Koa

今日内容

- koa中间件/错误处理
- 鉴权(session和cookie/jwt-token/oAuth)
- resetful api规范
- koa实战案例

简介

Koa 是一个新的 web 框架，由 Express 幕后的原班人马打造， 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。 通过利用 async 函数，Koa 帮你丢弃回调函数，并有力地增强错误处理。 Koa 并没有捆绑任何中间件， 而是提供了一套优雅的方法，帮助您快速而愉快地编

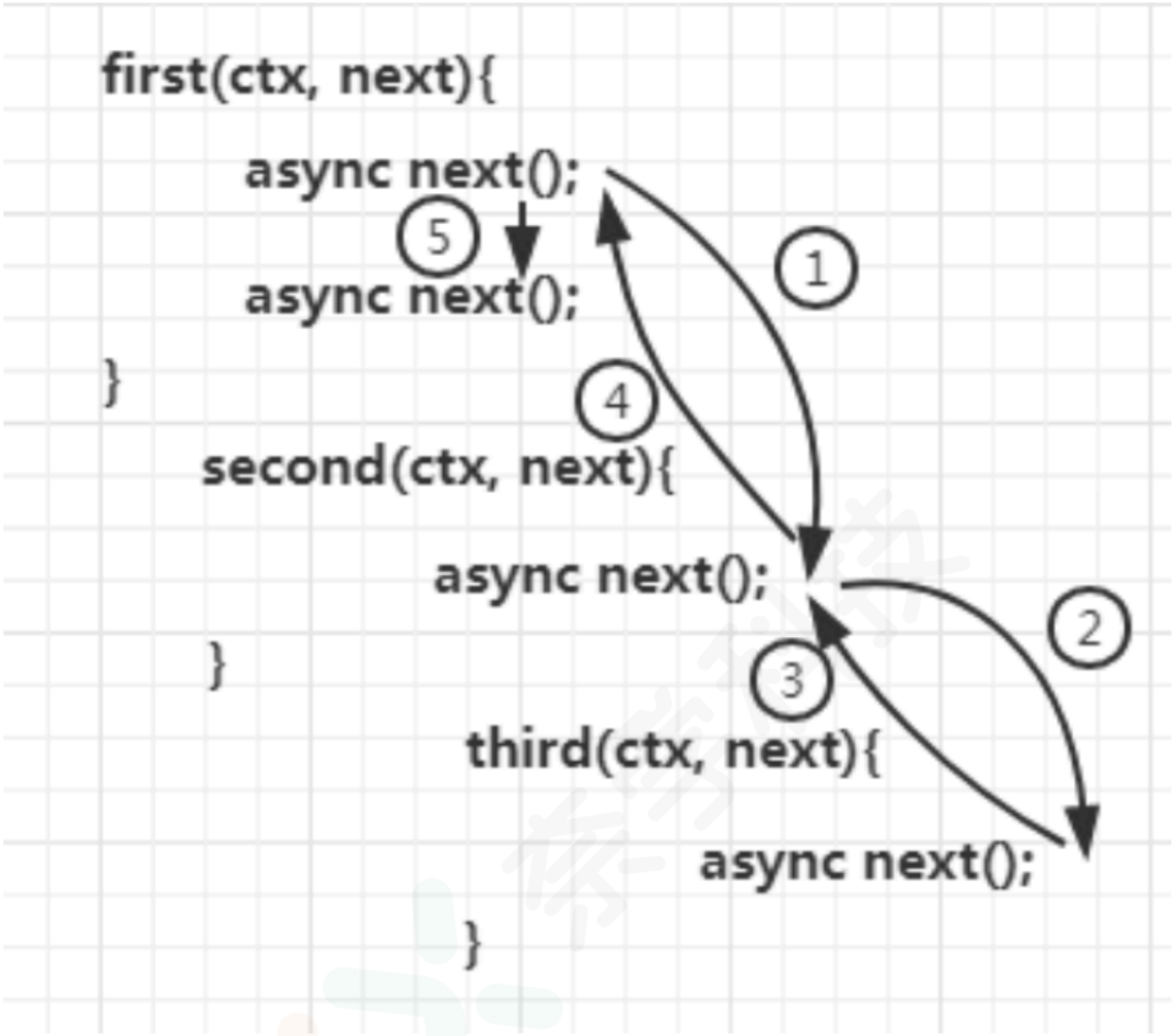
写服务端应用程序。

安装和应用

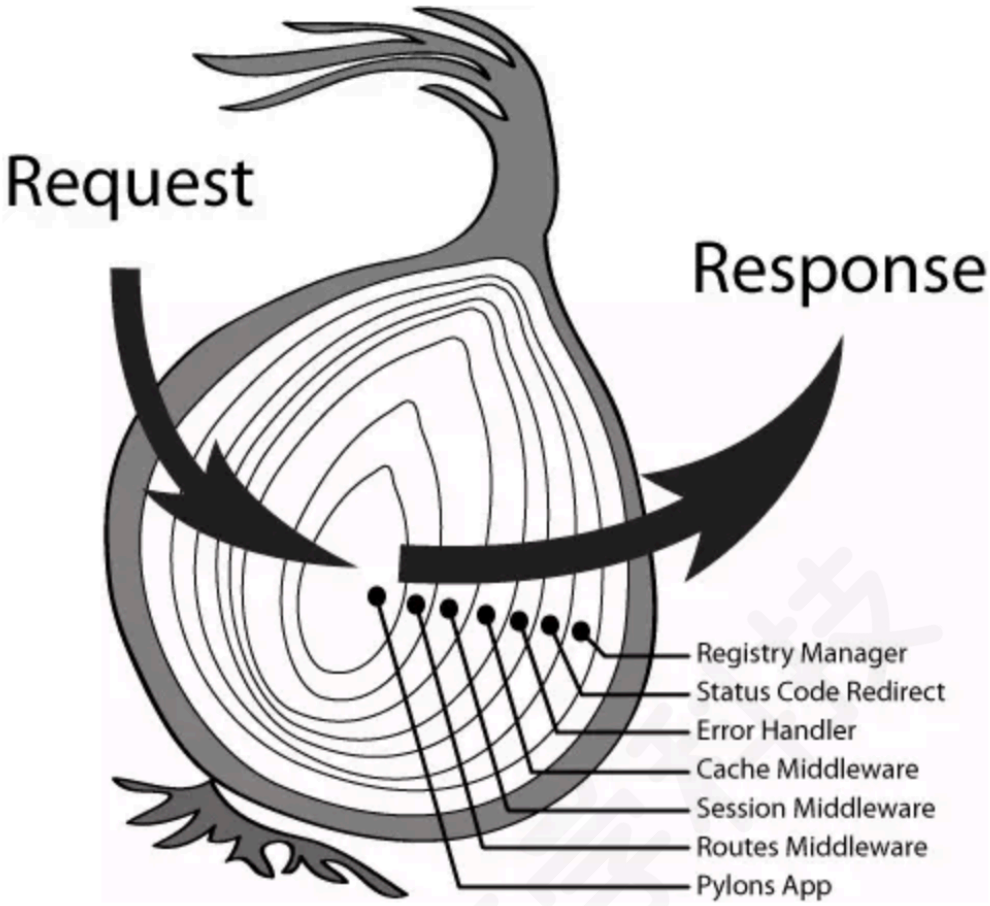
- `npm i koa -S`
- 快速搭建app应用

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

app.use(async ctx => {
  ctx.body = 'Hello World';
});

app.listen(3000);
```

中间件机制


first(ctx, next){
async next();
5
async next();
}
1
4
second(ctx, next){
async next();
2
3
}
third(ctx, next){
async next();
}

再配一张洋葱模型




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

// logger

app.use(async (ctx, next) => {
    console.log(1);
    await next();
    console.log(5);
    const rt = ctx.response.get('X-Response-Time');
```

```
    console.log(`${ctx.method} ${ctx.url} - ${rt}`);
});

// x-response-time

app.use(async (ctx, next) => {
    const start = Date.now();
    console.log(2);
    await next();
    console.log(4);
    const ms = Date.now() - start;
    ctx.set('X-Response-Time', `${ms}ms`);
});

// response
app.use(async ctx => {
    console.log(3);
    ctx.status = 200;//设置响应状态码
    ctx.type = 'html';//等价于ctx.set('Content-Type','text/html')
    ctx.body = 'Hello World'; //设置响应体
});
app.listen(3000); //语法糖 等同于 http.createServer(app.callback()).listen(3000)
```

打印结果：1 2 3 4 5

结论:当请求开始时先通过`X-Response-Time`和`logger`中间件,然后继续交给`response`中间件

当一个中间件调用`next()`则该函数暂停执行并将控制权传递给定义的下个中间件.当在`response`中间件执行后,下游没有更多的中间件.这个时候每个中间件恢复其上游行为

错误中间件

常见抛出异常和错误类型

- 代码语法不规范造成的JS报错异常
- 程序运行中发生的一些未知异常
- HTTP错误
- 自定义的业务逻辑错误

添加error全局事件侦听器

```
app.on('error',err=>{
    console.log('全局错误处理:',err.message);
})
```

```
const Koa = require('koa');
const http = require('http');
const https = require('https');
const app = new Koa();
/* app.use(async ctx=>{
    ctx.body = 'hello world'
}) */
```

```
// 错误处理中间件
app.use(async (ctx,next)=>{
    try {
        await next();
    } catch (error) {
        // 给用户显示状态码
        ctx.status = error.statusCode || error.status || 500;
        //如果是ajax请求,返回的是json错误数据
        ctx.type='json';
        // 给用户显示
        ctx.body = { ok: 0, message: error.message};
        // 系统日志
        ctx.app.emit('error',error,ctx);
        // console.log('捕获到错误',error.message);
    }
})

//.....

// 触发错误  koa帮咱们做了处理
app.use(async (ctx,next)=>{
    throw new Error('未知错误');
})
```

```
31
32 // response
33 //....
34
35 //全局错误处理  后台打印
36 app.on('error',err=>{
37     console.log('全局错误处理',err.message)
38 })
39 // app.listen(3000);
40 http.createServer(app.callback()).listen(3000);
```

koa-logger处理日志

koa-erros处理错误

`koa-log4` 比较好用的node环境下处理日志处理的模块

`koa-log4` 在 `log4js-node` 的基础上做了一次包装，是 `koa` 的一个处理日志的中间件，此模块可以帮助你按照你配置的规则分叉日志消息。

- 在根目录下新建 `logger/` 目录
- 在 `logger/` 目录下新建 `logs/` 目录，用来存放日志文件
- 在 `logger/` 目录下新建 `index.js` 文件

```
1 const path = require('path')
2 const log4js = require('koa-log4')
3
4 log4js.configure({
5   appenders: {
6     access: {
7       type: 'dateFile',
```

```
      // 生成文件的规则
      pattern: '-yyyy-MM-dd.log',
      // 文件名始终以日期区分
      alwaysIncludePattern: true,
      encoding: 'utf-8',
      // 生成文件路径和文件名
      filename: path.join(__dirname, 'logs', 'access')
    },
    application: {
      type: 'dateFile',
      pattern: '-yyyy-MM-dd.log',
      alwaysIncludePattern: true,
      encoding: 'utf-8',
      filename: path.join(__dirname, 'logs', 'application')
    },
    out: {
      type: 'console'
    }
  },
  categories: {
    default: { appenders: ['out'], level: 'info' },
    access: { appenders: ['access'], level: 'info' },
```

```
    application: { appenders: ['application'], level: 'WARN' }
  }
})

// // 记录所有访问级别的日志
// exports.accessLogger = () => log4js.koaLogger(log4js.getLogger('access'))
// // 记录所有应用级别的日志
// exports.logger = log4js.getLogger('application')

module.exports = {
  // 记录所有访问级别的日志
  accessLogger: () => log4js.koaLogger(log4js.getLogger('access')),
  // 记录所有应用级别的日志
  logger: log4js.getLogger('application')
}
```

- 访问级别的，记录用户的所有请求，作为koa的中间件，直接使用便可。修改 `app.js` 文件

```
const { accessLogger } = require('./logger')

app.use(accessLogger())
```

- 访问级别的，记录用户的所有请求，作为koa的中间件，直接使用便可。修改 `app.js`

文件

```
const { accessLogger } = require('./logger')

app.use(accessLogger())
```

- 应用级别的日志，可记录全局状态下的 `error`，修改 `app.js` 全局捕捉异常

```
const { logger } = require('./logger')

// 在try-catch错误是无法监听的
// 需要手动释放：ctx.app.emit('error', err, ctx)
// 或者在try-catch中直接logger.error(e)
// 在需要的代码中放入即可监听
app.on('error', (err, ctx) => {
  logger.error(err)

  // 这里可以优化下，开发环境才记录日志
  // if (process.env.NODE_ENV !== 'development') {
  //   logger.error(err)
  // }
})
```

- 应用级别的日志，也可记录接口请求当中的错误处理。

```
const { logger } = require('../logger')


router.get('/test', async ctx => {
  try {
    ddd()
    ctx.body = 'test'
  } catch (e) {
    logger.error(e)
    // 用这种方式手动释放也可以 - app.js文件里面，已经监听了error事件
    // ctx.app.emit('error', e, ctx)
    ctx.body = { code: -1, msg: e.message }
  }
})
```

路由中间件koa-router

- 安装 `npm i @koa-router -S`
- 使用 新建router/index.js和router/users.js

```
//index.js

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


router.get('/',(ctx,next)=>{
    ctx.body = '首页';
})
module.exports = router;
```

```
//users.js

const Router = require('@koa-router');
const router = new Router();
router.prefix('/users')

router.get('/', (ctx, next) => {
    ctx.body = '用户界面';
})
module.exports = router;

```

在app.js中导入并注册

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

const http = require('http');

const app = new Koa();
// 引入路由
const index = require('./router/index')
const users = require('./router/users')

// 注册路由
app.use(index.routes(),
index.allowedMethods());
app.use(users.routes(),
users.allowedMethods());

app.use(3000);
```

get请求

```
//users.js
//访问http://localhost:3000/users/3/1
router.get('/:id/:pid', (ctx, next) => {
    console.log(ctx.params.id,ctx.params.pid);
 // 3 1
    ctx.body = '用户界面1';
})
 //访问http://localhost:3000/user/3?
name=zhangsan
router.get('/:id', (ctx, next) => {
   //通过ctx.query获取查询的参数
    console.log(ctx.query.name);
    ctx.body = '用户界面2';
})
```

post请求

- post解析请求的参数需要下载 `koa-bodyParser`
- `npm i koa-bodyParser -S`
-

```
//app.js
const bodyParser = require('koa-bodyparser')

app.use(bodyParser());
```

```
//index.js
router.post('/',(ctx,next)=>{
    console.log(ctx.request.body);
    ctx.body = {'ok':1}
})
```

重定向

```
//index.js
router.get('/login',ctx=>{
  //判断用户是否处于登录状态...

  ctx.redirect('/sign-in');
  ctx.status = 301;
})
router.get('/sign-in',(ctx,next)=>{
  ctx.body = '注册页面'
})
```

静态文件中间件

路由有了之后,我们希望数据展示到页面,在koa中提供了很多的模板引擎,像pug,ejs,hbs

以hbs为例

下载中间件,设置静态文件目录

下载：`npm i koa-static -S`

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

模板引擎

- 为什么要使用模板引擎
  - 简单来说，模板最本质的作用是“变静为动”，一切利于这方面的都是优势，不利于的都是劣势。

    要想很好地实现“变静为动”的目的，有这么几点：

    1．可维护性（后期改起来方便）；

    2．可扩展性（想要增加功能，增加需求方便）；

    3．开发效率提高（程序逻辑组织更好，调试方便）；

    4.看起来舒服（不容易写错）；

    从以上四点来看，前端模板引擎体现的优势都不是一点两点的。

    其实最重要的一点就是：**视图（包括展示渲染逻辑）与程序逻辑的分离**，分离的好处太多了，比如说后期的维护修改代码，增加代码，调试代码，和应用开发模式（MVC、MVVM）都方便很多

- `npm i koa-hbs@next -S`
- 使用
  - 在app.js

-

```
const hbs = require('koa-hbs');
// 注册中间件
app.use(hbs.middleware({
    viewPath: __dirname + '/views',//视图根目录
    defaultLayout:'layout',//默认布局页面
    partialsPath:__dirname +'/views/partials',//注册partials目录
    disableCache:true //开发阶段不缓存
}));
```

- 创建views目录,结构如下

```
├──views
    ├── index.hbs
    ├── layout.hbs
    └── partials
```

- index.js

```
router.get('/',async (ctx,next)=>{
  const data = {
    title:'hbs',
    subTitle: "hello hbs",
    htmlStr:'<h4>hello h4</h4>',

    isShow:true,
    username:'张三',
    users:[{name:"小黄",age:20},{name: "小红",age: 25},{name: "小蓝",age: 27}
    ]
   }
  await ctx.render('index', data);
})
```

- layout.hbs

```
<!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>{{title}}</title>
</head>
<body>
    <!--以后所有的子模板的内容渲染到当前位置 body为固定写法-->
    {{{body}}}
</body>
</html>
```

- index.hbs

```
{{!-- 1.插值绑定 --}}
<h2>{{subTitle}}</h2>
{{!-- 2.插html --}}
<div>{{{htmlStr}}}</div>

```

```
{{!-- 3.条件判断 --}}
{{#if isShow}}
<p>{{username}},欢迎你!</p>
{{else}}
<a href="">请登录</a>
{{/if}}

{{!-- 4.循环 --}}
<ul>
    {{#each users}}
    <li>{{name}}-{{age}}</li>
    {{/each}}
</ul>
```

- 公共模板
  - 在partical创建nav.hbs
  - 在views/layout.hbs使用

```
{{>nav}}
```

- 帮助方法:扩展handlebars的功能函数

  创建/utils/helpers.js

```
const hbs = require('koa-hbs');
const moment = require('moment');
hbs.registerHelper('date', (date, pattern)=>{
    try{
        return moment(date).format(pattern);
    }catch(error){
        return "";
    }
})
```

在app.js中导入

```
const helpers = require('./utils/helpers');
```

/router/index.js 修改数据

```
users:[
    {
        name:"小黄",
        age:20,
        birthday:new Date(1999,2,2)
    },
    {
        name: "小红",
        age: 25,
        birthday: new Date(1994, 3, 2)
```

```
    },
    {
        name: "小蓝",
        age: 27,
        birthday: new Date(1995, 10, 2)
    }
]
```

/views/index.hbs

```
<ul>
    {{#each users}}
    <li>{{name}}-{{age}}--{{date birthday 'YYYY/MM/DD'}}</li>
    {{/each}}
</ul>
```

- 调用helper库的方法

> **N多帮助方法参考**
>
> 1.下载
>
> `npm install --save handlebars-helpers`
>
> 2.使用

```
const helpers = require('handlebars-helpers');
helpers.comparison({handlebars:hbs.handlebars})
```

3.index.hbs

```
<div>{{#and a b}} a,b都是true {{/and}}</div>

<div>
 {{#contains arr 'e' }}
 这e在数组中
 {{else}}
 这e不在数组中
 {{/contains}}

</div>
```

还有众多帮助方法,大家自行浏览,当使用到某个帮助方法的时候,再来查阅

- 高级应用:代码搬家

  定义代码块,views/index.hbs

```
{{#contentFor 'jquery'}}
    <script>
        $(function(){
            console.log('content for jquery');
        })
    </script>
{{/contentFor}}
```

代码搬家,views/layout.hbs

```
<script src="https://cdn.bootcss.com/jquery/3.4.1/jquery.min.js"></script>
{{#block 'jquery'}}{{/block}}
```