Day01:

MintUI-- Unit01

1.组件库基础

1.1 什么是组件库?

组件(Component)是企业为提升开发效率而开发的针对应用的、结构、表现及行为的封装。组件的优点在于一次定义，多次使用。

1.2 组件库的优势

- 提升开发效率
- 提升项目的可维护性
- 便于团队的协作开发

1.3 组件库的分类

- 移动端组件库
  - Mint UI(饿了吗) -- http://mint-ui.github.io/#!/zh-cn

- Vant UI(有赞) -- https://youzan.github.io/vant/#/zh-CN/
- Cube UI(滴滴) -- https://didi.github.io/cube-ui/#/zh-CN

- 桌面端组件库
  - Element UI (饿了吗) -- https://element.eleme.cn/#/zh-CN
  - AT-UI(凹凸实验室) -- https://at-ui.github.io/at-ui/#/zh
  - View UI(视图更新) -- https://www.iviewui.com/

2.Mint UI

2.1 安装

```
npm install --save mint-ui
```

该命令在脚手架的根目录下执行或者说在package.json 文件所在的目录下执行

在命令行中实现安装的操作步骤如下：

A.启动 WINDOWS 操作系统的命令行(WIN+R),然后输入 cmd

B.有可能需要切换盘符，输入 盘符:，如 d:

C.输入 cd 路径

D.输入 `npm install --save mint-ui`

2.2 引入 Mint UI

在 `src/main.js` 中输入以下代码:

```
//导入 Mint UI
import MintUI from 'mint-ui'

//导入 Mint UI 的样式表文件(library,库)
import 'mint-ui/lib/style.min.css'

//通过 Vue.use()方法注册为插件
Vue.use(MintUI)
```

2.3 Mint UI 组件库的组成

- CSS 组件
- JS 组件
- 表单组件

2.4 使用 Mint UI

- Header 组件

Header 组件用于实现顶部导航，其语法结构是：

```
<mt-header title="标题信息" fixed>
    ...
</mt-header>
```

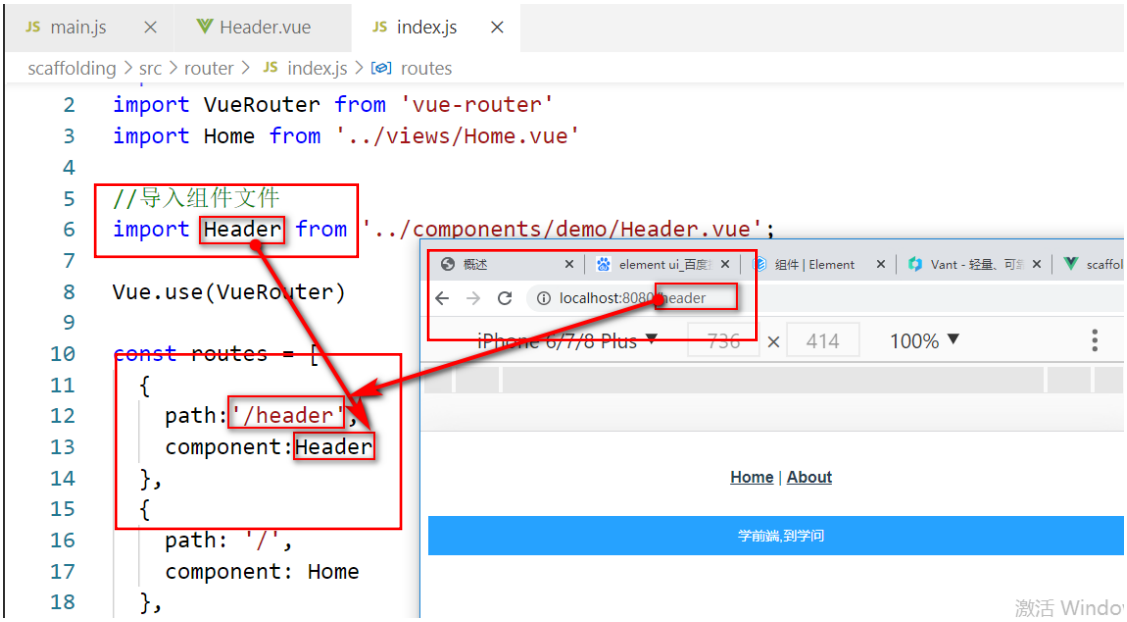
fixed 属性为布尔属性，用于控制顶部导航固定在顶部(不会随页面的滚动而滚动)

在 header 中可以嵌套子元素，可以为子元素添加 slot="left"或 slot="right"属性

示例代码如下：

```
<template>
    <div>
        <mt-header title="学前端,到学问">
        </mt-header>
    </div>
</template>
```

组件文件与路径的对应关系如下图所示：



- **reset.css**

所有的 HTML 元素都有默认样式，而且不同的浏览器的默认样式是不相同的，为了保证所有浏览器的默认样式相同 -- reset.css

操作步骤：

A.将 reset.css 放置在 public/css 目录内

B.编辑 public/index.html，在头部书写以下语句

引入 reset.css 文件

```
<link type="text/css" rel="stylesheet" href="/css/reset.css">
```

- Button 组件

  Button 组件用于实现按钮，其语法结构是：

```
<mt-button
  type="按钮类型"
    size="按钮尺寸"
    icon="按钮图标型"
    disabled
    plain>
    ...
</mt-button>
```

  按钮类型包括：default(默认)、primary(主要的)、 danger(危险的)

  按钮尺寸包括：small(小的)、normal(标准的)、large(大的)

  按钮图标包括：back(返回)、more(更多)

  可以在按钮内嵌套图像，并且为图像添加 slot="icon"属性，此时该图像将作为按钮的图标来呈现（优先级高于 icon 属性）

图标尺寸一般为：24X24、32X32、48X48、64X64

plain 属性为布尔属性，表示按钮是否为镂空按钮

布尔类型的属性是单个写的　　不要写 true/false 写上就代表为”true”

- Field 组件

Field 组件用于实现表单控件，其语法结构是：

```
<mt-field
  type="表单控件的类型"
    label="标签"
    placeholder="占位符"
    state="检测状态"
    disableClear
    :attr="原生属性"
    readonly
    disabled
    v-model="变量名称">
</mt-field>
```

表单控件的类型包括：

- text，单行文本框
- password，密码框
- textarea，多行文本框
- number，数字
- url,URL 地址，如网址

state 属性用于标识表单控件的检测状态，其值为 success(成功)、warning(警告)、error(错误)

disableClear 属性为布尔属性，表示是否禁用 clear 按钮

:attr 为组件的原生属性，对象类型，形如：

```
<mt-field
  type="text"
    :attr="{maxlength:'10',autocomplete:'off'}">
</mt-field>
```

readonly 属性为布尔属性，表示当前表单控件是否为只读

disabled 属性为布尔属性，表示当前表单控件是否禁用

示例代码如下：

```
<template>
    <div>
        <mt-field
                type="text"
                label="用户名"
                state="success"
                placeholder="请输入用户名">
        </mt-field>
        <mt-field
                type="password"
                label="密码"
                state="warning"
                disableClear
                placeholder="请输入密码">
        </mt-field>
        <mt-field
```

```
                    type="password"
                    label="确认密码"
                    state="error"
                    placeholder="请再次输入
密码"
                    :attr="{maxlength:'10',
autocomplete:'off'}">
        </mt-field>
    </div>
</template>
```

- **Toast 组件**

  Toast 组件用于显示短消息提示框，其语法结构是：

```
//简捷语法
this.$toast("提示内容")

//标准语法
this.$toast({
    message:"提示内容",
    position:"提示框的位置(top|middle|bo
```

ttom)"

duration:持续时间(单位为毫秒,默认为 3000)

})

- Messagebox 组件

Messagebox 组件用于显示消息提示框，语法结构是:

```
//简捷语法
this.$messagebox("标题信息","提示内容")
```

作业:

1:如何实时控制表单控件的状态

2:如何让表单控件在失去焦点时自动完成检测的业务(百度！百度！百度)

Day02:

MintUI-- Unit02

1.关于 Mint UI 中表单控件获取/失去焦点

```
<mt-field type="text"
    @focus.native.capture="函数方法"
    @blur.native.capture="函数方法">
</mt-field>
```

native、stop、prevent、capture 等属于

\

事件修饰符

在事件处理程序中调用 event.preventDefault() 或 event.stopPropagation() 是非常常见的需求。是：methods 只有纯粹的数据逻辑，而不是去处理 DOM 事件细节。

为了解决这个问题， Vue.js 为 v-on 提供了**事件修饰符**。通过由点(.)表示的指令后缀来调用修饰符。

.stop
.prevent
.capture
.self

.native //在某个组件的根元素上监听一个原生事件。可以使用 v-on 的修饰符 .native

通俗点讲：就是在父组件中给子组件绑定一个原生的事件，就将子组件变成了普通的

可以理解为该修饰符的作用就是把一个 vue 组件转化为一个普通的 HTML 标签，

```
<!-- 阻止单击事件冒泡 -->
<a v-on:click.stop="doThis"></a>
<!-- 提交事件不再重载页面 -->
<form v-on:submit.prevent="onSubmit"></form>
<!-- 修饰符可以串联  -->
<a v-on:click.stop.prevent="doThat"></a>
<!-- 只有修饰符 -->
<form v-on:submit.prevent></form>
<!-- 添加事件侦听器时使用事件捕获模式 -->
<div v-on:click.capture="doThis">...</div>
```

https://cn.vuejs.org/v2/guide/events.html#%E4%BA%8B%E4%BB%B6%E4%BF%AE%E9%A5%B0%E7%AC%A6

2.Mint UI 组件

· **Navbar 组件**

Navbar 组件用于实现顶部选项卡，其语法结构是：

顶部选项卡，与 Tabbar 类似，依赖 tab-item 组件。

```
<mt-navbar v-model="变量名称" fixed>  //fixed 可不写 用于在页面上是否固定当前选项卡
    <mt-tab-item id="当前选项卡的 ID">
        ...
```

```
    </mt-tab-item>
    ...
</mt-navbar>
```

navbar 绑定的变量的值应该为 mt-tab-item 中的 id

mt-tab-item 的 id 只要在当前容器内唯一即可。//每个选项卡子项都有唯一的爹　只要在唯一的爹内不重复使用 id 即可

可在<mt-tab-item>中嵌套图像，并且为图像设置 slot="icon"属性的话，该图像将作为选项卡的图标出现。

API:

navbar

参数	说明
fixed	固定在页面顶部
value	返回当前选中的 tab-item 的 id

tab-item

参数	说明	类型
id	选中后的返回值	*

Slot

navbar

name
-

tab-item

name	描述
-	显示文字
icon	icon 图标

· TabContainer 组件

面板，可切换显示子页面。

TabContainer 组件用于实现面板，其语法结构是：

API

tab-container

参数	说明	类型	可选值	默认值
value	当前激活的 id	*		
swipeable	显示滑动效果	Boolean		false

tab-container-item

参数	说明	类型	可选值	默认值
id	item 的 id	*		

Slot

tab-container

name	描述
-	内容

tab-container-item

name	描述
-	内容

```
<mt-tab-container v-model="变量名称" swipeable>
    <mt-tab-container-item  id="当前面板的ID">...</mt-tab-container-item>
    ...
</mt-tab-container>
```

tab-container 绑定的变量的值应该为 mt-tab-container-item 中的 id

mt-tab-container-item 的 id 只要在当前容器内唯一即可。

swipeable 属性为布尔属性，表示面板是否具备滑动效果。

· Tabbar 组件

API

tabbar

参数	说明	类
fixed	固定在页面底部	Bo
value	返回当前选中的 tab-item 的 id	*

tab-item

参数	说明	类型
id	选中后的返回值	*

Slot

tabbar

name	描述
-	内容

tab-item

name	描述
-	显示文字
icon	icon 图标

Tabbar 组件用于实现底部选项卡，其语法结构是：底部选项卡，点击 tab 会切换显示的页面。依赖 tab-item 组件。

```
<mt-tabbar v-model="变量名称" fixed>
    <mt-tab-item id="当前选项卡的 ID">...</mt-tab-item>
    ...
</mt-tabbar>
```

修改 Mint UI 组件样式

方式 1：直接修改 Mint UI 的样式表文件 -- mint-ui/lib/style.min.css 或 style.css

方式 2：在某个页面组件中重新定义关于该组件样式

方式 3：在重新定义组件样式时，可能会使用到 !important 以提升权重

作业:

1.根据 Field.vue 完成学子问答项目中用户注册（Register.vue）及用户登录(Login.vue)的页面搭建

2.参考知乎的首页的页面结构，完成内容列表的搭建，结构如下图所示

重庆万州公交坠江事故中，红车女司机会获得怎样的赔偿?

今年年初南京这边下大雪，我叔作为公职人员连夜去长江二桥扫雪。他车停在应急车道上时被一辆江北过江公交的首班车结结实实亲了一口，整个车屁股到后座撞得稀烂，车子当场报废，没有修的价值了。所幸气囊全开了，人没事…

:

Day03:

MintUI-- Unit03

1.学子问答的数据表结构

数据库名称 xzqa，编码方式 utf8

1.1 xzqa_category

xzqa_category 数据表用于存储文章的分类，数据表结构如下:

字段名称	数据类型	是否为空	默认值	键	其他	说明

id	SMALLINT UNSIGNED	NO	PRIMARY KEY	AUTO_INCREMENT	文章分类ID
category_name	VARCHAR(30)	NO	UNIQUE KEY		分类名称

1.2 xzqa_author

xzqa_author 数据表用于存储作者的相关信息，数据表结构如下：

字段名称	数据类型	是否为空	默认值	键	其他	说明
id	MEDIUMINT UNSIGNED	NO		PRIMARY KEY	AUTO_INCREMENT	作者ID

username	VARCHAR(30)	NO		UNIQUE KEY	用户名
password	VARCHAR(32)	NO			用户密码，采用MD5
nickname	VARCHAR(30)	YES	NULL		用户昵称
avatar	VARCHAR(50)	NO	unnamed.jpg		用户头像

article_number	MEDIUMINT UNSIGNED	N	0			用户发表的文章数量

1.3 xzqa_article-→本案例要使用的表

xzqa_article 数据表用于存储文章信息，数据表结构如下：

字段名称	数据类型	是否	默认值	键	其他	说明

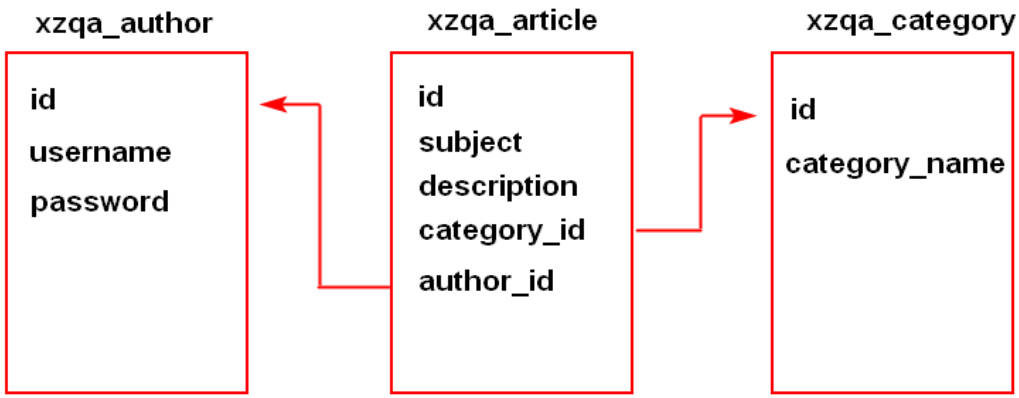
		为空			
id	INT UNSIGNED	NO	PRIMARY KEY	AUTO_INCREMENT	文章ID
subject	VARCHAR(50)	NO			文章标题
description	VARCHAR(255)	NO			文章简介
image	VARCHAR(50)	YES			文章缩略图
content	MEDIUMTEXT	NO			文章

category_id	SMALLINT UNSIGNED	NO	正文文章分类ID，外键，参照分类表中的ID字段

author_id	INT UNSIGNED	NO	作者ID，外键，参照料作者表中的ID

数据表的 ER 图如下：

ER图（Entity-Relationship）



什么是 ER 图?

实体 bai--联系图（entity-relation），用来反映现实世界中实体之间的联系的图形。

E-R 图中包括的元素主要有：实体（矩形框内写上实体名表示）

属性（用短横线连接实体，椭圆内写上属性名表示）

联系（短横线连接不同的实体，在菱形框内写上联系名）

联系的类型（联系连接不同实体的线上标示出来联系的类型）

联系的类型主要有1:1、1:n、m:n 三种类型。

1:1 表示联系两端的实体相互间都是 1:1 的联系，如：

一个学校有一个校长，一个校长在一个学校里任职；则校长和学校之间就是一对一的联系。

1：n 表示联系两端的实体之间是一对多的联系，如：

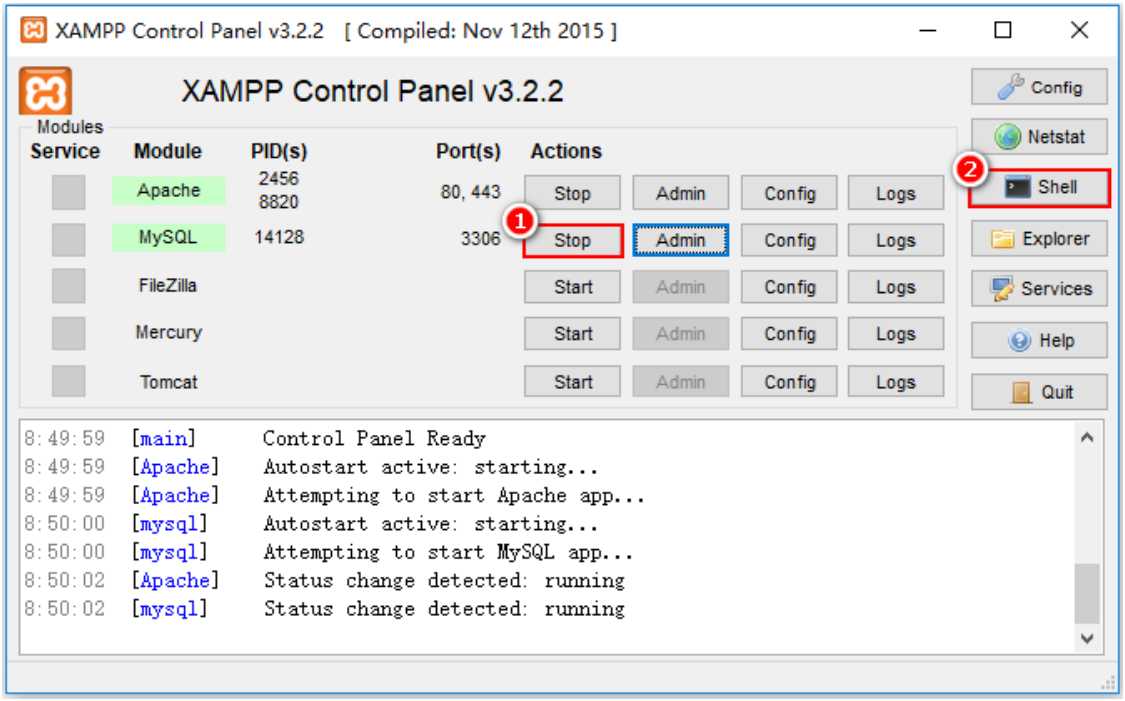
一个班级有很多学生，一个学生只能属于一个班级，则班级和学生实体之间就是一对多的联系；

m:n 表示联系两端的实体之间是多对多的联系，如：

一个学生可以学习很多课程，一门课可以被很多同学学习，则学生和课程之间就是多对多的联系。

1.4 MySQL 数据的导入

A.先启动 XAMPP，再启动 MySQL,最后单击 Shell 按钮



B.在命令提示符下输入以下命令:

`mysql -uroot -p < SQL 脚本文件的位置及名称`

2.学子问答项目的实践……

2.1 顶部选项卡的实现

对于顶部选项卡的数量有两种实现方式:

A. 静态的，指在书写<mt-navbar>时固定好其中包含的<mt-tab-item>的数量及内容

a) 缺点: 无法动态生成选项卡

B. 动态的，指<mt-navbar>中包含的<mt-tab-item>的数量及内容是数据表决定的

在学子问答项目中包含的 xzqa_category 数据表用于存储文章的分类，所以其用作决定顶部选项卡的数量及内容。

· 在什么情况发向 WEB 服务器发送请求以获取分类信息?

涉及 Vue 生命周期的钩子函数：beforeCreate、created、beforeMount、mounted、beforeUpdate、updated、beforeDestroy、destroyed

四个生命周期：创建(create)　挂载(mount)　更新(update)　销毁(destroy)

创建(create)	a. 创建 new Vue()对象，以及 data 对象， b. 暂时没有扫描 DOM 树，　也就没有虚拟
挂载(mount) **必经阶段**	a. 已经有 data 对象——既可操作 data 中 b. 扫描真实 DOM 树，　生成虚拟 DOM 树
更新(update)	只有 new VUe 中的程序修改了 data 中的

销毁(destroy)　　只有手动调用$destroy()函数，销毁当前组件

进入阶段之前触发	beforeCreate(){ ... }
离开创建阶段后触发	created(){ ... }
进入挂载阶段前触发	beforeMount(){ ... }
离开挂载阶段后触发	mounted(){ ... }　　**使用场景最多的
进入更新阶段后触发	beforeUpdate(){ ... }
离开更新阶段后触发	updated(){ ... }
进入销毁阶段前触发	beforeDestroy(){ ... }
离开销毁阶段后触发	destroyed(){ ... }

现在将采用 mounted 钩子函数触发时，发送请求到 WEB 服务器，于是在 Home.vue 中进行如下修改，示例代码如下：

```
<script>
export default{
  mounted(){
      //
  }
```

```
}
</script>
```

现在既然要向 WEB 服务器发送请求，请问通过什么来发送请求呢？-- 通过 axios 发送请求

但是，现在需要进行 axios 的安装与配置才可以使用，所以：

A.安装 Axios

```
npm install --save axios
```

B.配置 -- main.js 文件内

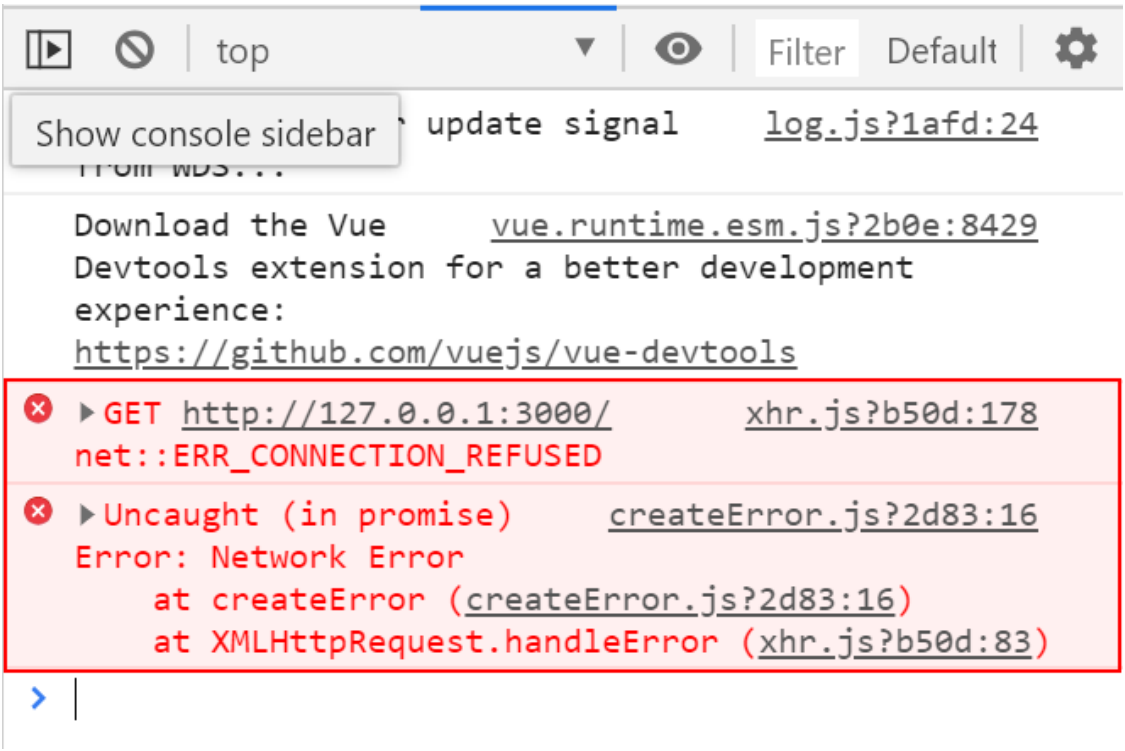
```
//导入 Axios 模块
import axios from 'axios';

//配置 Axios 的基础地址
axios.defaults.baseURL = 'http://127.0.0.1:3000';

//在 Vue 的原型上扩展属性
Vue.prototype.axios = axios;
```

在学子问答项目案例中，Node.js 服务器端口为 3000

经过上述的操作，已经完成了 Axios 的安装与配置，现在可以在 Home.vue 的 mounted 钩子函数中发送请求了，示例代码如下：

```
<script>
export default{
  mounted(){
      this.axios.get('/');
  }
}
</script>
```

此时脚手架的运行结果如下图所示：



产生上述错误的原因是：根本没有做 WEB 服务器

创建 Node 服务器

脚手架在发送请求时，指定协议名称为 HTTP，所以必须在 Node 服务器上安装 express 框架：

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

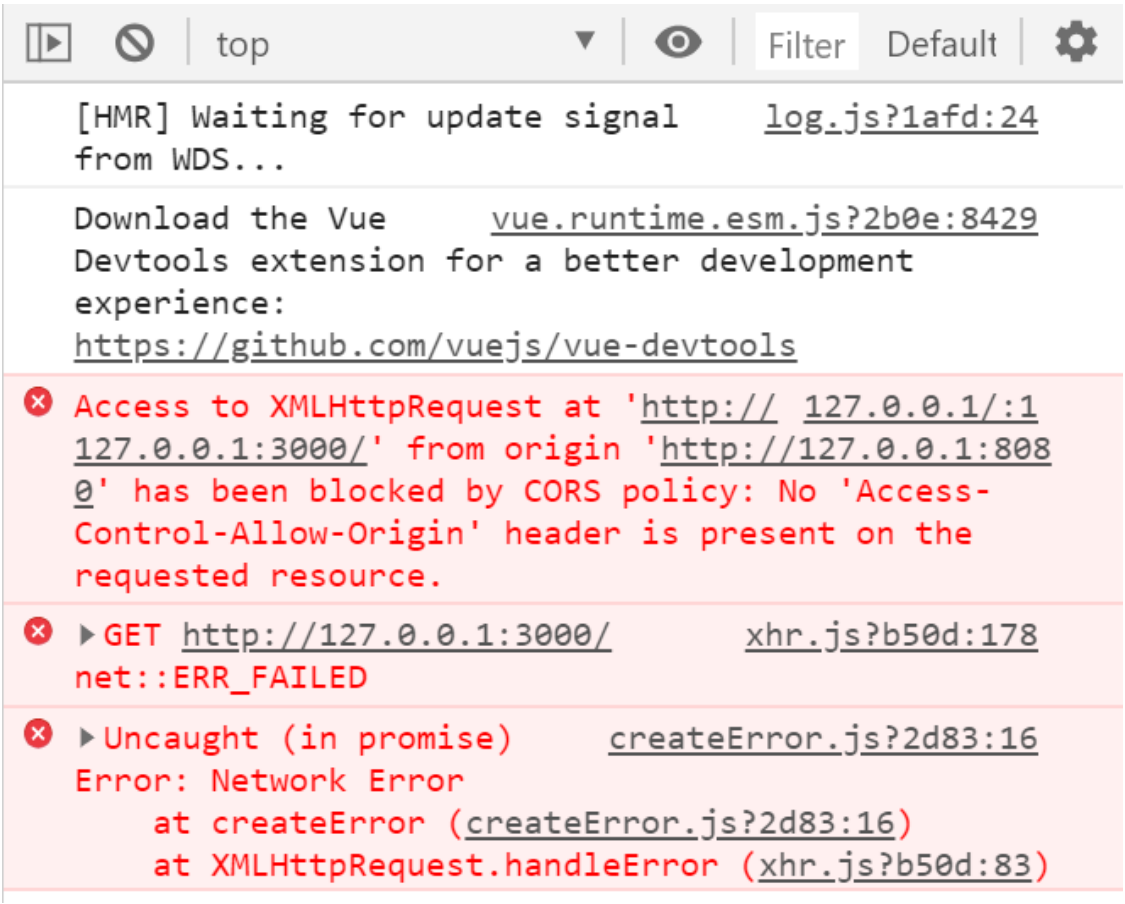
安装成功后，在服务器的根目录下创建 app.js，示例代码如下：

```
//引入 Express 模块
```

```
const express = require('express');

//创建 Express 实例
const server = express();

//指定服务器的监听端口号
server.listen(3000);
```

在 VSCode 编辑器的终端中输入 node app.js

此时脚手架的运行结果如下图所示：



此时出现的错误原因是：脚手架的端口号与服务器端口号不一致而导致的跨域错误！所以：

· 解决 Node 服务器的跨域错误

在 Node 服务器上安装 CORS 模块并且进行配置：

第一步：安装

```
npm install --save cors
```

第二步：配置

```
//引入 Express 模块
const express = require('express');

//引入 CORS 模块
const cors = require('cors');

//创建 Express 实例
const server = express();

//将 CORS 作为 Server 的中间件使用
server.use(cors({
```

```
  origin:['http://127.0.0.1:8080','http://
localhost:8080']
}));

//指定服务器的监听端口号
server.listen(3000);
```

第三步：重新启动 Node 服务器

此时脚手架的运行结果如下：

```
[HMR] Waiting for update signal from WDS...        log.js?1afd:24

Download the Vue Devtools extension for a better development experience:
https://github.com/vuejs/vue-devtools        vue.runtime.esm.js?2b0e:8429

▶ GET http://127.0.0.1:3000/ 404 (Not Found)        xhr.js?b50d:178

▶ Uncaught (in promise) Error: Request failed with status code 404        createError.js?2d83:16
    at createError (createError.js?2d83:16)
    at settle (settle.js?467f:17)
    at XMLHttpRequest.handleLoad (xhr.js?b50d:61)

>
```

现在产生错误的根本原因是：没有对应的 API 地址

· 书写获取全部文章分类信息的 API

规定：获取全部分类信息的 API 地址为 --/category，请求方式为 GET

所以现在必须在 Node 服务器上创建/category 的 API,并且请求方式为 GET，其基本结构如下：

```
server.get('/category',(req,res)=>{

});
```

在当前的/category API 地址中要获取数据库中 xzqa_category 数据表的记录，所以还必须安装 MySQL 模块，同时进行相关的配置：

第一步：安装 MySQL 模块

```
npm install --save mysql
```

第二步：配置 MySQL 模块

```
//引入 MySQL 模块
const mysql = require('mysql');
```

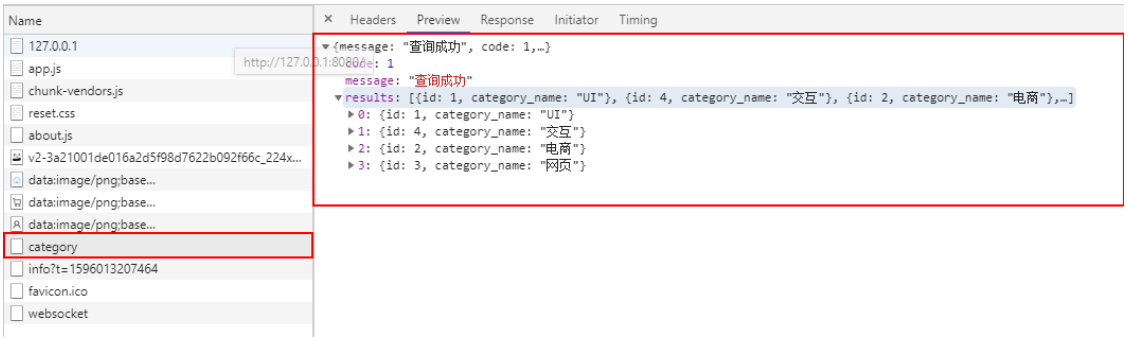
```
//创建MySQL 连接池
const pool = mysql.createPool({
    //数据库服务器地址
    host:'127.0.0.1',
    //数据库用户名
    user:'root',
    //数据库用户密码
    password:'',
    //数据库服务器端口号
    port:3306,
    //数据库名称
    database:'xzqa',
    //编码方式
    charset:'utf8',
    //连接限制
    connectionLimit:15 这个属性默认值就是 15
个
});
```

第三步：在/category 请求 API 中，获取数据表的记录并且返回到客户端(脚手架)，示例代码如下：

```
//获取所有文章分类信息的API
server.get('/category',(req,res)=>{
    //SQL 查询语句
    let sql = 'SELECT id,category_name FRO
M xzqa_category';
    //执行SQL 查询语句
    pool.query(sql,(err,results)=>{
        if(err) throw err;
        //响应到客户端的信息
        res.send({message:'查询成功',code:
1,results:results});
    });
});
```

第四步：重新启动 Node 服务器

此时脚手架的运行结果如下：



现在已经证明服务器返回了客户端期望的结果!!!

但是此时　客户端还没有接收并且在页面组件中显示该结果信息，所以：

· 在客户端接收并显示显示信息

在刚刚发送请求的代码如下：

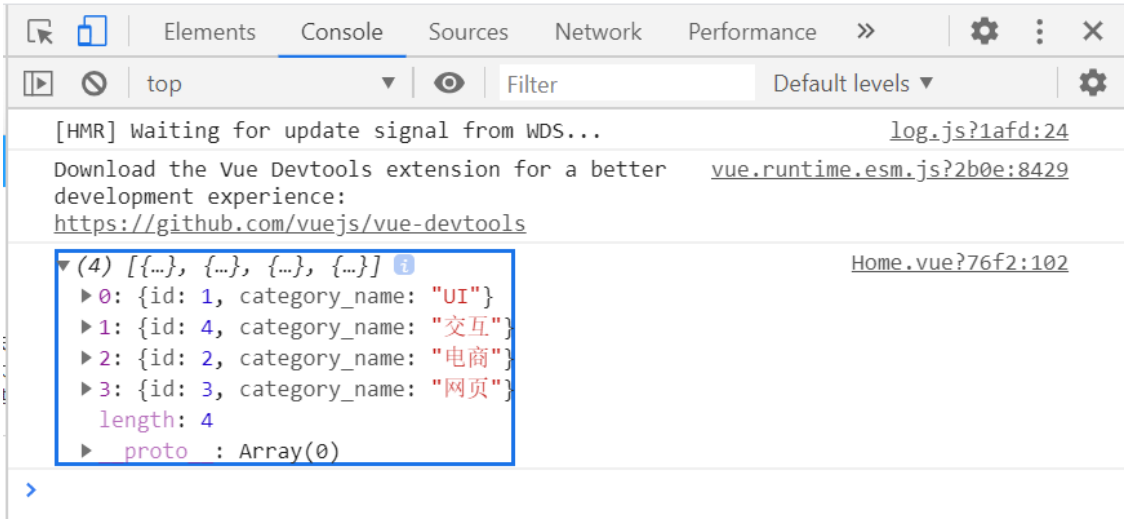
```
mounted(){
    this.axios.get('/category');
}
```

所以现在要接收服务器返回的数据了，示例代码如下：

```
mounted(){
    this.axios.get('/category').then(res=>
```

```
{
        console.log(res.data.results);
    });
}
```

此时脚手架的运行结果如下：



这些数据最终要在页面组件显示的话，必须将其存储到 Vue 的一个变量中，然后再通过 v-for 指令进行循环输出就可以了，所以：

```
data(){
    return {
        //用于存储文章分类信息
        category:[]
```

```
    }
}
```

另外在 mounted 的钩子函数中应该将服务器返回的数据存储到 Vue 变量中，示例代码如下：

```
mounted(){
    this.axios.get('/category').then(res=>
{
        this.category = res.data.results;
    });
}
```

最后通过 v-for 指令来动态决定并生成<mt-tab-item>的数量及内容，示例代码如下：

```
<mt-navbar v-model="active">
    <mt-tab-item
     :id="item.id"
        v-for="(item,index) of category"
        :key="index">
     {{item.category_name}}
```

```
    </mt-tab-item>
</mt-navbar>
```

最后当循环输出之后发现没有任何一个顶部选项卡被选定，根本原因是因为：active 变量原来为字符串类型，而现在为数字类型的，所以：需要将原来的 active 变量的默认值由字符串类型的"1"，改为整数 1 即可。

扩展一步：

面板的数量与顶部选项卡的数量是一致的，所以也应由 category 变量动态来决定面板的数量，故：

```
<mt-tab-container v-model="active">
    <mt-tab-container-item
            :id="item.id"
            v-for="(item,index) of category"

            :key="index">
        <!-- 单一文章信息开始 -->
        <div class="InfoItem">
            ...
        </div>
```

```
<!-- 单一文章信息结束 -->
    </mt-tab-container-item>
</mt-tab-container>
```

作业:

A.推倒重写

B.文章列表的业务功能

Day04:

MintUI-- Unit04

1.Mint UI 组件库

· v-lazy 指令

v-lazy 指令用于实现图像的懒加载，其语法结构是:

```
<img v-lazy="图像的 URL 地址">
```

```
<!-- 图片列表区域 -->
    <ul class="photo-list">
      <router-link v-for="item in list" :key="item.id" :to="'/home/photoinfo/' + item.id" tag="li">
        <img v-lazy="item.img_url">
        <div class="info">
          <h1 class="info-title">{{ item.title }}</h1>
          <div class="info-body">{{ item.zhaiyao }}</div>
        </div>
      </router-link>
    </ul>
```

拓展：

若列表不在 window 上滚动，则需要将被滚动元素的 id 属性以修饰符的形式传递给 v-lazy 指令

```
<div id="container">
  <ul>
    <li v-for="item in list">
      <img v-lazy.container="item">
    </li>
  </ul>
</div>
```

· Infinite scroll 指令

Infinite scroll 指令用于实现无限滚动，其语法结构：

定义：为 HTML 元素添加 v-infinite-scroll 指令即可使用无限滚动。滚动该元素，当其底部与被滚动元素底部的距离小于给定的阈值（通过 infinite-scroll-distance 设置）时，绑定到 v-infinite-scroll 指令的方法就会被触发。

```
<HTML 元素
      infinite-scroll-distance="阈值"
      v-infinite-scroll="方法名称"
      infinite-scroll-disabled="变量名称"
      infinite-scroll-immediate-check="true
">
</HTML 元素>
```

`infinite-scroll-distance` 属性指在滚动到距离容器底部还有多少个像素时触发无限滚动指令

`v-infinite-scroll` 属性指当触发无限滚动指令时需要执行的函数名称

`infinite-scroll-disabled` 属性指在执行滚动函数时，既使再滚动到指定范围内是否还要继续触发滚动函数。

API

参数	说明
infinite-scroll-disabled	若为真，则无限滚动不会被触发
infinite-scroll-distance	触发加载方法的滚动距离阈值（像素）
infinite-scroll-immediate-check	若为真，则指令被绑定到元素上后会立即检查是态下内容有可能撑不满容器时十分有用。
infinite-scroll-listen-for-event	一个 event，被执行时会立即检查是否需要执行

2.学子问答项目实践

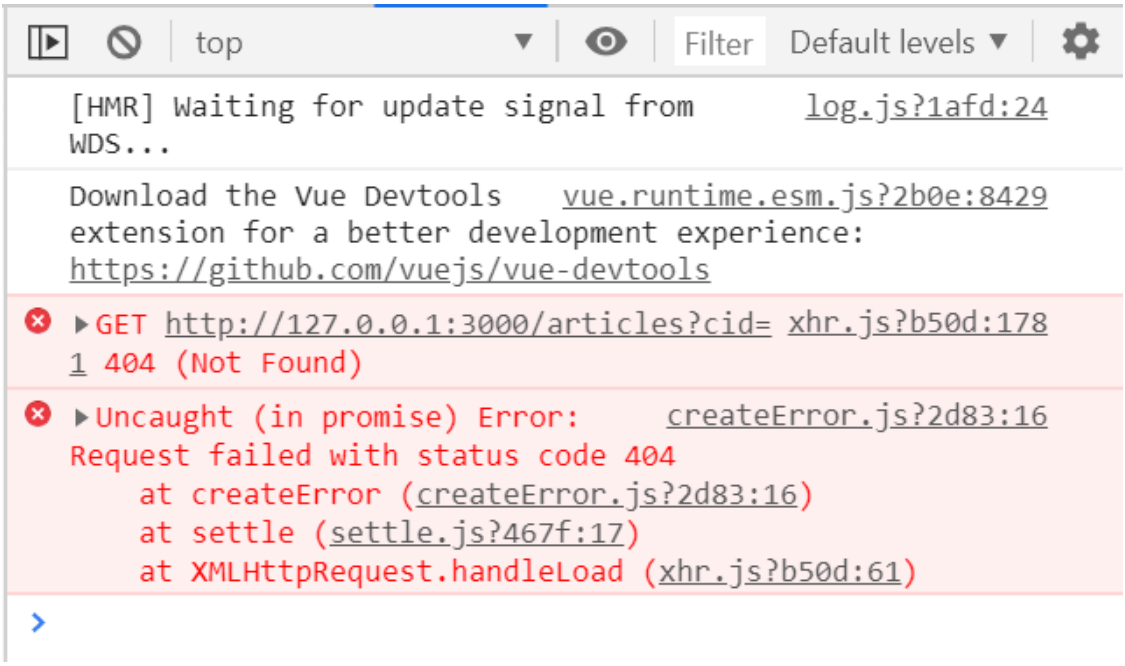
2.1 文章列表的业务实现

· 初始化的情况

初始情况下要去服务器获取默认被选定的顶部选项卡分类所包含的文章列表，此时引申出 Vue 生命周期的钩子函数 -- mounted，所以 Home.vue 中的示例代码如下：

```
this.axios.get('/articles?cid=' + this.active)
```

此时脚手架的运行结果如下图所示：



出现以上错误的原因是：服务器不存在指定的接口，所以：

第一步：在 Node 服务器的 app.js 中创建/articles 的接口，请求方式为 GET，示例代码如下：

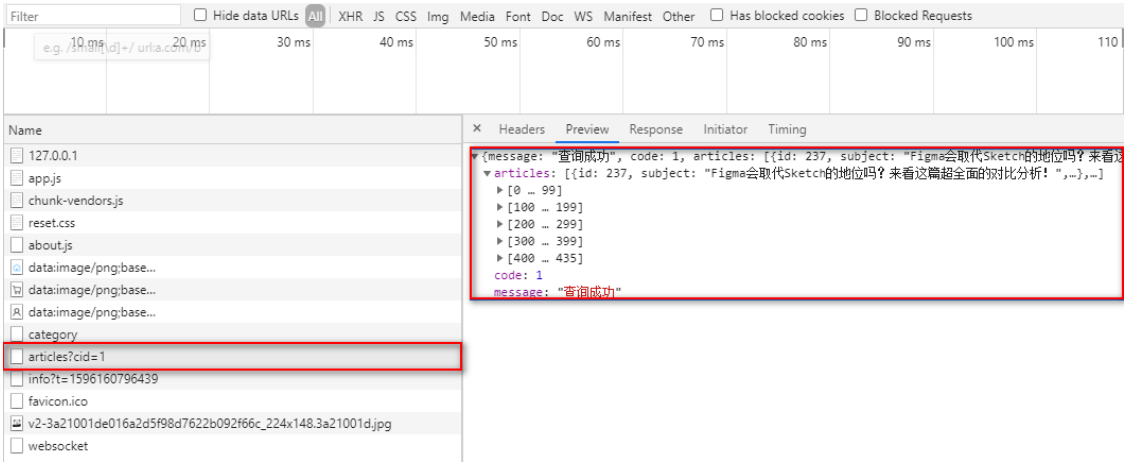
```
//获取特定分类下的文章数据
server.get('/articles',(req,res)=>{
    //获取客户端URL 地址栏的参数
    let cid = req.query.cid;
    //以获取到的参数为条件在文章数据表中进行
```

查找操作

```
    let sql = 'SELECT id,subject,description,image FROM xzqa_article WHERE category_id = ?';
    pool.query(sql,[cid],(err,results)=>{
        if(err) throw err;
        res.send({message:'查询成功',code:1,articles:results});
    });
});
```

完成上述代码后，要重新启动服务器!

此时脚手架的运行结果如下图所示：



此时代表服务器已经返回数据到客户端了，所以：

第二步：客户端接收服务器返回的数据，并且显示在页面组件，此时示例代码如下：

```
this.axios.get('/articles?cid=' + this.act
ive).then(res=>{
    console.log(res);
})
```

还需将返回的数据存储到 Vue 的变量中，然后再通过 v-for 指令进行循环输出，所以：

```
data(){
  return {
    //存储文章数据
    articles:[] //默认值
  }
}
```

除此之外，还需要将服务器返回的数据存储到刚刚声明的变量中，示例代码如下：

```
this.axios.get('/articles?cid=' + this.act
ive).then(res=>{
```

```
    this.articles = res.data.articles;
});
```

最后通过 v-for 指令进行循环输出即可，示例代码如下：

```
<div
     class="InfoItem"
     v-for="(article,index) of articles"
     :key="index">
    <!-- 标题信息开始 -->
    <div class="InfoItemHead">{{article.subject}}</div>
    <!-- 标题信息结束 -->
    <!-- 简介与缩略图区域开始 -->
    <div class="InfoItemContent">
        <!-- 简介开始 -->
        <div class="InfoItemDes">{{article.description}}</div>
        <!-- 简介结束 -->
        <!-- 缩略图开始 -->
        <img src="../assets/articles/v2-3a
```

```
21001de016a2d5f98d7622b092f66c_224x148.jp
g" class="InfoItemImg-s">
        <!-- 缩略图结束 -->
    </div>
    <!-- 简介与缩略图区域结束 -->
</div>
```

此时脚手架的运行结果如下图所示：

现在整个结构中除缩略图外，其余均是正常数据，所以：

如果此时使用 JS 中的模板字符串进行图片输出的的话，示例代码如下：

```
<img :src="`../assets/articles/${article.image}`" class="InfoItemImg-s">
```

但此时脚手架的运行结果是：

其根本原因是：动态图像需要通过 require 命令进行加载才可以！所以还需要重新修改原来在 mounted 钩子函数中的相关代码才行：

```
this.axios.get('/articles?cid=' + this.active).then(res=>{
    //将服务器返回的文章数据存储到 articles 变
```

```
量中
    let data = res.data.articles;
    //循环动态加载图像
    data.forEach(item=>{
        //因为有的文章没有图像，所以无需动态加载图像了
        //故进行判断操作
        if(item.image != null)  {
            //动态加载图像，此时 item 中 id,subject,description 没有发生任何变化
            //但是图像已经成为了动态加载的图像了
            //所以现在只能依次将最新的 item 添加到 articles 变量中了
            item.image = require('../assets/articles/' +  item.image) ;
        }
        this.articles.push(item);
    });
})
```

在通过 v-for 指令进行循环操作时，进行输出就可以了，示例代码如下：

```
<img :src="article.image" class="InfoItemImg-s" v-if="article.image != null">
```

之所以要添加 v-if 指令的原因是：因为既使没用图片的文章也已经加载了，但是无需要页面中显示空的、灰色的矩形，所以才进行判断操作。

因为初始情况下要显示的数据多达几百条，而且绝大部分文章都带有缩略图，所以这会增大服务器的请求压力，而且客户端将长时间面对空白页面，这样客户体验变得很差，所以最好的解决方案是：对于图像采用懒加载!!!

- **切换顶部选项卡时的情况**

切换顶部选项卡时，同样要向服务器发送请求，以获取当前分类下包含的文章数据，引申出 -- 如何在程

序中获知顶部选项卡发生了变化呢? -- 通过监听顶部选项卡所绑定的变量来实现，示例代码如下：

```
//监听顶部选项卡的变化
active(){
    //切换顶部选项卡时,清空之前存储的文章数据
    this.articles = [] ;
    this.axios.get('/articles?cid=' + this.active).then(res=>{
        let data = res.data.articles;
        data.forEach(item=>{
            if(item.image != null){
                item.image = require('../assets/articles/' + item.image);
            }
            this.articles.push(item);
        });
    });
}
```

现在已经可以实现初始化及切换顶部选项卡时的数据变化，但是由于数据太多，无形中在浪费服务器的请

求资源，所以最好的解决方案是：初始情况下只加载部分数据，然后向下滚动时再加载另外的一部分数据，实质上也就是采用分页实现。

· 分页显示数据

其涉及到两个技术问题：无限滚动、分页技术

· 无限滚动的实现

第一：无限滚动的指令要添加给哪个元素呢？-- 面板容器的 div 元素

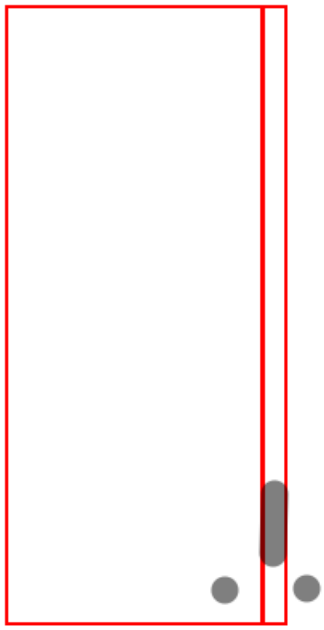
第二：在滚动到距离容器底部还有多少个像素时触发滚动函数呢？-- 10 像素

第三：触发的滚动函数是谁呢？-- loadMore

现在要进行分页显示数据了，其分为两种情形：

第一种：在初始情况下只显示部分数据或者说只显示第一页的数据

第二种：在滚动距离容器底部还有指定距离时，再加载第二页、第三页，...的数据



· 初始情况下显示第一页的数据

MySQL 的分页原理是：利用 SELECT 语句的 LIMIT 子句，每次只返回部分结果，即形成所谓的分页！

LIMIT 的语法结构是：

```
SELECT ... LIMIT [offset,] row_count
```

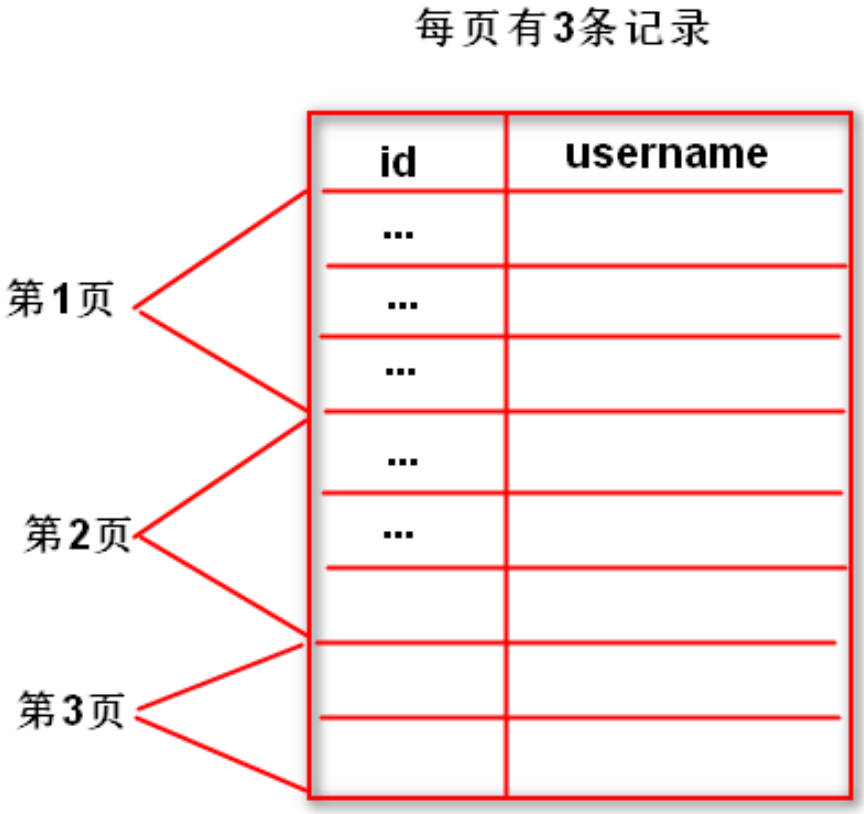
offset 参数指从第几条记录开始返回，其从 0 开始编号(与 id 无任何关系)

row_count 参数指返回多少条记录

```
# 从 0 开始返回，返回 5 条记录
SELECT * FROM users LIMIT 5;
```

在进行分页时，offset 参数值是不固定的，其标准的计算公式是:

(页码-1) * 每页显示的数据的量



那么必须在 Home.vue 中声明一个变量用于存储当前的页码，其初始值为 1，所以示例代码如下：

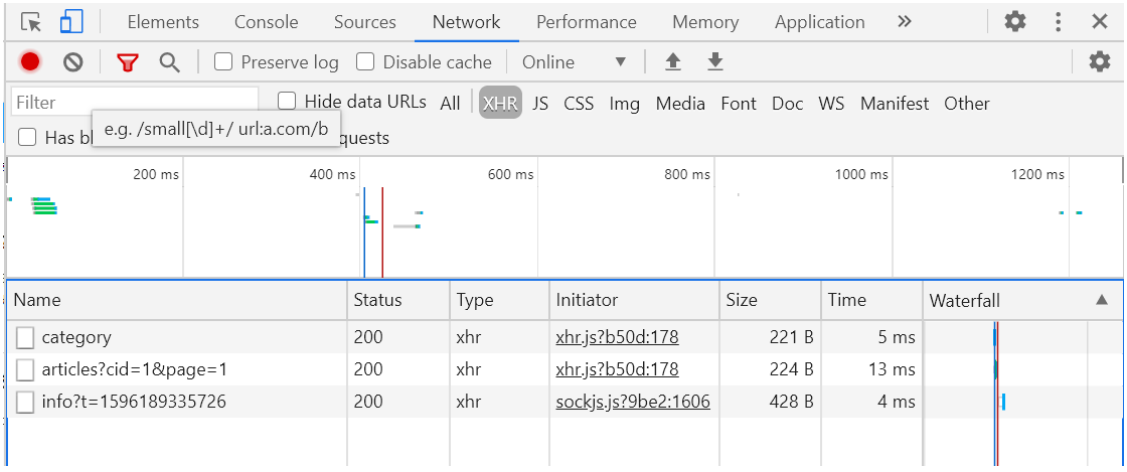
```
data(){
  return {
    //表示当前页码
    page:1
  }
}
```

现在在 mounted 钩子函数中获取初始化文章数据的时候，不仅要告诉服务器：要获取哪一类的文章，还要告诉服务器 -- 要获取该分类下的第 1 页的数据才行！于是还需要发送请求时再加额外的参数，示例代码如下：

```
this.axios.get('/articles?cid=' + this.active + '&page=' + this.page).then(res=>{

    //....
});
```

注意：URL 参数之间以&进行连接，形如：cid=5&page=9&mid=8

此时脚手架的运行结果如下图所示：



现在必须修改服务器的相关代码才可以 --/articles API

此时示例代码如下：

作业：尝试通过无限滚动实现页面数据的变化。

Day05:

MintUI-- Unit05

1.学子问答项目实践

1.1 文章列表的业务实现

- **初始情况下显示第一页的数据**

现在必须修改服务器的相关代码才可以 --/articles API

此时示例代码如下：

```
//获取特定分类下的文章数据
server.get('/articles',(req,res)=>{
```

```
    //获取客户端URL 地址栏的参数
    let cid = req.query.cid;
    //获取当前的页码
    let page = req.query.page;
    //分页实质上利用了SELECT 语句的LIMIT 子句
    //其标准计算公式为（页码-1） * 每页显示记录数
    //所以现在必须通过上述公式计算出 offset 参数值
    let offset = (page-1) * 20;
    //以获取到的参数为条件在文章数据表中进行查找操作
    let sql = 'SELECT id,subject,description,image FROM xzqa_article WHERE category_id = ? LIMIT ' + offset + ',20';
    pool.query(sql,[cid],(err,results)=>{
        if(err) throw err;
        res.send({message:'查询成功',code:1,articles:results});
    });
});
```

这样已经实现初始化的第一页数据可正常显示了!

· 滚动到距离页面底部还有指定像素时再加载第二页、第三页的数据

在滚动到距离页面底部还有指定像素时，要加载第二页、第三页的数据，那么就意味着无限滚动需要与分页结合使用，而且现在的页码会在原始的基础上发生累加操作。

上节课中的无限滚动指令调用的方法名称为 --loadMore，所以现在需要在该方法中：

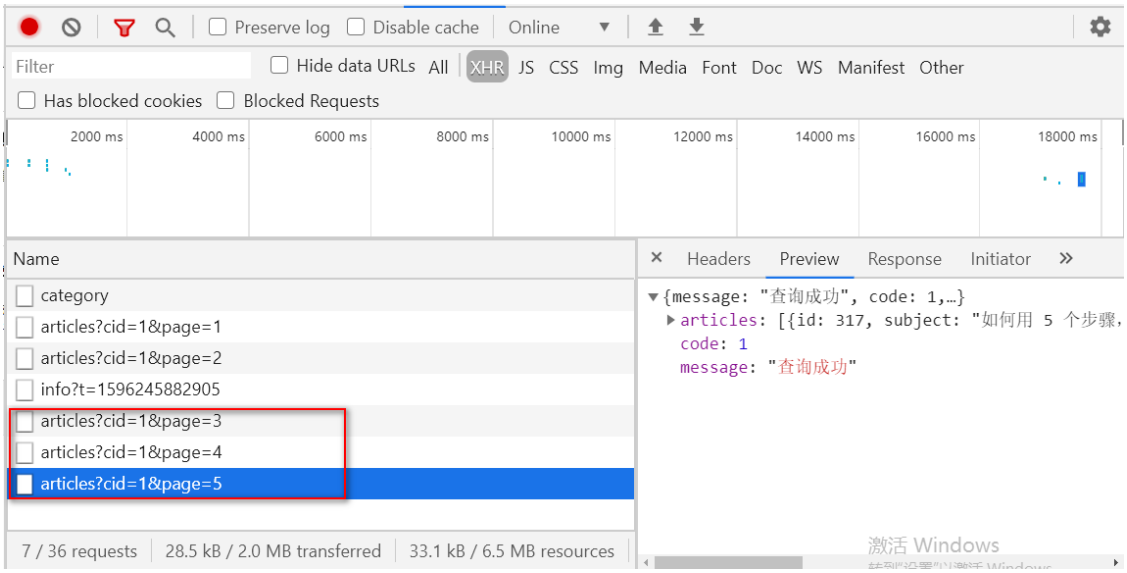
第一：让页码发生累加操作

```
loadMore(){
    //页码累加
    this.page++;
}
```

第二：将当前的页码发送到服务器以获取该指定分类下的第几页的数据，示例代码如下：

```
loadMore(){
    //页码累加
    this.page++;
    //.......
    this.axios.get('/articles?cid=' +
```

```
this.active + '&page=' +
this.page).then(res=>{

    });
}
```



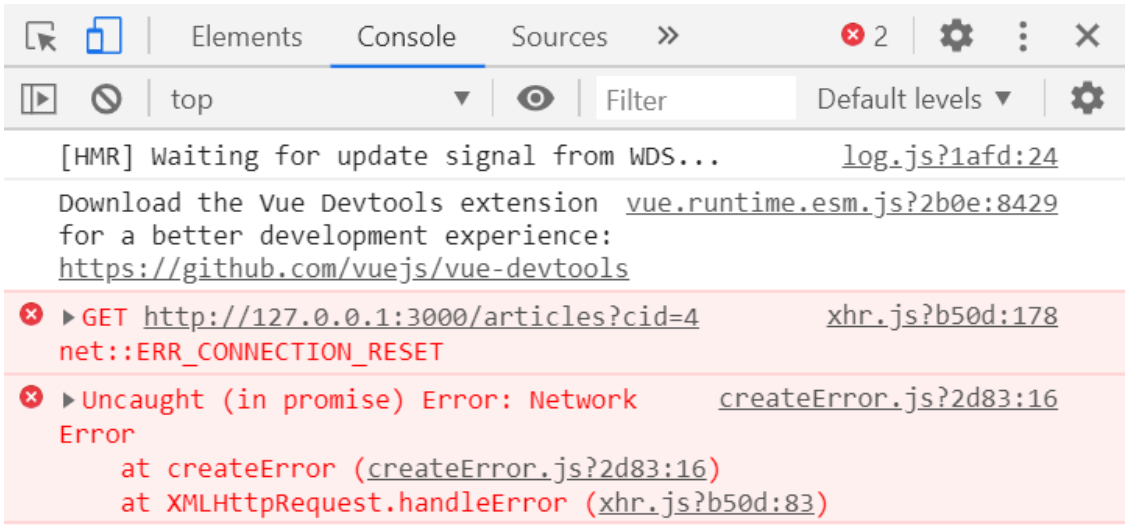
另外，app.js 中已经获取了当前分类的 ID 及页码参数，所以无需进行任何修改。但客户端现在还需要接收服务器返回的数据，另外，在接收过程中仍然需要进行图像的动态加载，示例代码如下：

```
loadMore(){
    //页码累加
```

```
        this.page++;
        //获取指当前分类下的第几页的数据
        this.axios.get('/articles?cid=' + this.active + '&page=' + this.page).then(res=>{
            let data = res.data.articles;
            data.forEach(item=>{
                if(item.image != null){
                    item.image = require('../assets/articles/' + item.image);
                }
                this.articles.push(item);
            });
        });
        }
```

到现在为止，可以在初始情况下获取第一页的数据及滚动情况下获取第 N 页数据了！

如果进行顶部选项卡切换时，脚手架运行效果如下图所示：



现在服务器提示：连接重置。现在查看一下 HTTP 请求的地址为 http://127.0.0.1:3000/articles?cid=4，而原来书写的/articles API 中需要接收客户端提交的两个参数 -- cid 和 page ，而现在的请求中只包含一个 cid!!! 接着再推断一下服务器的代码的执行过程：

```
server.get('/articles',(req,res)=>{
  let cid = req.query.cid; //该语句可以正常执行，且值为 4
    let page = req.query.page ; //该语句也可以正常执行，且值为 undefined
    let offset = (page - 1) * 20 ; // (unde
```

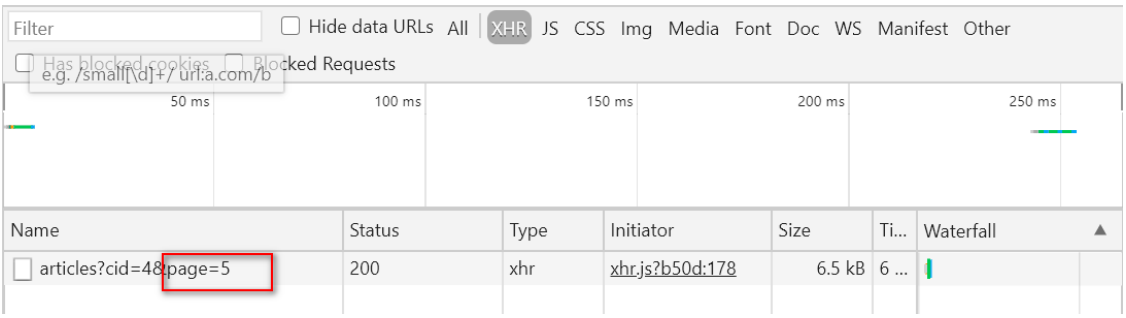
```
fined-1) * 20 ==> NaN
    let sql = 'SELECT * FROM xzqa_article WHERE category_id= ? LIMIT ' + offset + ', 20';
    //此时的SQL 结构如下:
    //SELECT * FROM xzqa_article WHERE category_id=4 LIMIT NaN,20;
    //当在执行上述SQL 时
    pool.query(sql,.....);
    //上述SQL 在MySQL 数据库中执行时 LIMIT 中出现了NaN, 它认为这是一个变量名称,所以在服务器
    //端出现 undeclared variable : NaN 的错误信息
});
```

所以其根本原因是因为在切换项部选项卡的时候缺少了一个请求参数 -- page

所以需要在 watch()监听 active 变量时，添加 page 参数即可，示例代码如下：

```
active(){
    //切换顶部选项卡时,清空之前存储的文章数据

    this.articles = [] ;
    this.axios.get('/articles?cid=' + this.active + '&page=' +  this.page).then(res=>{
        let data = res.data.articles;
        data.forEach(item=>{
            if(item.image != null){
                item.image = require('../assets/articles/' + item.image);
            }
            this.articles.push(item);
        });
    });
```

现在虽然可以进行顶部选项卡的切换了，但是在切换后会发现当前选项卡的数据发生了变化，此时查看一下脚手架的控制台可以发现：

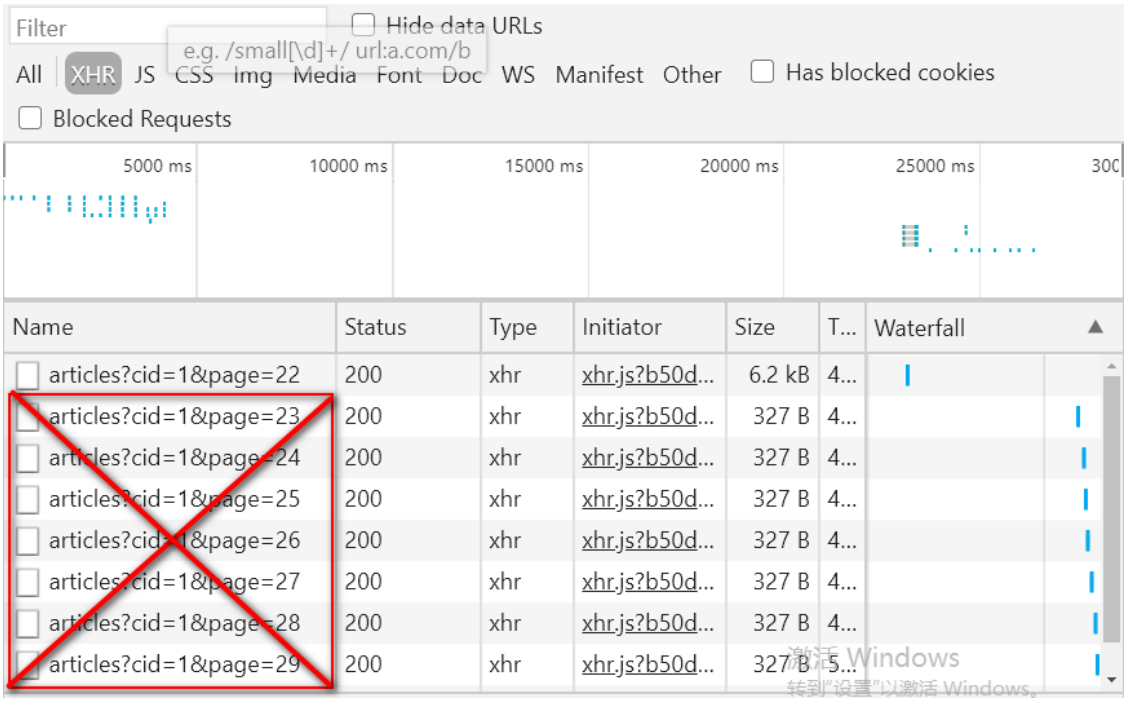


其原因是：因为在 active()函数中使用 this.page 变量，而这个变量的值可能已经由其他的分类在滚动时动态的发生了变化，所以现在在切换时将带有之前的数据。而切换选项卡时必须要从该分类下的第 1 页获取数据，所以必须人为将 this.page 变量值改变 1 即可，示例代码如下：

```
active(){
    //切换顶部选项卡时,清空之前存储的文章数据

    this.articles = [] ;
    //切换顶部选项卡时,页码初始值为 1
    this.page = 1;
    this.axios.get('/articles?cid=' + this.active + '&page=' +  this.page).then(res=>{
```

```
            let data = res.data.articles;
            data.forEach(item=>{
                if(item.image != null){
                    item.image = require('../as
sets/articles/' + item.image);
                }
                this.articles.push(item);
            });
        });
}
```

因为数据是有限的，但在滚动"最后一页"后继续向下滚动时，仍然可以发送请求，实质是错误的!



所以在向下滚动时，判断一下当前页码是否超出总页数，如果没有超出，则继续请求，否则不发送请求。

所以这有一个根本的问题：如何知道总页数呢?

再想一下，数据是存在于数据库中的，而数据库的访问操作都是在服务器上实现的，所以必须让服务器返回给客户端总页数，然后客户端才能接收该数据，并且利用该数据进行判断操作，以决定是否发送请求。

所以现在修改 app.js

总页数标准计算公式为：

Math.ceil(总记录数 / 每页显示的记录数)

目前必须要获知到总记录数，才可以计算出总页数 ， SQL 语句结构如下：

```
SELECT COUNT(id) AS count FROM xzqa_article WHERE category_id=?
```

MySQL 中的聚合函数有：COUNT、SUM、AVG、MIN、MAX ,只有一个返回结果

获取总记录数及总页数的示例代码如下：

```
server.get('/articles',(req,res)=>{
    //获取客户端URL 地址栏的参数
    let cid = req.query.cid;
    //获取当前的页码
    let page = req.query.page;
    //分页实质上利用了SELECT 语句的LIMIT 子句
    //其标准计算公式为 (页码-1) * 每页显示记录数
    //所以现在必须通过上述公式计算出 offset 参数值
    let offset = (page-1) * 20;
    //获取总记录数的操作
    let sql = 'SELECT COUNT(id) AS count FROM xzqa_article WHERE category_id=?';
    pool.query(sql,[cid],(err,result)=>{
        if(err) throw err;
        //获取出总记录数
        //因为聚合函数只有一个返回结果,所以result[0]将返回该结果
        //而结果是一个对象,包含有 count 的属性
```

```
(count 属性实质是 SQL 语名中字段的别名)
        let rowCount = result[0].count;
        //声明变量用于存储每页显示的记录数
        let pagesize = 20;
        //声明变量 pagecount 用于存储计算出来
总页数
        let pagecount = Math.ceil(rowCount
 / pagesize);
    });
});
```

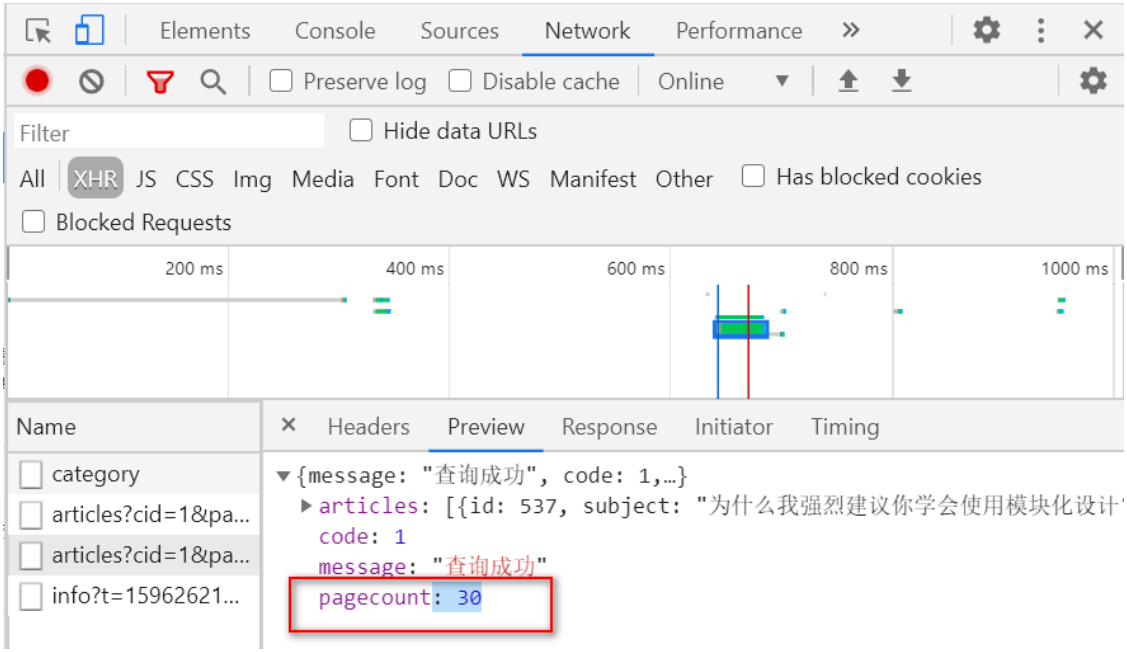
还需要将总页数的结果返回给客户端，除此之外，还要保证两个 SQL 可以都顺序执行到，最后其示代码如下：

```
//获取特定分类下的文章数据
server.get('/articles',(req,res)=>{
    //获取客户端 URL 地址栏的参数
    let cid = req.query.cid;
    //获取当前的页码
    let page = req.query.page;
```

```
    //分页实质上利用了SELECT 语句的LIMIT 子句
    //其标准计算公式为（页码-1）* 每页显示记录数
    //所以现在必须通过上述公式计算出 offset 参数值
    let offset = (page-1) * 20;
    //获取总记录数的操作
    let sql = 'SELECT COUNT(id) AS count FROM xzqa_article WHERE category_id=?';
    pool.query(sql,[cid],(err,result)=>{
        if(err) throw err;
        //获取出总记录数
        //因为聚合函数只有一个返回结果,所以result[0]将返回该结果
        //而结果是一个对象,包含有 count 的属性(count 属性实质是 SQL 语名中字段的别名)
        let rowCount = result[0].count;
        //声明变量用于存储每页显示的记录数
        let pagesize = 20;
        //声明变量pagecount 用于存储计算出来总页数
```

```
        let pagecount = Math.ceil(rowCount / pagesize);
        //以获取到的参数为条件在文章数据表中进行查找操作
        sql = 'SELECT id,subject,description,image FROM xzqa_article WHERE category_id = ? LIMIT ' + offset + ',' + pagesize;
        pool.query(sql,[cid],(err,results)=>{
            if(err) throw err;
            res.send({message:'查询成功',code:1,articles:results,pagecount:pagecount});
        });

    });

});
```

此时可以在脚手架查看到返回结果中存在总记录的信息，截图如下：



所以现要客户端必须要接收返回数据，故在 data()中声明 pagecount 属性用于存储该返回结果，示例代码如下：

```
data(){
   return {
         //用于存储服务器返回的总页数
      pagecount:0
   }
}
```

以下代码中都需要接收服务器返回的总页数：

第一：mounted 钩子函数中（因为初始化的

pagecount 为 0,但当 mounted 运行之后，即可获取到服务器的总页数，所以客户端需要接收)

第二：在监听 active 变量的函数中(因为顶部选项卡发生了变化，也就意味着分类发生了变化，同样总页数也可能发生变化)

第三：在 loadMore()函数中，实际上根本无需在该方法中接收服务器返回的总页数，之所以添加接收语句的原因是 -- 自定义函数的封装

既然已经接收到了总页数的数据，那么就应该在 loadMore()函数调用时，来判断当前页码与总页数的比较情况，如果在总页数范围内，则继续发送请求，否则无需发送请求。示例代码如下：

```
loadMore(){
  this.page++;
  if(this.page <= this.pagecount){
    //.....
  }
}
```

纵观 mounted()、active()及 loadMore()函数中，都存在发送异步请求的过程，而且代码完全相

同，所以最好的解决方案是：将发送异步请求的过程封装为自定义函数，然后多次调用即可！！！如装loadData()的函数，示例代码如下：

```
methods:{
    loadData(){
        //....
    }
}
```

当封装完自定义函数后，需要分别在 mounted()、active()及 loadMore()函数中进行调用即可。

当触发无限滚动时，如果服务器处理请求的速度非常慢，此时最好：

1.当用户再次进入到滚动范围时不再触发滚动指令

第一步：在相关无限滚动指定的 HTML 元素上添加 infinite-scroll-disabled="busy"属性

第二步：在 data 属性变量 busy，并且指定初始值为 false

第三步：在 *loadData()函数中将 busy 属性*设置为 true,代表当前服务器正在处理相关的请求，既 使现

在再次滚动到指定范围内，也不再触发滚动指令。

第四步：在 loadData()函数处理完成之后，应该将 busy 属性修改为 false，代表现在服务器已经处理完之前的相关请求了，现在可以继续处理其他后续请求.

2.最好在页面中显示加载提示框，以提示用户

Indicator 组件

indicator 组件用于显示加载提示框，其语法结构是：

```
//打开的简捷语法
this.$indicator.open("提示内容")

//打开的标准语法
this.$indicator.open({
  text:"提示内容",
  spinnerType:"snake|double-bounce|triple-bounce|fading-circle"
//中间等待动画的样式
})
```

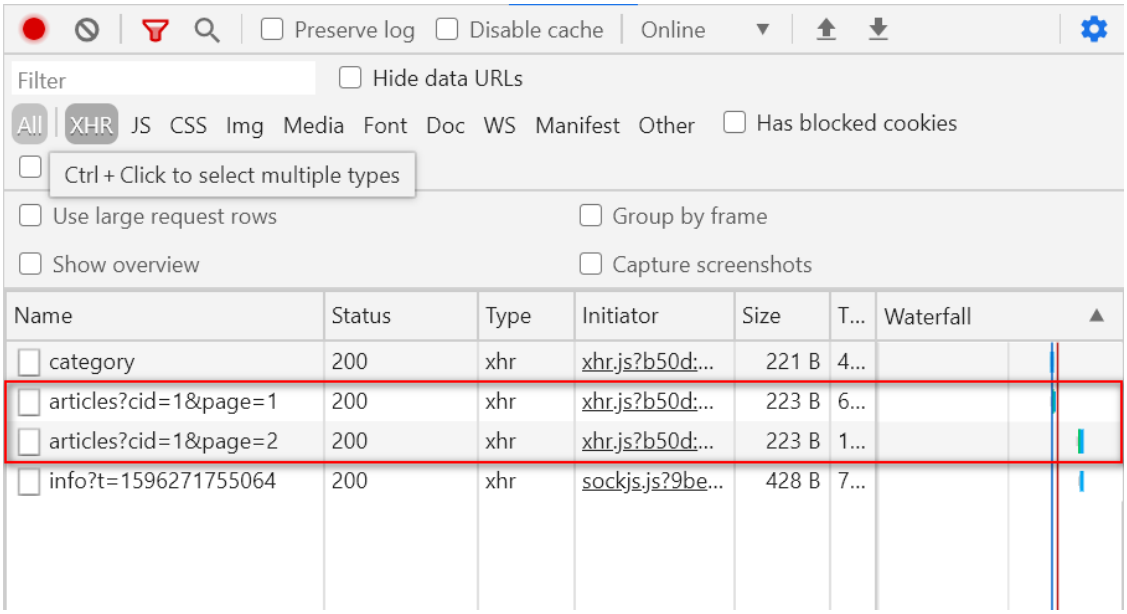
snake，蛇形，double-bounce，双弹跳，triple-bounce，三弹跳，fading-circle，渐消圆

```
//关闭的语法
this.$indicator.close()
```

在应用中应该是在发送请求时出现加载提示框，而在请求完毕后，关闭加载提示框！所以应该在loadData()函数调用时出现加载提示框，而该函数执行完成后，关闭加载提示框，示例代码如下：

```
loadData(){
    //代表当前服务器正在处理请求，既使再次进入滚动范围也不再触发滚动指令
    this.busy = true;
    //显示加载提示框
    this.$indicator.open('加载中...');
    this.axios.get('/articles?cid=' + this.active + '&page=' + this.page).then(res=>{
        //获取服务器返回的文章数据
        let data = res.data.articles;
        //将服务器返回的总页数存储到 pagecount 变量中
```

```
            this.pagecount = res.data.pagecoun
t;
            data.forEach(item=>{
                //因为有的文章不存在缩略图，所以
无需动态加载
                if(item.image != null){
                    //动态加载缩略图(实际上是动态
加载后,属性重新赋值的过程)
                    item.image = require('../as
sets/articles/' + item.image);
                }
                //将item(文章对象,包含id,subjec
t 等属性添加到articles 数组的末尾)
                this.articles.push(item);

            });
            //代表服务器已经可以继续处理接下来的
滚动指令了
            this.busy = false;
            //关闭加载提示框
            this.$indicator.close();
        });
```



初始浏览时的异步请求如上图所示，发现竟然有两个请求，根本原因是：是因为 Mint UI 将自动触发一次无限滚动的指令。其依靠于

infinite-scroll-immediate-check="true"属性 解决。

1.2 文章详情页

现在在首页中单击文章的标题及缩略图可以链接到文章详情页面，以查看详细信息，所以：

第一步：创建页面组件，并且在 router/index.js 中

导入该页面组件，并且设定路由信息

第二步：在首页中分别为文章的标题及缩略图添加链接，以链接到跳转目标的详情页面组件

第三步：在文章详细页面中要**显示刚刚单击的那篇文章的详细信息** -- 如何知道刚才单击的是哪一篇文章呢？只能通过 URL 传参的方式进行传递了，其有两种实现方法：

A.article?id=5 的形式

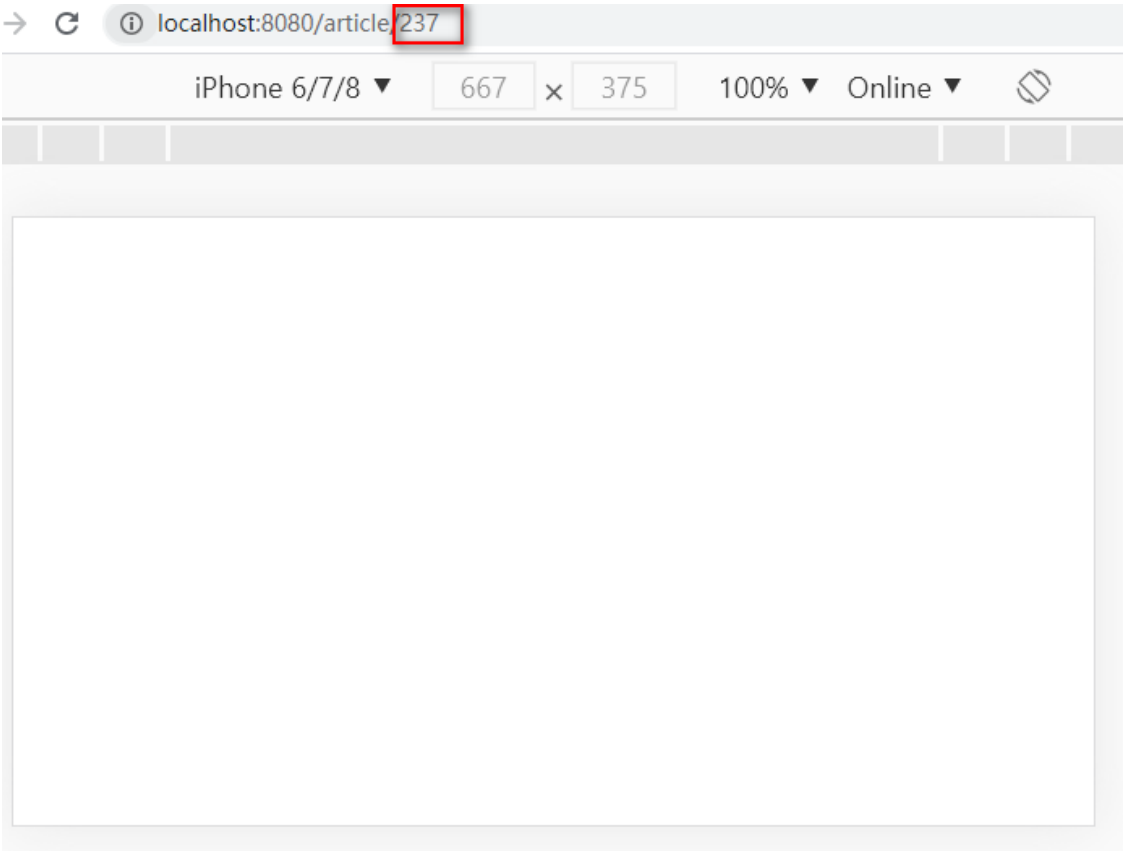
B.article/5 的形式

目前在本案例中准备使用第二种方式

如果要采用第二种方式的话，必须在书写链接时，带有后面的 id 参数(也是就说必须带有 5、12 等数字才行)，所以首页中的链接必须进行修改，示例代码如下：

```
<div class="InfoItemHead">
    <router-link :to="`/article/${article.id}`">
    {{article.subject}}
    </router-link>
</div>
```

此时单击某一篇文章的标题后显示的结果如下图所示:

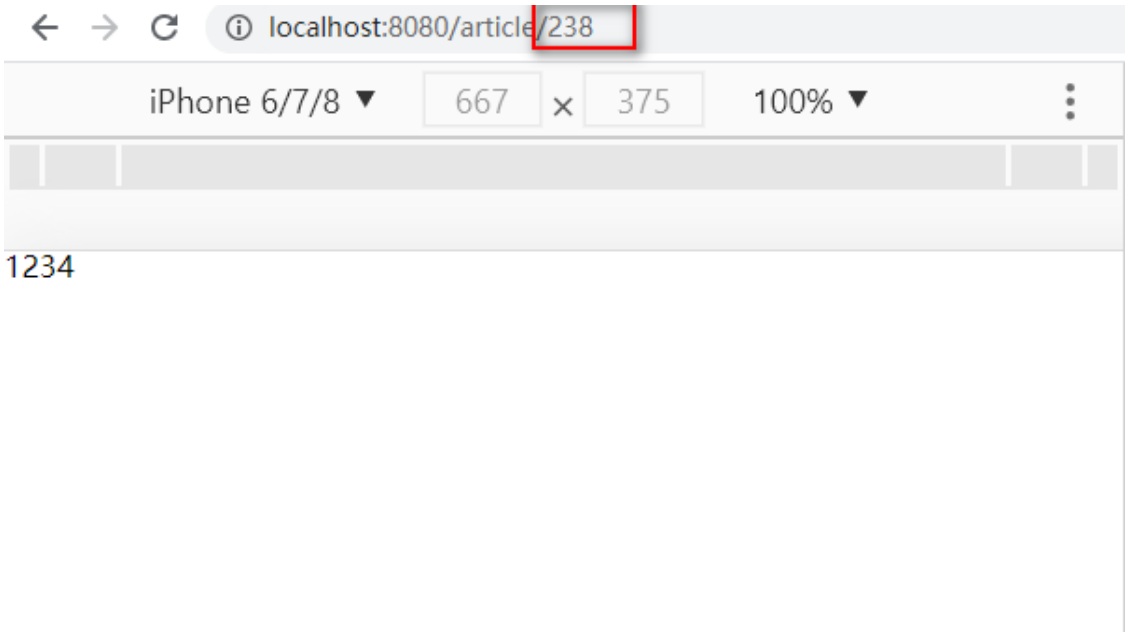


原因是因为: 路由信息与当前地址不匹配!!!

所以还需要修改路由信息，代码如下:

```
{
    path:'/article/:id',
    component:Article
}
```

此时可以显示出正常的、预期的结果了，截图如下：



现在需要获取出地址栏中的参数值，并且在 mounted 钩子函数中发送异步请求，去获取该篇文章的详细信息，所以：

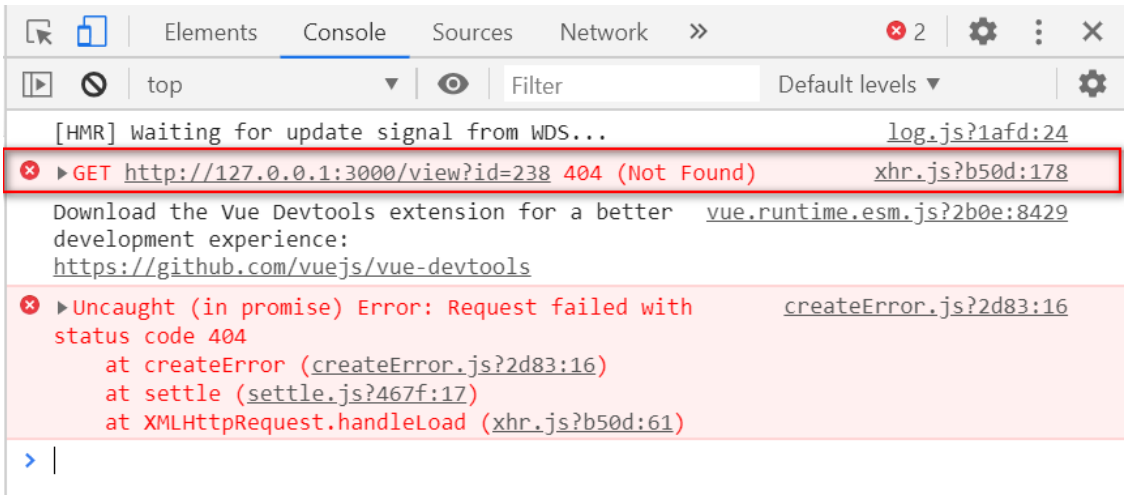
第一步：获取地址栏中的参数值

```
this.$route.params.id
```

第二步：在 mounted 钩子函数中发送请求，示例代码如下：

```
mounted(){
```

```
  let id = this.$route.params.id;
    this.axios.get('/view?id=' + id).then
(res=>{
        //....
    });
}
```



作业:

1.完成显示文章详情信息的功能 -- 文章的详细信息存储到 xzqa_article 数据表中。

2.尝试完成用户的注册操作

Day06:

MintUI-- Unit06

1.文章详情的实现

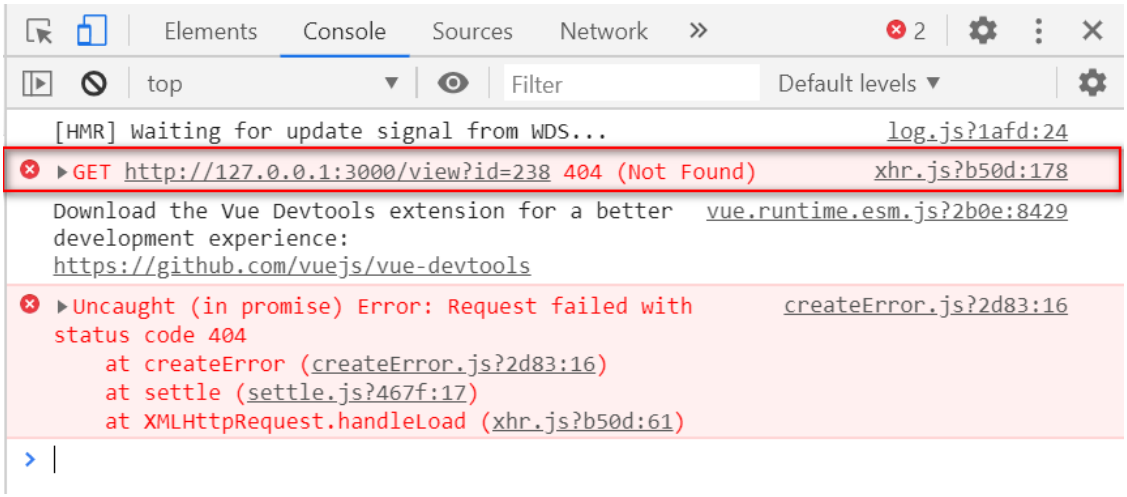
现在需要获取出地址栏中的参数值，并且在 mounted 钩子函数中发送异步请求，去获取该篇文章的详细信息，所以：

第一步：获取地址栏中的参数值

this.$route.params.id

第二步：在 mounted 钩子函数中发送请求，示例代码如下：

```
mounted(){
  let id = this.$route.params.id;
   this.axios.get('/view?id=' + id).then
(res=>{
      //....
   });
}
```



其根本原因是：服务器根本没有对应的路由地址!!!

所以：

A.在 app.js 中创建/view 的 GET 类型的路由，示例代码如下：

```
// 获取指定文章详细信息的接口
server.get('/view',(req,res)=>{
    //获取地址栏中的 id 参数
    let id = req.query.id;
    //以 id 为条件进行文章相关信息的查找操作(一会儿还要进行调整)
    let sql = 'SELECT id,subject,content FROM xzqa_article WHERE id=?';
    pool.query(sql,[id],(err,results)=>{
```

```
        if(err) throw err;
        res.send({message:'查询成功',code:
1,results:results[0]});
    });
});
```

B.在脚手架中进行服务器数据的接收，示例代码如下:

```
data(){
  return {
    info:{}
  }
},
mounted(){
    //获取动态参数
    let id = this.$route.params.id;
    //发送请求
    this.axios.get('/view?id=' + id).then
(res=>{
        this.info = res.data.results;
    });
```

```
}
```

C.直接在页面组件中输出内容，示例代码如下：

```
<h1 style="border:2px solid #f00;">{{info.
subject}}</h1>
<div v-html="info.content"></div>
```

采用 v-html 指令的原因是：在数据库中文章的正文存储了大量的 HTML 标签，而默认情况下，Vue 为防止 XSS 攻击，将对内容的特殊符号，如>、<等进行转换操作 -- HTML 实体，而现在本案例中依然要输出正常的 HTML 标签，所以需要通过 v-html 指令。

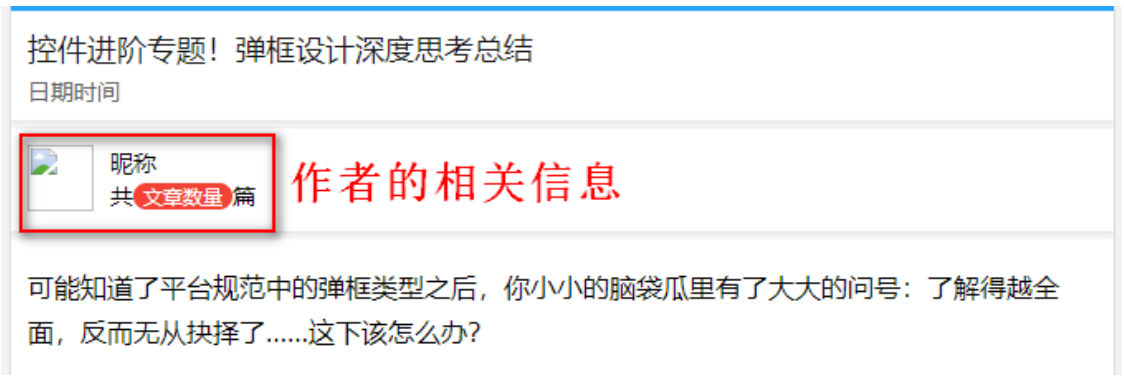
<style scoped>内的样式不会对 v-html 指令内的内容生效。如果要修饰 v-html 指令内的内容的话，应该使用<style>定义全局样式(有可能对外界重复 ID 或其他 class 产生冲突)，示例代码如下：

```
<style>
    .article-content p{
```

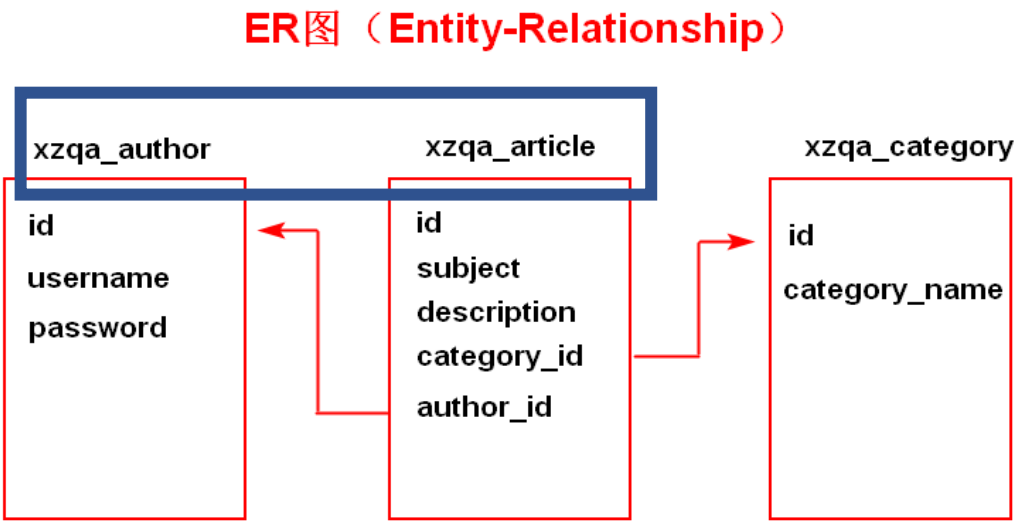
```
        padding:5px 0;
        font-size:16px;
    }
    .article-content img{
        max-width: 100%;
        display: block;
    }
</style>
```

https://cn.vuejs.org/v2/api/#v-html

文章详细信息中还包括作者的相关信息，如下图所示：



回顾一下原来的数据表 ER 图



根据上述的 ER 图可以发现：数据表 xzqa_article 与 xzqa_author 之间存在外键关联。所以现在需要修改 app.js 中的/view 路由，以进行连接的查找操作，示例代码如下：

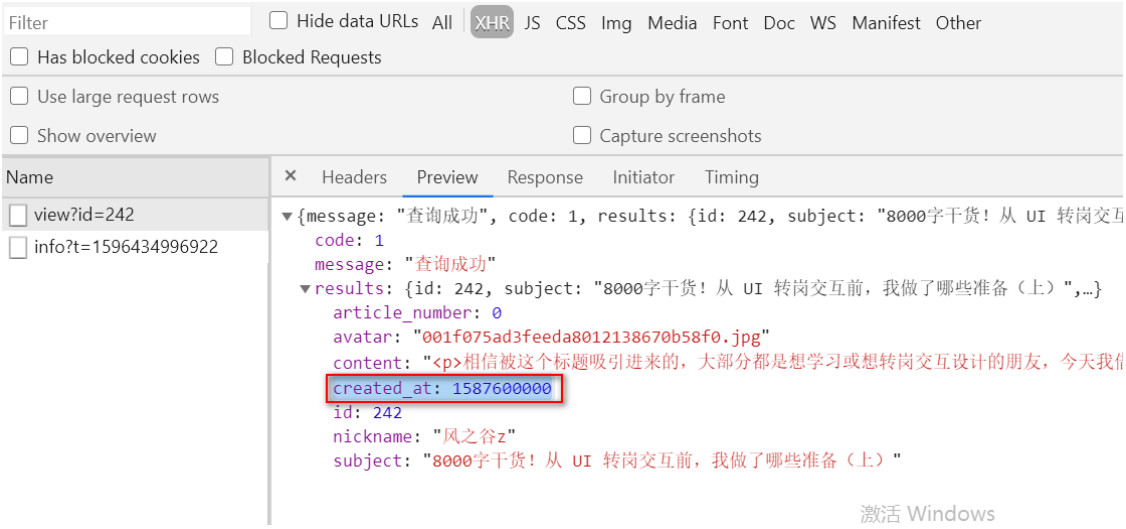
```
// 获取指定文章详细信息的接口
server.get('/view',(req,res)=>{
    //获取地址栏中的 id 参数
    let id = req.query.id;
    //以 id 为条件进行文章相关信息的查找操作
(一会儿还要进行调整)
```

```
    let sql = 'SELECT a.id,subject,conten
t,nickname,avatar,article_number FROM xzqa
_article AS a INNER JOIN xzqa_author AS u
ON author_id = u.id WHERE a.id=?';
    pool.query(sql,[id],(err,results)=>{
        if(err) throw err;
        res.send({message:'查询成功',code:
1,results:results[0]}); //返回的是[{}]格式
的数组  里边包含了有且只有一个对象
    });
});
```

现在在文章详细信息中，除发表日期外，其余都是完整的。于是还需要修改 app.js，并且/view 的 SQL 中添加该字段才可以。

```
let sql = 'SELECT a.id,subject,content,nic
kname,avatar,article_number,created_at FRO
M xzqa_article AS a INNER JOIN xzqa_author
 AS u ON author_id = u.id WHERE a.id=?'
```

此时脚手架的运行结果如下：



时间戳:

现在问题是：如何将类似 1587600000 的数字转换成可识别的日期时间格式？ -- Moment

2.Moment

Moment 是一个 JavaScript 日期时间处理类库。

http://momentjs.cn/

安装

```
npm install --save moment
```

解析 -- 将不同的日期格式转换成 Moment 对象

```
//将当前日期和时间转换成 Moment 对象
moment();
//将指定的日期转换成 Moment 对象
moment('2020-8-3');
//将 Unix 时间戳转换成 Moment 对象
//时间戳(Timestamp)，指公元 1970-01-01 00:00:00 到现在经历的秒数
moment.unix(Unix 时间戳)
```

moment 对象的 format()方法

该方法用于格式化 moment 对象，其语法结构是：

```
moment.format("格式")
```

- Y 四位的年份
- MM 两位的月份
- DD 两位的日期
- HH 两位的小时
- mm 两位的分钟
- ss 两位的秒

在 Vue 使用 moment：需要在 main.js 完成以下代码：

```
import moment from 'moment';
Vue.prototype.moment = moment;
```

详情界面：

```
                    <div class="article-header-datetime">
                     {{moment.unix(info.created_at).form
at('Y 年 MM 月 DD 日 HH:mm:ss')}}
                    </div>
```

Badge 组件

Badge 组件用于实现徽章，其语法结构是：

三个属性：type 选择徽章颜色

size 选择徽章大小 color 选择徽章颜色(自定义)

```
<mt-badge type="primary|success|error|warn
ing" size="small|large|normal" color="自定
```

义颜色">

...

</mt-badge>

徽章颜色:primary|success|error|warning

徽章大小: small|large|normal

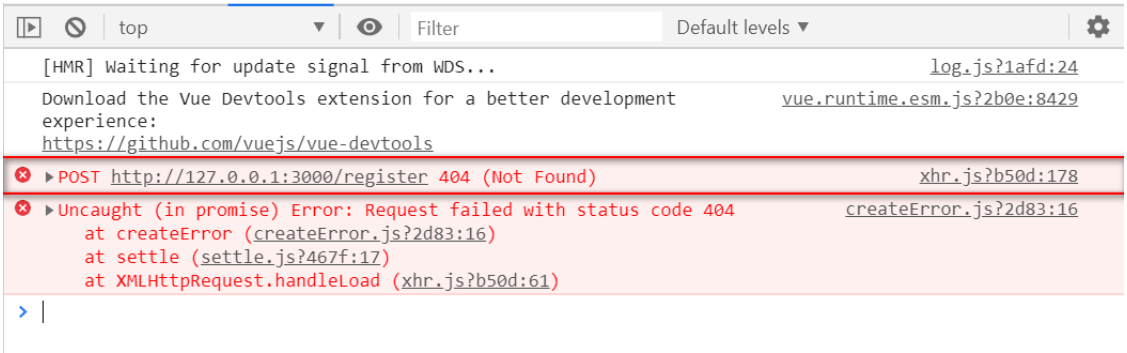
3.用户注册

在前期已经实现了用户注册的基本的页面结构及客户端的业务，但是在所有信息合法的情况下，必须去服务器上校验该用户是否已经存在，如果存在，则返回错误信息给客户端；如果不存在，则提示注册成功。所以现在：

注册验证： 当用户输入用户名时 验证是否和数据库中已有的用户名有重复(意思就是在数据库查询用户输入的用户名是否是已经存在的 存在的话 result[0].count 就应该是有值的 那么就应该提示用户该用户已存在 反之则通过验证 才记录其写入的密码 才能把完整的注册信息提交到数据库保存用户数据)

```
this.axios.post('/register','username=' +
this.username + '&password=' + this.passwo
rd).then(res=>{
    //...
});
```

如果在客户端所有的信息都合法的情况下，单击“免费注册”按钮的话，运行结果如下：



其原因是：在服务器上没有相应的 API,所以：

第一步：在 app.js 中创建/register 的 POST 类型的路由，示例代码如下：

```
//用户注册的接口
server.post('/register',(req,res)=>{
```

```
});
```

第二步：在获取以 POST 方式提交的相关信息 -- BodyParser 模块

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

进行相关的配置操作

```
//引入 body-parser 模块
const bodyParser = require('body-parser');


//将 bodyParser 作为 Server 中间件使用
server.use(bodyParser.urlencoded({
    extended:false
}));
```

复习：

```
app.use( bodyParser.urlencoded({      //urlencoded，可以将 po
extended:false     //extended:false 不使用扩展的第三方的 qs 模
}) );
```

```
//.在路由中获取 post 请求的数据，格式为对象
req.body
```

现在可以通过 req.body.xx 来获取以 POST 方式提交的数据了，示例代码如下：

```
//用户注册的接口
server.post('/register',(req,res)=>{
    //获取用户提交的用户名等信息
    let username = req.body.username;
    console.log(username);
});
```

接下来就应该以当前的用户名为条件，在 xzqa_author 数据表中查找该用户是否存在，如果用户存在,则产生合法的错误信息；如果用户不存在，则将用户的相关信息写入数据表，示例代码如下：

```
//用户注册的接口
server.post('/register',(req,res)=>{
    //获取用户提交的用户名等信息
```

```
    let username = req.body.username;
    //以用户名为条件进行查找操作,如果用户存在,则产生合法的错误信息
    //如果用户不存在,则将用户的相关信息写入数据表 -- xzqa_author
    let sql = 'SELECT COUNT(id) AS count FROM xzqa_author WHERE username=?';
    pool.query(sql,[username],(err,results)=>{
        if(err) throw err;
        if(results[0].count){ //如果已存在同名用户
            res.send({message:'注册失败',code:0});
        } else { //否则就是数据库不存在 就能执行注册业务
            //获取密码信息
            //如果合法
            let password = req.body.password;
            console.log(password);
        }
```

```
    })

});
```

密码是在 xzqa_author 表中是采用 MD5 进行加密的，所以在写入记录时，必须以密码字段通过 MySQL 数据库本身提供的 MD5()函数进行加密，此时的 SQL 成了：

```
let sql = 'INSERT xzqa_author(username,password) VALUES(?,MD5(?))';
```

综上，现在/register 路由示例代码如下：

```
//用户注册的接口
server.post('/register',(req,res)=>{
    //获取用户提交的用户名等信息
    let username = req.body.username;
    //以用户名为条件进行查找操作,如果用户存在,则产生合法的错误信息
    //如果用户不存在,则将用户的相关信息写入数据表 -- xzqa_author
```

```
    let sql = 'SELECT COUNT(id) AS count F
ROM xzqa_author WHERE username=?';
    pool.query(sql,[username],(err,result
s)=>{
        if(err) throw err;
        if(results[0].count){
            res.send({message:'注册失败',co
de:0});
        } else {
            //获取密码信息
            let password = req.body.passwor
d;
            //完成数据写入操作 此时的用户注册
信息为合法信息 就应存入数据表中保存,并将密码加
密处理
            let sql = 'INSERT INTO xzqa_aut
hor(username,password) VALUES(?,MD5(?))';
            pool.query(sql,[username,passwo
rd],(err,results)=>{
                if(err) throw err;
                res.send({message:'注册成功
',code:1});
```

```
            });
        }
    })

});
```

最后在客户端接收服务器返回的信息，并且根据返回的信息，在页面组件中完成不同的功能，示例代码如下：

```
this.axios.post('/register','username=' + this.username + '&password=' + this.password).then(res=>{
    //客户端的验证注册
            //如果数据库存在那么就提示错误
    if(res.data.code == 0){
        this.$messagebox("注册提示","对不起,用户已存在");
            //如果数据库不存在,就说明注册成功 顺带的操作就是  跳转到主页
    } else {
        this.$router.push('/');
```

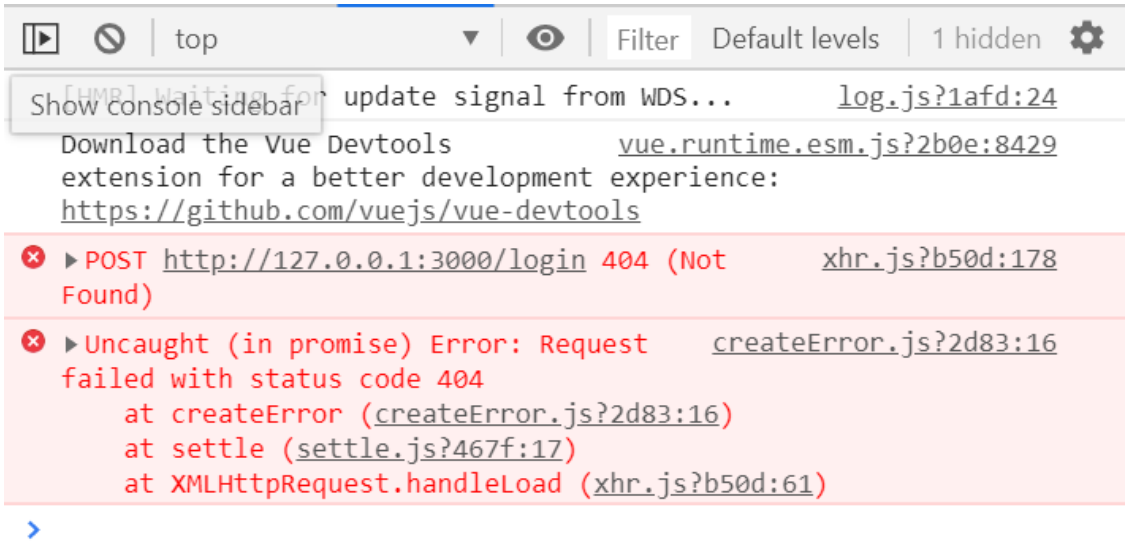
```
    }
});
```

4.用户登录

用户登录时仍然以发送 POST 类型的请求，请求地址为/login,所以现在脚手架中可以以下列示例代码发送请求了：

```
this.axios.post('/login','username=' + this.username + '&password=' + this.password).then(res=>{

});
```

此时运行后，结果如下图所示：



之所以产生错误：是因为服务器不存在指定的接口，所以：

第一步：在 app.js 创建/login 的 POST 类型的接口，示例代码如下：

```
// 用户登录接口
server.post('/login',(req,res)=>{

});
```

第二步：获取 POST 提交的数据，示例代码如下：

```
//获取用户名和密码信息
let username = req.body.username;
let password = req.body.password;
```

现在要以获取到的用户名和密码为条件进行查询操作了，示例代码如下：

```
// 用户登录接口
server.post('/login',(req,res)=>{
    //获取用户名和密码信息
    let username = req.body.username;
    let password = req.body.password;
    //以用户名和密码为条件进行查找，如果找到，则代表用户登录成功;否则代表登录失败
    let sql = 'SELECT id,username FROM xzqa_author WHERE username=? AND password=MD5(?)';
    pool.query(sql,[username,password],(err,results)=>{
        if(err) throw err;
        if(results.length == 1){
```

```
            res.send({message:'登录成功',co
de:1});
        } else {
            res.send({message:'登录失败',co
de:0});
        }
    });
});
```

最后在客户端接收服务器返回的信息，并且根据返回的信息进行不同的操作，示例代码如下：

```
this.axios.post('/login','username=' + thi
s.username + '&password=' + this.passwor
d).then(res=>{
    //登录成功
    if(res.data.code == 1){
        this.$router.push('/');
    } else {
        this.$messagebox("登录提示","对不
```

```
起,用户名或密码错误");
    }
});
```

完成正确登录操作后 接下来应该产生的效果:

自动跳转到主页 右上角的注册/登录 应该改变状态(登录成功后的展示) 但是:

---要想看到变化, 需要 Vuex 与 WebStorage 的配合使用!!!

The End~