# 前端基础和技术选型

## 实验介绍

本实验主要介绍一下前端的基础知识，对比认识一下各个框架的代码编写方式，并介绍我们本次技术选型的主要思路。对于前端三大技术 HTML、CSS、JavaScript，简单的介绍了基本情况和常用语法。中间介绍了现代框架的一些情况，并通过实际的案例，用代码直观的认识一下各种框架的实现方式。最后分析一下项目的技术选型。

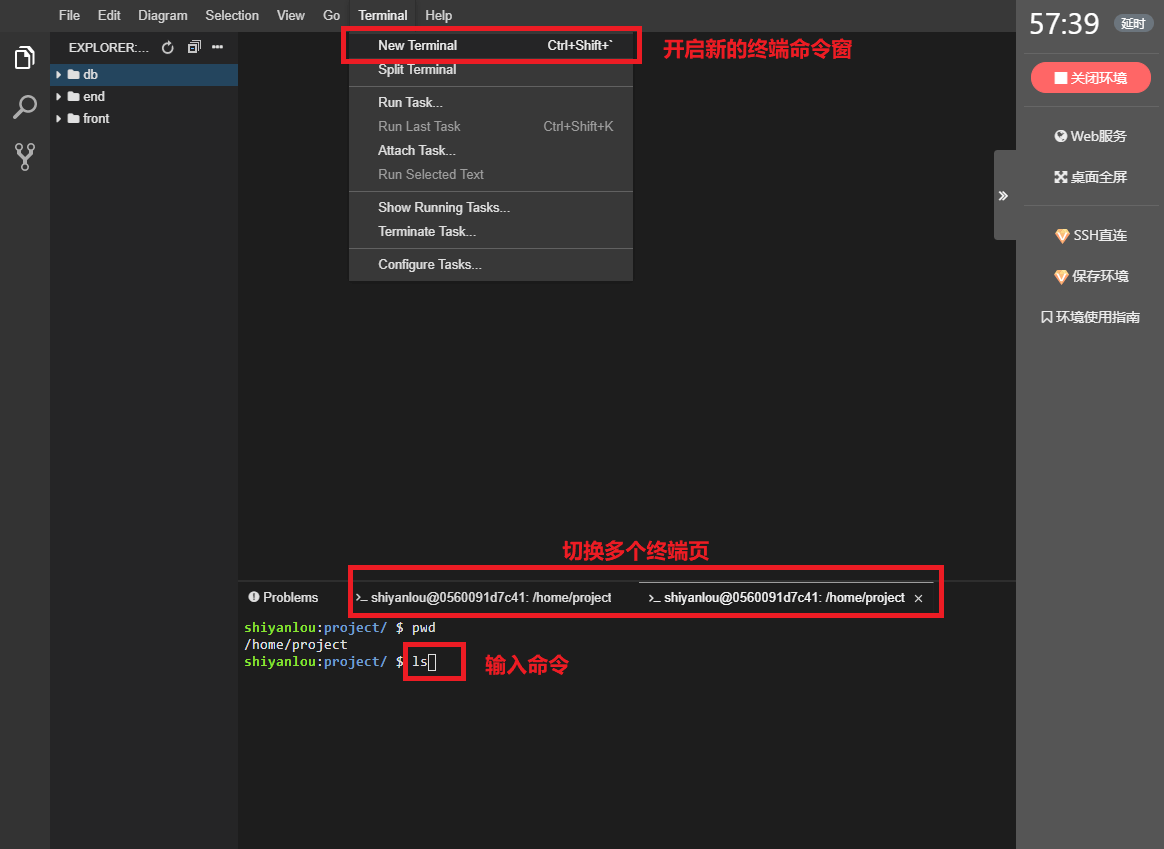
#### 知识点

* HTML、CSS、JavaScript 快速概览
* 前端框架概览和选型
* 后端选型
* 数据库选型
* Web 服务器选型

## 前端技术

本节我们简单介绍一下前端最基础 HTML, CSS, JavaScript 三驾马车。虽然本课程预设的读者为零基础开发者，但是前端开发一定会这三种技术的运用有要求。建议抽空学习 [《 Web 前端工程师路径》](https://www.lanqiao.cn/paths/12/) 中的阶段 1 甚至阶段 2。这里仅做语法介绍和基本使用的概览。

在此之前先认识一下实验环境。实验环境和 VS Code 使用体验基本一致。你可以启动一个终端，并在其中输入 Linux 命令。



后面的命令无特殊说明的都是在此**终端命令行**中输入。大多数命令可以多开终端窗口分别执行。

那么下面我们就快速的了解一下。

### HTML

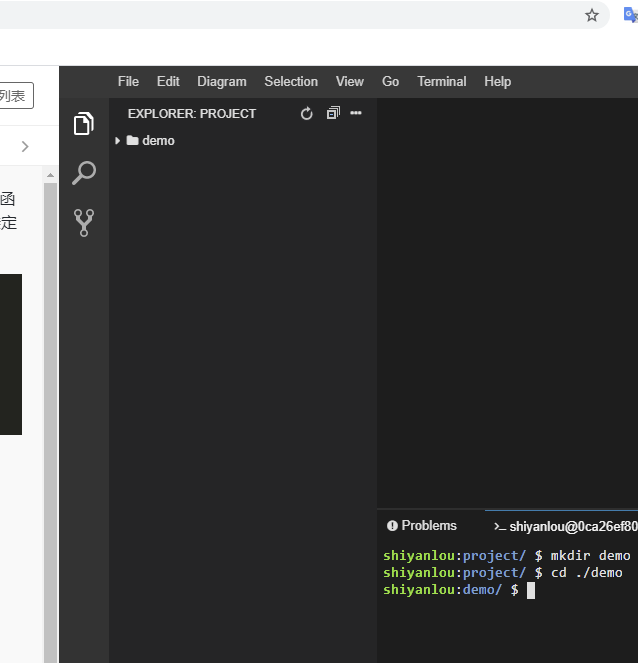
HTML 全称超文本标记语言，几乎是从万维网和浏览器产生伊始就存在的。主要用于结构化信息来方便浏览器展示。以标签对作为主要特征，如<p>这是一个段落</p>。这些标签会被浏览器解析成不同的模块，比如 p 标签就是一个段落，img 标签就是一个图片，a 标签就是一个超链接，标签名不区分大小写。

立刻就来尝试一下吧。首先通过命令行新建一个 demo 目录：

mkdir demo

然后命令行进入 demo 目录：

cd ./demo



新建一个 hello.html 文件，可以在实验环境左边的浏览框内在 demo 上右键选择 New File 然后命名为 hello.html；或者也可在命令行终端输入 touch hello.html，同样是新建文件。在其中输入以下内容：

<!DOCTYPE html>

<html>

<head>

<meta charset="UTF-8" />

<title>标题</title>

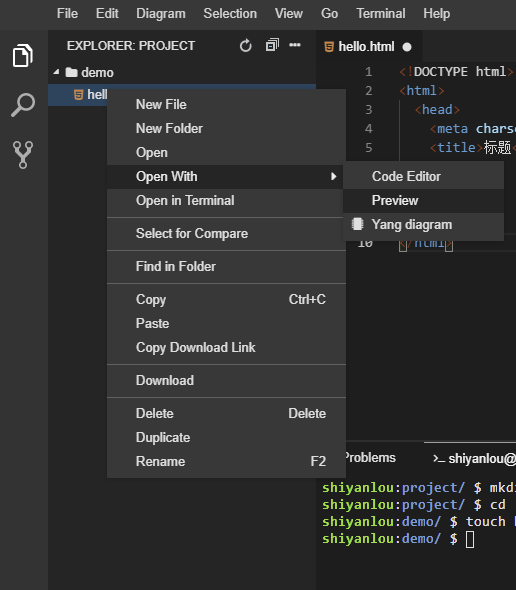
</head>

<body>

正文

</body>

</html>

然后右击文件选择 Open With → Preview。 

看到了吗？其实我们就是新建了一个 .html 后缀的文本文件，然后浏览器就可以将其中的内容展示出来。你也可以在自己的桌面上新建一个，然后使用浏览器打开看看效果。这里嵌套代码的缩进格式是为了美观和可读性，并无严格要求。

head 标签中是一些暂时无需用到的头部信息，渲染的主体是 body 标签。下面我们修改 body 标签里面的内容，填入一些常用标签来直观感受一下。

<body>

<h1>页面标题</h1>

<div>一个块容器</div>

<div>又一个块容器</div>

<p>这里是段落了，间距变大</p>

<div>一个块容器</div>

<div>

<div>

多层嵌套：

<div>内部第一个</div>

<div>内部第二个</div>

</div>

</div>

</body>

保存之后切换到浏览标签看一看，有没有感觉被忽悠了？嵌套没嵌套根本没体现出来，就像 Word 里排了一下版，按了几个回车。



因为我们没有对显示样式进行修改，那是 CSS 的事。HTML 主要管内容的组织结构。

*这里有一点针对学习的小建议，本课程中给到的全部代码请手动输入，忘记复制和粘贴快捷键。而且最好不要机械的一个字符一个字符照着抄，尽量看过一行或一小段代码之后，靠短暂的印象去输出，别怕出错，只有过脑子并输出实践，才是最快掌握一项技能的捷径。以上两句话是本课程中****最有价值****内容之一。*

下面我们接着修改刚才的代码，再给 body 中添加几个常用标签。每次修改和保存之后记得到预览页看看样式的变化。

<h4>4 级标题</h4>

<ul>

<li>

HTML

</li>

<li>

CSS

</li>

<li>

JavaScript

</li>

</ul>

<div>

<a href="https://www.lanqiao.cn" target="\_blank"

>点击超链接跳转至实验楼首页</a

>

</div>

<div>

<img

src="https://static.shiyanlou.com/frontend/dist/img/9f43b00.svg"

alt=""

width="200"

/>

</div>



最后的链接标签 a 和图片标签 img 出现了标签属性，属性为 attr="value" 格式，可以给标签增加更丰富的信息。

同时 img 标签还是一个单标签，不需要在后面添加 </img> 配合使用。

至此对 HTML 的简要介绍告一段落。

互联网上看到的各种五彩缤纷网页都是由这些 HTML 组成的，但是为什么我们写的这么难看？下一节我们就要学习一下如何用 CSS 美化页面。

### CSS

CSS 全称层叠样式表，是专门用来修饰 HTML 样式的一种语言。我们修改一下上节的 hello.html 文件来直观感受一下。

#### 内部代码块引入

在 head 标签内部增加以下 style 代码块：

<head>

<meta charset="UTF-8" />

<title>标题</title>

<style type="text/css">

div {

border: 1px solid blue;

padding: 2px;

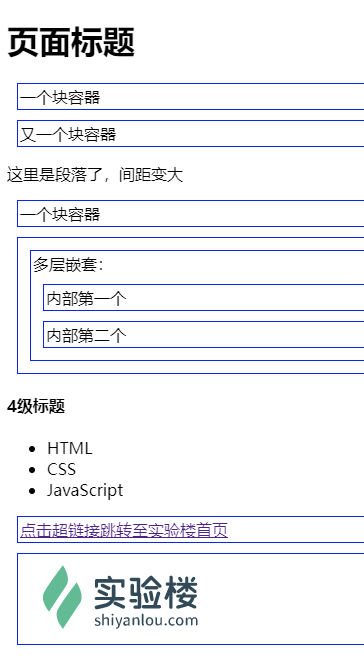
margin: 10px;

}

</style>

</head>

这是再切换到预览页，发现没那么平铺直叙了。



这就是 CSS 的第一种引入方式，HTML 内置代码块。

大括号外面的 div 是标签选择器，这里选中了本页面中的所有 div 元素。大括号里面是属性名与赋值，属性名都是固定的关键字，并已规定好了值的类型和可选范围。

读代码也就大概知道了，我们将 div 的边框设置为 1 像素宽、固体（单线型）、蓝色，填充（内边距）2 像素，边缘空白（外边距）10 像素。现在可以练习调整一下各个数字，预览看看发生了什么？

再说点题外话，懂一些英文对程序员来说非常必要，除了可以凭感觉就读懂没学过的代码，还可以在面向 Google 编程、面向 Stack Overflow 编程、面向 Github Issues 编程时游刃有余。

#### 外部文件引入

然后我们再试一下外部文件引入，在 hello.html 的同级目录新建 hello.css，输入以下内容保存：

div {

color: green;

border: 2px dotted red;

}

然后修改 hello.html，在 style 标签后面增加一行 link 标签，添加引入类型和地址：

<style type="text/css">

div {

border: 1px solid blue;

padding: 2px;

margin: 10px;

}

</style>

<link rel="stylesheet" href="hello.css" />

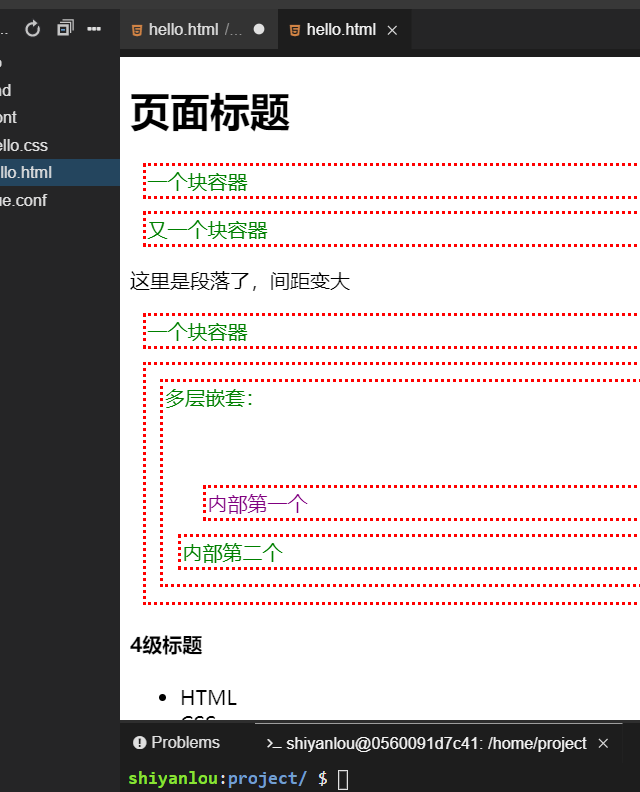
预览看看，文字颜色变为绿色，边框的样式也被更新为 2 像素宽、点线型红色。

同样是 div 选择器，为什么边框的样式被覆盖了呢？注意 CSS 在同样条件下会后面代码覆盖前面，可以尝试交换 link 标签和 style 标签块的顺序看看。

#### 行间样式

最后一种叫行间样式，这个结构更简单。修改 hello.html 中的 <div>内部第一个</div> 为

<div style="margin: 60px 0 10px 30px ;color:purple;">内部第一个</div>



样式覆盖前两种方式了，因为行间样式的优先级较高。这里涉及到选择器权重，先给一个简单公式了解一下。

!important > 行间样式 > ID > class | 伪类 | 属性选择 > 标签 > 继承 | 通配符。

多个选择器作用时权重相加。这里算 CSS 里有点复杂的部分，暂时不展开。

这里还有个小知识点是内外边距 margin 和 padding 接受的完整的值是四个，顺序固定为“上 右 下 左”。如果省略参数则从末尾计算对向合并。比如：

margin:40px 20px 50px; 三个参数时，左右同为 20px。

margin:40px 20px; 两个参数时 上下同为 40px, 左右同为 20px。

margin:40px; 一个参数时呢？请自行尝试理解。

CSS 先讲这么多，虽然没有把我们的页面变多好看，但最起码知道努力的方向了。

### JavaScript

制作 JavaScript 的快速入门简直非常伤脑筋。比起前两种技术 HTML 和 CSS，这是货真价实的编程语言了。也是我们后面要用到的 Vue.js 和 Node.js 中的根基，一下子又很难讲很多，所以还是希望同学们能重视起来系统学习一下，最起码读到后面的代码时不至于陷入“这是啥这又是啥”的窘境。

来段代码直观认知一下，还是先内部代码块引入。

在 hello.html 的 head 标签内增加一个代码块：

<link rel="stylesheet" href="./hello.css">

<script>

let message = "字符串提示";

function showMSG(msg) {

alert(msg);

}

</script>

修改 hello.html 的 h1 标签为：

<h1 onclick="showMSG(message)">页面标题</h1>

保存预览，点击“页面标题”，会弹出提示框。



JavaScript 代码加载之后就会执行，不存在编译阶段。行末的分号绝大多数时候可以省略。

我们先定义了一个变量 message，并赋值为“字符串提示”。定义变量关键字原是 var，ES6 新增关键字 let 有更清晰的作用域，可替代使用。

*学习 JavaScript 经常会碰到 ES6、ES7 之类的名词，实际上是 ECMAScript 标准的版本号的意思。可以简单理解为新版标准为 JavaScript 添加特定新特性。*

然后我们定义了一个函数 showMSG，并添加一个形参 msg。在函数体内部调用浏览器弹框方法，显示 msg 的值。function 是定义函数的关键字，暂时先把它当做一个功能封闭的盒子，当函数调用时，执行函数体内的代码。

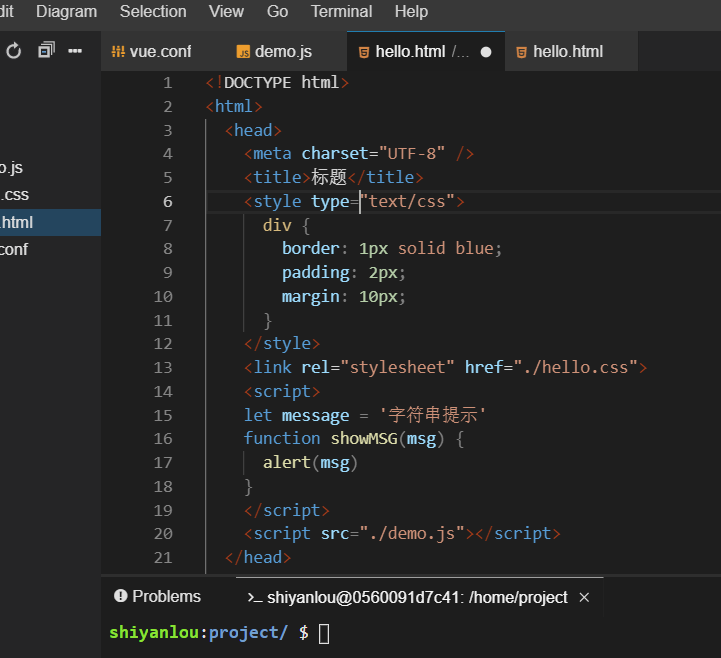
调用部分是先给 h1 标签添加了 onclick 点击事件，被点击时触发 showMSG(message)，也就是把 message 传给了 msg。

之后再试一下调用外部 js 文件，新建 demo.js 文件，写入下面内容并保存。

message = '修改一下字符串';

然后修改 hello.html 文件，在 script 代码块后面增加一行：

<script src="./demo.js"></script>



这次保存预览，点击“页面标题”，可以看到弹窗的文字变了。这个演示了 script 代码块在页面可以同时存在多个，也是顺序调用，而且互相之间可以直接访问。文件命名也没有要求，希望不会逼死强迫症。

JavaScript 就是为什么网页可以做那么多交互的源头了。掌握起来任重道远。

### 现代前端框架对比与选型

截止 2019 年底，前端三大主流框架为 VAR（Vue.js, Angular, React）应该是没有悬念的，基本终结了前些年 jQuery 一统天下的局面。

从我个人角度简单评价一下：

jQuery 有方便的选择器操作和万能的 DOM 操作。

三大框架无论哪一个都具有组件化，数据绑定，高性能，丰富的语法糖等优点，选哪一个深入下去都可以做的非常好。

| **名称** | **开发维护者** | **特点** | **热度** | **适用范围** | **学习难度** |
| --- | --- | --- | --- | --- | --- |
| Angular | Google | 强制 TS，工程化 | 中 | 大型项目 | 高 |
| React | Facebook | JSX 语法，生态最繁荣，资源最丰富 | 最高 | 所有项目 | 中 |
| Vue.js | 尤雨溪及社区 | 模板语法，国内生态繁荣、资源丰富 | 高 | 所有项目 | 低 |

这里为了更直观的了解一下这些技术，我专门制作了一组演示实例，用原生 JS, jQuery, React, Vue.js 分别实现一下。为什么没有 Angular 呢？官方只推荐 TypeScript 和 Node.js 去实现，难以在本案例下简单对比，就先把它看做更严格更复杂更庞大的 Vue.js 好了。

案例只实现一个最简单的功能，通过数组渲染一组无序列表，然后通过输入框和按钮给数组增加元素，然后重新渲染整个数组。还是老建议，除了地址引用复制，其他尽量手敲，用身体感受不同框架下代码的差异。

### 原生 JavaScript 版

在路径 /home/projest/demo 路径下新建 js.html 文件。输入以下内容：

<!DOCTYPE html>

<html>

<head>

<meta charset="UTF-8" />

<title>JavaScript Demo</title>

</head>

<body>

<h2>JavaScript Demo</h2>

<div>

<input type="text" id="input" />

<button onclick="changeList()">添加</button>

<ul id="list"></ul>

</div>

<script>

let items = ['Vue.js', 'Angular', 'React', 'jQuery'];

function appendText(value) {

var para = document.createElement('li');

var node = document.createTextNode(value);

para.appendChild(node);

var list = document.getElementById('list');

list.appendChild(para);

}

function clearList() {

let list = document.getElementById('list');

var listChildren = list.childNodes;

for (let index = listChildren.length - 1; index >= 0; index--) {

const element = listChildren[index];

list.removeChild(element);

}

}

function refreshList() {

clearList();

for (let index = 0; index < items.length; index++) {

const element = items[index];

appendText(element);

}

}

function changeList() {

let newItem = document.getElementById('input').value;

items.push(newItem);

refreshList();

}

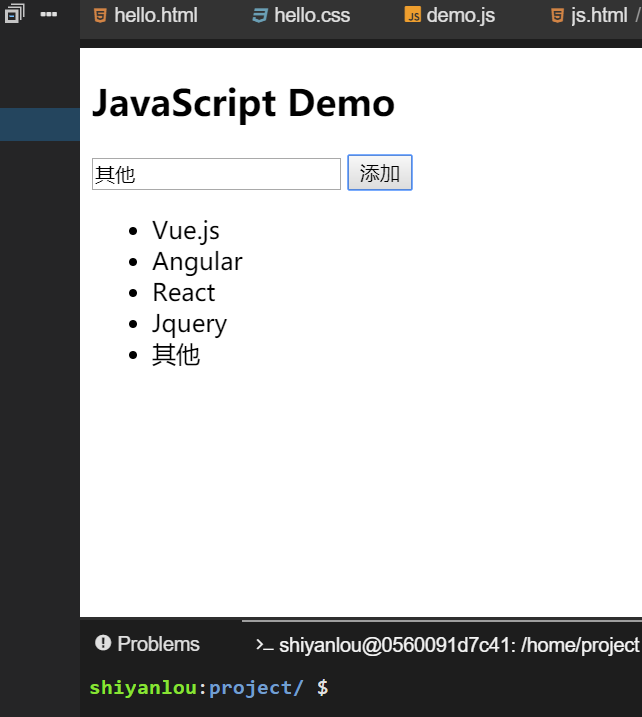
refreshList();

</script>

</body>

</html>

保存之后预览一下。尝试在输入框写入内容，然后点击添加按钮。



原生的代码比较直白。先定义了 items 数组并赋初始值。然后定义了新增节点、清空所有节点、重新渲染所有节点和按钮触发修改这四个函数。最后一行为初始化后执行第一次列表渲染。

这里用到了很多 DOM 操作，通过 id 拿到元素，再用 DOM 语法去增加或者删除。所以最好提前给 HTML 元素加好 id 方便获取。这段代码应该是很清晰好读的，不多做解释了。

再来一个小提示，编写函数时每个函数包含功能要尽可能少，但是命名一定要清晰有意义不怕长。好的编程习惯会让你事半功倍。

下面我们看看神器 jQuery 的表现。

### jQuery 版

在路径 /home/projest/demo 路径下新建 jquery.html 文件。输入以下内容：

<!DOCTYPE html>

<html>

<head>

<meta charset="UTF-8" />

<title>jQuery Demo</title>

<script src="https://cdn.jsdelivr.net/npm/jquery/dist/jquery.min.js"></script>

</head>

<body>

<h2>jQuery Demo</h2>

<div>

<input type="text" />

<button>添加</button>

<ul></ul>

</div>

<script>

let items = ['Vue.js', 'Angular', 'React', 'jQuery'];

function appendText(value) {

var txt = $('<li></li>').text(value);

$('ul').append(txt);

}

function refreshList() {

$('ul').empty();

for (let index = 0; index < items.length; index++) {

const element = items[index];

appendText(element);

}

}

$(document).ready(function () {

refreshList();

$('button').click(function (event) {

let newItem = $('input').val();

items.push(newItem);

refreshList();

});

});

</script>

</body>

</html>

第 6 行引入了 jQuery 库。这个代码比之前就轻量多了，首先就是 $ 为首的各种选择器简化了选择操作，也基本不用提前定义 id 了，然后封装的 DOM 操作命令简洁又强大。清空子节点不用自己写了，事件也可以在脚本里很方便的添加。

### React 版

在路径 /home/projest/demo 路径下新建 react.html 文件。输入以下内容：

<!DOCTYPE html>

<html>

<head>

<meta charset="UTF-8" />

<title>React Demo</title>

<script src="https://unpkg.com/react@16/umd/react.development.js"></script>

<script src="https://unpkg.com/react-dom@16/umd/react-dom.development.js"></script>

<script src="https://unpkg.com/babel-standalone@6.15.0/babel.min.js"></script>

</head>

<body>

<h2>React Demo</h2>

<div id="root"></div>

<script type="text/babel">

class Welcome extends React.Component {

constructor(props) {

super(props);

const list = props.list;

const listItems = list.map((item, index) => (

<li key={index}>{item}</li>

));

this.state = { listItems, newItem: '' };

this.handleClick = this.handleClick.bind(this);

this.handleChange = this.handleChange.bind(this);

}

handleClick() {

this.setState((prevState) => ({

listItems: [

...prevState.listItems,

<li key={prevState.listItems.length}>{prevState.newItem}</li>,

],

}));

}

handleChange() {

this.setState({ newItem: event.target.value });

}

render() {

return (

<div>

<input

type="text"

value={this.state.newItem}

onChange={this.handleChange}

/>

<button onClick={this.handleClick}>添加</button>

<ul>{this.state.listItems}</ul>

</div>

);

}

}

const items = ['Vue.js', 'Angular', 'React', 'Jquery'];

ReactDOM.render(

<Welcome list={items} />,

document.getElementById('root')

);

</script>

</body>

</html>

到了 React 风格突变，首先就是用到了 JavaScript 语句里写标签的 return(<div></div>) 这种 JSX 写法，注意括号里不是字符串哦。先是创建了一个组件类，在构造函数里增加了 state 并根据传入的 props 赋值。两个函数分别是更新列表和更新输入值，其中有一个 ... 是数组展开运算符（ES6）。后面 render() 的返回值为渲染模板。

如果说以前的网页是标签中夹杂着脚本，React 就像是脚本中夹杂着标签。通过这种结构可以轻松的写出可复用的组件，只要传入不同的数据即可。

### Vue.js 版

终于来到我的最爱了。

在路径 /home/projest/demo 路径下新建 vue.html 文件。输入以下内容：

<!DOCTYPE html>

<html>

<head>

<meta charset="UTF-8" />

<title>Vue.js Demo</title>

<script src="https://cdn.jsdelivr.net/npm/vue@2.6.0"></script>

</head>

<body>

<div id="root">

<h2>Vue.js Demo</h2>

<input v-model="newItem" type="text" />

<button @click="items.push(newItem)">添加</button>

<ul>

<li v-for="item in items" :key="item.id">

{{ item }}

</li>

</ul>

</div>

<script>

new Vue({

el: '#root',

data: {

items: ['Vue.js', 'Angular', 'React', 'jQuery'],

newItem: '',

},

});

</script>

</body>

</html>

在标签里加了些似曾相识的指令，功能就完成了。构思这个例子之前我也没觉得能对比出什么，就是直观的看一下各种框架的代码结构和外貌。可 Vue.js 确实体现出了低门槛和易用性，也许和创始人——前谷歌工程师尤雨溪后来居上地汲取了 Angular 和 React 两家的优点有关系。

因为后面实验还有详细的介绍，所以这里不解释语法了。

还有 Vue.js 中文文档完整，国内的相关资源丰富，这都是非常奢侈难得的优势。2020 年 Vue.js 即将推出新版本 3.0，会在兼容 2.0 的基础上增加一些更新更好的特性，正是入手学习的好时机。

### 后端技术选型

服务器后端的选择太多了，JAVA 应该还是老大没有悬念，PHP, C#, Go, Node.js, Python 也各有千秋。还是那句老话，技术都没什么大问题，主要的问题是技术不够精。前端做全栈推荐 Node.js 不纠结了，一种语言和生态了解和熟悉到极致比什么都强。尤其是 TypeScript 上马之后，有了静态类型大型项目也不慌。

确定 Node.js 之后，选择就简单多了。Express, Koa 和 Egg.js 几乎是一脉相承，Express 是 Node.js 生态广泛使用 Web 框架，网上很多教程和资源都是它的。Koa 就是 Express 团队吸收了很多新特性的下一代，2.0 版本以后除了 ES6 特性还增加了 Promise, async/await 解决回调地狱。Egg.js 是阿里巴巴基于 Koa2 的扩展封装。还有一些其他 Web 框架流行度和受众没那么高，暂不考虑了。

这里我的选择偏保守，还是使用 Koa。Express 就不推荐再用了——毕竟官方重构最为致命，Egg.js 新手入手又稍显复杂，如果真有需求可以在学通 Koa 以后再去考虑。

Koa2 还有个好玩的地方是库很小，只保留核心功能。需增加功能时再添加各种官方或者第三方中间件，多数中间件只做一件事，结构清晰源码好读，非常适合学习和练手。

### 数据库选型

开源数据库 MySQL, PostgreSQL, MongoDB 都非常流行，小项目大项目都可以应付下来。Node.js 生态对各种数据库支持也都不错。但是 MongoDB 作为存储键值对 NoSQL 数据库，数据文档本身就是以 JSON(JavaScript Object Notation) 的格式设计存在的，与 JavaScript 天然自洽。本身又是轻量好部署，扩展性强。所以直接选择 MongoDB。

如果自身对 SQL 语言有过学习和使用的经验，也可以考虑继续使用关系型数据库，或再配上如 Sequelize 这样的 ORM 类库，数据库使用并不是问题。

### Web 服务器选型

主流有 Apache, Nginx, tomcat, IIS 等。我们选一个轻量、简单、跨平台、性能高的，Nginx 完全搞定，实验环境又有预装，不用犹豫。

# 课程所需环境搭建

## 实验介绍

本节内容主要是指导在实验楼线上环境中如何快速搭建所需开发环境，以及简要的介绍所用到的命令和工具。前面介绍了 Linux 常用的几个基础命令，然后介绍了后面开发需要用到的一些工具的安装和初始化方法，如 Node.js、npm、Vue CLI、MongoDB 服务器，Nginx 服务器等。

#### 知识点

* 快速搭建环境
* npm 包管理器
* Git 代码管理
* Vue CLI 脚手架
* Node.js http 服务器
* MongoDB 服务器和客户端
* Nginx http 服务器

## 环境首次搭建

实验楼提供的线上实验环境已经为大家安装好了绝大部分所需工具，如：Git, MongoDB, Node.js, Vue CLI, Nginx 等。这样就为你省去了安装工作，并避免了工具版本号不一致带来的差异。

下面，我们附上一些基本的文件路径相关命令，这些命令在终端命令行输入使用。如果你不熟悉 Linux，可以学习 [《Linux 基础入门》](https://www.lanqiao.cn/courses/1) 课程。

cd /home/project

# 进入目录 /home/project，以 / 开头为绝对路径

cd ../

# 进入上级目录

pwd

# 查看当前的绝对路径

ls

# 查看当前目录中的所有目录名和文件名

ls -l

# 同上，并列出详细信息

cd project

# 进入同级目录下的 project 目录。或者严格一点使用 ./project

此外：

* Ctrl+C 快捷键为中断当前命令或操作。
* 运行命令时都要注意当前所在路径。
* 在输入中使用 Tab 键自动补全，可以更准确快速的输入目录名或文件名，返回上级目录尝试输入 cd p 然后按 Tab 键。

## 修改 npm 源

npm 作为 Node.js 默认的包管理器，可以下载 Node.js 环境所需要的各种包。这些源码包提供了各种各样的功能，使我们完成很多常见任务可以站在前人的肩膀上，无需重新发明轮子，快速的实现自己的需求。但是 npm 包的源地址是 https://registry.npmjs.org，国外的服务器下载起来的速度非常不理想，所以需要替换为了国内源。

在终端中输入下方命令更改 npm 源：

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

此源地址为阿里巴巴在国内架设的镜像站，10 分钟同步一次。

下面可以继续练习一下查看和修改源地址的命令。

查看当前源地址命令：

cat ~/.npmrc

修改回默认地址命令：

npm config set registry https://registry.npmjs.org

记得改回国内源：

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

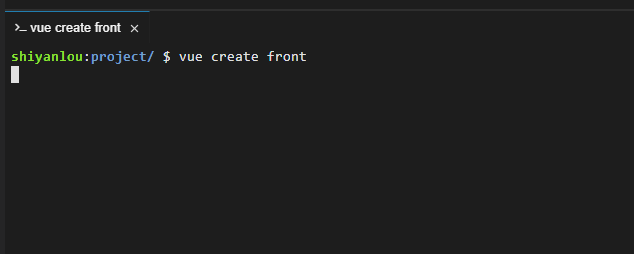
在 npm install 或者 yarn add 命令的下载包速度很慢时，请记得检查这一项。一般情况下，我们都会修改为国内的镜像源。

除了这种方式，可以使用优先级更高的命令行参数的方式临时修改源，只在单次命令有效，即在使用 npm i 安装命令时增加 --registry=https://registry.npm.taobao.org 参数，临时指定下载地址，如：

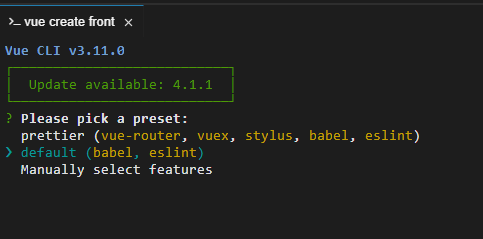
npm install axios --registry=https://registry.npm.taobao.org

## Vue 脚手架初始化前端项目

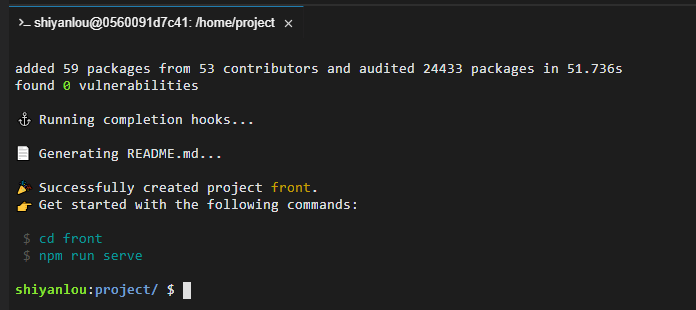
接下来，我们初始化项目。命令行进入 project 目录，在终端键入命令：vue create front 来以脚手架初始化 Vue.js 前端项目。



用上下箭头选择到 default 选项，回车确认。



由于要下载依赖，所以需要等待 3-4 分钟左右。如果你购买了实验楼会员，请保存一下实验环境，以免后续还要重新下载安装依赖。



在 front 目录下新建文件：vue.config.js。文件中输入如下内容并保存：

module.exports = {

publicPath: '',

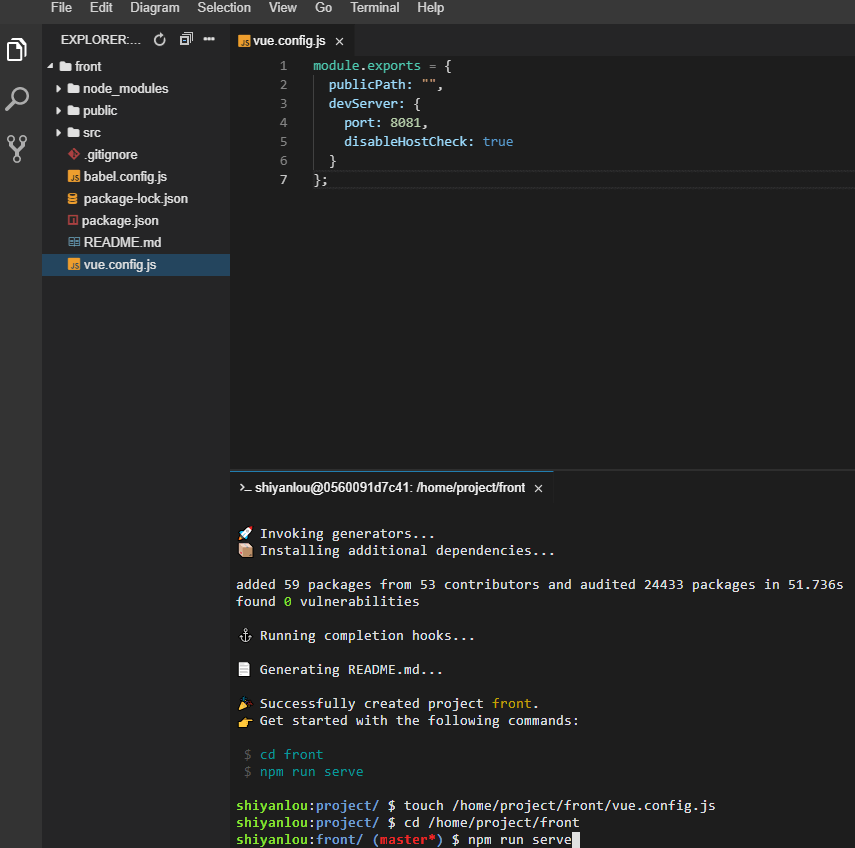
devServer: {

port: 8081,

disableHostCheck: true,

},

};



然后，在终端输入下面两行命令：

cd /home/project/front

npm run serve

即可在 8081 端口开启 Vue 调试服务器。

下面简单介绍一下这些命令和操作的作用。

vue create front 为脚手架的新建项目命令，会在当前目录下新建 front 目录（/home/project/font)，在其中加入必要的初始模板文件，并安装依赖包。

Vue 官方脚手架 Vue CLI（包名 @vue/cli)，v3.0 和 v4.0 的使用方法基本一致。

通过脚手架可以方便的开发单页面应用 (SPA)，并引入各种生态链工具。这里我们选择了默认的最简配置搭框架。

vue.config.js 为脚手架配置文件。也可以在其中配置 webpack。我们加入的几项 publicPath: "" 为使用相对目录调试和部署站点，设定调试服务器端口为 8081，并关闭 Host 检查。这些设定为本课程环境所需项，**请勿修改**。

npm run serve 为 Vue CLI 脚手架已帮我们添加好的调试服务器命令。更多命令可查看 /home/project/front/package.json 中的 scripts.

脚手架最核心是 webpack + Node.js，打包各种资源，并开启 http 调试服务器。由于涉及到的东西偏多，我们在后续章节展开，本节简要了解一下概念和大致作用就好。

## 开启 Node.js 后端服务器

开启新终端窗口（开启方式见上一节实验），并键入如下命令：

cd /home/project && mkdir back && cd back && touch index.js

这里实际上用了一个 Linux 小技巧，四行命令使用 && 连接成了一行，前一命令执行完毕后会继续执行下一命令。

接下来，我们在新建的 /home/project/back/index.js 中加入下面的内容并保存。

var http = require('http'); //加载内置 http 模块，无需安装

http

.createServer(function (request, response) {

response.writeHead(200, { 'Content-Type': 'text/plain' });

response.end('Node Server is OK. \n'); // 发送数据

})

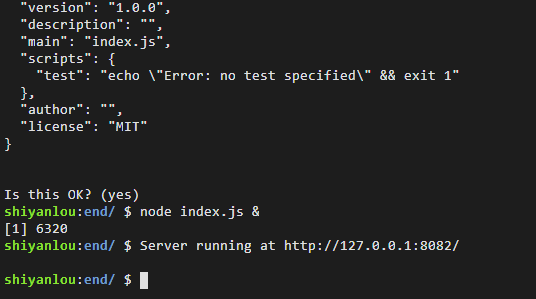
.listen(8082);

// 终端打印如下信息

console.log('Server running at http://127.0.0.1:8082/');

执行命令 npm init，一路回车使用默认值。从提供的选项可以看出，实际上这个命令是在新建 package.json 文件并填入相关的项目信息。之后可以自行尝试下修改输入值，然后检查 package.json 文件内容变化。

继续执行命令 node index.js &，启动后端服务器并挂载在后台运行。这里又给大家提到了一个 Linux 中后台运行的小技巧，如果没有末尾的 & 符号则会前台运行，影响在本终端输入其他命令。



本部分新建了一个 js 文件，其中引用了内置的 http 模块，在本机 8082 端口开启了 http 服务器，并发送了 Node Server is OK. 作为响应返回文本。通过 node index.js 这种方式使用 Node.js 执行 js 文件。

## 启动 mongo

开启新终端命令行窗口输入命令：

cd /home/project

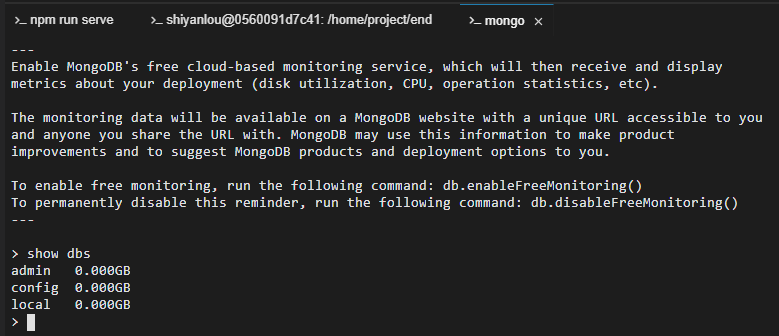
mkdir db

mongod --dbpath /home/project/db &

本命令新建了 db 数据文件夹，并后台启动 mongoDB 服务器。

再敲一次回车激活输入光标，然后在键入命令 mongo 启动命令行工具，启动正常的话输入框左侧变为 >，即已进入数据库交互命令工具。

在此状态下键入命令 show dbs 查看已有数据库列表，如下图。记得万能退出快捷键 Ctrl+C 。



MongoDB 为 NoSQL 数据库，可以简单的把它想象成一个 JSON 数据库，天然和 JavaScript 无缝对接。

## 修改 Nginx 配置

接下来，我们还需要修改 Nginx 配置。在终端键入命令新建配置文件：

touch /home/project/vue.conf

写入以下内容：

server {

listen 8080;

location / {

proxy\_pass http://127.0.0.1:8081/;

}

location /node/ {

proxy\_pass http://127.0.0.1:8082/;

}

}

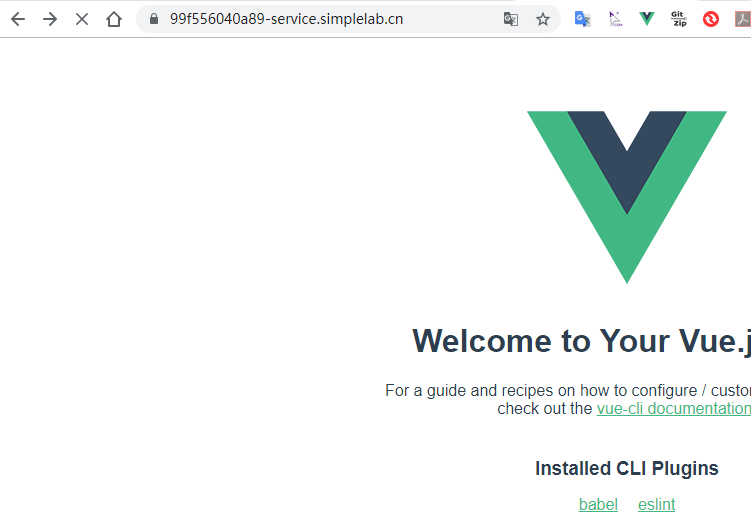
键入下面两行命令，拷贝配置文件到 Nginx 配置目录，开启 Nginx 服务器。

sudo cp /home/project/vue.conf /etc/nginx/conf.d/vue.conf

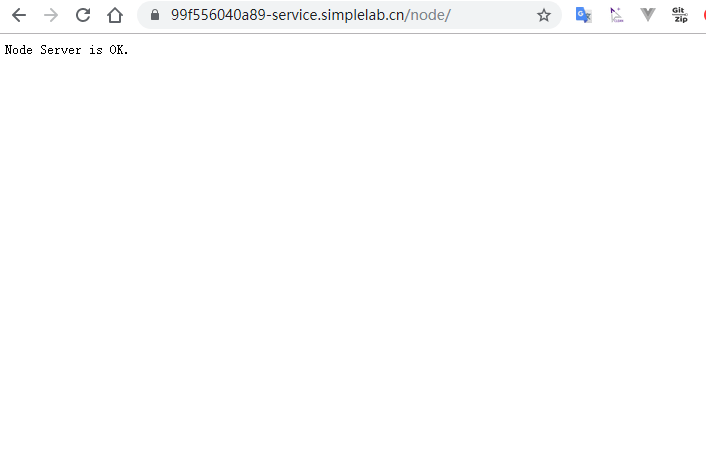
sudo nginx

启动 Nginx 之后如果修改了配置文件，需要键入命令 *sudo nginx -s reload* 重新加载配置。

现在点击实验环境右上角 Web 服务按钮，即可在浏览器新弹出的标签页中看到前端页面。



浏览器地址中加入 /node/ 后缀，即可访问到后端服务器。



Nginx 为轻量又强大的 http 反向代理服务器。由于实验楼环境只允许 8080 端口对外访问，所以我们修改配置使用 Nginx 将根路径代理到 8081 前端， /node/ 路径代理到后端服务器。实际上在本地环境中直接访问相应端口即可，无需这么麻烦。不过，这里又学习了新的知识点。

# Mongod数据库与数据模型

## MongoDB 命令行

上个实验已经给出了新建环境后启动 MongoDB 服务器的命令：

mkdir /home/project/db # 创建数据目录

mongod --dbpath /home/project/db & # 启动数据库服务器并挂载到后台

如果你使用的是保存后的环境，已经拥有 db 目录了，则无需再创建数据目录，可跳过第一句命令，直接启动服务器。如果服务器也已经启动了

然后最好开启一个新的终端，以免提示信息混杂在一起，然后输入命令启动命令行工具：

mongo

现在我们进入到 MongoDB 命令行工具中了，先敲一个 2+2 回车试试。

可以看到返回了计算结果 4。命令工具能输入表达式并计算结果，字符串连接和逻辑运算也可以。

尝试输入 "node"+".js"，注意不要漏输英文半角引号，然后回车，返回 "node.js"。

继续输入 true && false 回车，返回 false。

还可以用 JavaScript 语法定义变量和函数。逐行输入下列代码：

function plus(a) {

return a + 10;

}

plus(2);

返回了调用函数的计算结果：12。

## 数据库基础命令

简单的了解下命令行工具之后，我们开始输入一组 MongoDB 基本操作命令来查看一下各种命令的作用。

显示所有数据库命令：

show dbs

可以看到前一列是数据库名，后面一列是该库已经占用的存储空间，以 GB 为单位。系统初始化后已经有 3 个内置数据库，用户还可以根据需求创建更多的数据库。

查看当前激活的数据库命令：

db

很多数据库命令都是基于当前激活的数据库，所以操作之前使用 db 先查看一下是个很好的习惯。

激活数据库：

use demo

激活名为 demo 的数据库作为当前数据库。当然我们还没有 demo 数据库，所以现在这个命令几乎相当于新建数据库。只有添加了数据以后，才能使用 show dbs 看到新数据库 demo。

查看数据库中所包含的表名：

show tables

无返回，现在库中没有任何表。

为数据库添加一条数据：

db.users.insertOne({name:"Jack",sex:"male"})

这里的 db 其实指代了当前激活的数据库，也就是 demo。点后面的 users 是表名，insertOne() 为插入一条数据的命令。数据是 {name:"Jack",sex:"male"} 这个对象，会作为表中的一条记录保存。保存成功后，再次尝试 show dbs 和 show tables，现在有我们期待的结果了。数据库 demo 和表 users 都已经新建成功。

查询表中的全部数据：

db.users.find()

结果只有一条，因为我们只插入了一条数据。那就再来一条，输入 db.users.insertOne({name:"Pony",sex:"male"}) 之后，再试试 db.users.find()。两条都能看到了。

带上参数去查询表：

db.users.find({name:"Pony"})

只有满足限定条件的那一条数据返回了。

这里大家应该注意到了，虽然插入数据时没有添加 \_id，MongoDB 数据库自动添加了 ObjectID 类型的 \_id。这是数据表的主键，用来唯一确定和区分不同数据条目，所以不能重复。

我们可以尝试自己生成 \_id，并手动指定。

定义变量 testOID 并赋值为随机生成的的 ObjectId 值：

var testOID = ObjectId()

查看一下 testOID 的值：

testOID

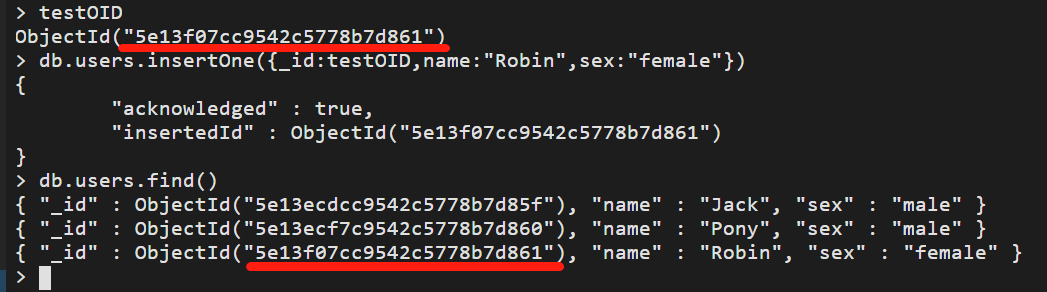
使用这个变量再插入一条数据：

db.users.insertOne({\_id:testOID,name:"Robin",sex:"female"})

查看结果：

db.users.find()

可以看到自行生成的 \_id 值已经作为主键加入新数据。



MongoDB 概念中的集合相当于表，文档相当于一条记录，只不过是一条可以嵌套的 JSON 记录。

## 更新命令

新增和查询都会了，再来试试更新命令：

db.users.updateOne({name:"Robin"},{$set:{sex:"male"}})

继续查看 db.users.find()，数据已发生变化。

前面的 find 和 insert 命令比较直观。find 传入的是过滤条件对象，insert 传入的是要新增的新对象。到了 update 就要传入用逗号分隔的两个参数，第一个大括号为筛选条件，第二个为带着操作符和数据的对象。

更新这里出现新的操作符关键字 $set，也就是修改值。用 sex 的新值 male 替换了原有值 female。除了 $set 之外还有些其他的操作符关键字，后面用到时还会再讲。

然后再看看删除操作：

db.users.deleteOne({ name: 'Robin' });

删除完了再查看一下 db.users.find()，该条数据已消失。

到这里应该已经有同学注意到了，我们的增改删命令都加了 One 作为修饰，也就是单条数据操作的意思。就算条件匹配了多条，也只会对第一条数据执行操作。那多条数据同时操作呢？要用inserteMany, updateMany, deleteMany，这样符合条件的多条数据都会进行操作。只有 find 特殊，单条查询命令使用 findOne，多条使用 find 而不加 Many。

以前版本的写法是 update，其中可填加参数如 {multi:true} 来允许多条操作。虽然目前这种写法仍然可以用，但是官方已经不推荐。我们自己写的函数也应该是这样的思路，编写一个需要传入标记参数去区分两种不同情况的函数，不如直接编写两个名称和作用更清晰的函数，并将内部的公共部分提炼成另外的无状态通用函数，这样可以做到好读、好改、好用，对自己和其他代码阅读者都有好处。

下面来试试多条数据操作吧。每条命令后都执行一次 db.users.find() 来查看一下操作结果。

插入多条数据，注意这里的数据项和值都有所不同：

db.users.insertMany([{name:"Robin",sex:"male",info:{age:18,height:180}},{name:"Richard",info:{age:19,height:190}},{name:"Gil",sex:"female",info:{age:20,height:200}}])

筛选多条数据，但是只修改其中第一条：

db.users.updateOne({},{$set:{sex:"female"}})

然后执行多条修改：

db.users.updateMany({},{$set:{sex:"male"}})

我的返回值是 { "acknowledged" : true, "matchedCount" : 5, "modifiedCount" : 3 }，匹配到 5 条，修改了其中的 3 条。因为这里匹配了所有了数据，但是跳过了修改后无变化的值，只操作需要修改和新增的那些条目。

然后再次修改：

db.users.updateMany({},{$set:{sex:"female"}})

这次是全量修改了。

再试试修改子文档中的值：

db.users.updateOne({name:"Robin"},{$set:{"info.age":22}})

嵌套子文档的深入修改可以直接用 "info.age" 这种方式去取，只是注意需要加上引号。

查询时可以加后缀 .pretty()，如可以尝试命令：

db.users.find().pretty()

可以让 JSON 格式化展开，更加美观易读。

还有就是查询的结果如果超过 20 条，就会自动分段展示，需要在命令行输入 it 继续展示后面内容。

最后再说一个大杀器，删表命令 db.users.drop()。想好了再执行。

现在增查改删的基本操作都会了，请在实验环境下自由练习和熟悉。避免实战时出错，造成从删库到跑路的后果

## Mongoose 库

了解了 MongoDB 命令行的基本操作后，我们接下来介绍一下 Mongoose。Mongoose 是一个基于 Node.js 的 ODM/ORM 类库，封装了 MongoDB 操作，拥有内置类型转换、验证、查询构建、业务逻辑钩子等功能，可以更方便简洁的操作数据库。

介绍起来挺复杂，不如直接从代码开始吧。

### 安装和连接数据库

在实验 2 中配置好的 /home/project/back 目录中安装 Mongoose。如果还没有 back 目录，请参照实验 2 中开启 Node.js 后端服务器的步骤配置完毕再继续。

然后检查是否按照本实验开头的提供方式开启了 MongoDB 服务器，然后命令行运行：

cd /home/project/back

npm i mongoose

touch db.js

在新建的 db.js 中写入下面的代码，保存：

var mongoose = require('mongoose');

var testDB = 'mongodb://localhost:27017/test';

mongoose.connect(

testDB,

{ useNewUrlParser: true, useUnifiedTopology: true },

function (err) {

if (err) {

console.log('MongoDB connect fail.');

} else {

console.log('MongoDB connect success.');

}

}

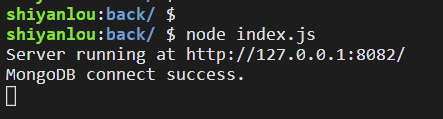
);

module.exports = mongoose;

这里的 mongodb://localhost:27017/ 即为本地 MogonDB 服务器的连接地址和端口，test 为要连接的数据库名，注意这里不是我们之前测试命令的 demo 数据库了。在回调函数中如果连接成功就会打印提示 MongoDB connect success.。

然后修改同级目录下的 index.js，在最上方插入一行 var db = require("./db");

现在可以运行试试了，命令行输入 node index.js。



数据库连接成功。

#### 数据库连接失败分析

如果已经有后台服务器运行在 8082 端口，这里会报错端口已被占用。

Server running at http://127.0.0.1:8082/

events.js:174

throw er; // Unhandled 'error' event

^

Error: listen EADDRINUSE: address already in use :::8082

...

记得结束掉之前开启的服务器进程以后再尝试。这里分为几种情况：

* 前台运行 到相应的终端用 Ctrl+C 结束进程。
* 后台运行 在之前执行的命令的窗口输入 jobs 命令，可查看后台进程序号如 [1] + running node index.js 中 [1] 为序号，使用命令 kill %1 结束该序号进程。或者输入 fg 命令将后台任务放入前台，然后再用 Ctrl+C 结束。
* 异常运行 由于环境异常退出等问题可能碰到，即报错时上面两种情况都不存在。先执行 ps -ef |grep node 查找行末尾为 index.js 语句所对应进程号如 1006，然后输入结束进程命令：kill -9 1006。

第二个可能出现的错误是 MongoDB connect fail.。那么就要检查 MongoDB 服务器是否启动正常，以及服务器地址或代码的输入是否正确。

### 建立模型

现在试试按照 Mongoose 的结构建立 Schema 和 Model 。在 back 目录下新建 model.js，写入如下内容：

let db = require('./db'); //引入 Mongoose

let Schema = db.Schema;

let UserSchema = new Schema({

name: String,

info: {

age: Number,

height: Number,

},

});

let Users = db.model('User', UserSchema);

module.exports = { Users };

上面这段代码新建了 UserSchema。Schema 的概念类似表结构的映射。然后使用内建方法 model()，由 UserSchema 定义出模型 Users。最后导出模型 Users 供其他程序调用。

我们再修改 index.js 测试一下。把前一步骤新加的这行代码 var db = require("./db"); 删掉，再添加一行对 model.js 的引用，同时在服务器回调方法中添加新增一条数据的代码。

var http = require('http'); //加载内置 http 模块，无需安装

let { Users } = require('./model'); //引入 model.js

http

.createServer(function (request, response) {

var newUser = new Users({

name: 'Steven',

info: { age: 30, height: 170 },

});

newUser.save(function (err, res) {

console.log('保存结束');

});

response.writeHead(200, { 'Content-Type': 'text/plain' });

response.end('Node Server is OK. \n');

})

.listen(8082);

console.log('Server running at http://127.0.0.1:8082/');

这段代码中的 save() 方法就是保存数据到数据库，其中传入的是回调函数，数据保存操作后就会执行。回调函数中可以拿到报错信息 err 或者执行结果 res。

现在我们重启 Node.js 服务器，再访问后台页面地址时，会触发数据新增。这里可以参照实验 2 以 Web 服务器的方式访问，或者直接在终端输入命令：

curl http://127.0.0.1:8082

然后使用 Mongo shell 去查看 test 数据库 users 表内容，具体怎么查看呢？作为一个小作业留给大家自己解决，检查一下自己对本实验前半部分命令行知识掌握的程度，记得先激活。

下一节我们开始介绍 Koa 框架，并在其中实际结合 Mongoose 的操作。

# Koa 和中间件

## 实验介绍

本实验介绍了 Koa 框架的基本结构，如何安装和使用。然后带大家理解中间件原理，重点是洋葱模型的逻辑顺序关系。之后通过编写中间件案例来掌握中间件写法和用法。最后介绍了组合中间件的方法和一些常用的中间件。

#### 知识点

* Koa 安装使用
* Koa 中间件逻辑
* Koa 中间件编写
* Koa 中间件应用

## 安装启动 Koa

首先恢复环境和代码。

如果保存了上节的实验环境，可以跳过恢复环境步骤直接继续。

定位到默认目录：

cd /home/project

下载代码压缩包命令：

curl -O https://labfile.oss.aliyuncs.com/courses/1505/code03.zip

解压命令：

unzip code03.zip

如果遇到：

replace back/index.js? [y]es, [n]o, [A]ll, [N]one, [r]ename:

输入 A 选择全部替换。

快速恢复脚本命令（注意命令开头是点-空格-点-斜杠，第一点表示执行，后面的点杠为当前目录）：

. ./tools/start.sh

环境恢复完成。脚本中的主要操作就是按照实验 2 的方式开启必要的环境，并恢复至截止到前一节实验的代码。

进入 back 目录下安装 Koa。

cd /home/project/back

npm i koa

将 index.js 修改为如下内容：

const Koa = require('koa');

const app = new Koa();

app.use(async (ctx) => {

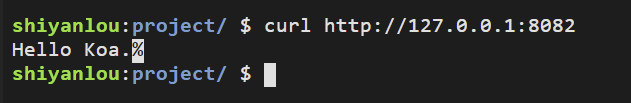
ctx.body = 'Hello Koa.';

});

app.listen(8082);

console.log('Server running at http://127.0.0.1:8082/');

重启后端服务器，可以测试是否能正常访问了。



这里我们先概括一下 **Koa 后端服务器的基本结构**：通过引入 Koa 库，然后初始化为 app 实例，通过一个或若干个连续的 app.use() 传入中间件，然后监听 8082 端口，之后还可以增加一些事件监听器。

Koa 的内核和逻辑结构非常精简。核心之外的功能都是加入各种中间件去解决。下面我们要重点介绍中间件的逻辑和用法。

## 中间件介绍

我们刚写过的这几行代码就是一个最简单的中间件：

async (ctx) => {

ctx.body = 'Hello Koa.';

};

这是一个带着 async 前缀的用箭头函数定义的函数。

为了看懂代码，这里再补充两个知识点。

1. 箭头函数

箭头函数 => 是 ES6 新特性，上面写法相当于：

async function (ctx){

ctx.body = "Hello Koa.";

}

唯一的不同点是箭头函数不改变作用域上下文，也就是 this 指向不会变。这里有点超纲暂时先不讲，只要能看懂函数的结构就够了。再额外举个小例子：

x=>x\*2 也是一个箭头函数，等于 (x)=>x\*2。这样的写法省略了大括号和 return 关键字，相当于

function (x)

{ return x\*2 }

这样的写法。

1. async/await

async/await 是 ES7 新特性，await 关键字可以阻塞异步操作，还可以获取 Promise 的返回值，但是必须在拥有 async 标记的异步函数内部才能用 await。简单的说这就是一种可以把回调嵌套的写法改为顺序执行的写法。我们在这里加了 async，就标记了这个函数为异步函数。

不用急于把提到的这些语法特性全搞明白，等用到的时候我们再进一步解释和尝试。这些新的东西都是为了解决特定问题或者偷特定的懒而产生的，等碰到这些问题的时候自然会明白这些新特性的好处到底在哪。

现在看明白了上面的中间件是函数，只执行一句代码，给 ctx.body 赋了值，这又是在干什么呢？等把下页看完就明白了。

## 中间件逻辑

再简单的说一下 Koa 的处理逻辑。Koa 定义了一个全局的 ctx（上下文 context）来处理各个请求，其中封装了 Node.js 的 request 和 response 对象以及其他方法和对象，收到一个请求后，所有中间件按顺序逐个处理 ctx，按栈返回。顺序是个重点，后面会举个实例来解释清楚。

这里先认识中间件的基本结构。标准的中间件应该长成下面的样子：

async (ctx, next) => {

doSomethingBefore();

await next();

doSomethingAfter();

};

留意下这个奇怪的 next()，我们写一个实例来看看如何工作。修改 index.js，写入如下代码：

const Koa = require('koa');

const app = new Koa();

let middle1 = async (ctx, next) => {

if (ctx.path == '/favicon.ico') return;

console.log('1 before');

await next();

console.log('1 after');

ctx.body = 'Hello Koa Middleware.';

};

let middle2 = async (ctx, next) => {

console.log('2 before');

await next();

console.log('2 after');

};

let middle3 = async (ctx, next) => {

console.log('3 before');

await next();

console.log('3 after');

};

app.use(middle1);

app.use(middle2);

app.use(middle3);

app.listen(8082);

console.log('Server running at http://127.0.0.1:8082/');

重启后端服务器并访问，可以看到命令行打印的返回值顺序是：

1 before

2 before

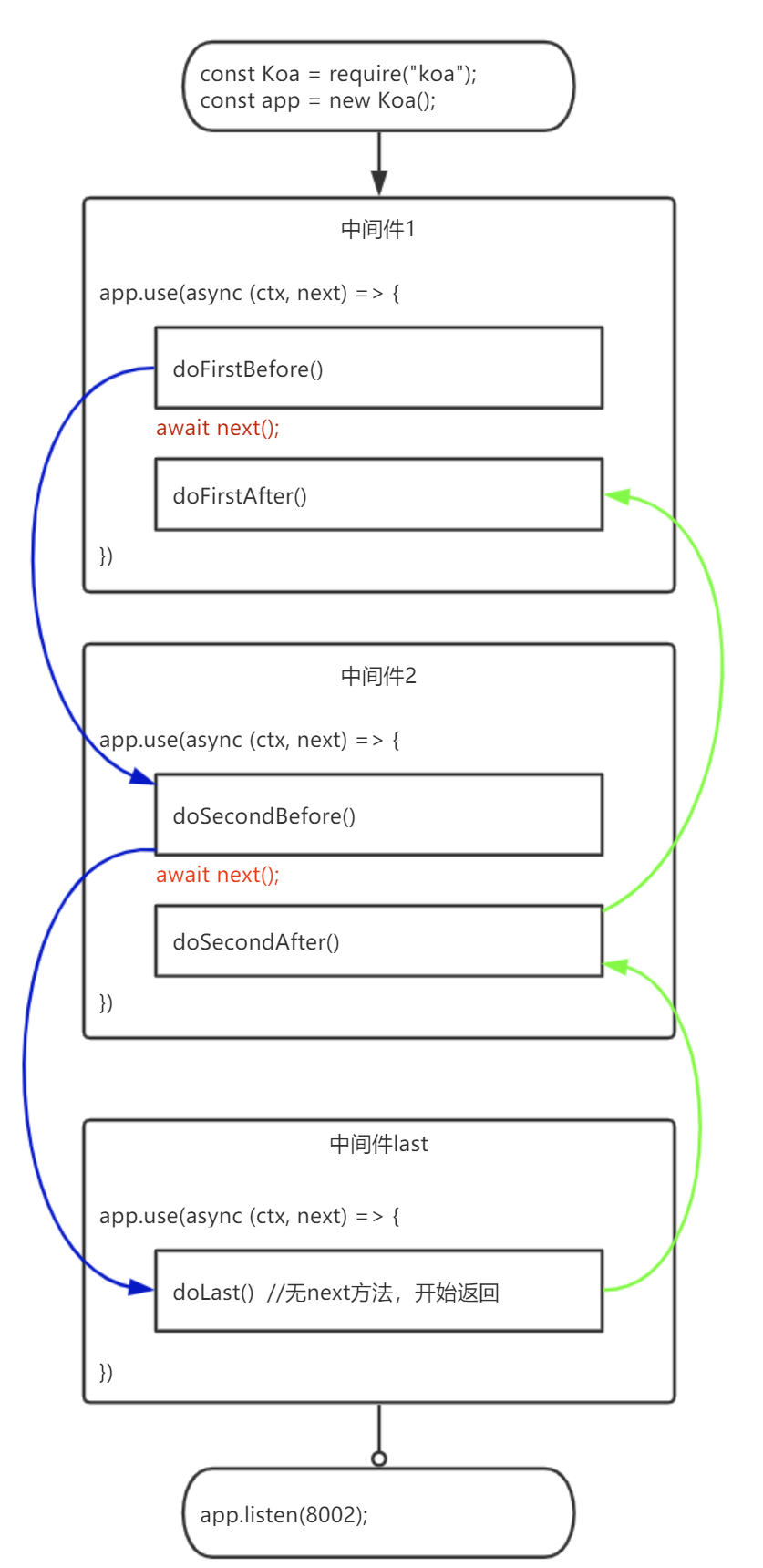
3 before

3 after

2 after

1 after

也就是，请求进来的时候是顺序执行。根据 app.use() 方法调用的前后顺序，执行到 next() 时就跳往下一个中间件，等待执行到没有 next() 或者最后一个中间件执行完毕，开始倒着返回执行 next() 后面的部分。也就是栈或者洋葱模型的意思。画一张图辅助大家理解：



请求进来先走蓝色线，执行中间件前半截代码，截止到 await next() 语句就跳往下一个中间件，一直到底。然后后半截代码按照绿色线反向执行。

示例中的中间件 1 有两句额外代码，一个 if 语句是禁用掉网站图标 favicon.ico 请求防止混淆理解，另一个 ctx.body 是为了给出返回内容以免页面报 404 找不到内容的错误。

如果上面中间件过程或者顺序没看懂，请回去再重新敲代码测试，并配合图片去理解，务必掌握了再继续后面的实验。

## 中间件实例编写

知道中间件的执行顺序以后，我们就可以看看一些常用的中间件是怎么实际联合起来工作的了。

来试试写几个简单的中间件，感受一下如错误捕获，日志，路由等功能如何去实现。当然这些功能都有设计完善的库可以供我们使用，这里主要是为学习和掌握。

继续修改 index.js 为如下内容：

const Koa = require('koa');

const app = new Koa();

// 全局错误捕获

app.use(async (ctx, next) => {

try {

await next();

console.log('handler 通过');

} catch (err) {

console.log('handler 处理错误');

ctx.response.status = err.statusCode || err.status || 500;

ctx.response.body = {

message: err.message,

};

}

});

// 记录请求地址和时间

app.use(async (ctx, next) => {

const start = Date.now();

console.log(`${ctx.method} ${ctx.url} in at ${start}.`);

await next();

const end = Date.now();

console.log(

`${ctx.method} ${ctx.url} out at ${end}.Time use: ${end - start}`

);

});

// 路由

app.use(async (ctx, next) => {

if (ctx.path == '/login') {

ctx.body = 'login';

} else if (ctx.path == '/signup') {

ctx.body = 'signup';

} else {

await next();

}

});

app.listen(8082);

console.log('Server running at http://127.0.0.1:8082/');

再来测试看看吧。可以尝试任意路由查看报错信息，以及测试已经设定可以正常返回的 login 和 signup 路由：

curl http://127.0.0.1:8082/test

curl http://127.0.0.1:8082/login

第一个中间件错误捕获实际上在请求进来的时候没有做任何处理，但是他作为最后一道出口把门，可以截获中间任何环节丢出的错误。

第二个中间件分别记录了进入和出去的时间，也就记录了整个响应内部的耗时。由于我们示例的代码是直接返回没有做耗时操作，所以几乎都返回 0 毫秒。

第三个中间件是一个简单路由，判断请求的地址是不是命中了 login 或 signup 这两个地址，命中则开始返回，否则直接跳过再去下一个中间件。

看懂这部分以后，已经可以自己写中间件了，剩下的就是熟悉 ctx 中的各种属性和方法了。我们在后面的编写过程中会用到几条常用的属性，更多请自行查阅 [Koa 官方文档](https://koajs.com/)。

## 组合中间件

再介绍一个库，koa-compose，可以将几个中间件方法组合成一个，适合共享和打包。

有个官方的例子正好可以参考。先执行安装命令：

cd /home/project/back

npm i koa-compose

之前强调过，执行终端命令都要注意当前路径。为了避免执行出错，提供的很多命令前都会额外增加一句 cd path 的命令。如果大家已经清楚当前路径是对的，就无需重复 cd 进入了。

在 back 目录下新建 mutimidd.js 文件，写入如下代码：

const compose = require('koa-compose');

async function random(ctx, next) {

if ('/random' == ctx.path) {

ctx.body = Math.floor(Math.random() \* 10);

} else {

await next();

}

}

async function backwards(ctx, next) {

if ('/backwards' == ctx.path) {

ctx.body = 'sdrawkcab';

} else {

await next();

}

}

async function pi(ctx, next) {

if ('/pi' == ctx.path) {

ctx.body = String(Math.PI);

} else {

await next();

}

}

const allMiddleGroup = compose([random, backwards, pi]);

module.exports = allMiddleGroup;

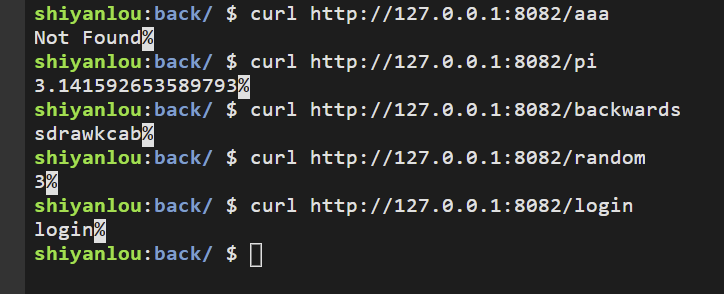
然后在 index.js 的适当位置添加下面两句代码。具体加在哪里呢？靠上面学到的知识试试吧。

let allMiddleGroup = require('./mutimidd');

app.use(allMiddleGroup);

最后请自行测试一下这三个路由地址。

请先开启后台服务器，再开启另一个终端进行测试。



## 常用中间件

我们再来介绍一下 Koa 中常用的中间件。

这里只提供 npm 包名和基本功能介绍，详细的用法还是要看具体的文档。有一些我们后面会用到。包名都有了，安装命令相信大家已经会了。

| **包名** | **用途** |
| --- | --- |
| koa-logger | 日志 |
| koa-compress | 压缩 |
| koa-jwt | jwt 鉴权 |
| koa-session | Session 会话 |
| koa-static | 静态文件请求 |
| koa-bodyparser | 参数处理 |
| @koa/router | 路由 |
| @koa/multer | 文件上传 |

之前提到过中间件的功能可以都做的很小，具体要用到什么功能直接安装引入就可以。源码也比较简单，安装之后在后端目录下的 node\_modules 找到包的名字，查看 index.js 基本就够了。阅读源码是快速学习的一种好方法。

去 [npm 官方网站](https://www.npmjs.com/) 输入包名可以查询到具体信息页，从里面可以找到基本的使用文档、官网和 Github 仓库地址。有助于我们进一步的学习和交流。

# 文章 API 编写

## 实验介绍

本节实验引入 @koa/router 路由中间件，介绍路由的配置和编写方法。

然后进行了 RESTful API 的设计。

之后在具体的 API 地址中编写 Mongoose 增查改删方法，来实现业务效果。

并学习如何访问接口，完成了与文章相关的 Web API 编写工作。

#### 知识点

* 路由中间件
* Mongoose 增查改删
* bodyparser 中间件
* curl 访问接口

## 路由中间件用法

首先恢复环境和代码。

如果保存了上节的实验环境，可以跳过恢复环境步骤直接继续。

定位到默认目录：

cd /home/project

下载代码压缩包命令：

curl -O https://labfile.oss.aliyuncs.com/courses/1505/code04.zip

解压命令：

unzip code04.zip

快速恢复脚本命令（注意命令开头是点-空格-点-斜杠，第一点表示执行，后面的点杠为当前目录）：

. ./tools/start.sh

环境恢复完成。脚本中的主要操作就是按照实验 2 的方式开启必要的环境，并恢复至截止到前一节实验的代码。

我们上节实验已经写过一个简略版的路由。这次我们引入比较完善的路由中间件包 @koa/router 。本包原名 koa-router，后来被官方接管维护，修改了包名。先来安装使用，在 back 目录下运行安装命令：

cd /home/project/back

npm i @koa/router

在 back 目录中新建 router.js ，写入以下演示代码：

const router = require('@koa/router')();

const { Users } = require('./model');

router.prefix('/api');

var getUsers = async function (ctx, next) {

let result = await Users.find();

if (result.length) {

ctx.body = {

status: 1,

msg: 'success',

body: result,

};

} else {

ctx.body = {

status: 0,

msg: 'no user',

body: [],

};

}

};

router.get('/users', getUsers);

module.exports = router;

修改 index.js，添加以下两行代码，引入中间件：

const router = require('./router');

app.use(router.routes(), router.allowedMethods());

输入 node index.js 开启服务器，如果之前已经开启了记得关掉再重启。

然后测试 API 地址。这里给大家提供两种方法：

1. 浏览器访问。

点击右上角 Web 服务开启，在自动弹出的浏览器地址中加上/node/api/users后缀。如我的环境是：

https://027df8d151aa-service.simplelab.cn/node/api/users

1. curl 工具，直接在命令行执行。
2. curl http://127.0.0.1:8080/node/api/users
3. curl http://127.0.0.1:8082/api/users # 或者
4. curl -i -X GET http://127.0.0.1:8082/api/users # 以及详细版

-i 参数可以看到详细的头信息，-X 是执行请求方法，默认是 GET 方法。

在实验 2 曾解释过，因为使用 Nginx 配置了转发，所以本地端口 8080/node/ 也可以访问。

可以看到返回结果，说明路由工作正常，获取到了所有 Users 表中的信息。