Nodejs

js的插值语法`${y}_${m}_${d}  ${HH}_${mm}_${ss}`，这两旁的``实现插值语法的

1.fs文件系统模块

- fs.readFile(path,[,options],callback)方法，用来读取指定文件中的内容

参数1：必选参数，字符串，表示文件的路径

参数2：可选参数，表示以什么编码方式来读取文件

参数3：必选参数，文件读取完后，通过回调函数拿到读取的结果

```
const fs = require('fs')   //引入模块
fs.readFile('./files/1.txt','utf-8',function(err,dataStr){
    console.log(err)//错误的时候返回的值是对象，想要知道错误,使用err.message
    console.log("----")
    console.log(dataStr) //读取的数据
})
```

- fs.writeFile(file,data,[,options],callback)方法，用来向指定的文件中写入内容，有这个文件的话，重写这个文件内容，没有则新建文件

参数1：必选参数，需要指定一个文件路径的字符串，表示文件的存放路径

参数2：必选参数，表示要写入的内容

参数3：可选参数，表示以什么格式写入文件内容，默认是utf-8

参数4：必选参数，文件写入完成后的回调函数

```
const fs = require('fs')
fs.writeFile('./files/1.txt','枫',(err)=>{
    console.log(err)//失败的原因，和上面的一样
})
```

3.整理文件内容

将`小红=70 小黄=80 小兰=90`变成竖着排布的

```
const fs = require('fs')
fs.readFile('./files/1.txt','utf-8',(err,data)=>{
    const dataList = data.split(' ')
    let s = ""
    dataList.forEach((item)=>{
        s+=item+'\n'
    })
    fs.writeFile('./files/1.txt',s,(err)=>{
        console.log(err)
    })
})
```

4.路径动态拼接问题：`__dirname`来获取当前的文件路径，文件拼接

```
console.log(__dirname)
```

5.使用path模块处理路径

先导入相关的模块`const path = require('path')`

- path.join([...paths])方法，用来将多个路径片段拼接成一个完整的路径字符串

参数1：...paths路径片段的序列

返回：，拼接之后的路径

```
const path = require('path')
console.log(path.join(__dirname,'/1.js')
```

- path.basename(path[,ext])方法，用来从路径字符串中，将文件名解析出来

参数1：path必选参数，表示一个路径的字符串

参数2：ext可选参数，表示文件的拓展名

返回：表示路径中的最后一部分

```
const path = require('path')
const fpath = '/a/b/c/index.html'
console.log(path.basename(fpath)) //index.html
console.log(path.basename(fpath,'.html')) //index
```

- path.extname(path)可以获取路径中的拓展名部分

参数1：path，必选参数，文件的路径

返回：返回得到的拓展字符串名

```
const path = require('path')
const fpath = '/a/b/c/index.html'
console.log(path.extname(fpath))//.html
```

2.HTTP模块

1.创建最基本的web服务器

- 导入http模块
- 创建web服务器实例
- 为服务器实例绑定`request`事件，监听客户端的请求
- 启动服务器

```
const http = require('http')
const server = http.createServer()
server.on('request',(req,res)=>{
    console.log("欢迎进入我的localhost")
})
server.listen(7878,()=>{
    console.log("启动的时候显示的代码")
})
```

2.req请求对象

req是请求对象，他包含了与客户端相关的数据和属性,例如：

- req.url是客户端请求的url地址
- req.method是客户端的method请求类型

3.res响应对象

res是响应对象，他包含了与服务器相关的数据和属性，例如：

- res.end() 向客户端发送指定的内容，并结束这次请求的处理过程

```
res.end(str)//将数据显示在页面上
```

注意中文乱码问题：`res.setHeader("Content-Type",'text/html;charset=utf-8')`

4.根据不同的url的相应不同的html内容

- 获取请求的url地址
- 设置默认的响应头内容为404 Not found
- 判断用户请求的是否是为`/`或`/index.html`
- 判断用户请求的是否是为`/about.html`
- 设置`Content-Type`响应头，防止中文乱码问题
- 使用`res.end()`将内容相应给客户端

```
const http = require('http')
const server = http.createServer()
server.on('request',(req,res)=>{
    const url = req.url
    let content = "<h1>404 not found </h1>"
    if(url==='/'||url==='/index.html'){
        content = "<h1>首页</h1>"
    }else if(url==='/about.html'){
        content = "<h1>关于页面</h1>"
    }
    res.setHeader("Content-Type",'text/html;charset=utf-8')
    res.end(content)
})
server.listen(7878,()=>{
    console.log("启动的时候显示的代码")
})
```

注意：读取图片`binary`的读取方式

```
const url = req.url
res.writeHead(200,{'Content-Type' : 'image/jpeg'});
if(url==='/favicon.ico'){
    const fpath = path.join(__dirname,url)
    fs.readFile(fpath,'binary',(err,data)=>{
        res.write(data,'binary')
        res.end()
    })
}
```

3.模块化

1.加载模块

使用require()方法，可以加载需要的内置模块，用户自定义模块，第三方模块进行使用。

2.**module.exports对象 === exports对象**　中间实例对象，传值用的

在自定义模块中，可以使用module.exports对象，将模块中的成员共享出去，共外界使用。外界使用`require()`方法导入自定义模块时，得到的就是module.exports所指向的对象，这个对象默认就是空的，这里以module.exports对象的指向为准

通过这个实例对象，可以实现模块间的传值

2.js

```
module.exports.username='feng'
module.exports.SayHello=()=>{
    console.log("Hello")
}
```

1.js

```
const out = require('./2')
console.log(out.username)  //feng
console.log(out.SayHello())   //Hello
```

让module.exports指向一个新的对象，通过这种方法可以解除属性的绑定

```
module.exports={
    name:'feng',
    sayHello:()=>{
        console.log("Hello")
    }
}
```

4.npm的初体验

1.导入npm的包

`npm install 包的名字` 或`npm i 包的名字`，指定版本号的下载，通过`npm i 包@版本号`

5.express [express]

0.先导入相关的包

`npm i express`

1.创建基本的web服务器

```
const express = require('express')
const app = express()
app.listen(7878,()=>{
    console.log("Hello")
})
```

2.get方法/post方法

参数1：客户端请求的URL地址

参数2：请求对应的处理函数

- req：请求对象(包含了与请求相关的属性与方法)
- res：响应对象(包含了与响应有关的属性与方法)

```
const express = require('express')
const app = express()
app.get('/index.html',(req,res)=>{
    res.setHeader("Content-Type","text/html;charset=utf-8")
    res.end("<h1>首页</h1>")
})
app.listen(7878,()=>{
    console.log("Hello")
})
```

1.函数res.send():向客户端发送JSON对象或字符串

2.函数req.query()对象，可以访问到客户端通过查询字符串的形式，发送到服务器的参数，这里的参数也就是通过get的方法来获得的，默认是一个空对象，通过网址+`?name=''&age=`等

```
const express = require('express')
const fs = require('fs')
const app = express()
app.get('/index.html',(req,res)=>{
    console.log(req.query)
    res.send(req.query)
})
app.listen(7878,()=>{
    console.log("Hello")
})
```

3.函数req.params对象，可以访问到URL中，通过**: 匹配的动态参数** 这里的风格是Restfull分风格

```
const express = require('express')
const fs = require('fs')
const app = express()
app.get('/index/:x/:y',(req,res)=>{
    console.log(req.params)
    res.send(req.params)//{"x":"1","y":"2"}  json对象
})
app.listen(7878,()=>{
    console.log("Hello")
})
```

4.函数express.static()用于托管静态资源

通过访问`localhost/feng`下资源目录文件可以访问这些文件，这里有顺序的关系

```
app.use('/feng',express.static('./files'))
```

5.express路由

express中的路由分为3部分组成，分别是`请求的类型，请求的URL地址，处理函数`

`app.method(path,handler)`，这里的**method**就是请求的类型，**path**就是请求的URL地址，**handler**就是处理函数，只有当**method**和**url**完全相同的时候才会调用该函数，还有顺序的关系

```
const express = require('express')
const fs = require('fs')
const app = express()
app.get('/index/:x/:y',(req,res)=>{
    console.log(req.params)
    res.send(req.params)//{"x":"1","y":"2"}  json对象
})
app.listen(7878,()=>{
    console.log("Hello")
})
```

6.模块化路由　　在这个路由中可以定义中间件

使用`express.Router()`函数创建一个新的路由

```
const express = require("express")
const router = express.Router()
router.get('/',(req,res)=>{
    res.send("Hello World")
})
module.exports = router
```

路由的注册和使用路由，为模块化路由添加访问前缀

```
const express = require('express')
const app = express()
const router = require('./2')
app.use('/feng',router)//app.use是注册全局中间件的方法，这里还添加了访问前缀
app.listen(7878,()=>{
    console.log("初始化")
})
```

7.express的中间件

这个中间件只要访问了网址，就会触发中间件的发生

```
const express = require('express')
const app = express()
const mw = (req,res,next)=>{
    console.log("这是一个中间件函数")
    next()
}
app.use(mw)
app.get('/',(req,res)=>{
    res.send("Hello World")
})
app.listen(7878,()=>{
    console.log("初始化")
})
```

多个中间件之间，共享同一份req和res，可以通过这样的特性，统一为req和res添加自定义的属性和方法，供下游的中间路由使用

```
const express = require('express')
const app = express()
app.use((req,res,next)=>{
    req.startTime = Date.now()
    next()
})
app.get('/',(req,res)=>{
    res.send({msg:"Hello World",time:req.startTime})
})
app.listen(7878,()=>{
    console.log("初始化")
})
```

注册多个中间件只需要多使用`app.use`函数就行，这个是按顺序来执行的

多个局部生效的中间件，这里不需要`app.use`函数来注册中间件

```
const mw = (req,res,next)=>{
    req.startTime = Date.now()
    console.log("这个是一个中间件函数")
    next()
}
const mw1 = (req,res,next)=>{
    req.username = "feng"
    console.log("这个第二个中间件")
    next()
}
app.get('/',mw,mw1,(req,res)=>{
    res.send({msg:"Hello World",time:req.startTime,username:req.username})
})
app.get('/user',[mw,mw1],(req,res)=>{
    res.send({msg:"Hello World",time:req.startTime,username:req.username})
})
```

错误级别的中间件

```
const express = require('express')
const app = express()
app.get('/',(req,res)=>{
    throw new Error("出错了")
})
app.use((err,req,res,next)=>{
    console.log(err.message)
    res.send("错了") // 这个不会使客户端出错而停止运行
})
app.listen(7878,()=>{
    console.log("初始化")
})
```

express内置的中间件　这些都是通过express.use来调用这些中间件

- express.static 托管静态资源（无兼容）
- express.json　解析JSON格式的请求体的数据　（有兼容）

- express.urlencoded 解析URL-encodeed格式的请求体数据 (有兼容)

通过express.urlencoded解析post的请求

```
const express = require('express')
const app = express()
app.use(express.urlencoded({extended:false}))
app.post('/user',(req,res)=>{
    res.send(req.body)
})
app.listen(7878,()=>{})
```

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>测试</title>
</head>
<body>
<form action="http://localhost:7878/user" method="post">
    <input type="text" name="username" id="username" value="" placeholder="请输入名字">
    <input type="text" name="age" id="age" value="" placeholder="请输入年龄">
    <input type="submit" value="提交">
</form>
</body>
</html>
```

可以搭配html来实现post的请求

自定义一个中间件

自己手动模拟一个类似于express.urlencoded这样的中间件，来解析post提交到服务器的表单数据

- 定义中间件
- 监听req的data事件
- 监听req的end事件
- 使用querystring模块解析请求数据
- 将解析出来的数据对象挂载为req.body

```
const express = require('express')
const qs = require('node:querystring')
const app = express()
const mw = (req,res,next)=>{
    let str = ''
    req.on('data',(chunk)=>{
        str += chunk
    })
    req.on('end',()=>{
        req.body = qs.parse(str)
        next()
    })
}
app.post('/user',mw,(req,res)=>{
    res.send(req.body)
})
app.listen(7878,()=>{})
```

8.使用express写接口

1.get接口

1.js

```
const express = require('express')
const router = require('./2')
const app = express()
app.use('/feng',router)
router.get('/user',(req,res)=>{  //http://localhost:7878/feng/user
    res.status(200).send({
        status:0,
        msg:"GET请求成功",
        data:req.query
    })
})
app.listen(7878,()=>{})
```

2.js

```
const express = require('express')
const router = express.Router()
module.exports = router
```

2.post接口

```
const express = require('express')
const router = require('./2')
const app = express()
app.use(express.urlencoded({extended:false}))// 全局的配置
app.use('/feng',router)
router.use(express.urlencoded({extended:false})) // 单个路由配置的
router.post('/user',(req,res)=>{
    res.status(200).send({
        status:0,
        msg:"POST请求成功",
        data:req.body
    })
})
app.listen(7878,()=>{})
```

跨域问题的html

```
<!DOCTYPE html>
<html lang="en">
    <head>
        <meta charset="UTF-8" />
        <meta name="viewport" content="width=device-width, initial-scale=1.0" />
        <title>Document</title>
        <script src="./jQuery/jquery-3.6.0.js"></script>
    </head>
    <body>
        <button id="btnGET">GET</button>
        <button id="btnPOST">POST</button>
```

```
        <button id="btnDelete">DELETE</button>
        <button id="btnJSONP">JSONP</button>

        <script>
            $(document).ready(()=>{
                $('#btnGET').on('click', function () {
                    $.ajax({
                        type: 'GET',
                        url: 'http://localhost:7878/feng/get',
                        data: {name: 'zh', age: 20},
                        success: function (res) {
                            console.log(res)
                        },
                    })
                })
                // 2. 测试POST接口
                $('#btnPOST').on('click', function () {
                    $.ajax({
                        type: 'POST',
                        url: 'http://localhost:7878/feng/post',
                        data: {bookname: '水浒传', author: '施耐庵'},
                        success: function (res) {
                            console.log(res)
                        },
                    })
                })
            })
        </script>
    </body>
</html>
```

浏览内有同源访问的限制，配置cors之后可以解决这种跨域访问的问题

接口的跨域问题(cors)

- 先安装cors的第三方库
- 在js文件中引入cors
- 注册cors中间件

```
const cors = require('cors')
const app = express()
app.use(cors())
```

cors的响应头部

- Access-Controller-Allow-Origin字段 允许那些网址能访问这个网页

```
res.setHeader('Access-Controller-Allow-Origin',URL)
```

- Access-Controller-Allow-Headers字段 header 配置额外的请求头

```
res.setHeader('Access-Controller-Allow-Headers','Content-Type,额外的请求头')
```

- Access-Controller-Allow-Methods字段 仅支持get，post，head的请求

```
res.setHeader('Access-Controller-Allow-Methods','POST,GET,DELETE,HEAD')
```

简单请求和预检请求：前是一次，后是两次