Koa2源码解读

课前准备

1. koa2 https://github.com/koajs/koa

回顾

基于Koa打造企业级MVC框架

课堂目标

1. 手写koa
2. 手写static中间件

课堂主题

1. koa 原理
2. context
3. Application剖析
4. 中间件机制
5. 常见中间件

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。

koa是Express的下一代基于Node.js的web框架

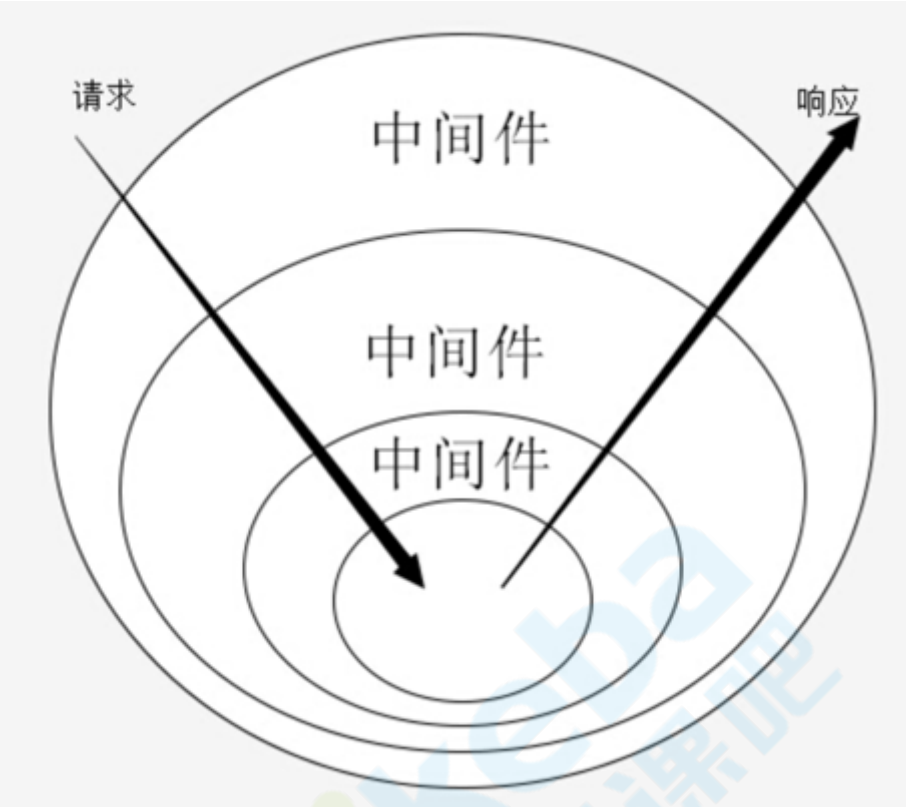
koa2完全使用Promise并配合`async`来实现异步

- 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) => {
    // 同步sleep
    const expire = Date.now() + 100;
    while (Date.now() < expire)

    // ctx.body && ctx.body.push(
    //     {
    //         name:'jerry'
    //     }
    // )
    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>`
}

const fun = router[ctx.url]
fun && fun(ctx)
```

- Koa中间件机制：Koa中间件机制就是函数式 组合概念 Compose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。



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

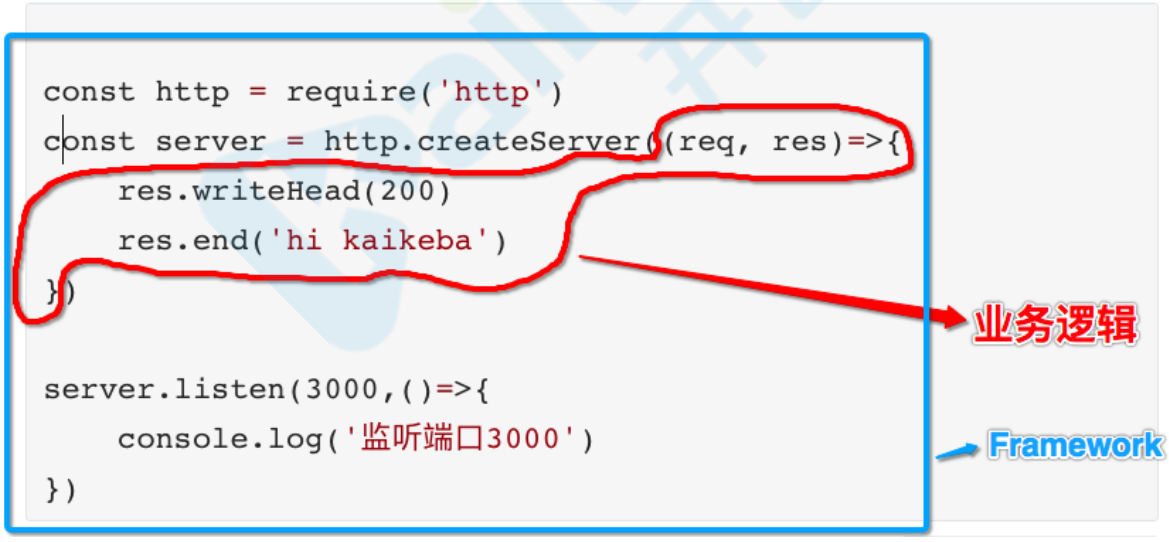
```
app.use(async (ctx,next) => {
    const start = Date.now()
    await next()
    const end = Date.now()
```

```
    console.log(`请求${ctx.url} 耗时${parseInt(end - start)}ms`)
})
app.use(async (ctx,next) => {
    const expire = Date.now() + 102;
    while (Date.now() < expire)
    ctx.body = [
        {
            name:'tom'
        }
    ]
})
```

koa 原理：

- 一个基于nodejs的入门级http服务，类似下面代码：

```
const http = require('http')
const server = http.createServer((req, res)=>{
    res.writeHead(200)
    res.end('hi kaikeba')
})

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



- 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，index.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");
});
```

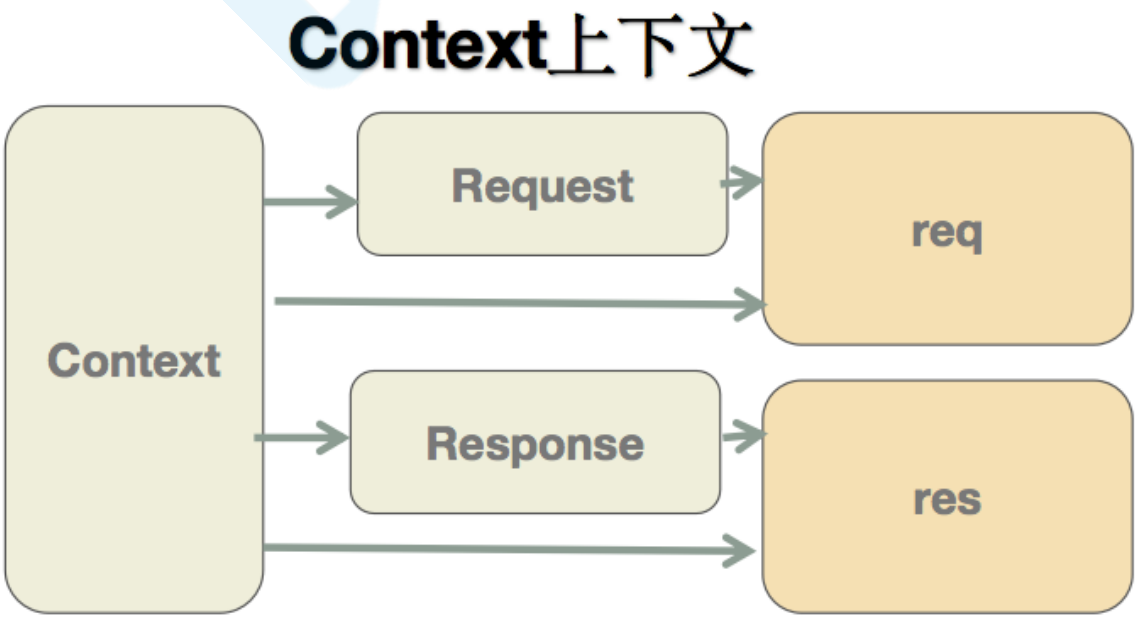
> 目前为止，KKB只是个马甲，要真正实现目标还需要引入上下文（context）和中间件机制（middleware）

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。

使用方法，接近koa了

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



- 知识储备：getter/setter方法

```
// 测试代码，test-getter-setter.js
```

```
const kaikeba = {
    info:{ name: '开课吧', desc: '开课吧真不错' },
    get name(){
        return this.info.name
    },
    set name(val){
        console.log('new name is' + val)
        this.info.name = val
    }
}
console.log(kaikeba.name)
kaikeba.name = 'kaikeba'
console.log(kaikeba.name)
```

- 封装request、response和context

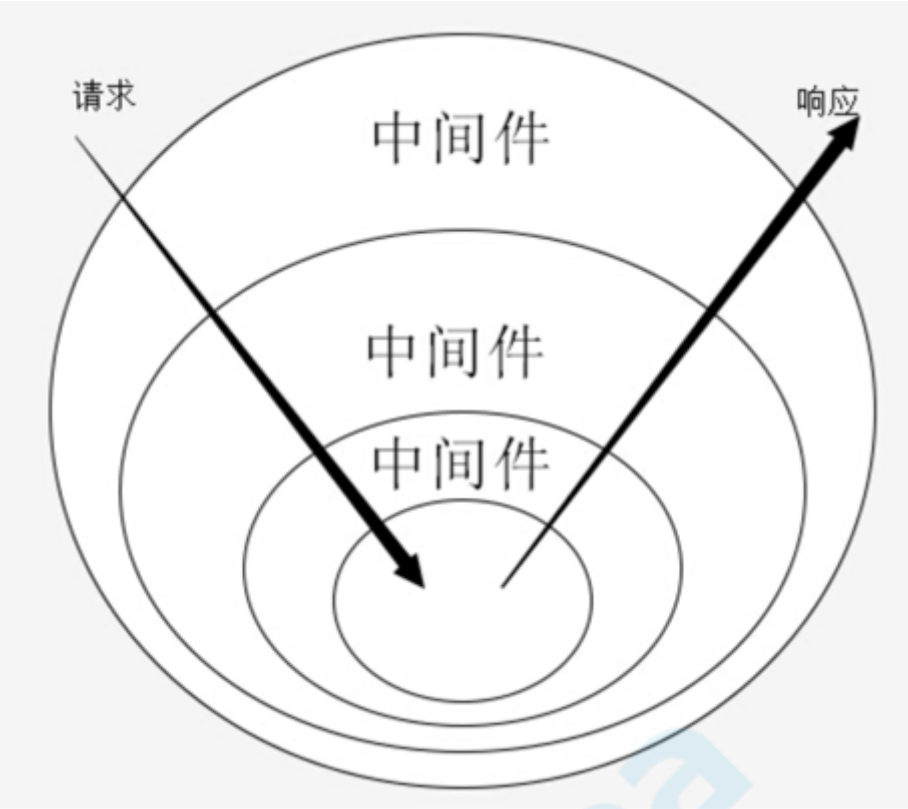
  https://github.com/koajs/koa/blob/master/lib/response.js

```
// 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
  }
};
```

```
// kkb.js
// 导入这三个类
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);
    });
    // ...
  }
  // 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;
  }
}
```

中间件

- Koa中间件机制：Koa中间件机制就是函数式 组合概念 Compose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。



- 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))
```

上面就算是两次函数组合调用，我们可以把他合并成一个函数

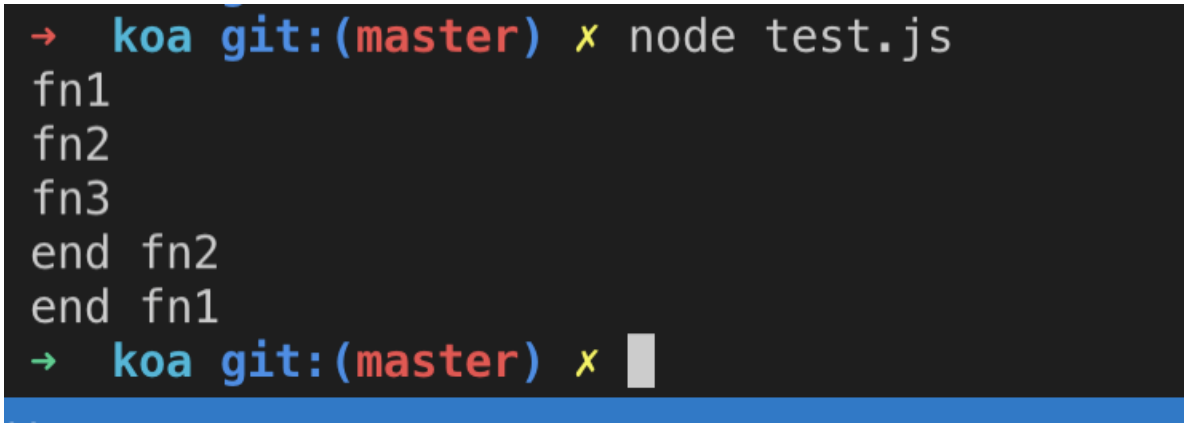
```
const compose = (fn1, fn2) => (...args) => fn2(fn1(...args))
const fn = compose(add,square)
```

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，再执行下一个中间件。

```
function compose(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，再执行下一个
          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((reslove, reject) => {
    setTimeout(() => {
      reslove();
    }, 2000);
  });
}

const middlewares = [fn1, fn2, fn3];
const finalFn = compose(middlewares);
finalFn();
```



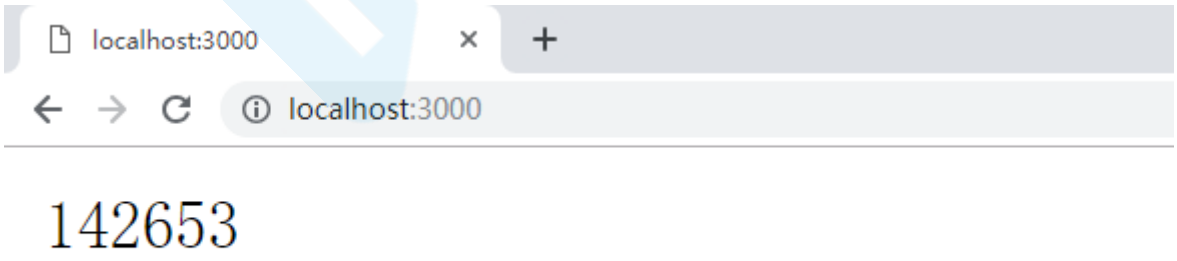
- compose用在koa中，kkb.js

```
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) => {
      const ctx = this.createContext(req, res);
      // 中间件合成
      const fn = this.compose(this.middlewares);
      // 执行合成函数并传入上下文
      await fn(ctx);
      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);
          })
        );
      }
    };
```

```
  }
  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 delay = () => new Promise(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";
});
```



> koa-compose的源码

> - 兼顾OOP和AOP
> - 函数编程 函数即逻辑 (React 函数即组件 组件即页面)

- 看一下Express

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
};
```

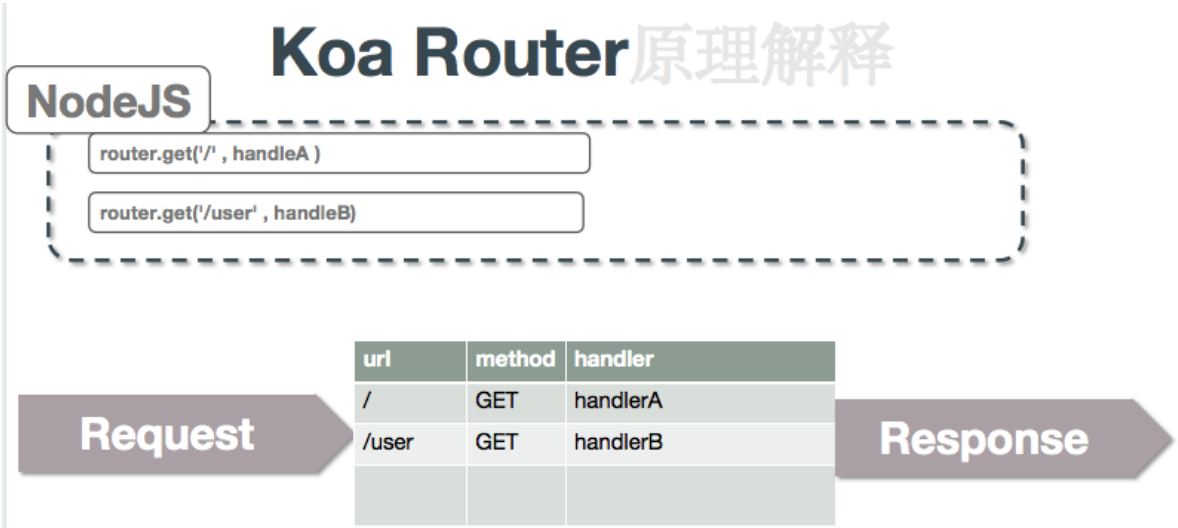
- 中间件常见任务：
  - 请求拦截
  - 路由
  - 日志
  - 静态文件服务

- 路由 router

  将来可能的用法

```
const Koa = require('./kkb')
const Router = require('./router')
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 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，修改request.js




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

```
// context.js
module.exports = {
    // add...
    get method() {
        return this.request.method
    },
}
```

```
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;
```

使用

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

router.get('/index', async ctx => {
  console.log('index,xx')
  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 page'; });

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

app.listen(3000,()=>{
  console.log('server runing on port 9092')
})
```

- 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
  - 静态文件读取
  - 返回

```
// static.js
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", "");
      console.log(filepath);
      // console.log(ctx.url,url, filepath, fileBaseName)
      try {
        stats = fs.statSync(filepath);
        if (stats.isDirectory()) {
          const dir = fs.readdirSync(filepath);
          // const
          const ret = ['<div style="padding-left:20px">'];
          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statSync
            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>");
          ctx.body = ret.join("");
        } else {
          console.log("文件");

          const content = fs.readFileSync(filepath);
          ctx.body = content;
        }
      } catch (e) {
        // 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 { res, req } = ctx;
  const blackList = ['127.0.0.1'];
  const ip = getClientIP(req);

  if (blackList.includes(ip)) {//出现在黑名单中将被拒绝
    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
  );
}

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

> 请求拦截应用非常广泛：登录状态验证、CORS头设置等。

总结

1. 回顾知识点
2. 提示学习方法
3. 提示本次课重点和必会知识

作业 && 答疑

扩展内容

Object.create的理解

https://juejin.im/post/5dd20cb3f265da0bf66b6670

中间件扩展学习

https://juejin.im/post/5dbf9bdaf265da4d25054f91

策略模式：

https://github.com/su37josephxia/frontend-basic/tree/master/src/strategy

中间件对比

https://github.com/nanjixiong218/analys-middlewares/tree/master/src

责任链模式

https://blog.csdn.net/liuwenzhe2008/article/details/70199520

作业：
实现compose函数。
要求: 提交代码截图
- 实现洋葱圈模型