`Express框架

Express是一个基于node.js平台的web应用开发框架，它可以快速高效的搭建网络应用平台

> 从这一刻开始，我们将进入项目式授课（BOP），我们本次授课以《学生综合管理系统》来完成

安装Express框架

在一个项目当中，如果我们要使用`express`的框架，则要先安装这个框架

```
$ npm install express --save
```

安装完成以后，我们接下来就要根据Express来构造项目

> 安装包的命令也可以简写成`npm i express -S`

Express创建项目

首先我们当前在的项目的目录下面创建一个`app.js`，同时在`package.json`的文件下面，将`main`后面的值也改成`app.js`，这里要保持一致

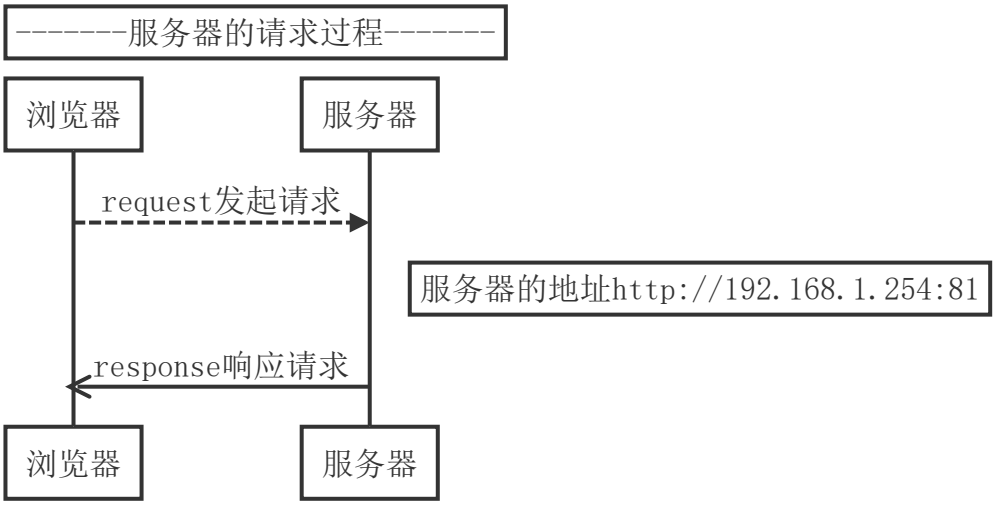
```
const express = require("express");
const http = require("http");

// 使用express生成了一个app的应用，这个应用是用于做web应用的，web应用应该是基于http的
const app = express();
const server = http.createServer(app);       //这一行代码的意思就是根据app的应用创建一个http的web服务器，提供给别人访问


//让当前的服务器去监听某一个IP地址面的某一个端口号
server.listen(81, "0.0.0.0", () => {
    console.log("服务器启动成功");
});
```

处理GET请求

> 在以前讲ajax的时候，我们都提到过，通过浏览器地址栏的请求是`get`请求，现在我们通过`http://192.168.1.254:81/`去请求的时候，它提示`cannot get /`这个错误，这是因为express服务器还没有做好处理请求的准备




```
//开始处理请求、这里的get代表的是处理get请求
app.get("/", (req, resp) => {
    //使用回调函数来处理这个请求
    //req代表request也就是浏览器到服务器的请求
    //resp代表response也就是服务器到浏览器的响应过程
    console.log("你的请求我已经接收到了");
    resp.send("我收到你的请求了");
});
```

上面的`app.get("/")`是使用express来处理一个`get`的请求，这个路径就是`/`，后面的`resp.send`代表的是服务器发送一个内容到浏览器

在这个知识点下面，有两个重要的对象，分别是

1. `req`全称叫`request`，指浏览器到服务器的请求
2. `resp`全称叫`response`，指服务器到浏览器的响应

请求处理	处理路径
`http://192.168.1.254:81/`	`app.get("/")`
`http://192.168.1.254:81/login`	`app.get("/login")`
`http://192.168.1.254:81/register`	`app.get("/register")`

在开发的时候，我们可以根据不同的路径来处理不同的页面场景

Express模板引擎的渲染

我们通过上面的方式可以实现不同的路径处理不同的请求，但是在处理不同的请求的时候，我们怎么样向浏览器渲染一个页面呢?

在这里，我们就要结合我们之前的思路，我们最开始的时候是学过模板引擎的概念，模板引擎是可以直接生成HTML字符串的

同时在express的框架里面，它也是支持模板引擎的，并且它是支持`art-template`

第一步：先安装express下面的模板引擎

```
$ npm install art-template express-art-template --save
```

第二步：配置模板引擎

```
const template = require("express-art-template");        //导入模板引擎
//配置视图的模板引擎的位置
app.set("views", path.join(__dirname, "./views"));
app.engine("html", template);
app.set("view engine", "html");
```

第三步：渲染模板

```
app.get("/login", (req, resp) => {
    //现在要渲染views目录下面的login.html这个页面，怎么办呢？
    resp.render("login");
});
```

👉 **注意**：

1. 在上面的渲染模板的时候，我们使用的是`resp.render()`这个方法，不是使用的`resp.send()`这个方法
2. 因为之前已经设置了模板引擎的路径文件夹，所以在渲染的时候，我们没有写`./views`
3. 因为我们之前已经设置了模板引擎的后缀名为`html`，所以在我们渲染的时候，也没有加入`.html`的后缀

模板引擎与数据的结合渲染

我们刚刚已经可以让一个视图渲染出来的，但是它是一个固定的数据，我们需要在这个模板上面渲染指定的数据

```
app.get("/stuList", (req, resp) => {
    //这里渲染的是views目录下面的stuList.html这个文件
    let stuArr = [
        { stuName: "程文来", stuSex: "男", stuAddr: "湖北武汉" },
        { stuName: "刘俊焱", stuSex: "男", stuAddr: "湖北武汉" },
        { stuName: "郑重阳", stuSex: "男", stuAddr: "湖北武汉" },
        { stuName: "蒋雨晴", stuSex: "女", stuAddr: "湖北汉川" },
        { stuName: "姚维雄", stuSex: "男", stuAddr: "湖北武汉" },
        { stuName: "吴港", stuSex: "男", stuAddr: "湖北武汉" },
        { stuName: "邓娜", stuSex: "女", stuAddr: "湖北赤壁" }
    ]
    //我在渲染stuList的时候，顺便还给了一个数据stuArr
    resp.render("stuList", {
        stuArr: stuArr
    });
});
```

接下来在views文件下面创建一个`stuList.html`的模板，然后编写代码如下

```
<!DOCTYPE html>
<html lang="zh">

<head>
    <meta charset="UTF-8">
    <meta http-equiv="X-UA-Compatible" content="IE=edge">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>学生列表</title>
```

```
    <style>
        .page-title {
            text-align: center;
        }

        .table-box {
            width: 900px;
            margin: auto;
        }

        .table1 {
            border: 1px solid black;
            border-collapse: collapse;
            width: 100%;
        }

        .table1 td,
        .table-box th {
            border: 1px solid black;
        }
    </style>
</head>

<body>
    <h2 class="page-title">学生列表</h2>
    <hr>
    <div class="table-box">
        <table class="table1">
            <tr>
                <th>学生姓名</th>
                <th>学生性别</th>
                <th>学生地址</th>
            </tr>
            {{each stuArr item index}}
            <tr>
                <td>{{item.stuName}}</td>
                <td>{{item.stuSex}}</td>
                <td>{{item.stuAddr}}</td>
            </tr>
            {{/each}}
        </table>
    </div>
</body>
</html>
```

这就是在渲染模板的时候向模板里面传递了数据， 然后再使用我们之前所学习的模板引擎的语法将数据展示出来

重点注意事项:`views`文件夹下面的所有视图都不能够直接在浏览器里面打开， 一定是通过`express`渲染才能显示的

express中静态文件的设置

在express里面，视图必须是通过`render`渲染出来的，同时在视图模板引擎渲染视图的时候，还可以使用第三方的样式CSS或JS文件

在项目当中，我们一般会新建一个`public`的目录，这个目录下面的文件是不需要经过`express`的`get`请求来处理的，应该是直接就可以请求 到的

```
//在当前这个位置，我们要公开public这个文件夹，让任何人都可以直接访问到这个文件夹下面的内容，而不需要经过express的处理
app.use("/public", express.static(path.join(__dirname, "./public")));
```

当设置完了上面的静态文件夹以后，我们就可以直接使用这个`/public`目录下面的文件，而不再经过express的处理了，后面在模板引擎里面，我们就可以正常的使用了

```
<link rel="stylesheet" href="/public/bootstrap/css/bootstrap.min.css"
type="text/css">
<!--[if lt ie 9]>
<script src="/public/bootstrap/html5shiv.min.js"></script>
<script src="/public/bootstrap/respond.min.js"></script>
<![endif]-->
<script src="/public/bootstrap/js/jquery.js"></script>
<script src="/public/bootstrap/js/bootstrap.min.js"></script>
```

express的分部模板

在express使用`art-template`做为模板引擎的时候，它支持分部模板技术，就是可以把一个模板拆分多个部分，然后再来结合使用，这样可以极大的优化我们的代码，减少代码的冗余量

我们现在把某一个模板引擎页面的页头与页单独的拆分出来

layout_header.html

```
<!DOCTYPE html>
<html lang="zh">

<head>
    <meta charset="UTF-8">
    <meta http-equiv="X-UA-Compatible" content="IE=edge">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>教师列表</title>
    <link rel="stylesheet" href="/public/bootstrap/css/bootstrap.min.css">
    <!--[if lt ie 9]>
        <script src="/public/bootstrap/html5shiv.min.js"></script>
        <script src="/public/bootstrap/respond.min.js"></script>
    <![endif]-->
    <script src="/public/bootstrap/js/jquery.js"></script>
    <script src="/public/bootstrap/js/bootstrap.min.js"></script>
</head>
<body>
```

layout_footer.html

```
</body>
</html>
```

当我们拆分出这两个部分以后，我们后期就可以直接调用这两个分部模板就可以了

```
{{include "./layout_header.html"}}
```

```
<div class="container">
    <div class="page-header">
        <h2 class="text-center text-primary">教师列表</h2>
    </div>
    <div role="form" class="form-inline">
        <div class="form-group">
            <label for="" class="control-label">教师编号</label>
            <input type="text" placeholder="输入编号查询" class="form-control">
        </div>
        <div class="form-group">
            <label for="" class="control-label">教师姓名</label>
            <input type="text" placeholder="输入姓名查询" class="form-control">
        </div>
        <div class="form-group">
            <label for="" class="control-label">教师性别</label>
            <select class="form-control">
                <option value=""></option>
                <option value="">男</option>
                <option value="">女</option>
            </select>
        </div>
        <div class="form-group">
            <button type="button" class="btn btn-info">
                <span class="glyphicon glyphicon-search"></span>查询
            </button>
        </div>
    </div>
    <div class="btn-group">
        <button type="button" class="btn btn-primary">
            <span class="glyphicon glyphicon-plus"></span>新增
        </button>
        <button type="button" class="btn btn-warning">
            <span class="glyphicon glyphicon-pencil"></span>修改
        </button>
        <button type="button" class="btn btn-danger">
            <span class="glyphicon glyphicon-trash"></span>删除
        </button>
    </div>
    <div class="table-responsive">
        <table class="table table-hover table-striped table-bordered">
            <tr>
                <th>教师编号</th>
                <th>教师姓名</th>
                <th>教师性别</th>
                <th>教师年龄</th>
                <th>民族</th>
                <th>籍贯</th>
                <th>密码</th>
                <th>手机号</th>
            </tr>
            {{each teacherArr item index}}
            <tr>
                <td>{{item.tid}}</td>
                <td>{{item.tname}}</td>
                <td>{{item.tsex}}</td>
                <td>{{item.tage}}</td>
                <td>{{item.tnation}}</td>
                <td>{{item.tacion}}</td>
```

```
                <td>{{item.tpwd}}</td>
                <td>{{item.tphone}}</td>
            </tr>
            {{/each}}
        </table>
    </div>
</div>

{{include "./layout_footer.html"}}
```

当我们完成了上面的代码以后，我们就可以把模板引入进来使用了，但是现在有个问题就是在`layout_header.html`里面，它中间有一个网页的标题`<title>`是需要改变的，所以我们要把这一块设置成一个变量，不然每个的页面标题都是一样的

第一种解决方案：通过`app.js`向模板传递标题数据

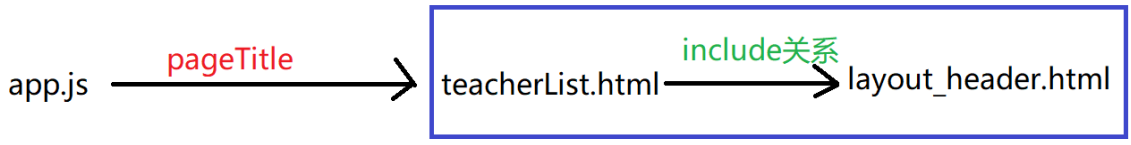
layout_header.html

```
<title>{{pageTitle}}</title>
```

然后使用下面的方法向页面传递数据

app.js

```
resp.render("teacherList", {
    teacherArr: results,
    pageTitle:"【教师列表】"
});
// 或
resp.render("instructorList", {
    instructorArr: results,
    pageTitle:"班主任列表"
});
```



数据本质上面并没有传递给`layout_header.html`，只是传递给了`teacherList.html`模板，但是`layout_header.html`仍然可以拿到值，这就说明，子级模板是共享父级模板的数据的

上面的方案，是我们的第一种方案，把数据通过`app.js`来传递过程

第二种方案：

之前学习模板引擎的时候，我们就已经知道，模板引擎里面的数据主要来源于2个地方

1. 渲染的时候给你的
2. 模板自身本来就有的

所以我们能不能直接在模板里面定义标题呢?

teacherList.html

```
<% var obj = {pageTitle:"标哥哥的网页"} %>
{{include "./layout_header.html" obj}}
```

上面的代码就是在`teacherList.html`的模板里面定义了一个变量`obj`，然后在`include layout_header.html`这个的时候把`obj`传递进去

instructorList.html

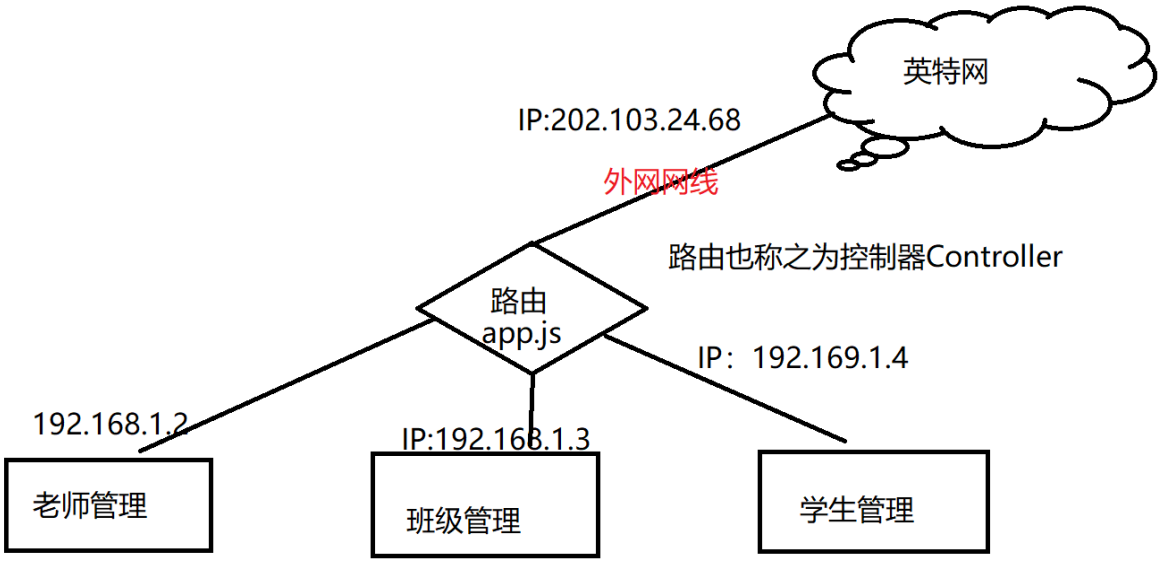
```
{{set obj = {pageTitle:"班主任列表12312312"} }}
{{include "./layout_header.html" obj}}
```

上面使用的就是模板引擎在简洁版的语法里面定义变量的方法，两种定义情况最终实现的效果是一模一样的

Express路由

路由可以看成是一个小型的app.js，路由主要的作用就是用于控制请求路径的，所以路径通常情况下也叫控制器

路由主要的作用就是用于分担app.js当中的请求，这样可以使怎么项目的层次清晰明确，我们现在通过一个通俗的图来理解路由



app.js的下面可以有多个路由文件

第一步：在当前项目下面创建一个一个routes的目录

第二步：在这个routes的目录下面根据不同的需求创建不同的路由文件，这里的路由文件最好以Router.js结尾,如`teacherInfoRouter.js`

第三步：在路由文件里面编写代码

```
/**
 * @file teacherInfoRouter.js  老师信息的路由
 * @author 杨标
 * @description 数据表teacherinfo的相关控制请求
 */
const express = require("express");
// const app = express();
//这就创建了一个路由
const router = express.Router();

router.get("/teacherList", async (req, resp) => {
    resp.render("teacherInfo/teacherList");
```

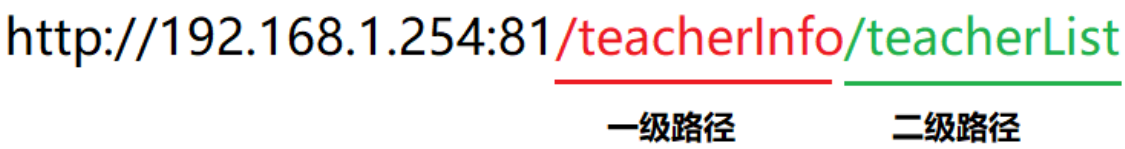
```
});

//导出这个路由
module.exports = router;
```

第五步：在app.js里面链接这个路由文件

```
const teacherInfoRouter = require("./routes/teacherInfoRouter");
app.use("/teacherInfo", teacherInfoRouter);
```

经过上面的操作以后，我们就可以通过下面的地址来访问我们的请求



一级路径是进入路由，二级路径是进入到路由以后的请求

经过像这样的拆分以后，我们的项目层次就会非常明确

Express前端后向端传值

在Express的项目里面，如果前端要向后端传值是有很多种方式的，前端浏览器发起请求的时候是可以携带数据到后端服务器的，请的传值方式主要有两种情况，是根据请求方式方式

请求方式又分为两种

1. `get`请求

   在之前讲请求协议方式的时候，我们定了一个通俗的说法，通过浏览器地址栏的请求都是GET请求【get请求俗称浏览器地址栏请求，它会把所有的请求信息暴露在浏览器的地址栏，这样很不安全】

   ```
   http://192.168.1.254:81/teacherInfo/teacherList?tsex=男
   ```

   在上面的URL地址里面，我们在某一个请求地址的后面添加了`?tsex=男`这就相当于向这一个地址传递了一个参数`tsex`并且值为“男”

   关键的问题就在于后端如何接收浏览器通过get方式传递过来的值？？？？

   express本身就支持get传值的接收，所有通过`get`方式请求过来的值都在`req.query`这个对象里面【`req.query`只能够接收get方式传递过来的值】

2. `post`请求

   post请求通俗一点说叫报文请求，也称之为机密请求，它不会把请求信息暴露在外边，这样很安全

   ```
   <form id="addClassInfoForm" role="form" class="form-horizontal"
   method="POST" action="/classInfo/doAddClassInfo">
   </form>
   ```

   前端如果通过`POST`方式向后台传值以后，后台的`express`如果要接收这个`post`过来的值，是不行的，它需要依赖第三方插件，这个插件叫`body-parser`

   第一步：安装插件

```
$ npm install body-parser --save
```

第二步：配置插件

在`app.js`的代码当中，添加如下代码

```
//配置body-parser的插件
const bodyParser = require("body-parser");
//接收以x-www-form-urlencoded这种方式过来的值
app.use(bodyParser.urlencoded({ extended: false }));
// 接收以application/json这种方式过来的值
app.use(bodyParser.json({ limit: "20m" }));
```

第三步：在post请求的路由里面接收值

```
//添加一个请求，接收form表单新增的数据
router.post("/doAddClassInfo", (req, resp) => {
    // console.log(req.query);  这里是post请求，所以不能使用req.query
    console.log(req.body);
    resp.send("我收到你的请求了");
})
```

我们的`post`请求去接收值的时候是以`req.body`来进行接收的

Express条件查询与模糊查询

express的条件查询就是将前台传传递给后台的值接收到了以后然后再去拼接SQL语句，模糊查询的原理也是一样的，现在先看下面的图

教师编号	教师姓名	教师性别	手机号码	Email	地址
1	杨标	男	18627109999	123@qq.com	湖北省武汉市
2	柴柳	男	18712345671	aaa@163.com	湖北省荆州闹市
3	邱芬芬	女	19827384738	ff@163.com	湖北省鄂州市
4	老冯	男	13456789876	feng@qq.com	湖北省武汉市
5	杨清云	男	15543456789	yqy@qq.com	陕西省西安市
6	阮强胜	男	19827831234	rqs@163.com	湖北省武汉市

在上图里面，我们是希望通过姓名与性别是查询相关信息，这个时候在代码里面应该这样完成

前台的模板引擎代码

```
<form method="GET" class="form-inline" role="form"
action="/teacherInfo/teacherList">
    <div class="form-group">
        <label for="" class="control-label">姓名</label>
        <input type="text" placeholder="输入姓名查询" class="form-control"
name="tname">
    </div>
    <div class="form-group">
        <label for="" class="control-label">性别</label>
        <select name="tsex" class="form-control">
```

```
            <option value="">-请选择-</option>
            <option value="男">男</option>
            <option value="女">女</option>
        </select>
    </div>
    <div class="form-group">
        <button type="submit" class="btn btn-info">
            <span class="glyphicon glyphicon-search"></span>查询
        </button>
    </div>
</form>
```

我们通过表单向后台提交了数据，注意表单元素里面的`name`属性以后`value`，同时在`form`里面，我们设置它的`method="GET"`以及`action="/teacherInfo/teacherList"`

后台处理的代码

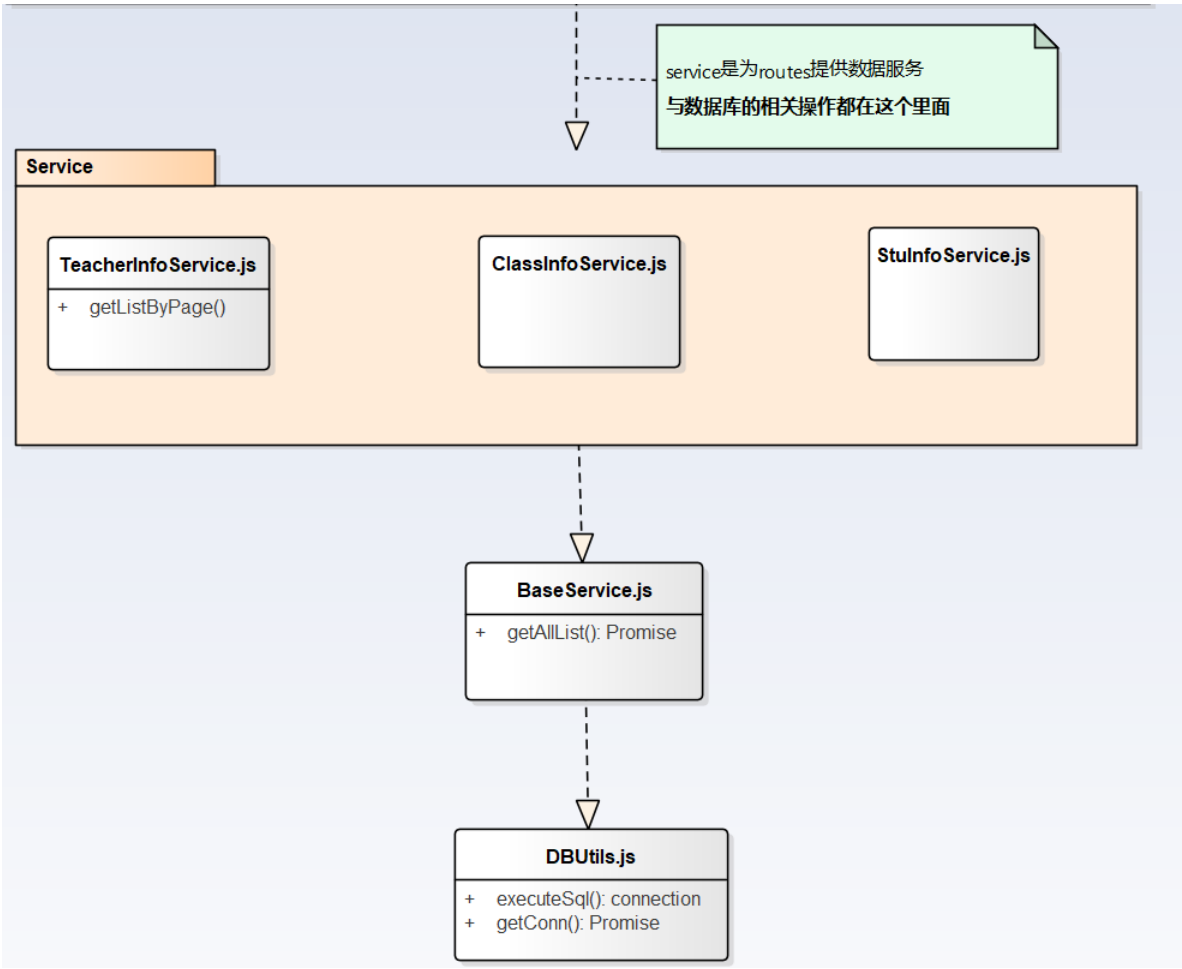
```
router.get("/teacherList", async (req, resp) => {
    try {
        //req.query是专门用于接收get传递过来的值的
        let { tsex, tname } = req.query;
        //tsex可能有值，可能没有值，没值就是undefined
        let strSql = " select * from teacherinfo where 1=1 ";
        let ps = [];   //SQL语句的参数
        if (tsex) {
            strSql += " and tsex = ? ";
            ps.push(tsex);
        }
        if (tname) {
            strSql += " and tname like ? ";
            ps.push("%" + tname + "%");
        }
        let db = new DBUtils();
        let results = await db.executeSql(strSql, ps);
        resp.render("teacherInfo/teacherList", {
            teacherArr: results
        });
    } catch (error) {
        console.log(error);
        resp.send("服务器错误");
    }
});
```

在上面的代码里面，注意一点就在于我们在SQL语句的后面添加了一个`where 1=1`这么写的好处就是为了后期动态的添加查询条件

同时在进行模板查询的时候，我们使用的是`like`，并且在参数的前面添加了`%`这个通配符

Service数据服务层的使用

在之前我们编写代码处理请求的时候，我们把数据库的相关操作都放在了路由里面，这个时候我们发现的冗余量太大，同时层次结构不太清晰



第一步：先新建一个BaseService.js这是一个公共服务对象，我们后期可以将一些公共的方法都写在这里

```
/**
 * @name BaseService Server的公共对象
 * @author 杨标
 * @version 1.1
 */
const DBUtils = require("../utils/DBUtils");
class BaseService extends DBUtils {
    constructor() {
        super();
        // tableMap得到结果就是一个代理对象，这个对象只有get的代理，所以只能取值不能赋值
        this.tableMap = (() => {
            let _obj = {
                teacherInfo: "teacherinfo",
                classInfo: "classinfo",
                stuInfo: "stuinfo"
            }
            return new Proxy(_obj, {
                get(target, propertyName, receiver) {
                    return target[propertyName];
                },
                set(target, propertyName, value, receiver) {
                    //告诉外边赋值已经完成了
                    return true;
                }
            })
        })();
    }
```

```
    getAllList() {
        let strSql = ` select * from ${this.tableName} where 1=1 `;
        return this.executeSql(strSql);
    }
}

module.exports = BaseService;
```

第二步：针对不同的数据表创建不同的Service对象

```
/**
 * @name TeacherInfoService 数据表teacherinfo的相关操作信息
 * @author 杨标
 * @version 1.1
 */

const BaseService = require("./BaseService");
class TeacherInfoService extends BaseService {
    constructor() {
        super();
        this.tableName = this.tableMap.teacherInfo;
    }

    /**
     * @name getListByPage 根据条件查询老师的列表
     * @param {*} param0
     * @returns {Promise} 返回数据库执行以后的Promise
     */
    getListByPage({ tname, tsex }) {
        let strSql = ` select * from ${this.tableName} where 1=1 `;
        let ps = [];
        if (tname) {
            strSql += ` and tname like ? `;
            ps.push("%" + tname + "%");
        }
        if (tsex) {
            strSql += ` and tsex = ? `;
            ps.push(tsex);
        }
        return this.executeSql(strSql, ps);
    }

}

module.exports = TeacherInfoService;
```

有了Service以后，我们每次与数据库操作相关的东西我们就全部都放在Service里面去完成

配置项目的启动信息

在之前我们启动项目的时候，我们都是直接在控制台里面输入要执行的文件有，如`node app.js`，但是正常情况下，我们还可以把启动信息写入`package.json`这个文件当中

```
"scripts": {
    "start": "node app.js"
},
```

当配置好上面的信息以后，我们就可以直接在控制台执行以下命令

```
$ npm run start
```

同时我们在配置这个`start`它是有提示的，说明这个start是内置的一个命令，对于这种内置的命令，我们可以进一步简化

如果是内置的命令，我们可以把run省略掉，这个时候上面的`npm run start`就成了`npm start`

所以如果我们配置的不是内置的命令，则不能省略`run`，如下

```
"scripts": {
    "bgg":"node app.js"
},
```

对于上面的命令，我们如果要执行则是`npm run bgg`

加载路由方法的封装

在之前我们在`routes`目录下面创建好了路由文件以后，我们就需要手动的在`app.js`里面去导入这个路由，然后去构造路径，这样做非常麻烦，我们可以直接写一个方法去完成

第一步：我们在`utils`的目录下面创建一个`loadRoute.js`的文件，代码如下

```
/**
 * @file loadRoute.js 专门用于加载路由文件
 */
const path = require("path");
const fs = require("fs");

/**
 * @name loadRoute 用于加载routes目录下面的路由文件
 * @param {Express} app
 */
const loadRoute = app => {
    let list = fs.readdirSync(path.join(__dirname, "../routes"));
    for (let item of list) {
        app.use(`/${item.replace("Router.js", "")}`,
require(path.join(__dirname, "../routes", item)));
    }
};
module.exports = loadRoute;
```

第二步：在app.js里面去调用这个方法

```
const loadRoute = require("./utils/loadRoute.js");
loadRoute(app);
```

经过上面的封装以后， 我们就可以在创建路由的时候自动的构造路径

工厂模式的使用

我们现在已经有很多个`Services`的文件了，这样做非常方便，但是也存在一定的隐患，如下所示

```
const TeacherInfoService = require("../services/TeacherInfoService");
```

我们如果直接使用上面的方式去`require`某一个文件，但是这个文件后期的名子发生变化以后，怎么办呢？这样所有`require`这个文件的地方都要发生改变

首先，我们在当前的项目下面创建了一个`factory`的文件夹，在里面创建了一个`ServiceFactory`的文件，代码如下

```
/**
 * @name ServiceFactory 服务器的工厂
 * @author 杨标
 * @version 1.0
 * @description 专门用于生产服务Service的工厂
 */
class ServiceFactory {
    static createTeacherInfoServcice() {
        let TeacherInfoService = require("../services/TeacherInfoService");
        return new TeacherInfoService();
    }
    static createClassInfoService() {
        let ClassInfoService = require("../services/ClassInfoService");
        return new ClassInfoService();
    }
}

module.exports = ServiceFactory;
```

通过上面的代码可以看到，这个对象全部都是静态方法，里面的所有方法都是用于返回某一个Service的对象【这就相当于一个工厂，这个工厂专门用于生产Service的对象，并且这个对象已经new好了】

抽象工厂模式

在上面的工厂模式里面，我们发现一个问题，我们每新建一个Service文件以后都需要手动在工厂里面去添加一个方法，这个做很麻烦，所以我们需要将这个工厂做进一步的改造

```
/**
 * @name ServiceFactory 服务器的工厂
 * @author 杨标
 * @version 1.0
 * @description 专门用于生产服务Service的工厂
 */
const path = require("path");
const fs = require("fs");

/**
 * @typedef ServiceFactoryType
```

```
 * @property {import("../services/ClassInfoService.js")} classInfoService
 * @property {import("../services/TeacherInfoService.js")} teacherInfoService
 * @property {import("../services/StuInfoService.js")} stuInfoService
 */


/**
 * @type {ServiceFactoryType} 工厂对象
 */
const ServiceFactory = (() => {
    let _obj = {}
    let list = fs.readdirSync(path.join(__dirname, "../services"));
    for (let item of list) {
        //第一步：导入这个文件
        let currentService = require(path.join(__dirname, "../services", item));
        //第二步：将去掉.js后缀，然后再将首字母变成小写，将这个结合做为属性名
        let temp = item.replace(".js", "")
        let propertyName = temp[0].toLocaleLowerCase() + temp.substr(1);
        _obj[propertyName] = new currentService();
    }
    return _obj;
})();


module.exports = ServiceFactory;
```

使用

```
/**
 * @file classInfoRouter.js
 * @author 杨标
 * @version 1.1
 */

const express = require("express");
const ServiceFactory = require("../factory/ServiceFactory");
const router = express.Router();
const { formatDate } = require("../utils/DateUtils");

router.get("/classList", async (req, resp) => {
    try {
        //在这个请求里面，我不仅仅是要拿班级的数据，还要拿老师的数据
        let teacherInfoService = ServiceFactory.teacherInfoService;
        let teacherArr = await teacherInfoService.getAllList();

        let classInfoService = ServiceFactory.classInfoService;
        let results = await classInfoService.getListByPage(req.query);
        resp.render("classInfo/classList", {
            classArr: results,
            formatDate,
            teacherArr
        });
    } catch (error) {
        resp.send("服务器错误");
        console.log(error);
    }
});
```

```
module.exports = router;
```

配置nodejs程序的热启动

当我们开发的时候每次更改了nodejs的代码以后都要手动的去重启，这个时候我们能不能让程序自动的重启呢?

在`nodejs`的项目下面，有一个工具叫`nodemon`，它可以实现这个功能。

Nodemon 是一款非常实用的工具，用来监控你 node.js 源代码的任何变化和自动重启你的服务器。Nodemon 是一款完美的开发工具，可以使用 npm 安装。

第一步：安装

```
$ npm install nodemon --save-dev
```

第二步：配置

安装完了nodemon以后，需要在`package.json`里面去配置启动信息

```
"main": "app.js",
"scripts": {
    "start": "node app.js",
     "dev": "nodemon ./"
},
```

> 在注意在配置nodemon的命令的时候，一定要注意main是必须要配置正确的

第三步：启动

```
$ npm run dev
```

Ajax向服务器后台发送静默情求

Ajax称之类异步的无刷新请求，它可以在后台悄悄的去发送（请求）数据，而无需刷新整个页面，在项目当中会经常使用到这种技术，如下图所示

在新增班级的时候，我们需要对班级编号做处理，不能重复（因为数据表当中它是主键），这个时候我们就存在一个需求，当鼠标离开这个输入的时候，应该把值发送到后台去，但是页面又不能够刷新，所以在这里要使用的Ajax来进行数据请求与发送

```
<input id="cid" type="text" class="form-control" placeholder="请输入班级编号">
```

接下来我们对这个输入框使用jQuery去做处理，在这个下面的当中，我们使用了`blur`这个事件

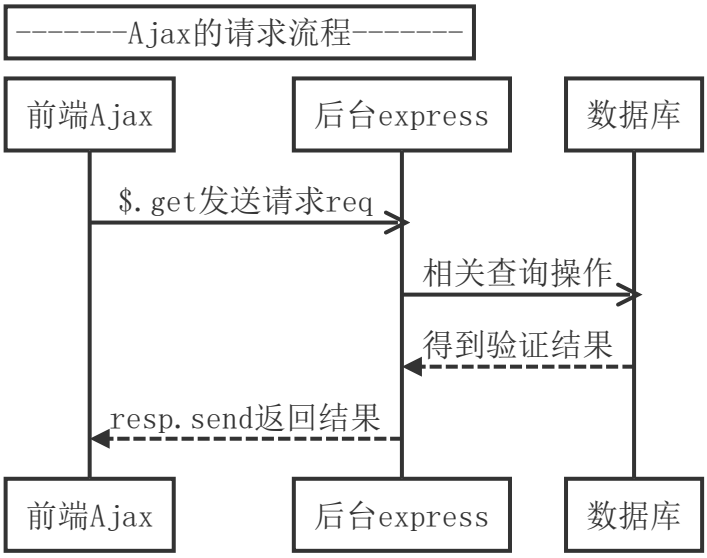
```
$(function () {
    $("#cid").blur(function () {
        //把这个值发送到后台，到数据库检测一下，看是否已经存在了
        var cid = $(this).val().trim();
        if (cid != "") {
            //说明有值
            //第一步：怎么样把值发送到后台去，使用ajax向后台发送请求
            //第二步：发送到哪个地址
            $.get("/classInfo/checkCidExists?cid=" + cid, function (str) {
                //这里的回调接收到的就是后台响应给你的东西resp
                if (str == "1") {
                    //说明这个cid已经存在
                    alert("当前班级编号已存在");
                    $("#cid").val("");
                }
            });
        }
    });
})
```

接下来，我们的代码转入到了`express`的后台，后台就要接收这个`cid`，并且到数据库当中去查询

classInfoRouter.js的代码

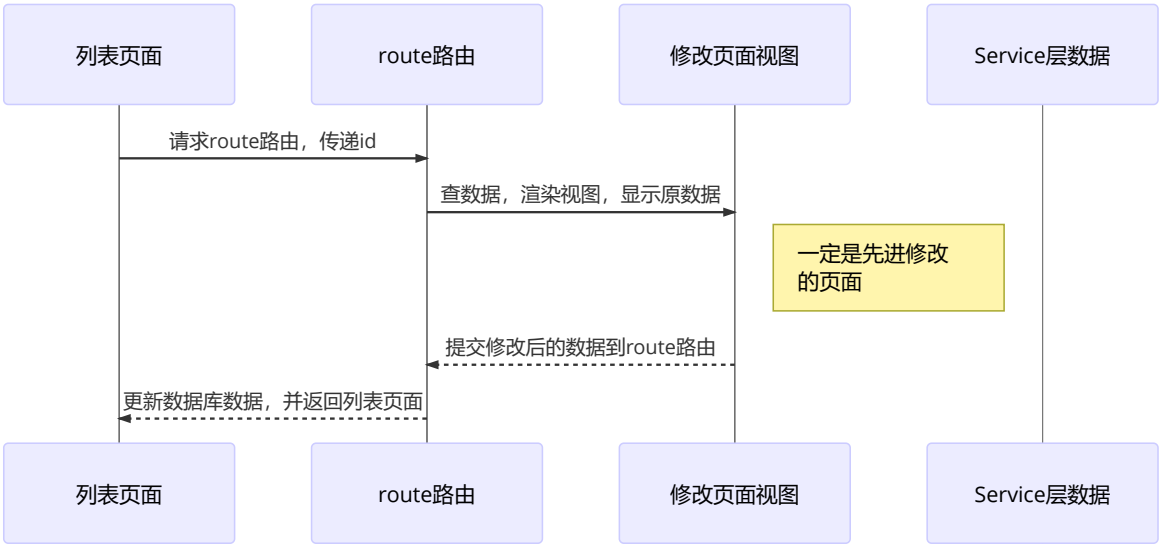
```
//检测cid是否已经存在
router.get("/checkCidExists", async (req, resp) => {
    try {
        let { cid } = req.query;
        //调用了Service里面的相关方法，来进行相应的处理
        let flag = await ServiceFactory.classInfoService.checkCidExists(cid);
        //flag为true就证明存在，flag为false则证明不存在
        resp.send(flag ? "1" : "0");
    } catch (error) {
        console.log(error);
        resp.send("服务器错误");
    }
});
```

> 上面的`resp.send()`就是响应，也就是把结果重新响应给前端，前端的Ajax就可以接收到这个结果



这个请求过程与我们之前所学习的请求过程是一样的，只是它页面没有刷新，因为Ajax就是在后台默默的给你发送请求（无需刷新页面）

修改数据的流程



分页查询数据展示（重点）

如图所示当列表的数据过多的时候，我们再去展示列表就不是很方便，这个时候为了更好的体验，我们需要做到分页的数据展示

在mysql的数据库里面本身是支持分页查询的，我们可以使用`limit`去完成

```
select 表名 from 表外 where 条件 limit 跳过多少条,取多少条;
```

在分页查的SQL语句里面，有两个值是一定要弄清楚的，这个分别是`pageIndex`代表页码，`pageSize`代表每页显示多少条，有了这两个值以后，我们就可以推导出一个公式

页码 pageIndex	跳过多少条	取多少条 pageSize
1	0	10
2	10	10
3	20	10

根据上面的表格，我们就得到了一个公式：`(pageIndex-1)*pageSize`

第一步：先设置数据库mysql可以执行多条语句

```
let conn = mysql.createConnection({
    host: "127.0.0.1",
    port: 3306,
    user: "h2003",
    password: "123456",
    database: "scms",
    multipleStatements:true          //开启多条SQL语句执行，因为分页查询里面要使用到
});
```

第二步：准备要返回的分页数据对象PageList

我们在项目的目录下面创建一个`model`的文件夹，然后创建一个PageList的对象，如下

```
/**
 * @name PageList 页面分页查询的所需要的数据
 * @author 杨标
 * @version 1.0
 * @description 页面分页查询的所需要的数据
 */
class PageList {
    /**
     *
     * @param {number} totalCount 总共有多少条
     * @param {number} pageSize  每页显示多少条
     * @param {Array} listData 要渲染的列表数据
     */
    constructor(pageIndex,totalCount, pageSize, listData = []) {
        this.pageIndex = pageIndex;
        this.totalCount = totalCount;
        this.pageCount = Math.ceil(totalCount / pageSize);      //分页是向上取整
        this.listData = listData
    }
}
module.exports = PageList;
```

> 后期这个文件要更改

第三步：在Service里面编写相应代码

```
async getListByPage({ cid, tid, pageIndex = 1 }) {
    //第一条SQL语句是查询的SQL语句，第二条SQL语句是计数的SQL语句
    let strSql = ` select * from ${this.tableName} a inner join
${this.tableMap.teacherInfo} b on a.tid = b.tid where a.isDel=false `;
```

```
    let countSql = ` select count(*) totalCount from ${this.tableName} a where
a.isDel = false `;

    let strWhere = "";
    let ps = [];
    if (cid) {
        strWhere += ` and a.cid like ? `;
        ps.push("%" + cid + "%");
    }
    if (tid) {
        strWhere += ` and a.tid= ? `;
        ps.push(tid);
    }
    //接第一条SQL语句的查询条件
    strSql += strWhere + ` limit ${(pageIndex - 1) * this.pageSize} ,
${this.pageSize} `;
    //接第二条SQL语句的查询条件
    countSql += strWhere;
    let results = await this.executeSql([strSql, countSql].join(";"), [...ps,
...ps]);
    //我现在要根据查询结果里面的totalCount计算pageCount
    let pageList = new PageList(pageIndex,results[1][0].totalCount,
this.pageSize, results[0]);
    return pageList;
}
```

> 这里的代码的重点就在于SQL语句的拼接，同时参数的设置，以及最后pageList的生成

第四步：在路由里面去设置相应的操作

```
router.get("/classList", async (req, resp) => {
    console.log(req.originalUrl);
    try {
        //在这个请求里面，我不仅仅是要拿班级的数据，还要拿老师的数据
        let teacherInfoService = ServiceFactory.teacherInfoService;
        let teacherArr = await teacherInfoService.getAllList();

        let classInfoService = ServiceFactory.classInfoService;
        let pageList = await classInfoService.getListByPage(req.query);  //返回的
是PageList对象
        //处理URL数据
        let originalUrl = req.originalUrl.replace(/&pageIndex=\d*/, "");
        if (!originalUrl.includes("?")) {
            originalUrl += "?1=1";
        }
        resp.render("classInfo/classList", {
            pageList,
            formatDate,
            teacherArr,
            originalUrl
        });
    } catch (error) {
        resp.send("服务器错误");
        console.log(error);
    }
});
```

> 上面代码当中最重要的是`originalUrl`这个处理过程，因为我们之前的查询是GET请求，GET请求所有的信息都暴露在地址栏，我们只要拿到地址栏的信息就可以得到之前的查询条件，然后在之前的查询条件上面去分页就可以了

第五步：前台生成分页的页码

```
<div class="clearfix">
    <div class="pull-left">
        当前第{{pageList.pageIndex}}页，共{{pageList.pageCount}}页，共
{{pageList.totalCount}}条数据
    </div>
    <ul class="pagination pull-right" style="margin-top: 0;">
        <li><a href="{{originalUrl}}&pageIndex=1">首页</a></li>
        <%for(var i=1;i<=pageList.pageCount;i++){%>
        <li class={{i==pageList.pageIndex?"active":""}}><a href="
{{originalUrl}}&pageIndex={{i}}">{{i}}</a></li>
        <%}%>
         <li><a href="{{originalUrl}}&pageIndex={{pageList.pageCount}}">尾页</a>
</li>
     </ul>
</div>
```

第六步：把分页的HTML代码拆分成分部模板

将这个HTML代码放在一个新建的模板彩虹城 面，然后使用导入操作引入就可以了

```
{{include "../layout_pagination.html"}}
```

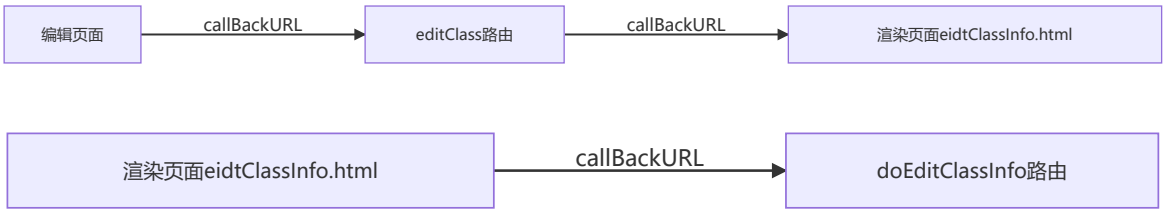
关于callBackURL的应用

请看下在的场景

当我在班级列表页面查询邱芬芬老师班级的信息的时候，我查询完毕跳转到了第2页，然后我们对班级编号为`J2023`的班级进行编辑，跳转到编辑页面

当我们修改完数据以后，点击保存，数据会提交到后台，并且重新返回列表页面

这个时候问题就来了，我们发现当我们修改数据完成了以后页面并没有回到之前的查询页面的第2页，而是直接回到了所有数据的第1页，这就存在问题



当它完成上面的步骤以后，我们先看代码

```
 <a href="#" data-href="/classInfo/editClassInfo?cid={{item.cid}}"
       class="btn btn-edit btn-warning btn-xs">编辑</a>
<script>
    $(function () {
        $(".btn-edit[data-href]").each(function (index, ele) {
            var href = $(this).attr("data-href");
            href +="&callBackURL="+encodeURIComponent(location.href);
            $(this).attr("href", href);
        })
    })
</script>
```

> 代码说明：上面的`encodeURIComponent`是将一个URL地址进行编码，因为URL地址在进行传递的时候是需要进行编码【要求，也是规范】

有了上面的代码以后，当我们再去点击编辑的时候，就会把当前的地址通过`callBackURL`传递到后台去，这个时候后台就可以接收到这个值

```
router.get("/editClassInfo", async (req, resp) => {
    try {
        let { cid, callBackURL } = req.query;
        console.log(callBackURL);
        let classInfoList = await ServiceFactory.classInfoService.findByCid(cid);
        if (classInfoList.length > 0) {
            //说明有数据
            let teacherArr = await ServiceFactory.teacherInfoService.getAllList();
            resp.render("classInfo/editClassInfo", {
                teacherArr,
                classInfo: classInfoList[0],
                formatDate,
                callBackURL
            });
        }
        else {
```

当接收到这个值以后，又重新渲染到了页面

```
-->
<form id="addClassInfoForm" role="form" class="form-horizontal"
    method="POST" action="/classInfo/doEditClassInfo"
    enctype="application/x-www-form-urlencoded">
    <!-- type="hidden"相当于一个看不见的type="text" -->
    <input type="hidden" name="callBackURL" value="{{callBackURL}}">
    <div class="form-group">
        <label for="" class="control-label col-sm-2">班级编号</label>
        <div class="col-sm-7">
            <input id="cid" type="text" class="form-control" placeholder="请输
```

这个时候页面上面添加一个隐藏域，当提交表单的时候，又会把这个`callBackURL`再次提交到后台，后台又可以接收到值

```
router.post("/doEditClassInfo", async (req, resp) => {
    try {
        let { callBackURL } = req.body;
        let flag = await ServiceFactory.classInfoService.editClassInfo(req.body);
        if (flag) {
            resp.redirect(callBackURL || "/classInfo/classList");
        }
        else {
            resp.send(`<script>alert("修改失败");history.back();</script>`);
        }
    } catch (error) {
        console.log(error);
        resp.send("服务器错误");
    }
});
```

当后台编辑完成以后，就可以通过这个`callBackURL`再次跳回到你编辑之前的列表页面

express文件上传

在express框架里面，如果要实现文件上传则需要使用一个第三方的模块，这个模块叫`multer`

第一步：安装模块

```
$ npm install multer --save
```

第二步：配置模块

```
//配置文件上传
const multer = require("multer");
const upload = multer({
    //配置上传文件保存在什么位置
    dest: path.join(__dirname,"./uploadImgs")
});
```

第三步：使用模块

```
//在这个请求里面，有可能会接收1个图片
//并且这个图片的name属性为sphoto
app.post("/doAddStuInfo",upload.single("sphoto"), (req, resp) => {
    console.log(req.body);      //代表post过来的值
    console.log(req.file);      //代表post过来的文件
    resp.send("我收到你的信息了");
});
```

> 我们需要将这个模块加载到请求里面去，`single`代表的是只接收1个文件，里面的参数代表文件在页面上面的`name`属性，接收到的文件在`req.file`里面

第四步：配置页面的enctype

在页面里面，我们以前的`form`表单使用的是`urlencoded`这种方式，现在要换成下面的方式

```
<form action="/doAddStuInfo" method="POST" enctype="multipart/form-data">
</form>
```

> 以前的`enctype="application/x-www-form-urlencoded"`

第五步：对上传的文件进行重命名

在后台的express里面，我们可以看到文件对象是一个如下的对象

```
{
  fieldname: 'sphoto',
  originalname: 's_photo-1533798916938.jpg',
  encoding: '7bit',
  mimetype: 'image/jpeg',
  destination: 'D:\\杨标的工作文件\\班级教学笔记
\\H2003\\0226\\code\\022601\\uploadImgs',
  filename: '2f3602e4f7943ce295cdcb542663382a',
  path: 'D:\\杨标的工作文件\\班级教学笔记
\\H2003\\0226\\code\\022601\\uploadImgs\\2f3602e4f7943ce295cdcb542663382a',
  size: 4894
}
```

这个默认上传的文件名是没有后缀名的，我们要改后缀名

```
app.post("/doAddStuInfo", upload.single("sphoto"), (req, resp) => {
    console.log(req.body);      //代表post过来的值
    console.log(req.file);      //代表post过来的文件
    //接收到文件以后，一定要改名
    if(req.file){
        // 判断一下是否有文件
        fs.renameSync(req.file.path, req.file.path + req.file.originalname);
    }
    resp.send("我收到你的信息了");
});
```

第六步：将上传保存图片的文件夹公开，方便直接访问

```
//把图片上传的目录设置为静态目录
app.use("/uploadImgs",express.static(path.join(__dirname,"./uploadImgs")));
```

> 设置为公开的静态目录以后，我们就可以直接访问了

多图片上传

多图片上传的时候，配置有一些不一样

```
<input type="file" name="sphoto" id="sphoto">
<input type="file" name="sphoto2" id="sphoto2">
```

上面有两个文件选择框，但是`name`属性值不一样，所以我们在加载`multer`这个插件的时候，要如下配置

```
upload.fields([
    { name: "sphoto" }, { name: "sphoto2" }
]),
```

这个时候，我们接收到的`req.files`就是一个如下的对象

```
{
  sphoto: [
    {
      fieldname: 'sphoto',
      originalname: 's_photo-1533655937571.jpg',
      encoding: '7bit',
      mimetype: 'image/jpeg',
      destination: 'D:\\杨标的工作文件\\班级教学笔记
\\H2003\\0226\\code\\022601\\uploadImgs',
      filename: 'aa74d57b51ed5cbb17058835ec331ada',
      path: 'D:\\杨标的工作文件\\班级教学笔记
\\H2003\\0226\\code\\022601\\uploadImgs\\aa74d57b51ed5cbb17058835ec331ada',
      size: 5777889
    }
  ],
  sphoto2: [
    {
      fieldname: 'sphoto2',
      originalname: 's_photo-1533799257285.jpg',
      encoding: '7bit',
      mimetype: 'image/jpeg',
      destination: 'D:\\杨标的工作文件\\班级教学笔记
\\H2003\\0226\\code\\022601\\uploadImgs',
      filename: 'de1b3e809d3b88683fe69aa4b45268a5',
      path: 'D:\\杨标的工作文件\\班级教学笔记
\\H2003\\0226\\code\\022601\\uploadImgs\\de1b3e809d3b88683fe69aa4b45268a5',
      size: 100147
    }
  ]
}
```

对于上面这种数据格式，我们需要改后缀名，关键就在于怎么处理上面的数据格式

```
app.post("/doAddStuInfo2", upload.fields([
    { name: "sphoto" }, { name: "sphoto2" }
]), (req, resp) => {
    console.log(req.body);
    let v = Object.values(req.files);
    if(v.length>0){
        v.forEach(item => {
            let file = item[0];
            fs.renameSync(file.path, file.path + file.originalname);
        });
    }

    resp.send("我收到你的请求了");
})
```

多图片上传二

还有一种情况的多图片上传指的是一个文件选择框里面可以选择多个文件，所以有如下情况产生

```
<input type="file" name="sphoto" id="sphoto" multiple>
```

针对上面的情况，我们可以采用如下方式解决

```
//多图片上传，一个选择框先多个文件
app.post("/doAddStuInfo2", upload.array("sphoto"), (req, resp) => {
    console.log(req.files);
});
```

这个时候，我们接收到的文件数据格式就是如下的数据格式了

```
[
  {
    fieldname: 'sphoto',
    originalname: 's_photo-1533708439617.JPG',
    encoding: '7bit',
    mimetype: 'image/jpeg',
    destination: 'D:\\杨标的工作文件\\班级教学笔记
\\H2003\\0226\\code\\022601\\uploadImgs',
    filename: '502b44475bd6263f92db6e98c48c7f5c',
    path: 'D:\\杨标的工作文件\\班级教学笔记
\\H2003\\0226\\code\\022601\\uploadImgs\\502b44475bd6263f92db6e98c48c7f5c',
    size: 997601
  },
  {
    fieldname: 'sphoto',
    originalname: 's_photo-1533709034337.JPG',
    encoding: '7bit',
    mimetype: 'image/jpeg',
    destination: 'D:\\杨标的工作文件\\班级教学笔记
\\H2003\\0226\\code\\022601\\uploadImgs',
    filename: 'a84d8ea7c72ef653510cbe20a90dec0d',
    path: 'D:\\杨标的工作文件\\班级教学笔记
\\H2003\\0226\\code\\022601\\uploadImgs\\a84d8ea7c72ef653510cbe20a90dec0d',
    size: 747948
  }
]
```

它是一个数组，我们只要把这个数组去遍历，然后再重命名就可以了

针对上面接收到的数据，我们可以进行如下重命名处理

```
//多图片上传，一个选择框先多个文件
app.post("/doAddStuInfo2", upload.array("sphoto"), (req, resp) => {
    console.log(req.body);
    if (req.files.length > 0) {
        //说明我接收到文件服
        for (let file of req.files) {
            fs.renameSync(file.path, file.path + file.originalname);
        }
    }
    resp.send("我收到你的请求啦");
});
```

省市区三级联动

首先如果要实现省市区三级联动，我们需要将数据准备好

areaId	areaCode	areaName	level	cityCode	center	parentId
1683	420102	江岸区	3	027	114.30911,30.60	1682
1684	420103	江汉区	3	027	114.270867,30.6	1682
1685	420104	硚口区	3	027	114.21492,30.58	1682
1686	420105	汉阳区	3	027	114.21861,30.55	1682
1687	420106	武昌区	3	027	114.31665,30.55	1682
1688	420107	青山区	3	027	114.384968,30.6	1682
1689	420111	洪山区	3	027	114.343796,30.5	1682
1690	420112	东西湖区	3	027	114.137116,30.6	1682
1691	420113	汉南区	3	027	114.084597,30.3	1682
1692	420114	蔡甸区	3	027	114.087285,30.5	1682
1693	420115	江夏区	3	027	114.319097,30.3	1682
1694	420116	黄陂区	3	027	114.375725,30.8	1682
1695	420117	新洲区	3	027	114.801096,30.8	1682

接下来完成我们的指定功能

在之前做学生信息的新增的时候，我们发现我们的地址这里是输入的，这是不符合要求的，所以我们现在将这里进行修改

前台页面代码

```
<div class="form-group">
    <label for="" class="control-label col-sm-2">学生地址</label>
    <div class="col-sm-7">
        <div class="row">
            <div class="col-sm-4">
                <select id="province" class="form-control">
                    {{each provinceList item index}}
                    <option value="{{item.areaId}}">{{item.areaName}}</option>
                    {{/each}}
                </select>
            </div>
            <div class="col-sm-4">
                <select id="city" class="form-control"></select>
            </div>
            <div class="col-sm-4">
                <select id="area" class="form-control"></select>
            </div>
        </div>
        <div class="row">
            <div class="col-sm-12">
                <textarea placeholder="请输入详细信息" class="form-control">
</textarea>
            </div>
        </div>
    </div>
    <div class="col-sm-3"></div>
</div>
```

前台JS代码

```
// 省的下拉选项框发生了变化
$("#province").change(function () {
    let province = $(this).val();
    //接下来要把province的数据发送到后台，要使用无刷新的ajax请求
    $.get("/area/getListByParentId?parentId=" + province, function (data) {
        $("#city").html(data);
        $("#city").change();
    });
});

$("#city").change(function () {
    let city = $(this).val();
    $.get("/area/getListByParentId?parentId=" + city, function (data) {
        $("#area").html(data);
    });
});
```

后台JS代码

```
//接收前台传递过来的parentId,然后获取列表，最后返回
router.get("/getListByParentId", async (req, resp) => {
    try {
        let { parentId } = req.query;
        let areaList = await
ServiceFactory.areaService.getListByParentId(parentId);
        resp.render("stuInfo/partial_area_select",{
            areaList
        });
    } catch (error) {
        resp.send("服务器错误");
        console.log(error);
    }
});
```

更详细的步骤，可以参考下个章节的ajax与部分模板的结合完成

ajax请求与部分模板的结合

> 这个技术是目前比较流行的一种技术，叫模板的异步渲染，是MVC开发下面经常会使用到的

之前同学们都意识一点，如果请求是Ajax，我们一般返回的都是一个JSON的字符串，如上面的省市区三级联动里面，我们可以返回JSON字符串

前台代码：发送Ajax请求

```
// 省的下拉选项框发生了变化
$("#province").change(function () {
    let province = $(this).val();
    //接下来要把province的数据发送到后台，要使用无刷新的ajax请求
    $.get("/area/getListByParentId?parentId="+province,function(data){
        console.log(data);
    });
});
```

后台代码：返回了JSON的数据格式

```
//接收前台传递过来的parentId,然后获取列表，最后返回
router.get("/getListByParentId", async (req, resp) => {
    try {
        let { parentId } = req.query;
        let areaList = await
ServiceFactory.areaService.getListByParentId(parentId);
        let jsonStr = JSON.stringify(areaList);
        //发送了一串JSON字符串到前台
        resp.send(jsonStr);
    } catch (error) {
        resp.send("服务器错误");
        console.log(error);
    }
});
```

关键问题，这里的需求是需要再将这个JSON字符串渲染到下面的HTML当中

```
 <div class="col-sm-4">
     <select id="city" class="form-control"></select>
</div>
```

说得再确切一点就是，我拿到了JSON字符串以后，我还是要再进行一次渲染，这样做就很麻烦，那么能不能是服务器直接渲染好了再给我呢?

答案是可以的。这个时候就要使用部分模板

第一步：准备部分模板的内容partial_area_select.html,里面的代码如下

```
<!-- 我假设这个模板有一组数据叫areaList,并且就是数据表t_area里面的数据 -->
{{each areaList item index}}
<option value="{{item.areaId}}">{{item.areaName}}</option>
{{/each}}
```

第二步：修改后台路由里面返回的数据类型，之前我们返回的数据是一个JSON数据，现在我们渲染刚刚的分布模板

```
//接收前台传递过来的parentId,然后获取列表，最后返回
router.get("/getListByParentId", async (req, resp) => {
    try {
        let { parentId } = req.query;
        let areaList = await
ServiceFactory.areaService.getListByParentId(parentId);
        //在这里，我们渲染了分布模板
        resp.render("stuInfo/partial_area_select",{
            areaList
        });
    } catch (error) {
        resp.send("服务器错误");
        console.log(error);
    }
});
```

第三步：分析前台的Ajax请求的结果

Express中的验证码

在nodejs的web项目当中，我们经常会使用到验证码的功能，这个时候我们可以借用于第三方的包来完成

```
$ npm install svg-captcha --save
```

接下来在项目当中去使用

```
const svgCaptcha = require("svg-captcha");
//当请求这个地址的时候，我就帮你生成一个图片显示出去
app.get("/getVCode", (req, resp) => {
    let svgObj = svgCaptcha.create();   //创建了一个对象
    // 在这里要告诉浏览器，我send给你的不是html标签，是一个svg的图片
    resp.setHeader("Content-Type", "image/svg+xml");
    resp.send(svgObj.data);
});
```

在上面的代码当中，我们一定要设置响应头为`image/svg+xml`，只有设置为这种格式，在页面上面才能够正确的显示

```
<img src="/getVCode" alt="验证码" >
```

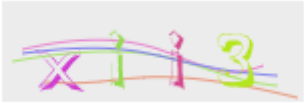
这样我们就可以在网页上面显示一个图片了

上面就是我们最基本的用法，在我们使用这个验证码的时候，里面有创建验证码的时候有一些特殊的设置选项，也跟大家介绍一下

```
let svgObj = svgCaptcha.create({
    size:4,
    noise:15,
    ignoreChars:"abcdefgABCDEF",
    color:true,
    background:"#ececec"
});
```

1. `size`用于设置生成验证码的字符的数量，默认值为4
2. `noise`代表生成的验证码干扰线的数量
3. `ignoreChars`生成验证码的时候要忽略哪些字符
4. `color`生成的验证码文字是否带颜色，默认为false不带颜色，就是黑色，如果设置为true，则代表生成彩色的
5. `background`代表生成的验证码的背景颜色，如果设置了这个属性，则上面的`color`就自动为true

通过上面的方式所生成的验证如下图所示



它是产生了4位随机数，如果想调整生成验证码字符的数量，可以设置`size`的属性值

验证码除了可以生成上面的四位随机数以外，还可以生成算术运算符，方法也很简单，如下

```
let svgObj = svgCaptcha.createMathExpr({
    mathMin: 10,
    mathMax: 100,
    mathOperator: ["+", "-"][parseInt(Math.random() * 2)],
    noise: 4
});
```

上面就是生成数学运算符的验证码

1. `mathMin`出现的最小数字
2. `mathMax`出现的最大数字
3. `mathOperator`运算符的符号，这里用了一个随机数。同时要注意，这里只支持“+”与"-"

通过上面的方式产生的随机数如下

Express中的session使用

Session是服务器的存储机制，它可以在服务器上面开辟一个独立的互不干扰的空间去存储数据，这个空间不受客户端的请求的影响，每个session代表的是浏览器与服务器的连接

Express默认是不支持session的，它依赖于第三方模块`express-session`

第一步：安装模块

```
$ npm install express-session --save
```

第二步：配置模块

```
const session = require("express-session");
app.use(session({
    secret: "biaogege123123aa",           //session是否加密
    resave: true,                         //session是否可以重新保存
    saveUninitialized: false              //保存的时候是否初始化，这里不需要
}));
```

第三步：使用session，这里我们以验证为为例子

```
router.get("/getVCode", (req, resp) => {
    let svgObj = svgCaptcha.create({
        size: 4,
        noise: 2,
```

```
        color: true,
        background: "#ececec"
    });
    //使用session去存储验证码，因为session是根据用户的链接请求产生的，session之前互不影响
    //每个用户只要链接了服务器就会有session
    req.session.VCode = svgObj.text;
    resp.setHeader("Content-Type", "image/svg+xml");
    resp.send(svgObj.data);
});

//接收前台传递过来的验证码，用于检测是否输入正确
router.get("/checkVCode", (req, resp) => {
    let { VCode } = req.query;
    let VCodeText = req.session.VCode;
    if(VCodeText.toUpperCase()==VCode.toUpperCase()){
        resp.send("验证成功");
    }
    else{
        resp.send("验证失败");
    }
});
```

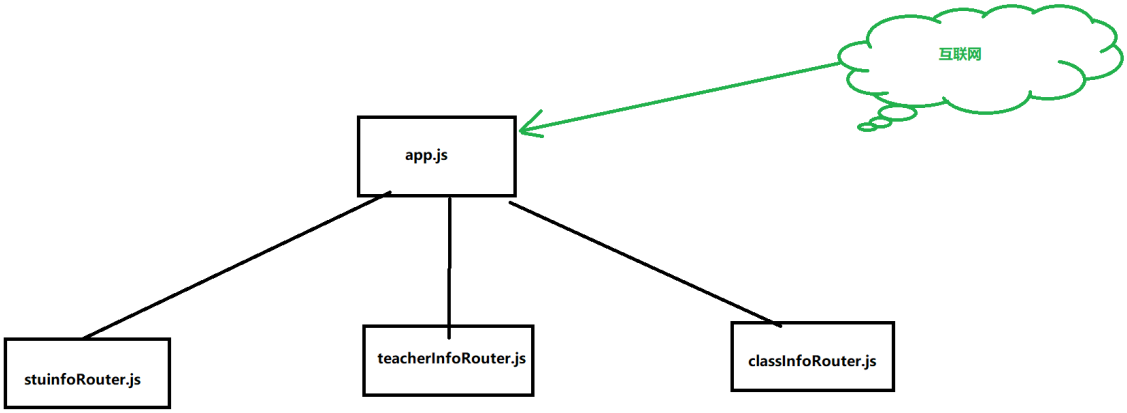
在上面的代码当中，我们是把验证码存在了`session`里面，因为`session`是互相独立互不干扰的的，每个请求在验证的时候是独立的

总结来说，session具备以下几个特点

1. `session`是独立的互不干扰的
2. `session`是会随着请求自动产生，随着会话的中断而自动消失
3. `session`是存在服务器的

express的拦截器

拦截器顾名思义就是把某请求给拦下来，通过这种方式，我们可以实现很多功能，如登陆验证的功能



在上面的图片里，我们如果要想实现整个的的拦截功能，怎么办呢?

拦截器既可以设置在`app.js`里面，也要可以设置在某一个路由里面，因为我们现在要做的是登陆验证，所以我们应该把拦截器设置在`app.js`这个地方，做到全局拦截

拦截器的语法非常简单，如下

```
app.use((req,resp,next)=>{
    //这里就是拦下来以后的处理
});
```

注意：拦截器应该设置在路由加载之前，静态文件加载之后

AppConfig.js的代码

```
const AppConfig = {
    userRole: {
        student: Symbol(),
        teacher: Symbol(),
        admin: Symbol(),
        superAdmin: Symbol()
    },
    //这里配置的是不需要经过拦截器拦截的页面
    excludePath:[
        "/admin/login",
        "/admin/getVCode",
        "/admin/checkVCode",
        "/admin/checkLogin"
    ]
}

module.exports = AppConfig;
```

app.js里面拦截器的代码

```
const AppConfig = require("./config/AppConfig");

app.use((req, resp, next) => {
    //只用在这里判断一下用户有没有登陆就行了，用户登陆以后有什么特征
    if (req.session.userInfo) {
        //说明你登陆了
        next();     //正常放行
    }
    else {
        //说明你没有登陆,但是如果是一些不需要拦截的，我们也要放行
        if (AppConfig.excludePath.includes(req.path)) {
            next();
        }
        else{
            //拦下来，跳转到登陆页面
            // resp.redirect("/admin/login");
            resp.send(`<script>top.location.replace("/admin/login");</script>`)
        }
    }
});
```

Express项目退了登陆

一个程序登陆以后必然会涉及到退出的功能，退出的的时候应该有两个操作

1. 销毁之前的登陆凭证，也就是`session`
2. 重新跳回登陆页面

```
//退出登陆
router.get("/logOut",(req,resp)=>{
    // 销毁存储的所有 session
    req.session.destroy();
    //跳转到登陆页面
    resp.send(`<script>top.location.replace("/admin/login");</script>`);
})
```

同时只用在前台如下处理就可以了

```
<a href="/admin/logOut" data-btn-confirm="你确定要退出吗"><span class="text-
danger">退出系统</span></a>
```

Express的重定向应用

在`app.js`里面的加载路由之前，添加下面的代码，这样我们在访问`http://127.0.0.1`的时候就会自动的跳转到`http://127.0.0.1/admin/login`

```
//处理重定向
app.get("/",(req,resp)=>{
    resp.redirect("/admin/login");
});
```

Express的路径变量

> 路径变量是请求方式一种，叫`pathVariable`,就是把变量写成路径传递到后台去

express里面的路径变量是目前一种比较流行的前端向后端服务器传值的方式

以前的时候，有`get`传值或`post`传值

1. `get`传值对应的是`req.query`来进行取值
2. `post`传值对应的是`req.body`的方式来取值
3. `pathVariable`传值使用`req.params`的方式来取值

get请求

```
$.get("/abc?uid=1&pwd=abc", function (data) {
    console.log(data);
});
//后端代码
app.get("/abc", (req, resp) => {
    console.log(req.query);     //get值
    resp.send("1111");
});
```

post请求

```
$.post("/doAbc", {
    uid: "hello",
    pwd: "123"
}, function (data) {
    console.log(data);
})
//后端代码
app.post("/doAbc", (req, resp) => {
    console.log(req.body);
    resp.send("1111");
});
```

pathVariable的请求

```
 $.post("/def/biaogege/123456",{
     age:18
 }, function (data) {
     console.log(data);
 });
//后端代码
//路径变量请求
app.post("/def/:uid/:pwd", (req, resp) => {
    console.log(req.params);        //取路径变量里面的参数
    console.log(req.body);
    resp.send("路径变量");
})
```

通过上面的对比， 我们发现路径变量的请求就更适合于小范围的传值， 所以以前的get请求就有可能会使用路径变量的方式

传值方式	服务器接收值方式	备注
`router.get("/list")`	req.query	
`router.post("/list")`	req.body	
`router.get("/list/:sid")`	req.params	pathVariable具体参照上面的内容

总结

在express的整个项目里面，每个文件夹各有各的作用

1. utils存放一些第三方的常用方法或对象
2. views存放视图模板
3. routes存放路由文件，所以与请求相关的都应该在路由
4. service存放服务文件，所有与数据库操作相关的都在这里
5. public存放公开文件（或称之为静态文件）

MVC开发模式

Express框架就是一个MVC的开发方式，MVC是一种前后结合的开发方式

MVC是一种开发模式，并不是一种技术。它不针对于语言，而针于对于平台与架构

C#语言对应的是asp.net MVC或MVC core框架

Java里面以前是Structs，现在是 springMVC框架

php里面则是thinkPHP框架

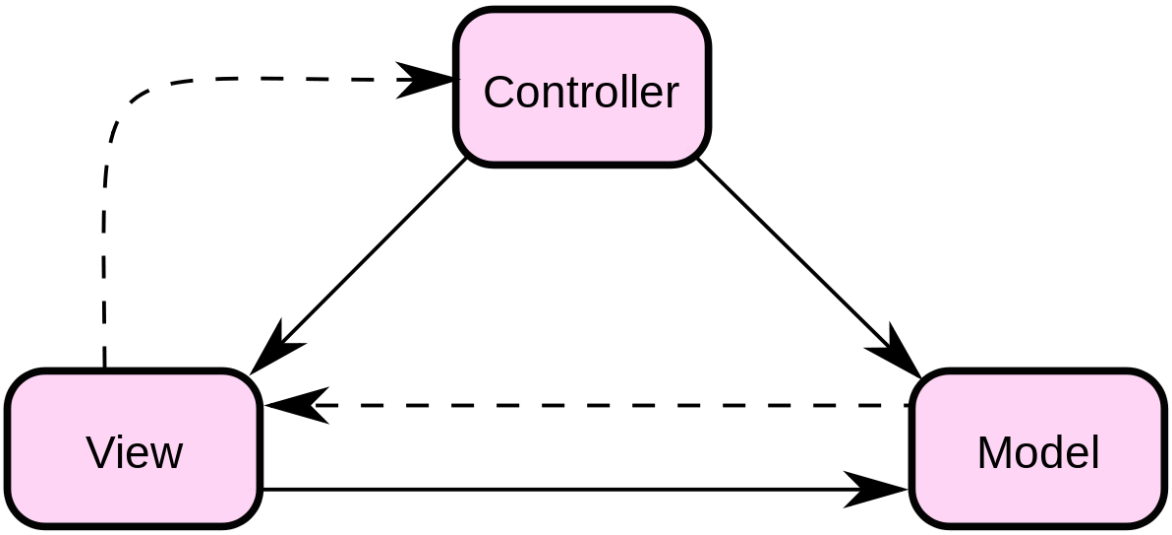
在我们的NodeJS里面，有很多MVC类型的框架 ， Express、KOA、EGG

所谓的MVC其实就是三大块

M:Model模型

V:View视图

C:Controller控制器



请求首先到达的是控制器，控制器根据你的请求路径去决定做相应的操作，例如通过service调用数据库的方法，获取数据（Model），然后将这个模块渲染到一个特定的视图模板文件里面，然后再将渲染好的页面返回到浏览器

优点

1. 层次结构非常明确，控制器和视图与模型分开，可以构建低耦合的开发模式
2. MVC并不是全套，可以根据相应的需求来决定是否结合

   例如如果仅仅只是需要返回一个视图view，则不需要model

   如果仅仅只是希望返回一串json数据，则不需要view

缺点

1. **前后端结合**的开发模式，视图必须要控制器渲染能够显示，前端的页面代码与后端的逻辑处理，数据库处理，模块处理的代码都放在了一起，项目比较臃肿
2. 视图与模型的结合渲染是在服务器完成的，当结合完成以后，再通过`resp`返回给浏览器展示出来，这样会消耗服务器的性能

   （为了解决这样的缺点，后期出现的开发模式就是MVVM的开发模式）