURL(file.raw); //
项目中用后台返回的真实地址
                } else {
                    this.$message.error('视频上传失败，请重新上传！');
                }
            },
            uploadProcess(event, file, fileList) {
                this.imgFlag = true;
                console.log(event.percent);
                this.percent = Math.floor(event.percent);
            },
        }
    })
</script>

</html>
```

ts项目结构

项目结构

1.package.json创建：npm init -y

2.开发依赖安装:npm i typescript ts-node-dev tslint @types/node -D

3.启动脚本

```
"scripts": {
    "start": "ts-node-dev ./src/index.ts -P tsconfig.json --no-cache",
    "build": "tsc -P tsconfig.json &&node ./dist/index.js",
    "tslint": "tslint --fix -P tsconfig.json"
  },
```

4.加入tsconfig.json

```
{
  "compilerOptions": {
      "outDir": "./dist",
      "target": "es2017",
      "module": "commonjs",//组织代码方式
      "sourceMap": true,
      "moduleResolution": "node",//模块解决策略
      "experimentalDecorators": true,//开启装饰器定义
      "allowSyntheticDefaultImports": true,//允许es6方式导入
      "lib": ["es2015"],
      "typeRoots": ["./node_modules/@types"],
  },
  "include": ["src/**/*"]
}
```

5.创建入口文件,./src/index.ts

```
console.log('hello')
```

6.运行测试:npm start

项目基础代码

1.安装依赖:npm i koa koa-static koa-body koa-xtime -S

2.编写基础代码,index.ts

```
import * as Koa from "koa";
import * as bodify from "koa-body";
import * as serve from "koa-static";
import * as timing from "koa-xtime";

const app=new Koa();


app.use(serve(`${__dirname}/public`))

app.use(bodify({
    multipart:true,
    strict:true
```

```
}))

app.use((ctx:Koa.Context)=>{
    ctx.body='hello'
})

app.use(router.routes())
app.listen(3000,()=>{
    console.log('服务器的启动')
})
```

3.测试：npm start

路由定义及发现

1.创建路由./src/routes/user.ts

```
import * as Koa from "koa";
import { get, post } from '../utils/route-decors'

const users = [{ name: 'tom', age: 20 }];

export default class User {

    @get('/users')
    public async list(ctx: Koa.Context) {
        // ctx.body={ok:1,data:users}
        const users = await model.findAll();
        ctx.body = { ok: 1, data: users }
    }

    @post('/users', {
        middlewares: [
            async (ctx, next) => {
                const name = ctx.request.body.name
                //用户名必须
                if (!name) {
                    throw '请输入用户名'
                } else {
                    await next();
                }
            }
        ]
    })
    public add(ctx: Koa.Context) {
        users.push(ctx.request.body);
        ctx.body = { ok: 1 }
    }
}
```

知识点补充:装饰器的编写，以@get('user')为例，它是函数装饰器且有配置项，其函数签名为:

```
function get(path){
return function(target,property,descriptor){}
}
```

另外需解决俩个问题：

1.路由发现

2.路由注册

2.路由发现及注册，创建./utils/route-decors.ts

```
import * as Koa from "koa";
import * as KoaRouter from "koa-router";
import * as glob from "glob";
import koaBody = require("koa-body");

type HTTPMethod = 'get' | 'put' | 'del' | 'post' | 'patch'
const router = new KoaRouter();
type LoadOptions = {
    /**
     * 路由文件扩展名，默认值是`.{js,ts}`
     */
    extname?: string
}

type RouteOptions = {
    /**
     * 适合用于某个请求比较特殊，需要单独制定前缀的情形
     */
    prefix?: string;
    /**
     * 给当前路由添加一个或多个中间件
     */
    middlewares?: Array<Koa.Middleware>;
}
//1.
// export const get=(path:string,options?:RouteOptions)=>{
//     return (target,property,descriptor)=>{
//         const url=options&&options.prefix?options.prefix+path:path;
//         router['get'](url,target[property])
//     }
// }

//2.
// export const method=method=>(path:string,options?:RouteOptions)=>{
//     return (target,property,descriptor)=>{
//         const url=options&&options.prefix?options.prefix+path:path;
//         router[method](url,target[property])
//     }
// }

//3.
export const decorate = (method: HTTPMethod, path: string, options:
RouteOptions = {}, router: KoaRouter) => {
    return (target, property, descriptor) => {
            const url = options && options.prefix ? options.prefix + path :
path;
            router[method](url,target[property])

        })

```

```
}
export const method = method => (path: string, options?: RouteOptions) =>
decorate(method, path, options, router)

export const get = method('get')
export const post = method('post')
export const put = method('put')
export const del = method('del')
export const patch = method('patch')

export const load = (floder: string, options: LoadOptions = {}): KoaRouter
=> {
    const extname = options.extname || '.{js,ts}'
    glob.sync(require('path').join(floder, `./**/*${extname}`)).forEach(item
=> {
        require(item)
    });
    return router;
}
```

3.使用

routes/user.ts

```
import { get, post} from '../utils/route-decors'
```

index.ts

```
import { load } from "./utils/route-decors";
import { resolve } from "path";
const router=load(resolve(__dirname,'./routes'));
app.use(router.routes())
```

4.数据校验：可以利用中间件机制实现

添加校验函数,./routes/user.ts

```
export default class User {

    @get('/users')
    public async list(ctx: Koa.Context) {
        ctx.body={ok:1,data:users}
        // const users = await model.findAll();
        // ctx.body = { ok: 1, data: users }
    }

    @post('/users', {
        middlewares: [
            async (ctx, next) => {
                const name = ctx.request.body.name
                //用户名必须
                if (!name) {
                    throw '请输入用户名'
```

```
                } else {
                    await next();
                }
            }
        ]
    })
    public add(ctx: Koa.Context) {
        users.push(ctx.request.body);
        ctx.body = { ok: 1 }
    }
}
```

更新decors.ts

```
export const decorate = (method: HTTPMethod, path: string, options: RouteOptions = {}, router: KoaRouter) => {
    return (target, property, descriptor) => {
            //添加中间件数据
            const middlewares = [];

            if(target.middlewares){
                middlewares.push(...target.middlewares)
            }

            //若设置了中间件选项则加入到中间件数组
            if (options.middlewares) {
                middlewares.push(...options.middlewares)
            }

            //添加路由处理器
            middlewares.push(target[property])

            const url = options && options.prefix ? options.prefix + path : path;
            // router[method](url,target[property])
            router[method](url, ...middlewares)
    }
}
```

5.类级别路由守卫

使用routes/user.ts

```
@middlewares([
    async (ctx, next) => {
        console.log('guard', ctx.header)
        if (ctx.header.token) {
            await next()
        } else {
            throw '请登录'
        }
    }
])
export default class User {

    @get('/users')
    public async list(ctx: Koa.Context) {
```

```
        ctx.body={ok:1,data:users}
        // const users = await model.findAll();
        // ctx.body = { ok: 1, data: users }
    }

    @post('/users', {
        middlewares: [
            async (ctx, next) => {
                const name = ctx.request.body.name
                //用户名必须
                if (!name) {
                    throw '请输入用户名'
                } else {
                    await next();
                }
            }
        ]
    })
    public add(ctx: Koa.Context) {
        users.push(ctx.request.body);
        ctx.body = { ok: 1 }
    }
}
```

添加中间装饰器，更新route-decors.ts

```
export const decorate = (method: HTTPMethod, path: string, options: RouteOptions = {}, router: KoaRouter) => {
    return (target, property, descriptor) => {
        //晚一拍执行路由注册：因为需要等类装饰器执行完毕
        process.nextTick(() => {
            //添加中间件数据
            const middlewares = [];

            if(target.middlewares){
                middlewares.push(...target.middlewares)
            }

            //若设置了中间件选项则加入到中间件数组
            if (options.middlewares) {
                middlewares.push(...options.middlewares)
            }

            //添加路由处理器
            middlewares.push(target[property])

            const url = options && options.prefix ? options.prefix + path : path;
            // router[method](url,target[property])
            router[method](url, ...middlewares)
        })


    }
}


```

```
export const middlewares =(middlewares: Koa.Middleware[]) =>{
    return (target) => {
        target.prototype.middlewares = middlewares;
    }
}

```

数据库整合

1.安装依赖:npm i sequelize sequelize-typescript reflect-metadata mysql2 -S

2.初始化，index.ts

```
import { Sequelize } from "sequelize-typescript";

const database=new Sequelize({
    port:3306,
    database:'zyl-kaikeba',
    username:'root',
    password:'yw@910714',
    dialect:'mysql',
    modelPaths:[`${__dirname}/model`]
})

database.sync({force:true})

```

3.创建模型

```
//model/user.js
import { Table,Column,Model,DataType } from "sequelize-typescript";
@Table({modelName:'users'})
export default class User extends Model<User>{
    @Column({
        primaryKey:true,
        autoIncrement:true,
        type:DataType.INTEGER
    })
    public id:number;
    @Column(DataType.CHAR)
    public name:string
}
```

4.使用模型，routes/user.ts

```
import * as Koa from "koa";
import { get, post, middlewares } from '../utils/route-decors'
import model from "../model/user";

const users = [{ name: 'tom', age: 20 }];
@middlewares([
    async (ctx, next) => {
        console.log('guard', ctx.header)
        if (ctx.header.token) {
            await next()
        } else {
```

```
            throw '请登录'
        }
    }
])
export default class User {

    @get('/users')
    public async list(ctx: Koa.Context) {
        // ctx.body={ok:1,data:users}
        const users = await model.findAll();
        ctx.body = { ok: 1, data: users }
    }

    @post('/users', {
        middlewares: [
            async (ctx, next) => {
                const name = ctx.request.body.name
                //用户名必须
                if (!name) {
                    throw '请输入用户名'
                } else {
                    await next();
                }
            }
        ]
    })
    public add(ctx: Koa.Context) {
        users.push(ctx.request.body);
        ctx.body = { ok: 1 }
    }
}
```

部署_nginx_pm2_docker

参考文档

https://yeasy.gitbook.io/docker_practice/

使用pm2+nginx部署koa2(https)https://www.zhaofinger.com/detail/5

构建一个高可用的node环境

主要解决问题

- 故障恢复‘
- 多核利用
- https://www.sohu.com/a/247732550_796914
- 多进程共享端口
-
-

参考资料

https://github.com/su37josephxia/compose-awesome 夏老师的开源社区compose https://www.jianshu.com/p/6af3846dbe68 window开启linux 子系统

http://fex.baidu.com/webuploader/ webuploader https://github.com/su37josephxia/frontend-basic 前端基础知识汇总 https://juejin.im/post/6844904116972421128 然叔http掘金缓存机制

https://www.processon.com/view/link/5ec52841e0b34d5f261e14e0#map http协议图

https://github.com/su37josephxia/compose-awesome 夏老师的compose git

https://github.com/su37josephxia/smarty-press 高效的 Markdown 网站制作工具 基于vue3

https://www.processon.com/view/link/5d661d8fe4b0145255caa5c9#map 学习方法论

https://www.juejin.im/post/6875236411349008398/ vue3的观后感

https://juejin.im/post/6844904196848762888 GraphQL