Koa学习

概要

- 介绍koa/compose/AOP思想

目的

- 通过AOP思想来优化代码

Node Http

先看个最简单的 用node 实现的 http请求，首先要判断request的method，request的url，才可以执行业务函数

```
const http = require('http');
const PORT = 8888;
const controller = { // 控制器
    index(req, res) {
      res.end('This is index page')
    },
    home(req, res) {
      res.end('This is home page')
    },
    _404(req, res) {
      res.end('404 Not Found')
    }
}
const router = (req, res) => { // 路由器
// 如果需要区分请求的方法是get还是post, 那么就要继续加request.method
  if( req.url === '/' ) {
    controller.index(req, res)
  } else if( req.url.startsWith('/home') ) {
    controller.home(req, res)
  } else {
    controller._404(req, res)
  }
}
const server = http.createServer(router) // 创建服务
server.listen(PORT, function() {
  console.log(`the server is started at port ${PORT}`)
})
console.log('Server running at http://127.0.0.1:8888/'); // 终端打印如下信息
```

如果这样写，一个项目上百个api，还要区别路径和方法，写在同一个路由器中 想想都疯了。

关于koa

koa核心功能就两部分，一部分是对中间件的处理，也就是 compose组合能力，另一个就是对http的请求的封装。重点是第一个compose能力 koa github源码

先看看koa基本用法， 改写上述node http请求

```
const Koa = require('koa');
const app = new Koa();
const controller = { // 控制器
  index(ctx) {
    ctx.response.body = 'This is index page'
  },
  home(ctx) {
    ctx.response.body = 'This is home page'
  },
  _404(ctx) {
    ctx.response.body = '404 Not Found'
  }
}
const router = ctx => {
  // 如果需要区分请求的方法是get还是post, 那么就要继续加ctx.request.method 判断
  if( ctx.request.url === '/' ) {
    controller.index(ctx)
  } else if( ctx.request.url.startsWith('/home') ) {
    controller.home(ctx)
  } else {
    controller._404(ctx)
  }
};
app.use(router);
app.listen(8888,() => console.log("Server running at http://127.0.0.1:8888/"))
```

和上面node原生http请求对比，app 封装了 server，ctx封装了http，包括request，response。

先来看简单部分，对http请求的封装
koa的文件很少，就四个文件。如下

- lib
  - application.js // 入口文件
  - context.js // 执行上下文

- request.js //httpRequest
- response.js // httpResponse

application.js

入口文件，只摘取了核心部分
整体的作用是做一些初始化配置，还有对一些错误的处理，以及对旧版的兼容处理。

> listen函数

```
listen(...args) {
    const server = http.createServer(this.callback());
    return server.listen(...args);
  }
```

```
callback() {
    const fn = compose(this.middleware);
    const handleRequest = (req, res) => {
      const ctx = this.createContext(req, res);
       // 创建执行上下文
      return this.handleRequest(ctx, fn);
      //  将http的请求返回封装ctx 并执行函数fn
    };
    return handleRequest;
  }
```

> use 函数

```
use(fn) {
    if (typeof fn !== 'function') throw new TypeError('middleware mus
t be a function!');
    this.middleware.push(fn);
    return this;
  }
```

user函数只是将所需要中间件处理或者说函数，塞进一个数组维护起来，

context.js

在application文件中，创建执行上下文 context 就指定其原型为context.js 中的proto 并返回
在context.js 中主要完成对proto的代理作用，间接对
比如ctx.body 其实就是代表 ctx.response.body等等，
通过npm 模块delegates达到委托代理目的，

```
delegate(proto, 'response')
  .method('attachment')
  .method('redirect')
  .method('remove')
  .method('vary')
  .method('has')
  .method('set')
  .method('append')
  .method('flushHeaders')
  .access('status')
  .access('message')
  .access('body')
  .access('length')
  .access('type')
  .access('lastModified')
  .access('etag')
  .getter('headerSent')
  .getter('writable');
```

- getter：外部对象可以直接访问内部对象的值
- setter：外部对象可以直接修改内部对象的值
- access：包含 getter 与 setter 的功能
- method：外部对象可以直接调用内部对象的函数

request.js response.js

这两个文件都是通过get set 方法，来对http的一些方法和属性就行访问设置和修改设置。

到这里只是说明了koa的一些封装作用，其实还没有体现出koa常说的组件处理能力，以及借用router更优雅实现路由。

compose

重点来了，
开始介绍之前，抛出一个问题，我们经常遇到一些js 串行的问题，需要先执行某个异步操作，完成后，再执行某个异步操作。当数量少的时候，Promise.then.then

就可以完成。。。试想，当这个异步操作数量上升到100个 1000个？？怎么办。。

Promise.all是并行操作，所有的异步操作同时执行。。是不可行的。

首先可以利用for 和 await 实现很简单

```
async function runFnArray(FnArray,context){
    for(let fn of FnArray){
        let res  = await fn(context);
    }
}
```

通过for循环和async await，将整个函数变成异步函数，等待每一个Promise执行完毕。。
或者用reduce更优雅来实现

```
function runFnArray(FnArray){
  FnArray.reduce((acc,item,index,array)=>
    acc.then(()=>item())
  ,Promise.resolve())
}
```

用reduce实现和for循环实现的唯一差别在于是否在构建队列的时候执行promise，
因为reduce是同步的，事先构建了一个promise串行队列，而for则是遇到某个promise就直接执行。

具体的reduce/for/forEach/map 性能测试参考 js几个函数性能测试

回过头来看，compose
koa 把compos独立出来，变成compose模块。。

```
module.exports = compose
function compose (middleware) {
  if (!Array.isArray(middleware)) throw new TypeError('Middleware stack must be an array!') // 判断是否是数组
  for (const fn of middleware) {
    if (typeof fn !== 'function') throw new TypeError('Middleware must be composed of functions!')
  }
  return function (context, next) {
    let index = -1
    return dispatch(0)
    function dispatch (i) {
      if (i <= index) return Promise.reject(new Error('next() called multiple times')) // promise 的快捷方式用法
      index = i
      let fn = middleware[i]
      if (i === middleware.length) fn = next // 这里的next 并没有添加进middleware ，而是在 调用compose函数时候 的参数，
      if (!fn) return Promise.resolve() // 相当于到栈底了，
      try {
        return Promise.resolve(fn(context, dispatch.bind(null, i + 1)));
      } catch (err) {
        return Promise.reject(err)
      }
    }
  }
}
```

把整个compose核心代码提取出来

```
function compose (middleware) {
  return function (context, next) {
```

```
    let index = -1
    return dispatch(0)
    function dispatch (i) {
      index = i
      let fn = middleware[i]
      return Promise.resolve(fn(context, dispatch.bind(null, i + 1))
  );
    }
  }
}
```

调用compose 返回一个匿名函数fn(contex,next),两个参数由调用者传入，注意到这个时候这个fn并未执行。
然后向下分发，返回一个新的匿名函数，同时回调函数next 变为dispatch.bind(null, i + 1)。对bind/call/apply不熟悉的可以参考优秀文章。

整个context，是贯穿的

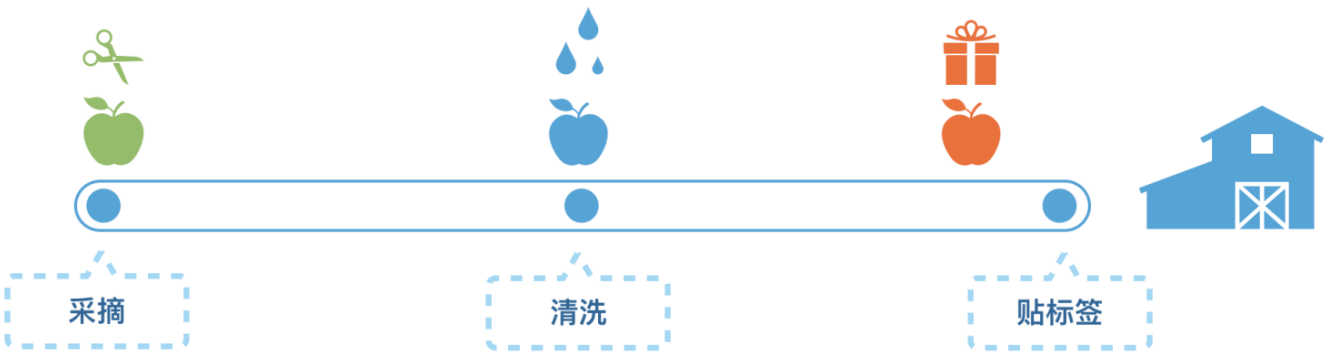
AOP

什么是AOP?

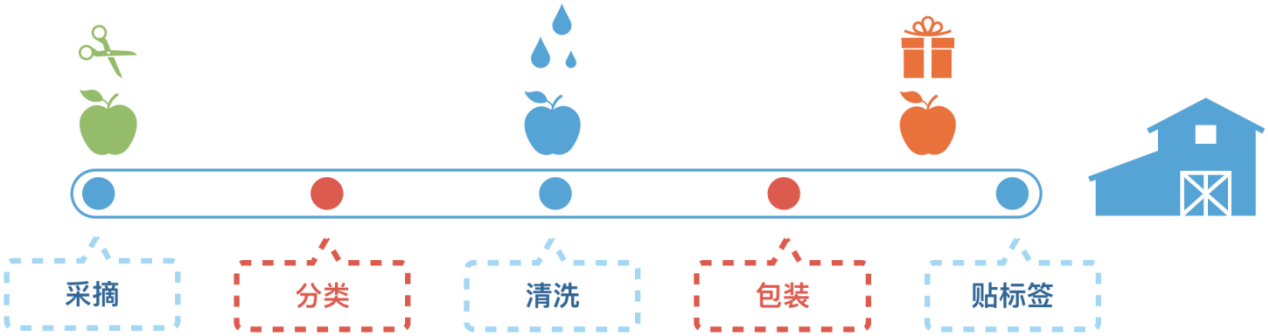
> 就是在现有代码程序中，在程序生命周期或者横向流程中 加入/减去 一个或多个功能，不影响原有功能。

举个例子。

- 农场的水果包装流水线一开始只有 采摘 – 清洗 – 贴标签



- 为了提高销量，想加上两道工序 分类 和 包装 但又不能干扰原有的流程，同时如果没增加收益可以随时撤销新增工序。



什么地方用到AOP?

- 当代码中if-else非常多。
- 当预想一个功能代码块很大，会由很多小部分组成，比如下单
- 当不想改变原有代码，想要额外增加功能，动态装饰原代码的时候
- 当想顺序执行多个操作的时候

AOP实现下单举例

下面用koa-compose 实现在AOP思想的下单过程

```
1   createOrder: async (context) => {
2
3     // -. 验证产品 售卖状态，是否库存量满足 (这里注意抢单)
4     // -. 生成订单
5     // -. 去库存
6     // -. 对比购物车删减产品数量
7
8     const validateGoods = async (context, next) => {
9       await Promise.all(
10        context.goods.map(async (item) => {
11          let product = await Product.findById(item.product)
12          let spec = product.specifications.find((spec) => spec.sku
  == item.sku)
13          if (!product.setting.ifSale || !spec.ifSale) {
```

```
            const err = new Error(`${product.name_ch},该产品已下架`)
            err.code = 406
            throw err
          }
          if (product.stock.sale < item.quantity.order) {
            const err = new Error(`${product.name_ch},该产品库存不足`)
            err.code = 406
            throw err
          }
          return
        })
      )
      await next()
      return //context.res
    }

    const create = async (context, next) => {
      let newOrder = context.goods.reduce((acc, item, index, arr) =
> {
        let key = item.supplyTag
        if (!acc[key]) {
          acc[key] = []
        }
        acc[key].push(item)
        return acc
      }, {})
      console.log(newOrder, "分单信息")
      let res = await Promise.all(
        Object.keys(newOrder).map(async (supplyTag) => {
          let amount = newOrder[supplyTag].reduce((acc, item, index
, arr) => {
            return acc + item.quantity.order * item.price
          }, 0)
          let orderInfo = {
            number: getCode(),
            canteen: context.canteen,
            deliveryTime: new Date(context.deliveryTime),
            supplyTag,
            goods: newOrder[supplyTag],
            amount,
```

```
52            }
53            console.log(orderInfo, "订单信息")
54            let ans = await Order.create(orderInfo)
55            return ans
56          })
57        )
58        await next()
59        // context.res = res
60      }
61
62      const clearStock = async (context, next) => {
63        await Promise.all(
64          context.goods.map(async (item) => {
65            let product = await Product.findById(item.product)
66            product.stock.sale -= item.quantity.order
67            product.stock.real -= item.quantity.order
68            product.save()
69            return
70          })
71        )
72        await next()
73        return
74      }
75
76      const clearCart = async (context, next) => {
77        let goods = []
78        context.goods.forEach((item) => {
79          goods.push({
80            product: item.product,
81            sku: item.sku,
82            count: item.quantity.order,
83          })
84        })
85        // 和购物车对比删除 减数量 不是直接删除
86        let canteen = await Canteen.findById(context.canteen.canteen)
  .select(
87          "cart"
88        )
89        let new_cart = goods.reduce((acc, item, index, arr) => {
90          let existIndex = acc.findIndex(
```

```
91            (cart_item) =>
92              cart_item.product == item.product && cart_item.sku == i
   tem.sku
93          )
94          if (existIndex == -1) {
95            return acc
96          } else {
97            if (item.count >= acc[existIndex].count) {
98              // 购买数量大于购物车的数量。直接清除该项
99              acc.splice(existIndex, 1)
100           } else {
101             acc[existIndex].count -= item.count
102           }
103         }
104         return acc
105       }, canteen.cart)
106       canteen.cart = [...new_cart]
107       canteen.save()
108     }
109
110     return compose([validateGoods, create, clearStock, clearCart])(
    context)
111   },
112 }
```

整个下单过程最主要的还是生成订单部分，但是我们要加许多验证，产品库存，产品是否可售卖，同时下完订单，还要对库存和购物车处理。。
当某天其他新的功能上来了，需要对下单时间做限制，对配送费做要求，对配送地址做要求等等，就只需要另外写功能函数，进入compose 参数 当中的数组中。
同时，某天觉得功能顺序应该是先清空购物车再清库存，也只需要简单换下两个函数在compose中的位置。

AOP的弊端

AOP也存在弊端，

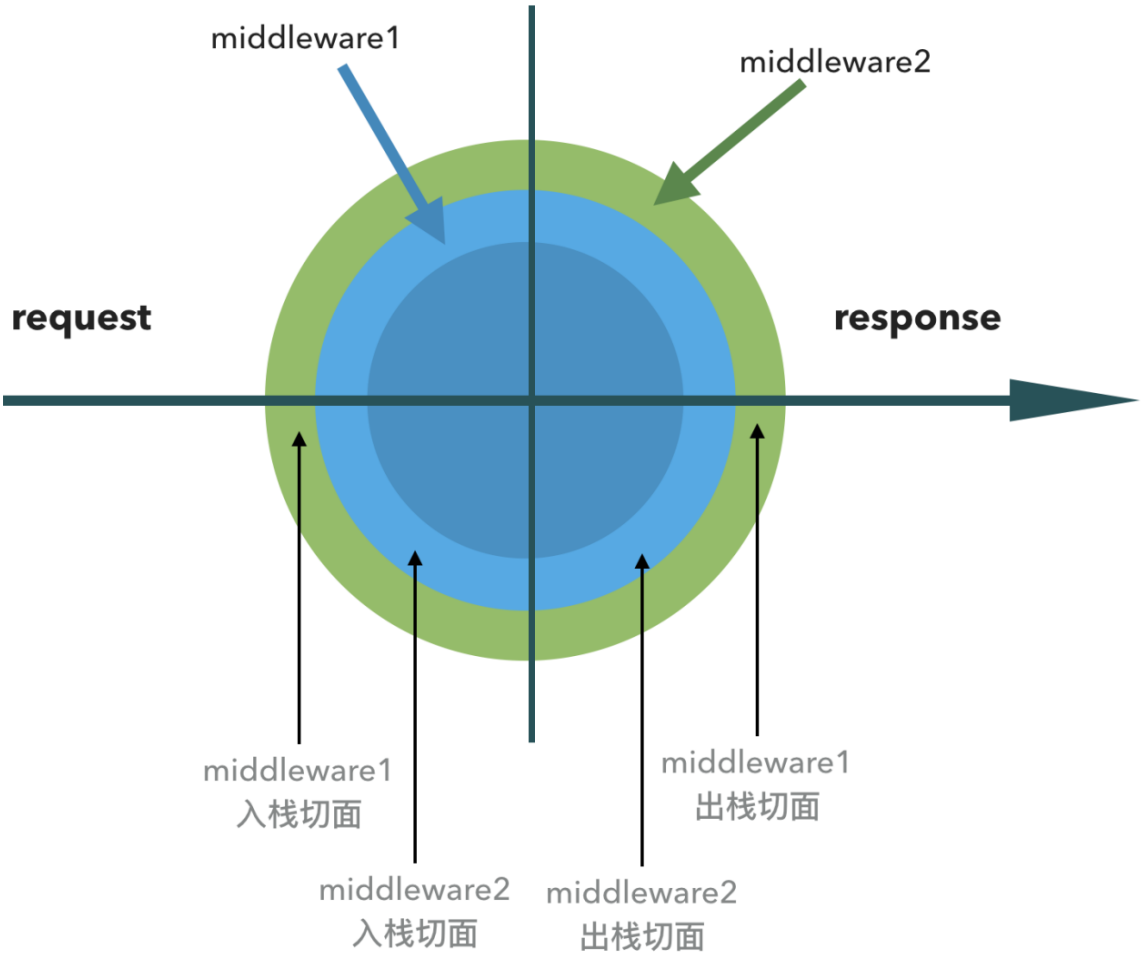
- **不能保证某个请求一定会被维护数组中每个函数所处理**
  每个函数维护的上下文都是同一个对象，所以对对象的属性和方法 的访问和修改权限都是相同的。

如果上个节点改动了下个节点所需的信息，那下个节点就崩溃了，自然报错了。。
那就需要一个默认的要求，不得已不允许改动上下文context，

- **不能保证传入的context格式或者数据，就一定能满足 compose函数的要求**
  可以在最外层对context加一层数据require的要求，或者是对传入数据转换成所需的数据格式。这也体现了AOP的好处，当多一个功能，只需要多写一个功能函数加入，不需要动原来的n模块
- **如果需要在某一部返回结果，怎么办**
  比如说，我新建完订单 想要返回这个新订单的信息，
  上面提到context是贯穿的，所以可以利用context携带新的订单信息返回，
  但是这和上面的 **不得已不允许改动上下文context** 矛盾，但是相比，一层一层向上返回，直接利用context贯穿的特性 新增context一个属性 来的会更好

Koa-Router

讲了这么多compose和AOP 回过头来看koa的洋葱模型，就更直观了



注意到compose处理的数组中 都是异步函数，最后一个是没有next回调函数的，所以返回。。**这个过程很像递归**

刚才AOP适用场景中提到，

> **当代码中if-else非常多。**
> 我们回到最开始，介绍api请求的时候，当api请求种类的很多，上百种，怎么办？
> 看看router如何实现。先看简单的使用例子

```
const Koa = require('koa')
const app = new Koa()
const Router = require('koa-router')
let home = new Router()
// 子路由1
home.get('/', async ( ctx )=>{
  let html = `
    <ul>
      <li><a href="/page/helloworld">/page/helloworld</a></li>
      <li><a href="/page/404">/page/404</a></li>
    </ul>
  `
  ctx.body = html
})
// 子路由2
let page = new Router()
page.get('/404', async ( ctx )=>{
  ctx.body = '404 page!'
}).get('/helloworld', async ( ctx )=>{
  ctx.body = 'helloworld page!'
})
// koa-router嵌套装载所有子路由
let router = new Router()
router.use('/', home.routes(), home.allowedMethods())
router.use('/page', page.routes(), page.allowedMethods())

// 或者利用new Router({prefix:'/'})  new Router({prefix:'/pages'})
// 用compose装载所有子路由
let router = compose([home.router(),home.allowMethods(),page.router(),page.allowMethods()])

```

```
// koa加载路由中间件
app.use(router.routes()).use(router.allowedMethods())
app.listen(3000, () => {
  console.log('[demo] route-use-middleware is starting at port 3000'
)
})
```

可以看到每个子路由，都会有自己的方法，get/post 等，很直观的看到请求是什么类型方法，而不需要从ctx.request.methods中判断。

扩展

前面讲的都是在异步操作基础上，同样，同步代码也会遇到很多条件判断。首先先模拟实现 异步操作中的then

```

Function.prototype.after = function(fn){
    let _self = this // this指向调用after的函数
    return function(){
        let haveReturn = _self.apply(this,arguments)
        // 调用_self  // this指向global
        // 返回的实质是function 由global调用
        if(haveReturn) return haveReturn
        return fn.apply(this,arguments)
    }
}
let get1 = function(){
    console.log(1)
}
let get2 = function(){
    console.log(2)
    return true
}
let get3 = function(){
    console.log(3)
}
let get4 = function(){
    console.log(4)
}

```

```
const context = {
    a:'参数'
}
let run = get1.after(get2).after(get3).after(get4)
run(context)
```

难点和重点就在于要时刻注意this的指向，这里就统一了 this的指向 全部指向global。这种场景在 做value验证时候能够常用。。

继续改写，把这种after的形式，改成compose形式，不想写过多的 after 或者then函数。

```

let get1 = function(){
    console.log(1)
    // return true
}
let get2 = function(context){
    console.log(2,context)
    return true
}
let get3 = function(context){
    console.log(3,context)
}
let get4 = function(){
    console.log(4)
}
const context = {
    a:'参数'
}
const compose = function(middleware){
    if (!Array.isArray(middleware)) throw new TypeError('Middleware stack must be an array!') // 判断是否是数组
    for (const fn of middleware) {
      if (typeof fn !== 'function') throw new TypeError('Middleware must be composed of functions!')
    }
    return function(){
        let args = arguments
        return dispatch(0)
        function dispatch(i){
            let fn = middleware[i]
```

```
            if (i === middleware.length) fn = undefined
            if(!fn) return
            let haveReturn = fn.apply(this,args)
            if(haveReturn) return haveReturn
            return dispatch.call(this,i+1)
        }
    }
}
let res = compose([get1,get2,get4,get3])
res(context)

```

可以观察到，这里的compose 核心采用了apply，所以在碰到函数就执行了，这样也符合同步代码的形式。调用compose就直接调用中间件中的函数。不需要再去执行。

优秀文章链接

> koa设计模式

> 用Reduce实现Promise串行执行

> JavaScript深入之call和apply的模拟实现

> JavaScript深入之bind的模拟实现

> js几个函数性能测试