Koa基础招式

一、基础

1、简介

koa 是由 Express 原班人马打造的，致力于成为一个更小、更富有表现力、更健壮的 Web 框架。使用 koa 编写 web 应用，通过组合不同的 generator，可以免除重复繁琐的回调函数嵌套，并极大地提升错误处理的效率。koa 不在内核方法中绑定任何中间件，它仅仅提供了一个轻量优雅的函数库，使得编写 Web 应用变得得心应手。 koa仅仅只是基于原生NodeJs的 `http` 模块做了一系列的扩展，让开发者更加简便的用NodeJs开发web服务应用。

koa只提供了4个核心模块，它们分别是：Application、Context、Request和Response。

2、初识

新建项目，运行安装命令，安装Koa

```
npm install koa
```

创建入口文件，开启第一个node服务

```
const Koa = require('koa')
const app = new Koa()
app.use(async (ctx)=>{
  const { name } = ctx.query
  ctx.body = {
    msg:name
  }
})
app.listen(3000,()=>{
  console.log('web server is running')
})
```

至此一个简单的node服务代码已经完成，启动该服务只需要运行命令 `node index.js` 。在这一段不到20行的代码中，包含了koa的4个核心模块。 `new Koa()` 实例化得到的就是一个Application应用； `use()` 方法接收的函数，这个函数的的参数 `ctx` 就是 `context` 上下文对象；从 `ctx.query` 中获取请求参数，是从 `Request` 对象中获取的；而将结果响应给浏览器的 `ctx.body` 实际上是 `Response` 对象发送响应数据。

3、应用

Application应用，通过用 `new` 关键字实例化Koa返回一个Application实例，这个Application实例是一个包含一系列中间件generator函数的对象。**Application**类实现了一系列的属性和方法，其中重要的属性、方法如下：

①、listen()

`listen()` 方法的作用是创建一个http服务并且监听指定的端口，实际上 `listen()` 方法是做了原生nodejs的创建并监听服务这两件事。

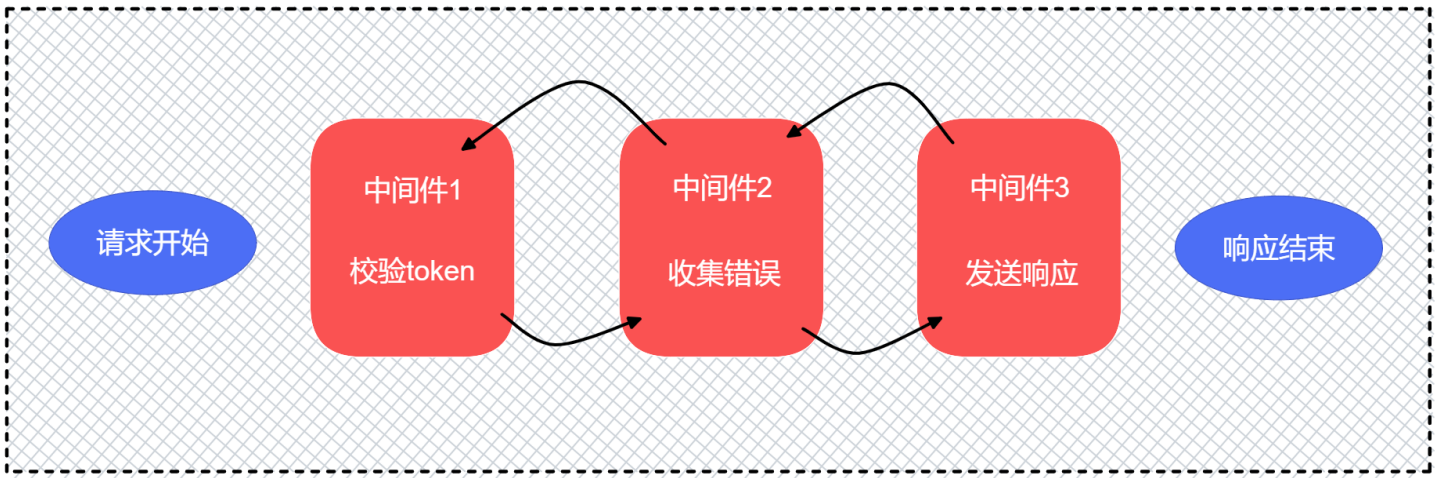
```
app.listen(3000)
// 等价于
const server = http.createServer((req,res)=>{})
server.listen()
```

②、callback()

`callback()` 方法返回一个用来作为 `http.createServer()` 的回调函数的函数处理请求，`http.createServer()` 的回调函数需要具备 `request` 和 `response` 参数。

③、use()

`use()` 方法的作用是为koa应用添加指定的中间件，该方法它会接收一个函数作为参数，接收的这个函数就是中间件，也叫作中间件函数。这个中间件函数就是用于处理http请求的函数，只要有请求进来，就一定会执行中间件函数。可以将它理解成，这个函数只存在于请求开始和响应结束这段期间，在这短期间，可以存在很多个中间件函数，也就是 `use()` 方法可以调用多次，添加多个中间件函数。



中间件的执行顺序是从第一个中间件到第二个一直到最后一个，再从最后一个反向执行回到第一个。

④、context

这里的 `context` 允许开发者通过编辑 `app.context` 向ctx添加属性或方法，当你的应用需要有一个公共的属性或方法用它添加是非常有用的。但是不允许直接获取它，比如你想看看 `app.context`

输出是什么，这种做法是不允许的。

```
app.context.sign='yiketang1234'
```

这样做之后，在每一个中间件，你都可以用 `ctx.sign` 获取到它。

4、上下文

koa的上下文 `context` 对象将原生nodejs的 `request` 和 `response` 对象单独做了一次封装，并且它还将许多属性做了一层映射，比如访问 `ctx.query` 时， `ctx.query` 映射到 `ctx.request.query` ，这么做极大的方便了编写web应用的便捷性。

context在每个request请求中被创建，每个中间件函数的ctx参数就是 `context` 上下文对象，所以在中间件函数中可以使用 `ctx.body` 、 `ctx.query` 等。 `context` 对象提供的方法和访问器如下：

①、ctx.req和ctx.res

`ctx.req` 和 `ctx.res` 分别是原生Nodejs中的request对象和response对象。

②、request和response

`ctx.request` 和 `ctx.response` 分别是koa实现的Request对象和Response对象

③、app

Application应用实例的引用

④、ctx.cookies.set()

用于设置cookie，它的语法是

```
ctx.cookies.set(name,value,[options])
```

name为设置的cookie，value为cookie值。options为可选参数，可用于配置签名、有效期等

⑤、ctx.cookies.get()

获取设置的cookie，语法是：

```
ctx.cookies.get(name,[options])
```

⑥、ctx.throw()

用于抛出包含status属性的错误，默认为500，该方法可以 让Koa准确的响应处理状态。

5、请求

Request对象是对原生nodejs的request进行了封装，kao的request对象内部定义了许多访问器，访问的属性都映射到原生nodejs的request上面。

```
1 console.log(ctx.request.method)
```

如上的代码当访问method属性，它会触发koa的request对象的method访问器，本质上还是从原生nodejs的req上访问

```
1  get method() {
2     return this.req.method;
3 }
```

如下Koa Request对象提供的常用的API

request.header	获取或设置请求头对象
request.method	获取或设置请求方法
request.length	以数字的习惯是返回request的内容长度(Content-Length)
request.url	获取或设置请求地址
request.originalUrl	获取请求的原始地址
request.origin	获取URL原始地址，包含protocol和host 如：http://example.com/
	获取完整的请求URL，包

request.href	含protocol、host和url 如：http://example.com/foo/bar?q=1
request.path	获取或设置请求路径名
request.querystring	获取查询参数字符串，？后面的部分，或设置查询字符串
request.search	获取查询参数字符串，包含?，或设置查询字符串
request.query	获取查询参数字符串的对象形式

6、响应

Response对象是对原生nodejs的response进行了封装，kao的response对象内部定义了许多访问器，访问的属性都映射到原生nodejs的response上面。

Koa Response对象提供的常用web服务开发API如下：

response.header	
response.status	获取或设置响应状态
response.message	获取或响应状态消息
response.body	获取或设置响应体
response.set()	设置header字段 ctx.response.set('Cache-Control','no-cache')

二、路由

Koa本身是没有提供路由功能的，你要使用路由根据访问的url地址不同而响应不同的数据，就需要引入配套的 `@koa/router` 这个路由库。 Koa路由的作用就是根据请求的方式(`GET` 、 `POST` 、 `PUT` 等)和请求的URL路径匹配到对应的响应程序。

在项目中安装

```
npm install @koa/router
```

1、初识

引入 `@koa/router` 并对其进行实例化，可以调用实例方法 `get` 、 `post` 等，接收的是一个url路径和一个回调函数。

```
const Router = require('@koa/router')
const router = new Router()
router.get('/home',(ctx)=>{
  ctx.body = {
    msg:'home'
  }
})
router.get('/news',(ctx)=>{
  ctx.body = {
    msg:'news'
```

```
  }
})
```

这里只是定义了路由，但是还没有真正使用它，使用它要调用 `routes()` 方法，该方法返回的是路由器中间件，将它作为中间件函数作为use方法的参数。

```
app.use(router.routes())
```

当访问 `/home` 和 `/news` 的时候就会响应对应的内容。

2、前缀

如果你的路由需要按模块设计，比如用户模块的路由统一前缀 `/user/**` ，订单模块的路由统一前缀为 `/order/**` 。为了方便，你可以在实例化的时候添加配置项 `prefix` ， `prefix` 配置的值将作为路由前缀，再定义路由的时候你可以不必每个都加上模块前缀。

```
const router = new Router({
  prefix:'/user'
})
router.get('/info',(ctx)=>{
  ctx.body = {
    msg:'userinfo'
  }
})
router.post('/update',(ctx)=>{
  ctx.body = {
    msg:'update'
  }
})
```

3、参数

这里的请求参数不是指koa中的查询参数，而是指路由参数。比如你的请求接口是新闻的详情数据，路由可以设计成如下方式访问，表示id为1的详情页、id为2的详情页和id为3的详情页。

```
/detail/1
/detail/2
/detail/3
```

这种访问方式可以通过路由参数的方式实现，路由定义成如下格式：

```
router.get('/detail/:id',ctx=>{
  console.log(ctx.params)
  ctx.body = {
    msg:'info'
  }
})
```

`:id` 可以看成是一个动态的id值路由，通过 `ctx.params` 可以获取到参数对象 `{id:**}`

4、中间件

`@koa/router` 也支持中间件的使用，通过调用 `use()` 方法给指定的访问路径添加中间件函数，当指定的路径添加上了中间件函数后每次访问该路径都会经过中间，基于此可以给指定的路径添加路由权限中间件。

router实例的 `use()` 方法语法如下：

```
router.use([path],middleware)
```

- path路径参数是可选参数，它可以是一个字符串，也可以是要给包含路径字符串的数组
- middleware参数是必填参数，一个中间件函数

三、取参

在服务端获取客户端向服务器发送请求时携带的参数，这里以 `GET` 请求和 `POST` 请求为例。

1、取get参数

在koa中 koa的Request对象已经实现如何获取 `GET` 请求的参数， `ctx.query` 和 `ctx.request.query` 都是以对象的形式返回查询参数。

```
// 请求路径为:  /user/info?name=yiketang
router.get('/user/info',(ctx)=>{
  console.log(ctx.query)

  console.log(ctx.request.query)
})
// 输出的结果都是:  {name:'yiketang'}
```

能通过上面两种方式取到GET参数，是因为在 `Request` 对象内部实现了 `query` 属性的访问器，当访问 `query` 属性的时候就会触发访问器，在访问器内部获取到查询参数并解析成对象形式返回。

2、取post参数

koa中并不能直接就获取到post请求携带的参数，要获取post参数成熟的方式就是引入已经成熟的库 `koa-bodyparser` ，这个库本质上就是koa的一个中间件，用于处理接收到请求后解析出post请求的参数。

安装

```
npm install koa-bodyparser
```

使用

只需要引入，并将它作为use的参数即可，然后在路由中就可以使用 `ctx.request.body` 获取到post的请求参数

```
const bodyParser = require('koa-bodyParser')
app.use(bodyParser)
router.post('update',(ctx)=>{
  console.log(ctx.request.body)
})
```

实现

`koa-bodyParser` 内部基于 `co-body` 这个库，核心就是通过原生nodeJs的 `data` 事件获取到请求体并解析，之后再给 `ctx.request` 添加 `body` 属性，值就等于请求体解析完成后的结果。来看下最简单的代码实现：

```
function bodyParser(req){
  req = req.req || req;
  return new Promise((resolve)=>{
    var postStr = ''
    req.on('data',function(chunk){
      postStr+=chunk
    })
    req.on('end',function(){
      console.log(postStr,'=====')
      resolve(JSON.parse(postStr))
    })
  })
}
app.use(async (ctx,next) => {
  ctx.request.body = await bodyParser(ctx)
  await next()
```

```
17 })
```