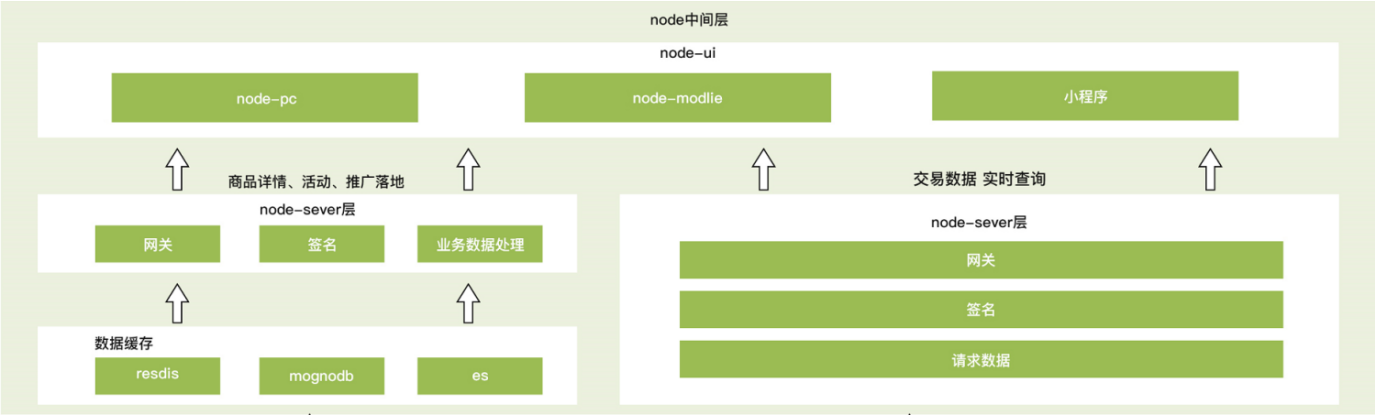
Node中间层的应用

Node中间层的作用

NodeJS中间层是面向端的后台接口系统，是介于端与平台之间的薄数据层，提供内外网接口供端上或开发平台调用，主要用来实现前后端分离，对数据进行二次加工，包括拼接转换和过滤，它在业务中的角色如下图所示：



Node中间层专为Web提供定制化接口整合服务，各个后端业务方只需要提供原始抽象数据接口，由中间层去整合，这样前端只需要请求一次中间层接口就获取到适合展现的数据，无需浏览器端进行多次处理，具体功能如下图

Egg.js

比如Express 是 Node.js 社区广泛使用的框架，简单且扩展性强，非常适合做个人项目。但框架本身缺少约定，标准的 MVC 模型会有各种千奇百怪的写法。Egg 按照约定进行开发，奉行『约定优于配置』，团队协作成本低。

Egg.js也是国内比较成熟的Node.js企业级框架，他的开放性也意味着Egg.js即拥有可靠的稳定性，也拥有高扩展性。而且Egg是基于Koa的，所以同理天然的支持

开发工具

关于Node.js的IDE和编辑器有很多选择，对比如下

名称	是否收费	断点调试	功能
Webstorm	收费	支持	是IDE，在代码提示、重构等方面功能非常强大，支持的各种语言、框架、模板也非常多，支持断点调试，好处是特别智能，缺点也是特别智能
Sublime/TextMate	收费	不支持	编辑器里非常好用的，textmate主要针对mac用户，sublime是跨平台的，相信很多前端开发都熟悉
Vim/Emace	免费	不支持	命令行下的编辑器，非常强大，难度也稍大，但更为酷炫，而且对于服务器部署开发来说是值得一学的
VSCode/Atom	免费	支持	Atom比较早，功能强大，缺点稍卡顿，VSCode是微软出的，速度快，对于Node.js 调试，重构，代码提示等方面支持都非常好

推荐开发工具

Webstorm

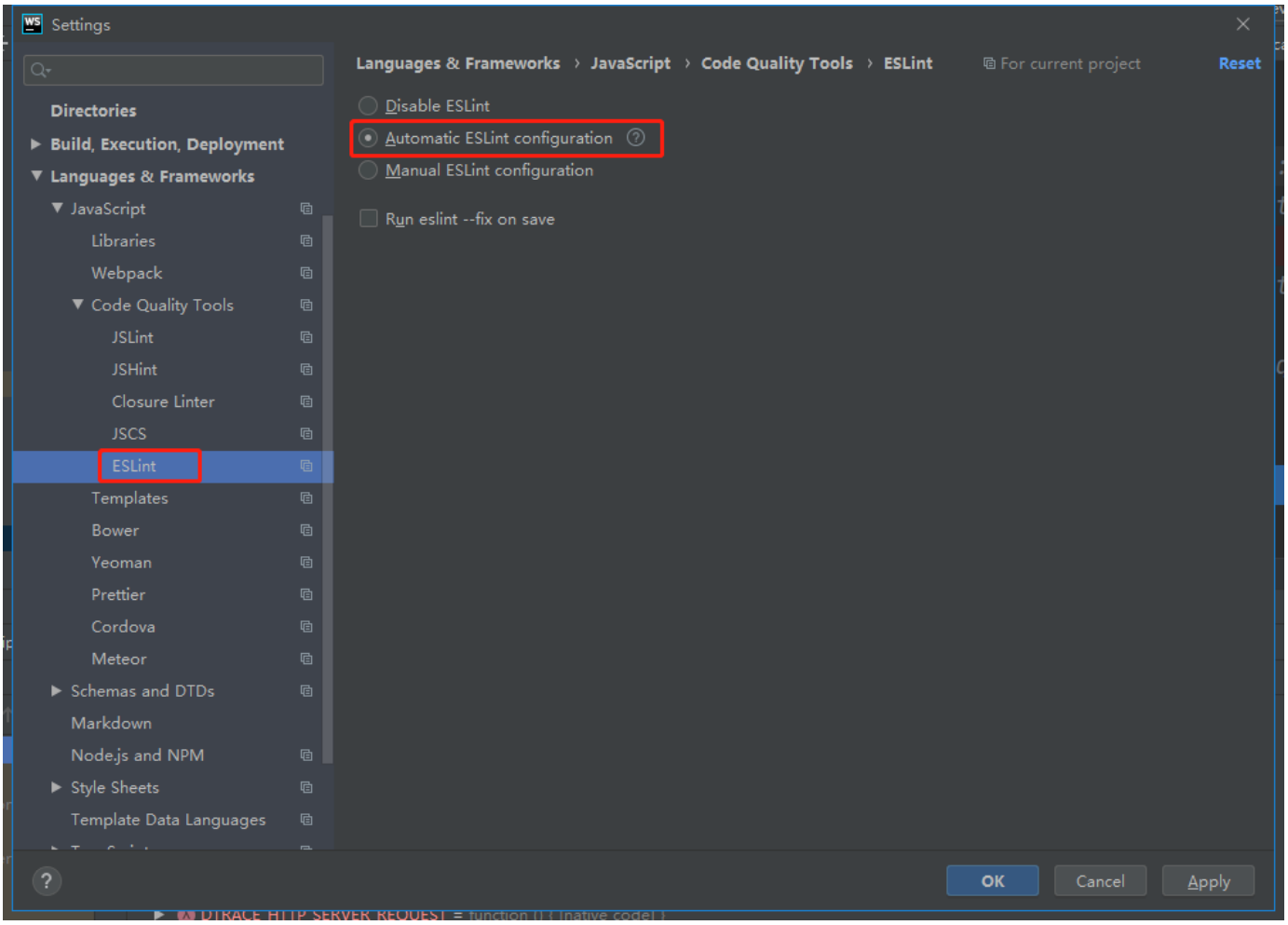
Webstorm是一款非常强大的IED，就是上面所说，他对代码提示和重构等方面应该是市场上最强大、最智能的一款前端开发工具。平常开完前端，可能只使用了10%还不到的功能，他的强大在于Node.js的支持。支持各种断点和调试方式。

VSCode

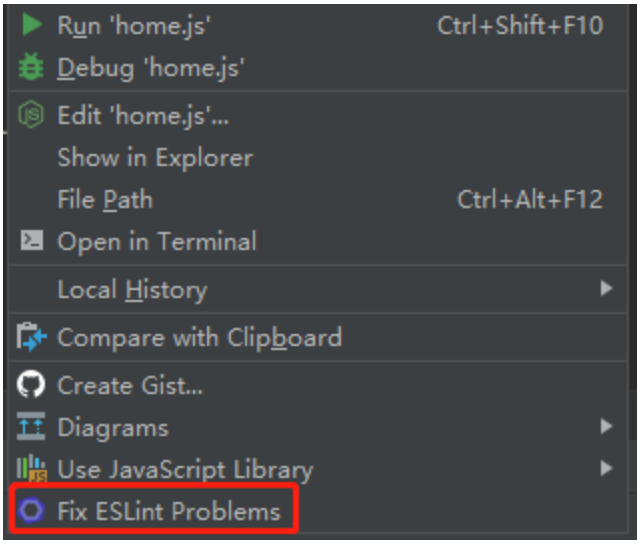
Visual Studio Code是一个运行于 Mac、Windows和 Linux 之上的，针对于编写现代 Web 和云应用的跨平台源代码编辑器。它功能强大，便于调试，加上它本身也是基于 Node.js 模块 electron 构建的。

语法风格检查

Egg内置了ESLint语法风格检测工具，我们在进行正式编码前，一定要开启各自使用的IDE的语法检查。以WebStorm为例开启ESLint语法检测：



既然开启语法检测，那么如何进行根据ESLint自动格式化代码呢？以WebStorm为例，可以只要开启ESLint语法检测后，可以直接右击鼠标，选择"Fix ESLint Problems"进行格式化。



Node.js常用调试

大家在开发客户端应用程序时，最常用的调试手段无疑是使用console.log打印，和浏览器断点调试。有一些基础比较好的同学，可能会使用debugger关键字进行断点调试（特别是现在使用框架进行工程化开发，浏览器断点出现不好打的情况）。

console.log

大家在前端开发时，console.log应该是最常用的调试方式。那么在后端服务器依然可以采用这种方式。但是这是最基础的方式，不能还原整个后台应用程序的执行逻辑。

debugger标签

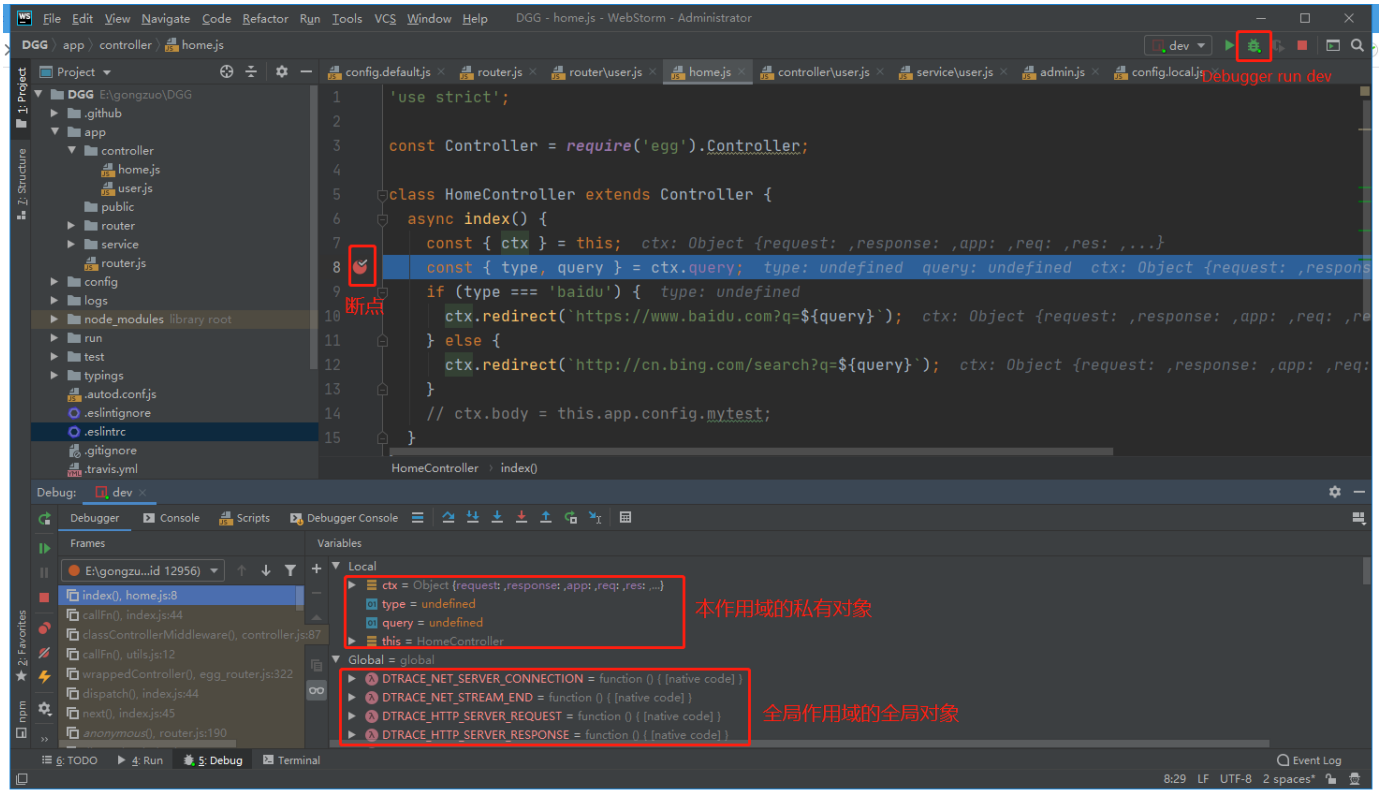
Nodejs提供了一个内建调试器来帮助开发者调试应用程序。想要开启调试器我们需要在代码中加入debugger标签，当Nodejs执行到debugger标签时会自动暂停（debugger标签相当于在代码中开启一个断点）。

```
1 //debug.js
2 var hello = 'hello';
3 var world = 'nodejs';
4 debugger;
5 var hello_world = hello + ' ' + world;
6 console.log(hello_world);
```

执行命令：node debug debug.js 就可以进入调试模式

当然，首先需要在程序代码中手动添加中断debugger;， 这样当以调试模式运行时，程序会自动中断，然后等候你调试。

WebStorm中调试



注意：打完断点一定要使用"debug dev"进入断点模式，Webstorm中的断电模式和Chrome断点非常类似。

VSCode与其他调试

VSC的调试方式也非常简单，和WebStorm类似，详细说明见狼叔的文档

其他还有很多的调试方式，可以参照狼叔的3法3例2模式的相关Node.js调试方式说明。

严格模式

在Node.js应用程序中，他是运行在后端服务器，他对代码的严谨性、规范性也非常高。所以在今后开发Node.js相关程序时，必须使用ES（ECMAScript）严格模式。在每个.js文件顶部必须书写"use strict"关键字。

```
1 'use strict';
```

快速开始

安装EGG

```
1 $ mkdir egg-example && cd egg-example
2 $ npm init egg --type=simple
3 $ npm i
```

启动EGG

```
$ npm run dev
$ open http://localhost:7001
```

正式环境部署

```
$ npm start
$ npm stop
```

项目配置

多环境配置

EGG运行环境分为本地开发环境（local）、测试环境（unittest）、和产线环境（prod），EGG支持根据环境来加载配置，定义多个环境的配置文件。

```
config
|- config.default.js
|- config.prod.js
|- config.unittest.js
`- config.local.js
```

config.default.js是默认配置文件，比如现在是正式环境（prod），那么EGG会加载config.default.js和config.prod.js这两个配置文件。假如现在是本地开发环境（local）那么加载的配置文件就是config.default.js和config.local.js这两个配置文件。

注意：在今后的Node中间层开发当中，共用配置放在config.default.js中，关于和环境有关的配置都放在每个环境对应的配置文件中。只需手动创建对应名称的配置文件即可，不同环境会自动加载，无需程序员去干预和进行其他设置。

项目中获取配置信息

我们可以通过 app.config 从 Application 实例上获取到 config 对象，也可以在 Controller, Service, Helper 的实例上通过 this.config 获取到 config 对象。

比如在contorller中获取配置文件中的配置项，在Service中也是一样。

```
'use strict';
const Controller = require('egg').Controller;
class HomeController extends Controller {
  async index() {
    const { ctx } = this;
    ctx.body = this.app.config;
  }
}
module.exports = HomeController;
```

配置文件加载顺序

一个配置文件大概分为应用 > 框架 > 插件三个部分，配置文件之间有加载顺序，文件中的配置项也有加载顺序。

```
-> 插件 config.default.js
-> 框架 config.default.js
-> 应用 config.default.js
-> 插件 config.prod.js
-> 框架 config.prod.js
-> 应用 config.prod.js
```

注意：假如两个配置文件出现相同的配置项，那么后加载的配置会覆盖先加载的配置（实际开发中，一定要避免相同配置的出现，否则将造成非常奇怪的且不报错的BUG）。

案例

我们这里可以将配置文件获取和配置问价加载顺序做一个案例。可以分别在默认配置(config.default.js)和本地开发环境(config.local.js)写入相同的配置，我们来看他们的输出结果。

```
//config.default.js
'use strict';
module.exports = appInfo => {
  const config = exports = {};
  config.mytest='默认测试';
  const userConfig = {};
  return {
    ...config,
    ...userConfig,
```

```
  };
};
```

```
//config.prod.js
'use strict';
module.exports = appInfo => {
  const config = exports = {};
  config.mytest='本地测试';
  const userConfig = {};
  return {
    ...config,
    ...userConfig,
  };
};
```

```
'use strict';
const Controller = require('egg').Controller;
class HomeController extends Controller {
  async index() {
    const { ctx } = this;
    ctx.body = this.app.config.mytest;//响应"本地测试"
  }
}
module.exports = HomeController;
```

MVC模式

Egg作为一个企业级的Node.js框架，它采用的是MVC模式，主要有视图(view)、控制器（Controller）、服务（Service）。因为我们会使用Nuxt.js来进行视图层（View）的渲染，所以MVC中对于Node中间层，我们只需要关注MC两个部分。接下来我们会来看MC如何编写。

路由（Router）

Router 主要用来描述请求 URL 和具体承担执行动作的 Controller 的对应关系， 框架约定了 app/router.js 文件用于统一所有路由规则。

```
module.exports = app => {
  const { router, controller } = app;
```

```
  router.get('/newsList', controller.news.list);
};
```

这里就描述了"/newsList"这个URL在GET请求方式下，调用控制器（contorller）news集合下的list方法。这就是一个完整的路由地址。

news是controller文件夹下的news.js文件,list是news.js文件中Class类中的一个属性方法。

路由重定向

内部重定向：router.redirect(来源路径，定向到的路径，[状态码（301）]);

我们可以在router中直接使用router对象，进行重定向，状态码可有可无，不填会默认为301。

```
module.exports = app => {
  const { router, controller } = app;
  require('./router/user')(app)
  router.redirect('/', '/user', 302);
};
```

还可以在Controller和service中使用ctx上下文对象进行重定向。

```
'use strict';
const Controller = require('egg').Controller;
class HomeController extends Controller {
  async index() {
    const { ctx } = this;
    ctx.redirect('/user');
  }
}
module.exports = HomeController;
```

外部重定向

```
'use strict';

const Controller = require('egg').Controller;
class HomeController extends Controller {
  async index() {
```

```
    const { ctx } = this;
     const { type, query } = ctx.query;
     if (type === 'baidu') {
       ctx.redirect(`https://www.baidu.com?q=${query}`);
     } else {
       ctx.redirect(`http://cn.bing.com/search?q=${query}`);
    }
    ctx.body = this.app.config.mytest;
  }
}

module.exports = HomeController;
```

注意：一般会在Router和Controller层会涉及到路由重定向，Service层是用于业务数据处理的抽象，他会被多次使用和调用，建议不要将路由重定向放到Service。

路由映射

思考：上边我们说到了，在app/router.js中定义所有的路由规则，那么假如一个非常庞大的项目，比如像咱们"薯片APP"，除了C端的PC和WAP会通过Node中间层，还有APP会通过Node中间层。如此庞大的项目，涉及到的接口也会非常多。这么多的路由都在router.js一个JS文件中定义，后期我们该怎么维护呢?

路由映射严格意义上说的是路由分组，一个Web应用中的路由件往往非常多，为了方便管理，可以根据功能划分为不同的路由模块中，在此模块绑定对应的控制器（controller）。熟悉Express的同学一定知道路由级中间件，下面我就看看在Egg中如何实现路由分组。

```
//router/user.js
module.exports=app=>{
    const { router, controller } = app;
    router.get('/user',controller.user.list)
}
```

我们在app目录下，创建router目录管理每个模块的路由规则，这是一段"用户"相关的路由规则，那么我们可以怎么样将他融合到app目录下router.js（因为上文也说到了，所有的路由规则必须在app/router.js中定义，他是所有的路由入口 ）的路由规则中?

了解Node.js的同学应该知道require()的作用，他可以导入属性和方法，并且可以传递参数（更多关于此知识点的Node.js知道可见模块的封装 和整个Node.js模块系统概述）。

```
//app/router.js
```

```
module.exports = app => {
  const { router, controller } = app;
  router.get('/', controller.home.index);
  require('./router/user')(app)
};
```

我们可以导入router文件夹下的user.js关于用户相关路由的规则，并且将app（Application）全局对象传递给user.js这个文件系统。在user.js中的路由规则写法是和router.js是完全一样的。

控制器（Controller）

控制器主要功能就是接受用户请求和响应请求结果。刚才也提到了，路由的请求会到控制器，控制器接收到客户端请求，以及客户端提交的数据。控制器会根据请求调用服务(service)，来处理业务相关的数据结果。

```
'use strict'

const  Controller =require('egg').Controller

class NewsController extends Controller{
    async list(){
        const { ctx, service } = this;
        const query = ctx.query;
        const newsList = await service.news.find(query.id)
        ctx.body=newsList
    }
}

module.exports=NewsController
```

GET参数获取

我们都知道GET请求是通过URL查询字符串进行传参的，那么要在控制器中获取GET请求的参数一样需要在query中去获取。

```
class PostController extends Controller {
  listPosts() {
    const query = this.ctx.query;
    consoloe.log('获取到的GET请求参数为：',query)
  }
```

```
}
```

POST参数获取

EGG内置了bodyParse中间件（这里为什么要使用中间件，需要了解详细？），所以我们在获取POST参数时，可以直接通过body进行获取。

```
class PostController extends Controller {
  listPosts() {
    const body = this.ctx.request.body
    console.log('获取到的POST请求参数为：',body)
  }
}
```

更多参数获取方式参考Egg官方文档。

服务（Service）

服务（service）主要负责业务逻辑的处理，比如我们获取一组复杂的业务数据，前端提交的一组复杂的业务数据，都需要在service中进行处理。这使得控制器(controller)的逻辑会更加简洁和清晰。一般抽象出来的Service是会被控制器多次调用的。

通常会在复杂的数据的获取处理、第三方服务的调用中使用到Service。

```
// app/service/user.js
const Service = require('egg').Service;
class UserService extends Service {
  async find(uid) {
    // 假如 我们拿到用户 id 从数据库获取用户详细信息
    const user = await this.ctx.db.query('select * from user wher
e uid = ?', uid);
    // 假定这里还有一些复杂的计算，然后返回需要的信息。
    const picture = await this.getPicture(uid);
    return {
      name: user.user_name,
      age: user.age,
      picture,
    };
  }
}
module.exports = UserService;
```

案例

```
//router.js
module.exports = app => {
  const { router, controller } = app;
  router.get('/newsList', controller.news.list);
};
```

```
//controller/news.js
'use strict'
const  Controller =require('egg').Controller

class NewsController extends Controller{
    async list(){
        const { ctx, service } = this;
        const query = ctx.query;
        const newsList = await service.news.find(query.id)
        ctx.body=newsList
    }
}

module.exports=NewsController
```

```
//service/news.js
'use strict'
const Service =require('egg').Service

class NewsService extends Service{
    async find(id){
        const user = {id,dataList:[{name:'汤姆',age:12},{ name:'玛利亚',age:32}]}
        return user
    }
}
```

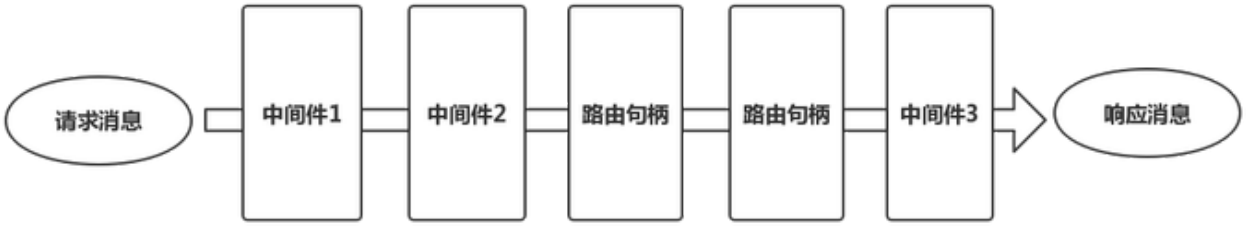
中间件

中间件（Middleware）是一个函数，它可以读取请求消息数据，并处理响应消息对象，并与应用中其他中间件交互。

具体来说，中间件可以实现如下功能：

（1）执行任何代码；
（2）修改请求和响应对象；
（3）终结请求-响应循环；
（4）调用堆栈中的下一个中间件

可以在路由句柄之前或之后声明多个中间件函数，组成一种栈式结构；
请求消息会按照声明顺序依次提交给每个中间件函数；
中间件或路由句柄内部可以控制next( )回调函数的执行以决定是否继续执行下一个中间件函数：



插件

使用插件

Egg插件是一组相对独立的业务逻辑的抽象，他和中间件不一样，中间件的定位是用于用户请求的拦截处理。比如我们需要在程序启动就加载初始化一段业务程序，中间件显然不能满足我们的需求。

使用npm下载插件

```
1 npm i egg-redis --save
```

启用插件

我们可以在config/plugin.js中启用egg-redis插件：

```
1 module.exports = {
```

```
  // 启用redis插件,并指定使用的包名
  redis: {
    enable: true,
    package: 'egg-redis',
  },
};
```

配置插件

我们在Egg中一提到配置可能就会想到config/config.default.js和其他环节的config.<环境>.js。所以我们在config.js中进行插件对应的配置。

```
config.redis = {
    client: {// 可以配置多Redis节点
      host: '127.0.0.1', // IP地址
      port: '6379', // 端口号
      family: 'root', // 用户名
      password: '', // 用户密码
      db: '0', // 数据库名称,redis默认16个数据库0-16
    },
}
```

使用插件

插件到此可以，插件的方法就已经在应用程序启动时，挂在到this.app(Application)全局对象中。可以直接使用this.app.redis进行使用了。

```
module.exports = app => {
  return class HomeController extends app.Controller {
    async index() {
      const { ctx, app } = this;
      // set
      await app.redis.set('foo', 'bar');
      // get
      ctx.body = await app.redis.get('foo');
    }
  };
};
```

错误处理

Node.js中错误处理也需要非常严谨，在以往客户端程序中，一个方法报错，顶多是部分功能异常。由于Node.js单线程的原因，一旦Node.js出现错误，会导致整个服务瘫痪。所以错误处理是重中之重，一旦出现报错还未处理的话，一定会造成产线事故。

使用if判断

所有的Node.js原生模块，他们关于回调时，都会至少有两个参数。第一个一定是err错误对象，第二个参数一般是回调结果值。所以在使用时必须使用if判断err错误对象，并且做错误处理。当没有错误时，err===null。

```
fs.readFile(path.resolve(__dirname, './../../logs/DGG/egg-wb.log')
, (err, data) => {
  if (err) throw err;
  console.log(data)
});
```

使用try catch

假如现在我们不是在使用Node.js原生模块，这段代码有出现异常的可能。亦或者，我们在使用async和await时，promise对象返回的错误我们依然无法捕获。那么我们可以使用try catch来进行错误处理。

```
'use strict';
const Service = require('egg').Service;
const fs = require('fs');
const path = require('path');

class UserService extends Service {
  async list(id) {
    try {
      const data = await new Promise((resolve, reject) => {
        fs.readFile(path.resolve(__dirname, './../../logs/DGG/egg
-wb.log'), (err, data) => {
          if (err) reject(err);
          resolve(data);
        });
      });
      return { id, data: data.toString() };
    } catch (err) {
```

```
      console.log(err);
      return { id, data: '出错啦' };
    }
  }
}
module.exports = UserService;
```

logger对象的使用

注意：在后端开发者错误信息一律不能使用console.log打印。因为后端一旦报错，涉及到整个前端可能无法访问。不统一进行错误收集，将无法进行错误排查。

Egg内置日志分类：

logger分类	对应的logs文件夹下的文件	说明
appLogger	example-app-web.log	应用相关日志，供应用开发者使用的日志。我们在绝大数情况下都在使用它。
coreLogger	egg-web.log	框架内核、插件日志。
errorLogger	common-error.log	实际一般不会直接使用它，任何 logger 的 .error() 调用输出的日志都会重定向到这里，重点通过查看此日志定位异常。
agentLogger	egg-agent.log	agent 进程日志，框架和使用到 agent 进程执行任务的插件会打印一些日志到这里。

Egg内置日志方法：

打印方法	说明
logger.debug()	生产环境默认无法打印debug级日志
logger.info()	
logger.warn()	
logger.error()	错误日志记录，直接会将错误日志完整堆栈信息记录下来，并且输出到 errorLog 中，为了保证异常可追踪，必须保证所有抛出的异常都是 Error 类型，因为只有 Error 类型才会带上堆栈信息，定位到问题。

```
'use strict';
```

```
const Service = require('egg').Service;
const fs = require('fs');
const path = require('path');

class UserService extends Service {
  async list(id) {
    const { ctx } = this;
    const { logger } = ctx;
    try {
      const data = await new Promise((resolve, reject) => {
        fs.readFile(path.resolve(__dirname, './../../logs/DGG/egg-wb.log'), (err, data) => {
          if (err) reject(err);
          resolve(data);
        });
      });
      return { id, data: data.toString() };
    } catch (err) {
      logger.warn(err);
      return { id, data: '出错啦' };
    }
  }
}
module.exports = UserService;
```

注意：开发者主动抛出的错误日志都记录在了"app/logs/DGG/DGG-web.log"日志文件中。

相关配置

日志切割

```
'use strict';
const path = require('path');
module.exports = appInfo => {
  const config = exports = {};
  config.logrotator = {
    filesRotateBySize: [
      path.join(appInfo.root, 'logs', appInfo.name, 'DGG-web.log'),
```

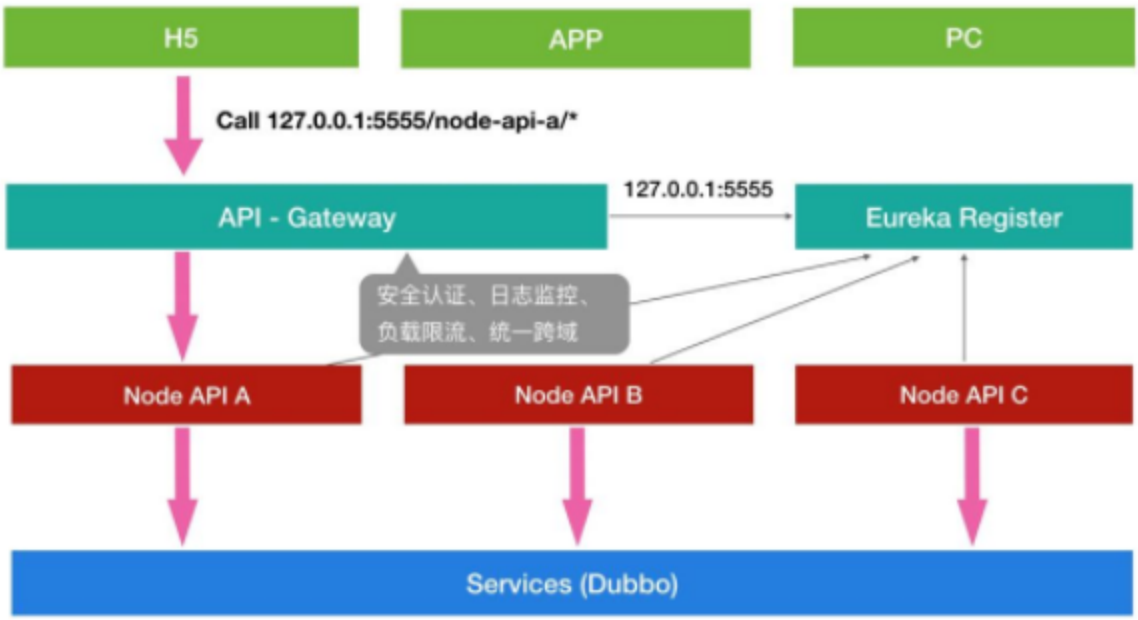
```
    ],
    // 设置日志文件需要切割的大小(kb为单位),超过配置大小(2GB),就会进行切割
    // 设置了按文件大小切割后,按时间切割的日子文件将不生效,都会安装文件大小切割
    maxFileSize: 2 * 1024 * 1024 * 1024,
  };
  // 在此处添加个人配置
  const userConfig = {
    // myAppName: 'egg',
  };
  return {
    ...config,
    ...userConfig,
  };
};
```

接入Eureka注册中心（手动）

Eureka是什么

在前后端分离架构中，服务层被拆分成了很多的微服务，微服务的信息如何管理？Spring Cloud中提供服务注册中心来管理微服务信息。微服务数量众多，要进行远程调用就需要知道服务端的ip地址和端口，注册中心帮助我们管理这些服务的ip和端口。更多关于eureka的文档可参考《Eureka工作原理》。

我们的Node API服务都需要统一接入网关，在Egg中我们使用的是egg-eureka-plugin（基于eureka-js-client包）模块，能够快速的能够让我们接入网关。



下载egg-eureka-plugin

```
npm i egg-eureka-pro --save
```

使用插件

```
// {app_root}/config/plugin.js
exports.eureka = {
  enable: true,
  package: 'egg-eureka-pro',
};
```

配置网关

因为eureka注册是基于eureka-js-client，所以更多相关详细配置查看相关文档。

```
// eureka中后端API节点集群的实例名称
  config.apiClient = {
    APPID: 'dgg-tac-msgsenter-channel',
  };
```

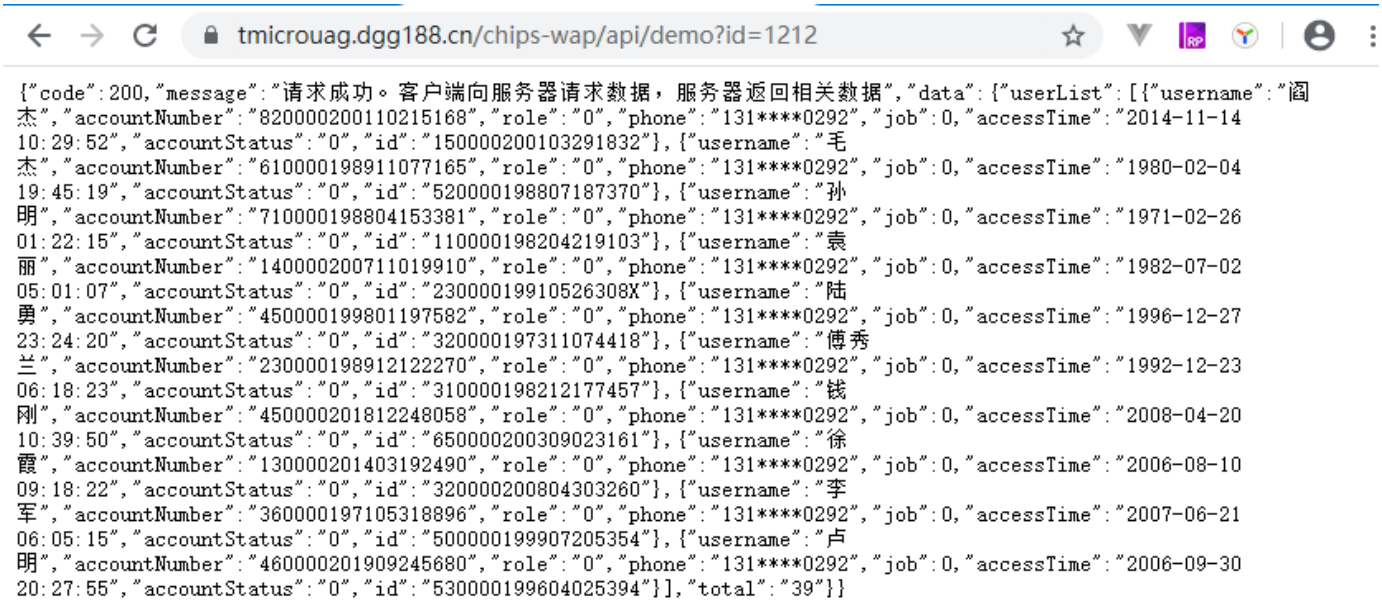
```
  // eureka相关配置
  config.eureka = {
    instance: {
      app: 'chips-wap',
      instanceId: `${getIPAdress()}:7001`, // 本地IP和端口
      hostName: getIPAdress(),
      ipAddr: getIPAdress(),
      port: {
        $: 7001,
        '@enabled': 'true',
      },
      homePageUrl: null,
      statusPageUrl: `http://${getIPAdress()}:7001/`, // 状态页面(判断心跳),
      healthCheckUrl: null,
      vipAddress: 'chips-wap',
      dataCenterInfo: {
        '@class': 'com.netflix.appinfo.InstanceInfo$DefaultDataCenterInfo',
        name: 'MyOwn',
      },
    },
    eureka: {
      servicePath: '/eureka/apps/',
      host: '192.168.254.27',
      port: 39817,
    },
  };
```

成功验证

假如运行无误，出现以下提示则意味着eureka注册中心接入成功。

```
2020-09-25 17:39:06,400 INFO 15308 registered with eureka:  chips-wap/172.16.132.78:7001
```

我们就可以直接通过网关地址进行对我们页面和API的访问。



前端走网管调用NodeAPI

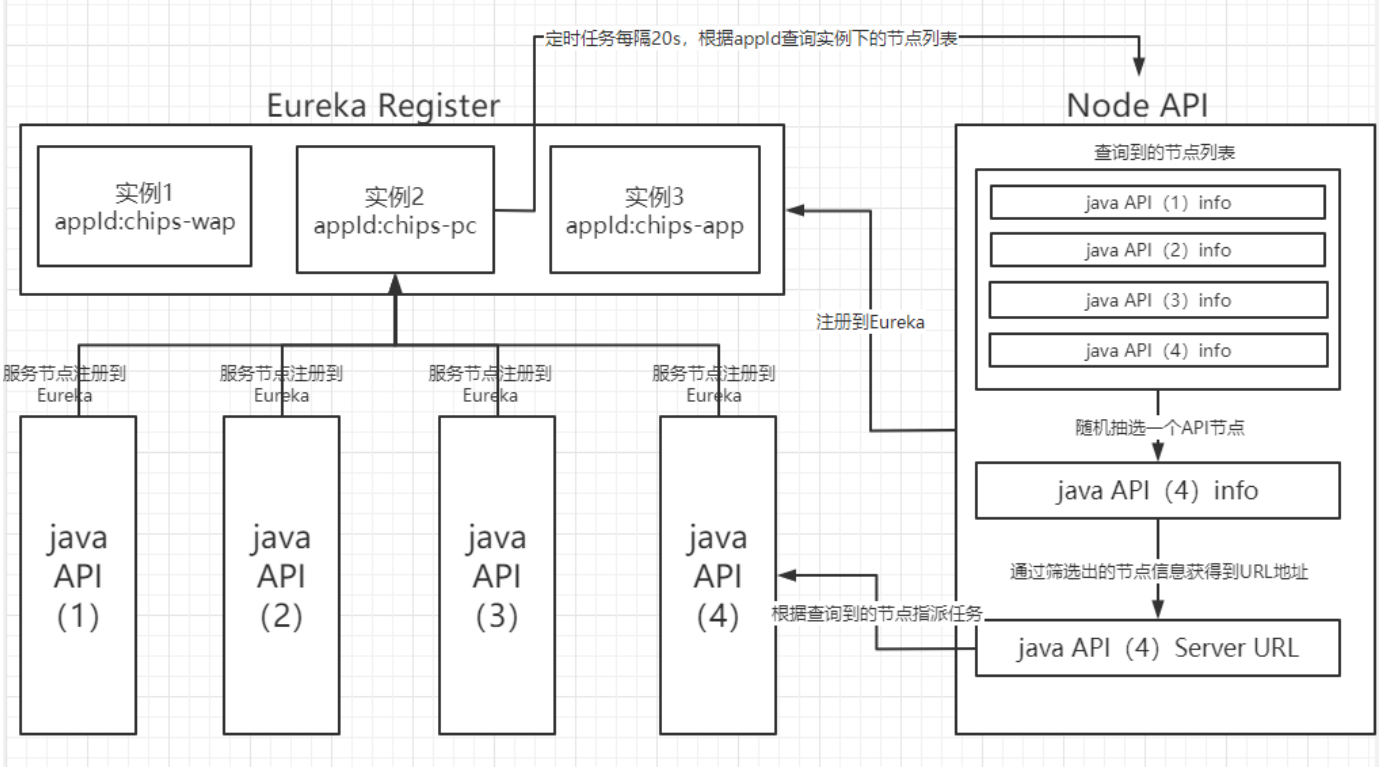
Node服务注册到Eureka成功之后，前端调用Node服务的接口就可以使用网管地址+实例名称的形式，进行访问。

```
/* T环境配置文件 */

module.exports = {
  baseURL: 'https://tmicrouag.dgg188.cn/chips-wap', // 网关地址
  mchMerchantSgin: '62070A76D7BF904888B75450D2D6B4C4',
  mchCoding: '20190620000111',
}
```

Node中间层服务进行任务分发

查询Eureka实例节点的方式可以使用appId和vipAddress进行查询。现在主流的方式是使用appId进行查询，所以会启动一个定时任务，在节点进程中随机出一个节点去查询到实例节点。我们在从节点集群中，抽出一个节点去进行我们所需要的任务计算。



定时任务

```
//app/schedule/eureka.js
'use strict';
module.exports = app => {
  return {
    schedule: {
      interval: '20s', // 20秒间隔
      type: 'worker', // 每台机器上只有一个 worker 会执行这个定时任务，每次执行定时任务的 worker 的选择是随机的。
      immediate: true, // 启动时立即执行一次
      env: [ 'local', 'unittest', 'prod' ], // 定时任务的运行环境
    },
    async task(ctx) {
    // 查询所有eureka的节点信息
      const appInstances = await app.eureka.getInstancesByAppId(app.config.apiClient.APPID);
      // 将节点信息放入egg缓存
      ctx.app.eurekaInstances = appInstances;
    },
  };
```

```
};
```

拦截中间件

假如未能从Eureka实例中，获取到节点信息。那么我们必须在请求执行最初就抛出错误结果。

```
//middleware/eureka.js
'use strict';
/**
 * 判断ereuka的API节点是否获取到
 */
module.exports = (option, app) => {
  return async function eureka(ctx, next) {
    const appInstances = app.eurekaInstances;
    if (!Array.isArray(appInstances) || appInstances.length === 0
) {
      const err = '未获取到eureka服务实例下的后端节点';
      ctx.logger.error(err);
      return await ctx.helper.fail({ ctx, code: 503, res: err });
    }
    next();
  };
};
```

从节点列表随机抽取一个节点

```
/**
   * 获取某个可用服务,随机取
   * @param {*} instances 所有实例
   * @return {*} json
   */
  getOneInstanceFromAll(instances) {
    if (instances != null) {
      const upInstances = [];
      for (const i of instances) {
        if (i.status.toUpperCase() === 'UP') {
          upInstances.push(i);
        }
```

```
      }
      if (upInstances.length > 0) {
        const instanceIndex =
          upInstances.length === 1 ? 0 : Date.now() % upInstances.length;
        return upInstances[instanceIndex];
      }
      return '';
    }
    return '';
  }
```

通过节点信息得到API Server URL

```
    /**
   * 根据实例获取一个完整的ip方式的服务地址。
   * @param {*} instance  app的实例。
   * @return {string}  url地址,包括协议，ip和端口。例如:http://192.168.1.100:8080。
   */
  getServerPath(instance) {
    let url = '';
    const http = 'http://';
    const https = 'https://';
    if (instance) {
      if (instance.port && instance.port['@enabled'] === 'true') {
        url = http + instance.ipAddr + ':' + instance.port.$;
      } else if (
        instance.securePort &&
        instance.securePort['@enabled'] === 'true'
      ) {
        url = https + instance.ipAddr + ':' + instance.securePort.$;
      }
    }
    return url;
  }
```

封装扩展方法helper用于获取节点信息

```
'use strict';
const Service = require('egg').Service;

class EurekaService extends Service {
  getUrl() {
    const { app } = this;
    const appInstances = app.eurekaInstances;
    const ins = this.getOneInstanceFromAll(appInstances);
    const serverUrl = this.getServerPath(ins);
    const url = serverUrl;
    return url;
  }
}
module.exports = EurekaService;
```

使用URL

我们可以通过使用helper的方法直接获取到随机的节点URL，我们可以使用获取到的URL直接访问Java服务器。

```
const serverUrl = ctx.helper.getUrl();
```

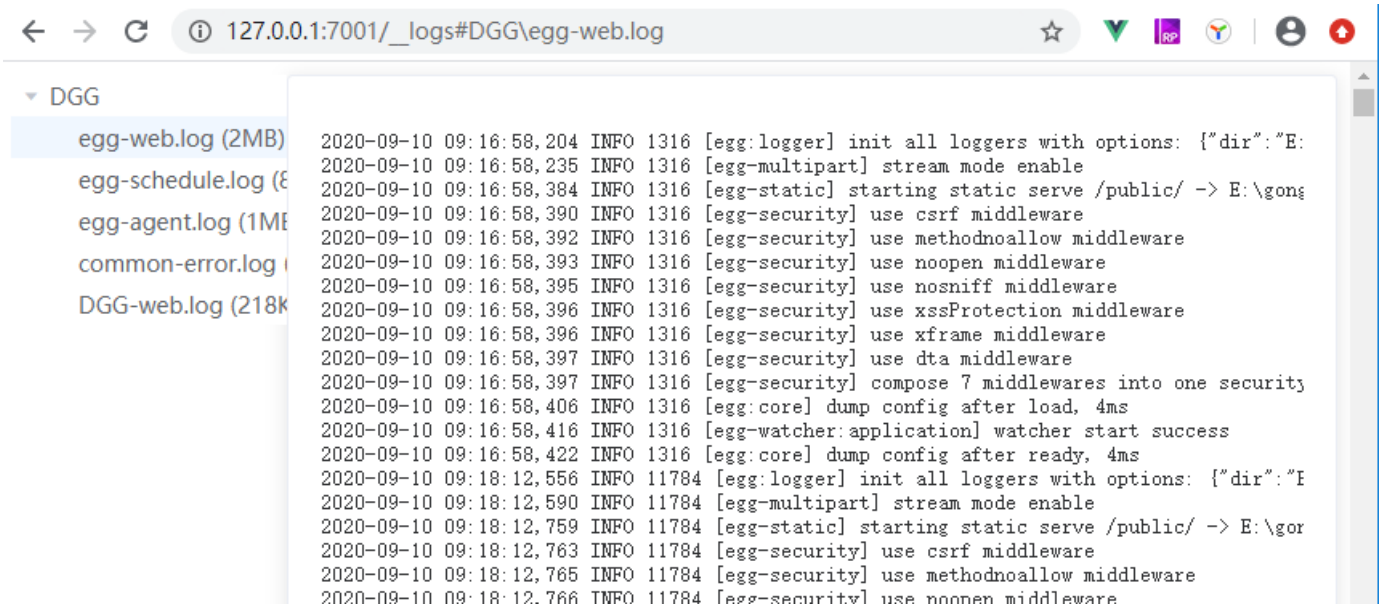
特别注意

每次开始新项目前，必须找架构组索取微服务的实例名称。注册Eureka时使用的必须是架构组给的实例名称，否则该服务无效。

封装使用

开发日志查看（自动）

使用了egg-logview插件，在本地服务 http://127.0.0.1:7001/__logs，可以查看到当前项目logs文件夹下的所有日志信息。egg-logview配置在plugin.local.js中，也就意味着（为了安全起见）只在本地开发环境可以查看到日志。



接口数据缓存（手动）

根据薯片项目的需求，实现后端接口可自由缓存的 要求。我们专门定制了dggCahe中间件，前端调用接口时，只需要告诉Node中间层，此接口是否需要缓存，即可以动态选择缓存接口数据。现在缓存数据默认存储一天。

前端使用

前端需要在请求头写入x-cache-control字段即可控制接口数据的缓存。
具体参数如下：

参数	说明
cache	缓存数据
no-cache/[不传]	不缓存数据

```
async asyncData({ $axios }){
  const res= await $axios.get('http://127.0.0.1:7001/api/demo',{
    headers: {'x-cache-control':'cache'}
  })
  if(res.status===200){
    return { ServerData:res.data.userList }
  }
}
```

关于egg-redis的拓展配置见，基于项目常见的egg-redis函数整理。

数据响应（手动）

我们为了避免服务端的异常错误，所以将所有响应信息给与包装。以后凡是开发中所以响应消息必须使用我们封装的响应方法，否则将导致异常错误或服务宕机（Nuxt一旦接收到错误数据格式，会导致Nuxt服务宕机，整站无法访问）。

响应状态码

```
//extend/helper.js
errorCode: {
    200: '请求成功。客户端向服务器请求数据，服务器返回相关数据',
    201: '资源创建成功。客户端向服务器提供数据，服务器创建资源',
    202: '请求被接收。但处理尚未完成',
    204: '客户端告知服务器删除一个资源，服务器移除它',
    206: '请求成功。但是只有部分回应',
    400: '请求无效。数据不正确，请重试',
    401: '请求没有权限。缺少API token，无效或者超时',
    403: '用户得到授权，但是访问是被禁止的。',
    404: '发出的请求针对的是不存在的记录，服务器没有进行操作。',
    406: '请求失败。请求头部不一致，请重试',
    410: '请求的资源被永久删除，且不会再得到的。',
    422: '请求失败。请验证参数',
    500: '服务器发生错误，请检查服务器。',
    502: '网关错误。',
    503: '服务不可用，服务器暂时过载或维护。',
    504: '网关超时。',
  },
```

成功响应

```
ctx.helper.success({ ctx, code: 200, res: result });
```

错误响应

```
ctx.helper.fail({ ctx, code: 500, res: err });
```

错误拦截（自动）

处理接口上我们可以预测到的错误，那么项目的异常错误怎么加以拦截呢？我们可以充分利用中间件（洋葱模型的特性），对响应内容进行判断和错误拦截，所以我们封装了错误处理中间件。无需程序员手动再去干预。

```
//middleware/errFilter.js
'use strict';
/**
 * 错误处理中间件,将项目中所有的错误信息进行包装拦截
 */
module.exports = (option, app) => {
  return async function errFilter(ctx, next) {
    try {
      await next();
    } catch (err) {
      const error = app.config.env === 'prod' ? '服务器错误,请联系管理员' : err;
      ctx.helper.fail({ ctx, code: 500, error });
    }
  };
};
```

数据压缩(自动)

数据压缩是指在不丢失有用信息的前提下，缩减数据量以减少存储空间，提高传输效率、存储和处理效率，或按照一定的算法对数据进行重新组织，减少数据的冗余和存储空间的一种技术方法。

常见的压缩技术有：

-gzip：

由GUN软件基金会在1992年发布的开源的数据压缩格式；

-deflate：

由Phil katz在1993年发布的非专利型无损压缩算法；

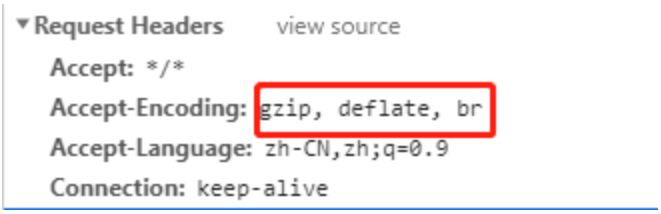
-rar：

由Eugene Roshal在1993年发布的一种压缩格式，是专利文件格式。

Gzip压缩

gzip中间件，得益于现代浏览器几乎都都支持gzip、deflate、 br等压缩格式自动解压，对于所有响应内容（JSON、String、Number、Html、File等）都采取gzip进行压缩处理。

我们可以在浏览器的任何一个响应文件的响应头看到Accept-Encoding字段，他表示当前浏览器对压缩文件格式的支持情况（一般经过Gzip压缩的内容，压缩率能达到80%）。




```
// config.default.js
config.gzip = {
  threshold: 1024, // 小于 1k 的响应体不压缩
};
```

参数校验（手动）

在Egg.js中，我们使用的参数校验使用的是egg-validate插件进行参数校验。他是基于parameter，所以他也支持parameter的所有API和配置项。

egg-validate的API：

API	结果	说明
ctx.validate(rule, [params])	检验失败的抛出一个异常，没有捕获的话，会返回一个422错误。	假如params不传，那么默认就会对ctx.request.body进行检验。假如需要校验ctx.query，直接将ctx.query当做params传入方法即可。
app.validator.validate	检验不通过会返回错误，不会抛出异常	参数同上

参数类型

- `'int'` => `{type: 'int', required: true}`
- `'int?'` => `{type: 'int', required: false }`
- `'integer'` => `{type: 'integer', required: true}`
- `'number'` => `{type: 'number', required: true}`

- `'date'` => `{type: 'date', required: true}`
- `'dateTime'` => `{type: 'dateTime', required: true}`
- `'id'` => `{type: 'id', required: true}`
- `'boolean'` => `{type: 'boolean', required: true}`
- `'bool'` => `{type: 'bool', required: true}`
- `'string'` => `{type: 'string', required: true, allowEmpty: false}`
- `'string?'` => `{type: 'string', required: false, allowEmpty: true}`
- `'email'` => `{type: 'email', required: true, allowEmpty: false, format: EMAIL_RE}`
- `'password'` => `{type: 'password', required: true, allowEmpty: false, format: PASSWORD_RE, min: 6}`
- `'object'` => `{type: 'object', required: true}`
- `'array'` => `{type: 'array', required: true}`
- `[1, 2]` => `{type: 'enum', values: [1, 2]}`
- `/\d+/` => `{type: 'string', required: true, allowEmpty: false, format: /\d+/}`

自定义验证规则

更多见自定义规则见官方文档

案例

```
'use strict';
const Controller = require('egg').Controller;
class UserController extends Controller {
  async list() {
    const { ctx, service, app } = this;
    // 定义参数校验规则
    const rules = {
      id: { type: 'id', required: true },
    };
    // 参数校验
    const valiErrors = app.validator.validate(rules, ctx.query);
    // 假如参数校验未通过
    if (valiErrors) {
      ctx.helper.fail({ ctx, code: 422, res: valiErrors });
      return;
    }
    // 参数校验通过,正常响应
```

```
    const result = await service.demo.list(ctx.query.id);
    ctx.helper.success({ ctx, code: 200, res: result });
  }
}
module.exports = UserController;
```

日期格式化（手动）

我们在helper中封装了很多工具方法，使用ctx.helper.moment()，可以对时间进行格式化和任意类型的转换。

```
ctx.helper.moment(new Date()).format('MM月DD日'); // 09月01日
ctx.helper.moment(new Date()).format('MMM'); // 9月
ctx.helper.moment(new Date()).format('MMMM'); // 九月
ctx.helper.moment(new Date()).format('dd'); // 六
ctx.helper.moment(new Date()).format('ddd'); // 周六
ctx.helper.moment(new Date()).format('dddd'); // 星期六
ctx.helper.moment(new Date()).isoWeekday(); // 6
ctx.helper.moment(new Date()).isoWeekYear(); // 2018
ctx.helper.moment(new Date()).format('LT'); // 16:56
ctx.helper.moment(new Date()).format('LTS'); // 16:56:34
ctx.helper.moment(new Date()).format('L'); // 2018-09-01
ctx.helper.moment(new Date()).format('LL'); // 2018年09月01日
ctx.helper.moment(new Date()).format('LLL'); // 2018年09月01日下午4点56分
ctx.helper.moment(new Date()).format('LLLL'); // 2018年09月01日星期六下午4点56分
ctx.helper.moment(new Date()).format('l'); // 2018-9-1
ctx.helper.moment(new Date()).format('ll'); // 2018年9月1日
ctx.helper.moment(new Date()).format('lll'); // 2018年9月1日 16:56
ctx.helper.moment(new Date()).format('llll'); // 2018年9月1日星期六 16:56
ctx.helper.moment(new Date()).format('A'); // 下午
ctx.helper.moment(new Date()).format('a'); // 下午
ctx.helper.moment(new Date()).format('ALT') // 下午17:09


// subtract 减法 、 add 加法
ctx.helper.moment().add(7, days).format('LL'); // 7天后的日期 2018
```

```
年09月08日
ctx.helper.moment().subtract(7, 'days').format('LL'); // 7天前的日期 2018年08月25日
ctx.helper.moment().add(9, 'hours').format('HH:mm:ss'); // 9小时后 01:56:34
ctx.helper.moment().add(1, 'week').format('LL'); // 1周后 2018年09月08日


// fromNow 时差 (之前) ; fromNow(true)  去除前或者内字
ctx.helper.moment([2017, 0, 29]).fromNow(true); //  2年
ctx.helper.moment([2017, 0, 29]).fromNow(); //  2年前
ctx.helper.moment([2019, 0, 29]).fromNow(true); //  5个月
ctx.helper.moment([2019, 0, 29]).fromNow(); //  5个月内
ctx.helper.moment("20120901", "YYYYMMDD").fromNow(); // 6年前
ctx.helper.moment(+new Date() - 1000 * 300).fromNow(); // 5分钟前
ctx.helper.moment(+new Date() - 1000 * 3).fromNow(); // 几秒前
ctx.helper.moment(+new Date() - 3 * 24 * 60 * 60 * 1000).fromNow(); // 3天前
ctx.helper.moment(+new Date() - 30 * 24 * 60 * 60 * 1000).fromNow(); // 1个月前
ctx.helper.moment(+new Date() - 365 * 24 * 60 * 60 * 1000).fromNow(); // 1年前


// toNow 时差 (之后 现在为基准) ; toNow(true)  去除前或者内字
ctx.helper.moment([2007, 0, 29]).toNow() // 12年内
ctx.helper.moment([2020, 0, 29]).toNow() // 1年前
ctx.helper.moment([2020, 0, 29]).toNow(true) // 1年

// 时差 (之后) ; to(true)  // 去除前或者内字
ctx.helper.moment([2007, 0, 29]).to() // 12年内
ctx.helper.moment([2020, 0, 29]).to() // 1年前
ctx.helper.moment([2020, 0, 29]).to(true) // 1年


// 时差 (毫秒)
ctx.helper.moment([2007, 0, 29]).diff(moment([2007, 0, 28])); // 86400000

```

```
58
59 // 时差 (天)
60 ctx.helper.moment([2007, 0, 29]).diff(moment([2007, 0, 28]), 'day
   s') //  1
61
62
63 // 天数 (月)
64 ctx.helper.moment("2012-02", "YYYY-MM").daysInMonth() //  29
```

常见问题

路径的使用

Path模块提供了对文件路径进行操作的相关方法；这些方法只是进行了字符串的相关转换，与文件系统本身没有任何关联。

为什么要使用Path模块

windows有盘符的概念，比如"E:\project\dgg"而linux则没有，"/"就表示更目录。

windows用的是"\",linux用的是"/"，那么就导致windows的代码在linux就不支持了。

Path模块

```
1  const path=require('path');
2
3  //解析路径字符串
4  console.log(path.parse('c:/user/local/img/1.jps'));
5  //{
6  //  root: 'c:/',
7  //  dir: 'c:/user/local/img',
8  //  base: '1.jps',
9  //  ext: '.jps',
10 //  name: '1'
11 //}
12
13 //将路径对象转化为字符串
14 var obj={dir:'c:/user/local/img',base:'1.jps'};
15 console.log(path.format(obj));
16 //c:/user/local/img\1.jps
17
```

```
//更具基础路径解析出一个目标路径的绝对路径
console.log(path.resolve('htdocs/css','../img/news'));
//H:\myNode\lesson01\htdocs\img\news

//更具基础路径，获取模板路径预期相对的关系
console.log(path.relative('htdocs/css','htdocs/img/news'));
//..\img\news
```

案例

下述代码在Windows能够正常运行，但是在Linux下由于操作系统路径规则的不同，则有可能报错。

```
fs.readFile('./../../logs/DGG/egg-wb.log', (err, data) => {
  if (err) throw err;
  console.log(data)
});
```

正确的写法为：使用path.resolve(__dirname, './../../logs/DGG/egg-wb.log')根据操作系统转换为各自支持的路径。

```
fs.readFile(path.resolve(__dirname, './../../logs/DGG/egg-wb.log')
, (err, data) => {
  if (err) throw err;
  console.log(data)
});
```

CDM提示错误但无错误信息

问题

windows使用cmd启动服务，包括Webstorm、VScode等IDE在windows环境启动都是基于cmd。启动过程提示遇到异常，需要结束异常退出当前进程，但是又没有错误提示。

```
yarn run v1.22.4
$ cross-env DGG_SERVER_ENV=development egg-scripts start --daemon
--title=egg-server-DGG
[egg-scripts] Starting egg application at E:\gongzuo\DGG
[egg-scripts] Run node E:\gongzuo\DGG\node_modules\egg-scripts\li
```

```
b\start-cluster {"title":"egg-server-DGG","baseDir":"E:\\gongzuo
\\DGG","framework":"E:\\gongzuo\\DGG\\node_modules\\egg"} --title
=egg-server-DGG
[egg-scripts] Save log file to C:\Users\Administrator\logs
[egg-scripts] Wait Start: 1...
[egg-scripts] Wait Start: 2...
[egg-scripts] tail -n 100 C:\Users\Administrator\logs\master-stde
rr.log
[egg-scripts] ignore tail error: Error: spawn tail ENOENT
[egg-scripts] Start got error, see C:\Users\Administrator\logs\ma
ster-stderr.log
[egg-scripts] Or use `--ignore-stderr` to ignore stderr at startu
p.
error Command failed with exit code 1.
info Visit https://yarnpkg.com/en/docs/cli/run for documentation
 about this command.

Process finished with exit code 1
```

解决方案

使用git和其他专业的命令行工具启动服务，遇到错误就可以正常输出错误相关信息。

```
yarn run v1.22.4
$ cross-env DGG_SERVER_ENV=development egg-scripts start --daemon
--title=egg-server-DGG
[egg-scripts] Starting egg application at E:\gongzuo\DGG
[egg-scripts] Run node E:\gongzuo\DGG\node_modules\egg-scripts\li
b\start-cluster {"title":"egg-server-DGG","baseDir":"E:\\gongzuo
\\DGG","framework":"E:\\gongzuo\\DGG\\node_modules\\egg"} --title
=egg-server-DGG
[egg-scripts] Save log file to C:\Users\Administrator\logs
[egg-scripts] Wait Start: 1...
[egg-scripts] Wait Start: 2...
[egg-scripts] tail -n 100 C:\Users\Administrator\logs\master-stde
rr.log
[egg-scripts] Got error when startup:
[egg-scripts]  WARN  mode option is deprecated. You can safely re
move it from nuxt.config
```

```
[egg-scripts]  WARN  No proxy defined on top level.
[egg-scripts]  WARN  No proxy defined on top level.
[egg-scripts] Start got error, see C:\Users\Administrator\logs\master-stderr.log
[egg-scripts] Or use `--ignore-stderr` to ignore stderr at startup.
error Command failed with exit code 1.
info Visit https://yarnpkg.com/en/docs/cli/run for documentation about this command.
```

Egg中前端代理失败

问题

集成Nuxt时@nuxt.js/proxy代理异常，当我们在客户端发送POST请求，浏览器上请求无响应，一段时间后Nuxt:proxy提示代理失败。但是所有GET请求无论是携带参数还是不携带参数都能正常代理和返回结果。

问题排查

1、GET请求在带参数和不带参数情况下都能正常代理和响应成功。√

2、POST请求在携带参数时会出现代理失败，请求无响应，不携带参数，则响应正常。√

3、猜想Nuxt是基于Express，那么放入Egg.js，使用的是Koa，会不会是两个中间件存在兼容问题。✕

@nuxt.js/proxy也是基于http-proxy-middleware，我们去http-proxy-middleware的Github查看相关兼容情况如下：

Compatible servers

`http-proxy-middleware` is compatible with the following servers:

- connect
- express
- browser-sync
- lite-server
- polka
- grunt-contrib-connect
- grunt-browser-sync
- gulp-connect
- gulp-webserver

经过查看issues我们尝试使用koa2-connect（express是一个老牌儿Node.js Web服务器程序，拥有强大的生态。当kao要使用express中间件时，可以使用该插件进行包装转换）进行一个中间件转换。下面是对@nuxt.js/proxy的源码修改：

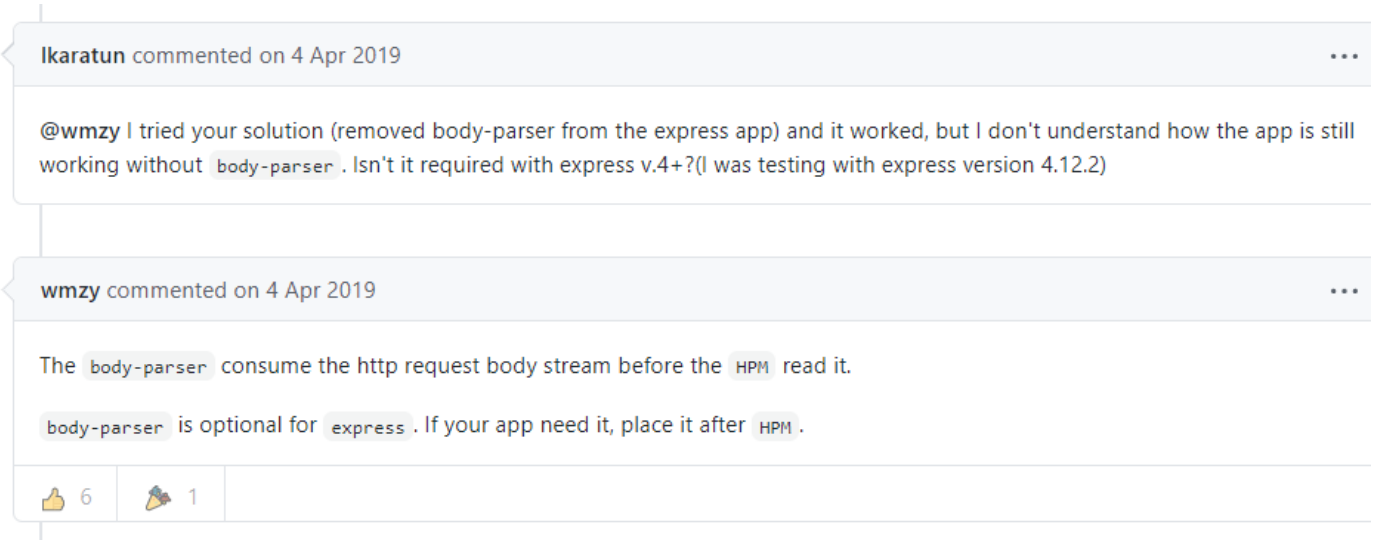
```
//before
proxy.forEach((args) => {
  const middleware = Proxy.apply(undefined, args)
  middleware.prefix = false
  this.options.serverMiddleware.push(middleware)
})
```

```
//after
const c2k = require('koa2-connect')
proxy.forEach((args) => {
  const middleware = Proxy.apply(undefined, args)
  middleware.prefix = false
  this.options.serverMiddleware.push((req, res, next)=>{
    return c2k(middleware)({req, res}, next)
  })
})
```

转换完成我们测试结果，POST请求下还是无法携带参数，猜想结果不成立。

4、我们测试了将http-proxy-middleware转为Koa中间件后，POST还是无法携带参数。那么问题会不会出在http-proxy-middleware本身。√

我们寻找关http-proxy-middleware相关的问题https://github.com/chimurai/http-proxy-middleware/issues/171可以看到，在处理代理请求时发生错误是因为代理中间件放在了body-parser中间件之后。因为POST请求的响应内容是放在请求对象以流的形式传递过来的，他需要使用监听req对象data事件，获得body内容。body-parser会对POST参数进行修改，所以后续的http-proxy-middleware中间件获取到的body内容是错误的，导致POST代理失败。



5、既然是中间件的执行问题，@nuxt.js/proxy中间件依托在Nuxt.js，Nuxt.js作为服务器渲染的中间件，又不需要使用body-parser这类处理api的中间件。那么我们直接将Nuxt中间件放入到Egg中间件执行开头即可。√

Egg.js中间件，分为应用级中间件和内置中间件。内置中间件优先于所有的应用级中间件，body-parser就是属于Egg.js内置5个中间件之一。内置中间件无法覆盖和修改，我们可以通过在入口主程序，服务启动之前，修改Egg.js的内置中间件相关配置。将nuxt中间件放到内置中间件最前。结果成功解决问题。√

```
// app.js
  app.config.coreMiddleware.unshift('nuxt');
```

服务注册到Eureka无法访问

问题

将一个新的Node服务注册到Eureka，在Eureke Server实例列表能够查看到注册的实例，但是就是无法访问。

问题原因

1、检查config配置中注册的Eureka APP名称是否是架构组所给的你项目对应的服务名称。

2、新的Node服务按架构组给的实例名称注册到Eureka后，要找架构组配置转发规则。

刚注册到Eureka的服务访问不了，几秒后回复正常

问题

新的Node服务注册到Eureka之后，Eureka Server列表能够看到我们注册的实例，但是接口访问依然访问不到，几秒后一切恢复正常。

问题原因

这种情况是网关心跳检测的问题，注册上去之后，网关会有几秒钟的检测延时，几秒后就正常访问了。这是属于正常情况，开发时遇到该问题无需过多考虑和忧虑。

通过Eureka访问接口接口偶尔404

问题

注册到Eureka的服务，接口偶尔访问404。

问题原因

因为我们的Node作为微服务，是多节点部署。我们多个人在本地开发，也会将自己的本地服务注册到Eureka。接口通过网关访问时，网关会做任务分发，将你的请求分发到别人的机器上。

解决方案

找架构组或找我，配置Node服务的灰度规则或隔离，将自己的请求指向到自己的本机。