KOA2的使用

1.KOA2的介绍

- 基于Node.js平台的Web服务器框架
- 由Express原班人马打造

  Express Koa,Koa2都是Web服务器的框架,他们之间的差别和关系可以通过下面这个表格表示出

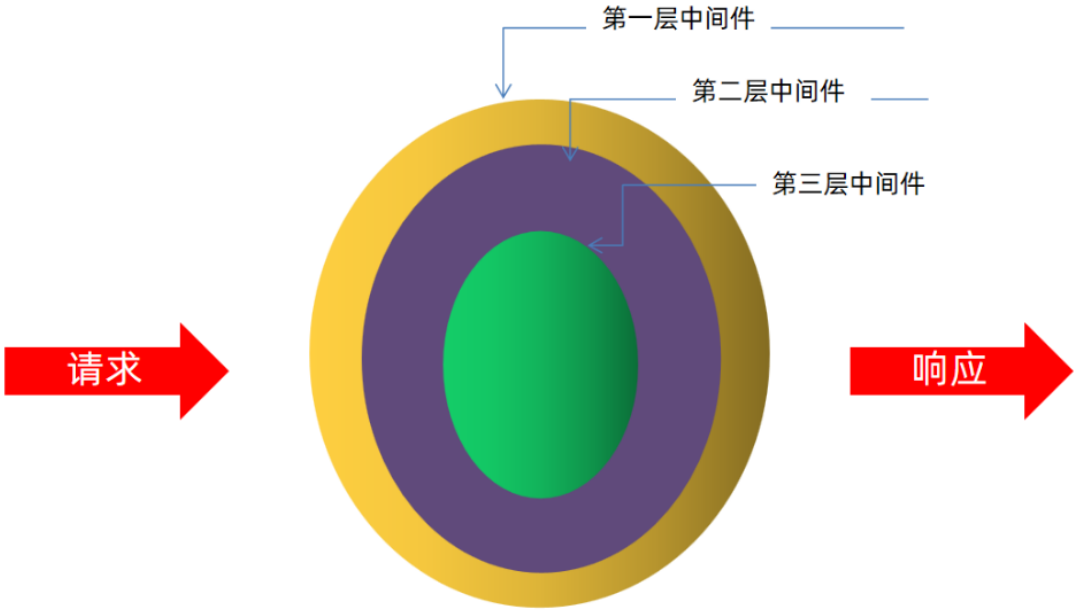
框架名	作用	异步处理
Express	web 框架	回调函数
Koa	web 框架	Generator+yield
Koa2	web 框架	async/await

- 环境依赖 Node v7.6.0及以上

  由于Koa2它是支持async和await，所以它对Node的版本是有要求的，它要求Node的版本至少是在7.6级以上,因为语法糖async和await是在Node7.6版本之后出现才支持

- 洋葱模型的中间件

  如下图所示, 对于服务器而言，它其实就是来处理一个又一个的请求，Web服务器接收由浏览器发过来的一个又一个请求之后，它形成一个又一个的响应返回给浏览器. 而请求到达我们的服务器是需要经过程序处理的,程序处理完之后才会形成响应，返回给浏览器，我们服务器处理请求的这一块程序，在Koa2的世界当中就把它称之为中间件



  这种中间件可能还不仅仅只有一个，可能会存在多个，比如上图所示, 它就存在三层中间件，这三层中间件在处理请求的过程以及它调用的顺序为:

  - 当一个请求到达咱们的服务器，最先最先处理这个请求的是第一层中间件
  - 第一层的中间件在处理这个请求之后，它会把这个请求给第二层的中间件
  - 第二层的中间件在处理这个请求之后，它会把这个请求给第三层的中间件
  - 第三层中间件内部并没有中间件了, 所以第三层中间件在处理完所有的代码之后，这个请求又会到了第二层的中间件，所以第二层中间件对这个请求经过了两次的处处理

- 第二层的中间件在处理完这个请求之后，又到了第一层的中间件, 所以第一层的中间件也对这个请求经过了两次的处理

这个调用顺序就是洋葱模型, 中间件对请求的处理有一种先进后出的感觉，请求最先到达第一层中间件，而最后也是第一层中间件对请求再次处理了一下

2.Koa2的快速上手

如何对`Koa2`进行快速的上手呢?需要有如下几个步骤

- 检查`Node`的版本
  - `node -v`的命令可以帮助我们检查`Node`的版本, `Koa2`的使用要求`Node`版本在7.6及以上

```
PS C:\Users\itheima\Desktop\数据可视化\day03\03.代码\koa_study> node -v
v12.14.1
```

- 安装`Koa2`
  - `npm init -y`

    这个命令可以快速的创建出`package.json`的文件, 这个文件可以维护项目中第三方包的信息
  - `npm install koa`

    这个命令可以在线的联网下载最新版本`koa`到当前项目中, 由于线上最新版本的`koa`就是`koa2`, 所以我们并不需要执行`npm install koa2`

    如果下载特别慢的话, 需要将`npm`的下载源换成国内的下载源, 命令如下

```
npm set registry https://registry.npm.taobao.org/
```

- 编写入口文件`app.js`
  - 创建`Koa`的实例对象

```
// 1.创建koa对象
const Koa = require('koa') // 导入构造方法
const app = new Koa() // 通过构造方法, 创建实例对象
```

  - 编写响应函数(中间件)

    响应函数是通过use的方式才能产生效果, 这个函数有两个参数, 一个是`ctx`,一个是`next`

    - `ctx`:

      上下文, 指的是请求所处于的Web容器,我们可以通过`ctx.request`拿到请求对象, 也可以通过`ctx.response`拿到响应对象
    - `next`:

      内层中间件执行的入口

```
// 2.编写响应函数(中间件)
app.use((ctx, next) => {
  console.log(ctx.request.url)
  ctx.response.body = 'hello world'
})
```

  - 指明端口号

    通过`app.listen`就可以指明一个端口号

```
// 3.绑定端口号 3000
app.listen(3000)
```

- 启动服务器

  通过 `node app.js` 就可以启动服务器了

随即打开浏览器, 在浏览器中输入`127.0.0.1:3000/` 你将会看到浏览器中出现 `hello world` 的字符串, 并且在服务器的终端中, 也能看到请求的`url`

3.KOA2中间件的特点

- `Koa2`的实例对象通过`use`方法加入一个中间件
- 一个中间件就是一个函数,这个函数具备两个参数,分别是`ctx`和`next`
- 中间件的执行符合洋葱模型
- 内层中间件能否执行取决于外层中间件的`next`函数是否调用
- 调用`next`函数得到的是`Promise`对象, 如果想得到`Promise`所包装的数据, 可以结合`await`和`async`

```
app.use(async (ctx, next) => {
     // 刚进入中间件想做的事情
     await next()
    // 内层所有中间件结束之后想做的事情
})
```

4.后台项目的开发

4.1.后台项目的目标

我们已经学习完了`KOA2`的快速上手, 并且对`KOA2`当中的中间件的特点页进行了讲解. 接下来就是利用`KOA2`的知识来进行后台项目的开发，后台项目需要达到这以下几个目标:

- 1.计算服务器处理请求的总耗时

  计算出服务器对于这个请求它的所有中间件总耗时时长究竟是，我们需要计算一下

- 2.在响应头上加上响应内容的`mime`类型

  加入mime类型, 可以让浏览器更好的来处理由服务器返回的数据.

  如果响应给前端浏览器是`json`格式的数据，这时候就需要在咱们的响应头当中增加`Content-Type`它的值就是`application/json`，`application/json`就是`json`数据类型的`mime`类型

- 3.根据URL读取指定目录下的文件内容

  为了简化后台服务器的代码，前端图表所要的数据, 并没有存在数据库当中，而是将存在文件当中的，这种操作只是为了简化咱们后台的代码. 所以咱们是需要去读取某一个目录下面的文件内容的。

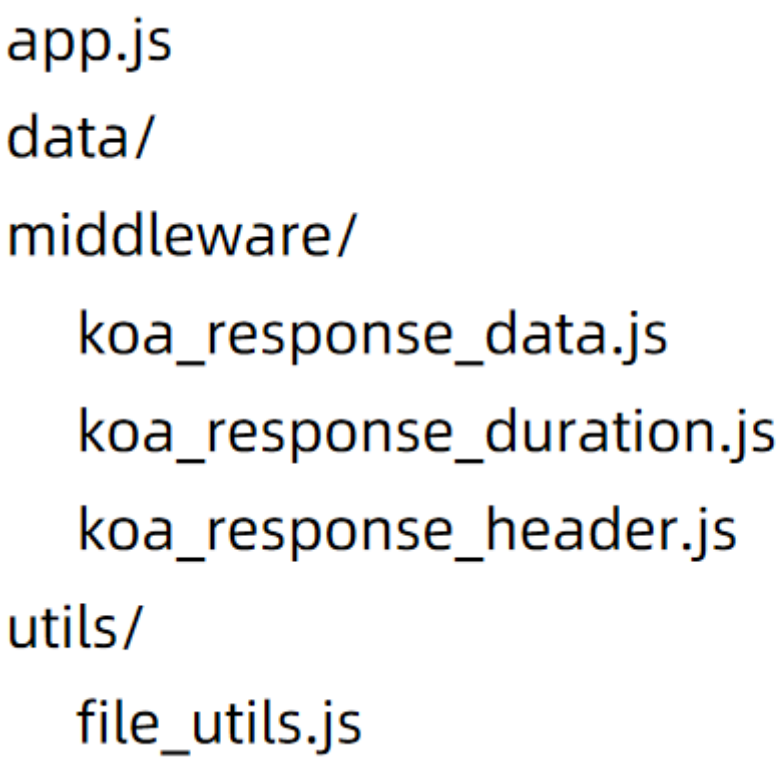
每一个目标就是一个中间件需要实现的功能, 所以后台项目中需要有三个中间件

4.2.后台项目的开发步骤

创建一个新的文件夹, 叫做`koa_server`, 这个文件夹就是后台项目的文件夹

- 1.项目准备

- 1.安装包
  - `npm init -y`
  - `npm install koa`
- 2.创建文件和目录结构



`app.js`是后台服务器的入口文件

`data`目录是用来存放所有模块的`json`文件数据

`middleware`是用来存放所有的中间件代码

`koa_response_data.js`是业务逻辑中间件

`koa_response_duration.js`是计算服务器处理时长的中间件

`koa_response_header.js`是用来专门设置响应头的中间件

接着将各个模块的`json`数据文件复制到`data`的目录之下, 接着在`app.js`文件中写上代码如下:

```
// 服务器的入口文件
// 1.创建KOA的实例对象
const Koa = require('koa')
const app = new Koa()
// 2.绑定中间件
// 绑定第一层中间件
// 绑定第二层中间件
// 绑定第三层中间件
// 3.绑定端口号 8888
app.listen(8888)
```

- 2.总耗时中间件
  - 1.第1层中间件

    总耗时中间件的功能就是计算出服务器所有中间件的总耗时，应该位于第一层，因为第一层的中间件是最先处理请求的中间件，同时也是最后处理请求的中间件
  - 2.计算执行时间

第一次进入咱们中间件的时候，就记录一个开始的时间

当其他所有中间件都执行完之后，再记录下结束时间以后

将两者相减就得出总耗时

- 3.设置响应头

  将计算出来的结果,设置到响应头的`X-Response-Time`中, 单位是毫秒`ms`

具体代码如下:

`app.js`

```
// 绑定第一层中间件
const respDurationMiddleware =
 require('./middleware/koa_response_duration')
app.use(respDurationMiddleware)
```

`koa_response_duration.js`

```
// 计算服务器消耗时长的中间件
module.exports = async (ctx, next) => {
  // 记录开始时间
  const start = Date.now()
  // 让内层中间件得到执行
  await next()
  // 记录结束的时间
  const end = Date.now()
  // 设置响应头 X-Response-Time
  const duration = end - start
  // ctx.set 设置响应头
  ctx.set('X-Response-Time', duration + 'ms')
}
```

- 3.响应头中间件
  - 1.第2层中间件

    这个第2层中间件没有特定的要求

  - 2.获取`mime`类型

    由于咱们所响应给前端浏览器当中的数据都是`json`格式的字符串，所以`mime`类型可以统一的给它写成`application/json`,当然这一块也是简化的处理了，因为`mime`类型有几十几百种，我们我们没有必要在我们的项目当中考虑那么多，所以这里简化处理一下

  - 3.设置响应头

    响应头的key是`Content-Type`，它的值是`application/json`,顺便加上`charset=utf-8`

    告诉浏览器，我这部分响应的数据，它的类型是`application/json`，同时它的编码是`utf-8`

具体代码如下:

`app.js`

```
// 绑定第二层中间件
const respHeaderMiddleware = require('./middleware/koa_response_header')
app.use(respHeaderMiddleware)
```

`koa_response_header.js`

```
// 设置响应头的中间件
module.exports = async (ctx, next) => {
  const contentType = 'application/json; charset=utf-8'
  ctx.set('Content-Type', contentType)
  await next()
}
```

- 4.业务逻辑中间件
  - 1.第3层中间件

    这个第3层中间件没有特定的要求
  - 2.读取文件内容
    - 获取`URL`请求路径

      ```
      const url = ctx.request.url
      ```

    - 根据URL请求路径,拼接出文件的绝对路径

      ```
      let filePath = url.replace('/api', '')
      filePath = '../data' + filePath + '.json'
      filePath = path.join(__dirname, filePath)
      ```

      这个`filePath`就是需要读取文件的绝对路径
    - 读取这个文件的内容

      使用`fs`模块中的`readFile`方法进行实现
  - 3.设置响应体

    `ctx.response.body`

具体代码如下:

`app.js`

```
// 绑定第三层中间件
const respDataMiddleware = require('./middleware/koa_response_data')
app.use(respDataMiddleware)
```

`koa_response_data.js`

```
// 处理业务逻辑的中间件,读取某个json文件的数据
const path = require('path')
const fileUtils = require('../utils/file_utils')
module.exports = async (ctx, next) => {
  // 根据url
  const url = ctx.request.url // /api/seller   ../data/seller.json
  let filePath = url.replace('/api', '') //  /seller
  filePath = '../data' + filePath + '.json'  // ../data/seller.json
  filePath = path.join(__dirname, filePath)
  try {
    const ret = await fileUtils.getFileJsonData(filePath)
    ctx.response.body = ret
```

```
  } catch (error) {
    const errorMsg = {
      message: '读取文件内容失败, 文件资源不存在',
      status: 404
    }
    ctx.response.body = JSON.stringify(errorMsg)
  }

  console.log(filePath)
  await next()
}
```

`file_utils.js`

```
// 读取文件的工具方法
const fs = require('fs')
module.exports.getFileJsonData = (filePath) => {
  // 根据文件的路径, 读取文件的内容
  return new Promise((resolve, reject) => {
    fs.readFile(filePath, 'utf-8', (error, data) => {
      if(error) {
        // 读取文件失败
        reject(error)
      } else {
        // 读取文件成功
        resolve(data)
      }
    })
  })
}
```

- 5.允许跨域
  - 设置响应头

```
app.use(async (ctx, next) => {
    ctx.set("Access-Control-Allow-Origin", "*")
    ctx.set("Access-Control-Allow-Methods", "OPTIONS, GET, PUT, POST,
DELETE")
    await next();
})
```