# 暑期项目执行日志

# So-vitic-so.4.1

## 1.1搭建环境（2024年6月28日星期五）

 **打开终端（或 Anaconda Prompt）**： 在你的操作系统中打开终端（Windows 用户可以使用 Anaconda Prompt）。

 **创建新的 conda 环境**： 使用以下命令创建一个包含 Python 3.8 的新环境。可以根据需要替换 myenv38 为你想要的环境名称：

conda create --name myenv38 python=3.8

 **激活新环境**： 创建完成后，激活新环境：

conda activate myenv38

 **安装所需的包**： 在新环境中，你可以使用 conda 或 pip 安装项目所需的包。

conda install numpy pandas# 或者pip install requests

 **切换回原环境**： 当你需要切换回原来的 Python 3.9 环境时

conda deactivate

conda activate your\_original\_env\_name

# 将 your\_original\_env\_name 替换为你原来环境的名称

这样，你就可以在同一台电脑上同时使用 Python 3.8 和 3.9，而不需要替换任何环境。

## 1.2跟着up走但是有些不一定和他一样因为咱没显卡哈（6月29日星期六）

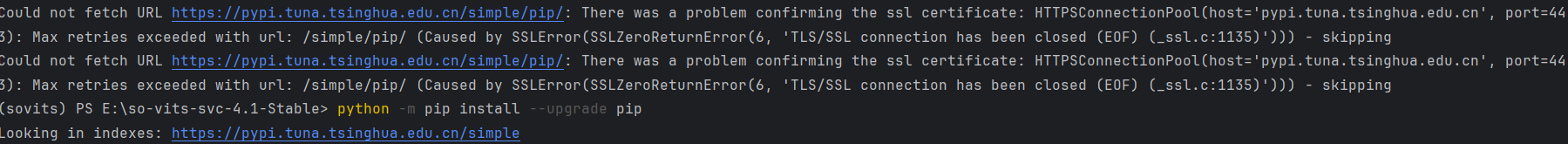
[so-vits-svc-Deployment-Documents/README\_zh\_CN.md at 4.1 · SUC-DriverOld/so-vits-svc-Deployment-Documents (github.com)](https://github.com/SUC-DriverOld/so-vits-svc-Deployment-Documents/blob/4.1/README_zh_CN.md)

安装pytorch相关包：

[【超详细教程】2024最新Pytorch安装教程（同时讲解安装CPU和GPU版本）-CSDN博客](https://blog.csdn.net/Little_Carter/article/details/135934842)

[没有独立显卡没有NVIDIA 如何安装pytorch\_没有独显可以安装pytorch吗-CSDN博客](https://blog.csdn.net/realgislander/article/details/127917670)（这是一篇没有独立显卡下载cpu版本的torch）

**1.3下载相关依赖**

****

在下载相关依赖时可能会有相关禁止，我觉得因为我将pip变为使用清华源然而我又使用外网所以会有禁止，我将外网关闭就好。

# Vue3项目实践

https://pan.baidu.com/s/1ZqjMxwZklZVYpB1ZbzEKKA&pwd=9987

首先我一直不明白别人老说要学会git仓库，这个是干嘛的怎么用有何益处？github我知道你可以保存自己的项目与他人共享，其中含有许多双赢的道理。

[06.git reset回退版本\_哔哩哔哩\_bilibili](https://www.bilibili.com/video/BV1HM411377j/?p=6&spm_id_from=pageDriver&vd_source=65efdae426f89a7b507fd7a3d43f4065)（这个up的专题讲得特别好）

对于这个只是简单了解一下如果要深入的掌握还是需要不断的实践操作才可以的。

## 小鲜兔电商实战

项目声明：本项目是在了解一定的vue3知识进行的新手入门项目

### 1.1第一步：创建项目框架+git仓库保存

### 1.2实战学习

**7月17日**

* 学习了利用pinia进行数据与方法的管理，使得发送请求的时候只进行一次减少了资源的浪费。但是在我第一次尝试中我并没有成功。我发现没有成功的原因就是没有下载pinia这个包而且这个也是要在main.js中去注册一下才能够展示出来。

在这个里我就有一个疑惑了，什么样的需要在main.js中注册才能使用其功能？

答：好吧内容挺多的我感觉我记下来也没有什么比较明确的好处。总结来说，任何需要在整个 Vue 应用中全局可用的插件、库或组件，都需要在 main.js 中进行注册。这样做可以确保这些功能在应用启动时就被初始化，并在各个组件中都能使用。

我还有一个疑惑就是store这个文件夹是作什么的？

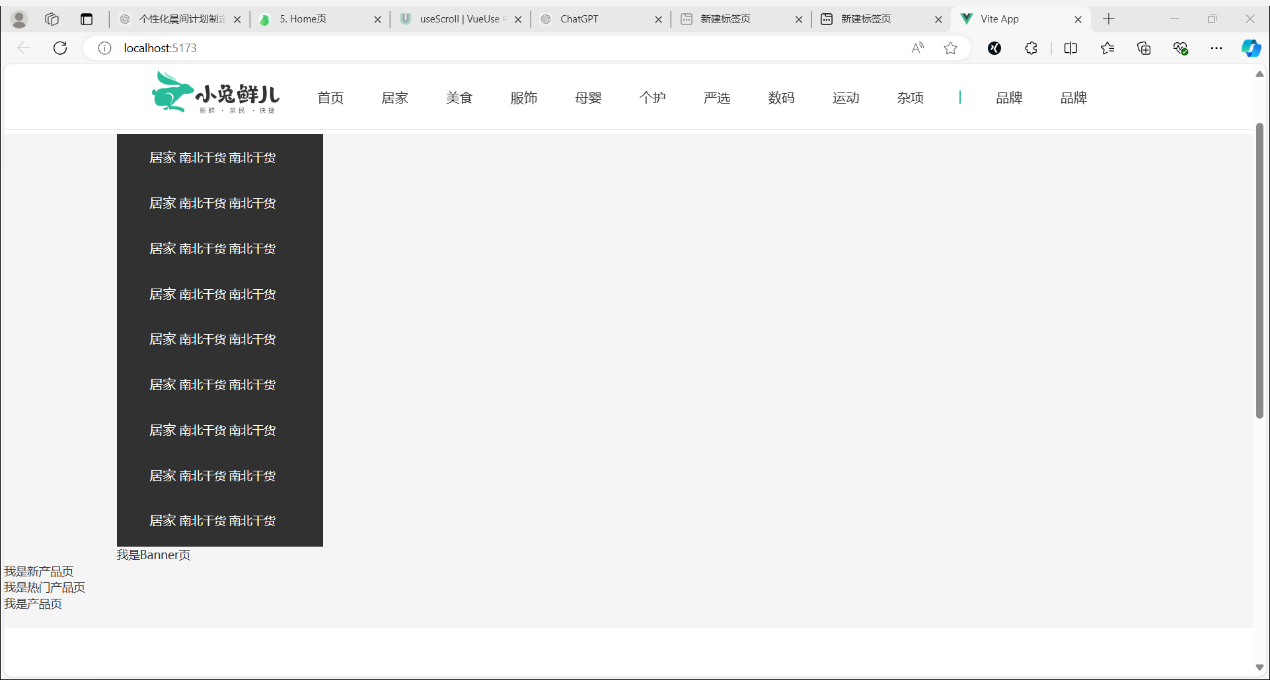
答：（好吧其实我是很懂）

 **定义 Store 的文件**： 每个 store 文件定义了一个独立的状态模块，包括状态（state）、动作（actions）、getters（计算属性）等。这些文件通常命名为与它们管理的状态相关的名称，例如 counter.js 或 user.js。

 **分模块管理**： 对于大型应用，将状态管理分为多个模块有助于更好地组织代码。例如，可以有一个 auth.js 文件管理用户认证相关的状态，一个 products.js 文件管理产品相关的状态等。

**7月18日（Home-整体结构拆分和分类）**

* 看我做成了这样的页面布局其实我页不知道这些是怎么做出来的？

****

答：请看代码注释

* 其实我一直很疑惑就是前端开发师怎么记住哪些颜色数值的，还有那些页面的大小怎么就直接写出来了。而且我就想不能够有这样的软件设计师直接在相关的画板中设计相关的UI界面，设计后软件直接给出相关的设计的布局代码。

答：

 大多数前端开发项目都会有设计稿，通常由设计师使用设计工具（如 Adobe XD、Sketch、Figma 等）制作。

 设计稿上会明确标注各个元素的尺寸、间距和位置，前端开发人员可以根据这些标注来设置相应的 CSS 属性。

 设计师通常会使用设计工具（如 Adobe XD、Sketch、Figma 等）创建设计稿。

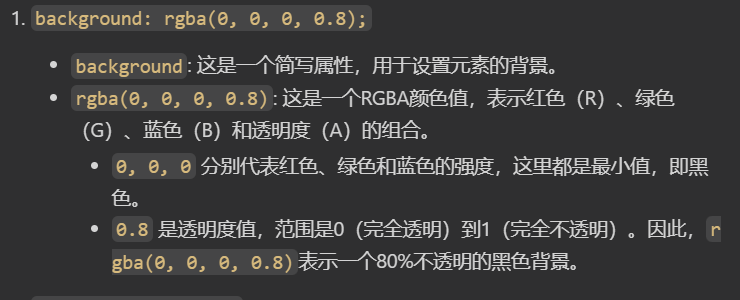
 这些工具中，设计师会使用特定的颜色并标注颜色值。

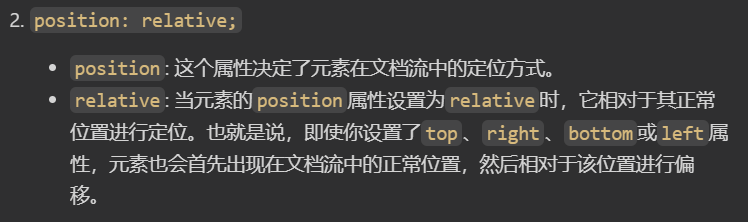
 前端开发人员可以直接从设计工具中获取颜色的十六进制值（如 #333）。

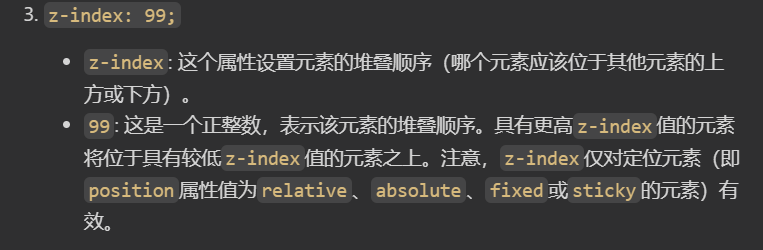
**Figma**

* **功能**：实时协作设计工具，支持团队成员同时在线编辑。提供丰富的 UI 元素和设计组件。
* **代码生成**：Figma 可以生成 CSS、iOS 和 Android 的代码片段。右键点击元素选择“Inspect”即可查看和复制相应的代码。
* **插件**：Figma 有许多插件可以生成代码或导出设计，如 Figma to HTML、Figma to React 等。

总结：现在有这样的UI设计工具并且能够直接给出代码的，我相信不久的将来这些类似的软件更够越来越完善完全体态员工，所以我打算的就是你就了解知道就好不必深究。





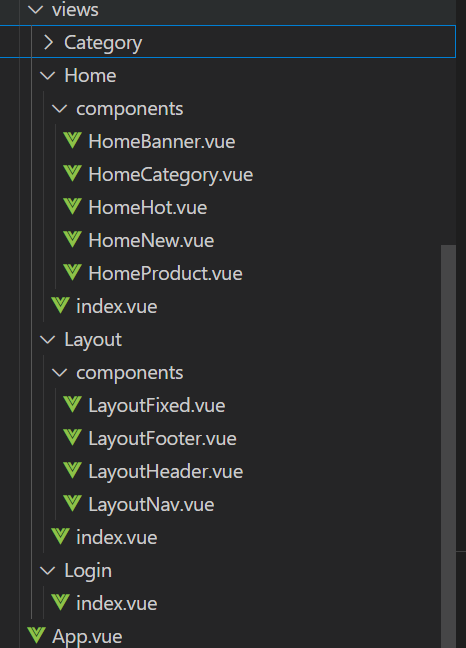


**7月21日**

* 今日遇到一个问题就是有些数据怎么显示都显示不来

答：后端的API接口不怎么稳定！

* 其实我有一个疑惑就是：这些布局到底是怎么进行管理起来的



答：这些布局页面我学习的是通过路由进行管理的

首先在router/index.js下构建路由结构：设置好一级路由、二级路由

在APP.vue中书写一级路由的出口，在一级路由的分类夹中的index.js中书写二级路由的出口

由此所有的布局页面就有条有理地展现出来了。

**7月22日**

* **（Home-banner轮番图功能实现）**

<template>

<div class="home-banner">

    <el-carousel height="500px" >

        <el-carousel-item v-for="item in bannerList" :key="item.id">

            <img :src="item.imgUrl" alt="">

        </el-carousel-item>

    </el-carousel>

</div>

</template>

这个挺有趣的但是那个布局我还是不是很懂，可是我觉得它完全可以被替代呀

 **<el-carousel>**：这是轮播图的容器组件。height="500px" 设置了轮播图的高度为 500 像素。

 **<el-carousel-item>**：这是轮播图中的每一个项目。使用 v-for 指令来遍历 bannerList 数组，为每个元素创建一个轮播图项。每个轮播图项的 key 属性使用元素的 id 进行标识，这是 Vue 在列表渲染中推荐的做法，以便在更新列表时能够高效地进行 DOM 操作。

 **<img>**：在每个轮播图项中显示一张图片。src 属性绑定到 item.imgUrl，显示来自 bannerList 数组中的图片。

**Element Plus**：Element Plus 是一个为 Vue 3 设计的基于 Element UI 的组件库，它提供了丰富的 UI 组件，帮助开发者快速构建美观的界面。你需要安装和引入 Element Plus 组件库。

* **Home-面板件封装**

这里主要涉及到的知识就是prop和插槽感觉好方面啊，但是我对这些知识还不是很会捏

* Giao git学得不精啊这个写错了不知道怎么退回我的项目



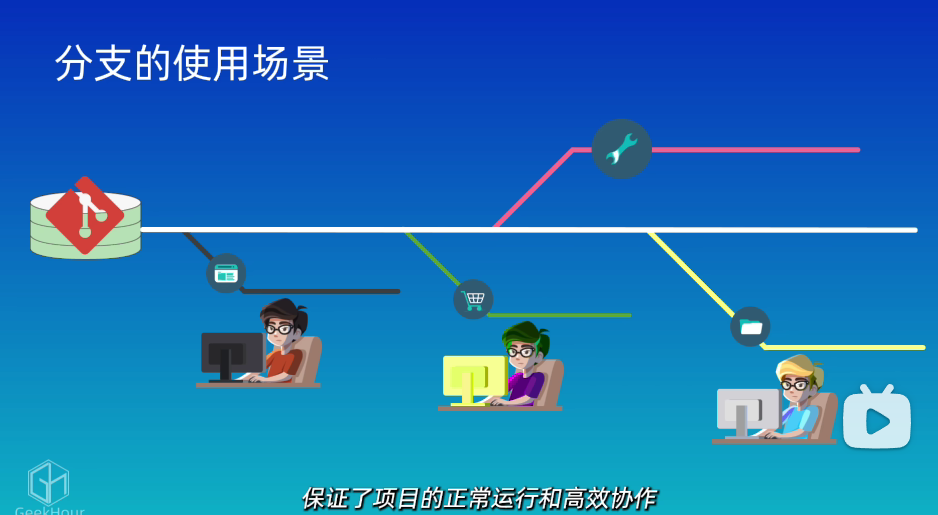
Git是什么：



日常生活中我们都会经历多个人修改某文件的情况，以修改竞赛文本为例，第一种当多人同时修改竞赛文本时之后还需要手工合成多人修改的部分。如果修改的版本过多可能会出现合成的版本不一致的问题，其他版本丢失等情况。第二种方式就是为了避免合成就一个个接替修改就会导致浪费时间等情况，那么我们急需版本控制系统对文件版本进行有效地管理。这里将要介绍的就是git。

Git采用的是分布式管理，每个人都有版本库，可在本地进行修改。修改后只需将仓库同步一下就好。使用它时可以知道谁在什么时间做了什么修改，也可以将文件恢复到某一版本，可为非常好用！（其实我觉得我讲得还不到位，我觉得我自己都没有理解透）

* 同时我还遇到了个问题就是git当中的分支到底是什么呀？



分支非常适合团队合作，每位成员都能够在自己相关的分支完成一定的开发测试任务，之后再汇总到主分支上。

* 还有一个我提交到仓库的提交信息搞错了，我该如何修改

### 修改最近一次提交的信息

1. 运行 git commit --amend 命令：

bash

复制代码

git commit --amend

1. 这将打开你默认的文本编辑器，显示最近一次提交的信息。在这里，你可以编辑提交信息。
2. 编辑提交信息后，保存并关闭编辑器。

* Home-新鲜好物和人气推荐实现

如果多个模块使用的模板相同，那我们就应该提取出需要修改的部分并进行封装。如果是类似于修改字符串这些简单的修改就可以使用props,如下所示：

<script setup>

defineProps({

    title:{

        type:String

    },

    subTitle:{

        type:String

    }

})

</script>

实际用法如下：

 <div class="head">

         <!-- 主标题和副标题 -->

        <h3>

          {{ title }}<small>{{ subTitle }}</small><!--这里用prop-->

        </h3>

此外为复杂的修改可以使用插槽。具体使用情况如下：

div class="home-panel">

    <div class="container">

      <div class="head">

         <!-- 主标题和副标题 -->

        <h3>

          {{ title }}<small>{{ subTitle }}</small><!--这里用prop-->

        </h3>

      </div>

      <!-- 主体内容区域 --><!--这里用插槽-->

      <div>

        <slot></slot>

      </div>

    </div>

  </div>

详细说明：

<div> 元素内有一个 <slot> 元素。这个 <slot> 元素表示一个默认插槽，当你在父组件中使用这个组件时，可以将一些内容传递给这个插槽。

<slot> 元素用于在组件中定义插槽（slot）。插槽是占位符，可以在使用组件时向组件内部传递内容。换句话说，插槽允许你在父组件中指定一部分内容，然后将其传递到子组件的特定位置。

Vue.js 插槽有几种类型：

1. **默认插槽**：如上例所示的基本插槽。
2. **具名插槽**：可以为插槽指定一个名称，在父组件中按名称传递内容。
3. **作用域插槽**：允许子组件向父组件传递数据，以便父组件能够自定义渲染内容。

#### 具名插槽示例

定义具名插槽：

<template>

<div>

<slot name="header"></slot>

<slot></slot> <!-- 默认插槽 -->

</div>

</template>

在父组件中使用具名插槽：

<template>

<div>

<MyComponent>

<template v-slot:header>

<h1>This is the header content</h1>

</template>

<p>This is the default slot content</p>

</MyComponent>

</div>

</template>

具名插槽的渲染结果：

<div>

<div>

<h1>This is the header content</h1>

<p>This is the default slot content</p>

</div>

</div>

**7月24日（23日没学习，不知道自己在干嘛了）**

* **Hom-图片懒加载指令实现（我如今也是学习到图片懒加载了）**

为什么要进行懒加载：

可能会遇到这种情况—网站资源过多，用户就不会浏览到下面，那么下面的资源数据就不必展现出来了。我觉得这个是减少浪费吧，具体是减少什么我就不知道了（减少请求），我只知道这是一个优化操作。其作用就是：图片通过懒加载优化手段可以做到只有进入视口区域才发送图片请求。

**懒加载的基本想法：**

使用vueUse判断图片是否进入视口——>如果图片进入视口，发送图片资源（img.src=url）请求。

1. 空指令实现：

app.directive('img-lazy', {

  mounted(el, binding) {

定义一个指令，其中传入的有指令名还有对象即该指令的具体操作

1. 在原代码中将src=“”改写成自己定义的指令

   <img v-img-lazy="item.picture" alt="" />

1. 指令的逻辑实现

app.directive('img-lazy', {

  mounted(el, binding) {

    // el: 指令所绑定的元素 img，DOM操作

    // binding:binding.value 指令等于括号后面绑定的表达式的值，图片地址url

    console.log(el,binding.value);

    useIntersectionObserver(

        el,//你要监听谁是否进入视口区域就把谁传过来

        ([{ isIntersecting }]) => {//isIntersecting 判断是否进入视口区域是一个布尔值

          console.log(isIntersecting);

          if (isIntersecting) {

            // 进入视口区域，把图片地址赋值给img的src属性

            el.src = binding.value; // 进入视口区域，把图片地址赋值给img的src属性

          }

        },

      )

  }

  })

**7月30日**

* **(通过插件的方法把懒加载指令封装为插件，main.js入口文件只需负责注册插件即可) 优化**

main.js文件本来主要是进行初始化的工作的，我们可在

src\directives定义全局指令将方法封装为插件：

export const lazyPlugin={

    install(app){//接下来是写该插件的逻辑

之后写好插件逻辑：

export const lazyPlugin={

    install(app){

        //懒加载指令逻辑

     //定义全局指令

// 使 v-focus 在所有组件中都可用

app.directive('img-lazy', {

    mounted(el, binding) {

      // el: 指令所绑定的元素 img，DOM操作

      // binding:binding.value 指令等于括号后面绑定的表达式的值，图片地址url

      console.log(el,binding.value);

      useIntersectionObserver(

          el,//你要监听谁是否进入视口区域就把谁传过来

          ([{ isIntersecting }]) => {//isIntersecting 判断是否进入视口区域是一个布尔值

            console.log(isIntersecting);

            if (isIntersecting) {

              // 进入视口区域，把图片地址赋值给img的src属性

              el.src = binding.value; // 进入视口区域，把图片地址赋值给img的src属性

            }

          },

        )

    }

    })

之后在mian.js注册+使用就好

//引入懒加载指令插件并注册

import { lazyPlugin } from './directives'

app.use(lazyPlugin)

* **useIntersectionObserver对于元素的监听是一直存在的，除非手动停止监听，存在内存浪费。优化**

解决：在监听的图片第一次完成加载之后就停止监听（可使用如下代码）

const { stop } = useIntersectionObserver(

target,

([{ isIntersecting }], observerElement) => {

targetIsVisible.value = isIntersecting

},

)

整体改为：

export const lazyPlugin={

    install(app){

        //懒加载指令逻辑

     //定义全局指令

// 使 v-focus 在所有组件中都可用

app.directive('img-lazy', {

    mounted(el, binding) {

      // el: 指令所绑定的元素 img，DOM操作

      // binding:binding.value 指令等于括号后面绑定的表达式的值，图片地址url

      console.log(el,binding.value);

      const { stop } = useIntersectionObserver(

          el,//你要监听谁是否进入视口区域就把谁传过来

          ([{ isIntersecting }]) => {//isIntersecting 判断是否进入视口区域是一个布尔值

            console.log(isIntersecting);

            if (isIntersecting) {

              // 进入视口区域，把图片地址赋值给img的src属性

              el.src = binding.value; // 进入视口区域，把图片地址赋值给img的src属性

              stop(); // 停止监听

            }

          },

        )

    }

    })

    }

}

* **Home-Product产品列表实现**

**方法：**

产品列表展示即每个模块的产品展示一部分，这样的话那静态模板是一样的呀（之前学习过了）——>封装接口——>获取数据渲染模板——>图片懒加载

**8月1日**

* **Home-Goodsltem组件封装**

在这里的核心思想是把要显示的数据对象设计为props参数，传入什么数据对象就显示什么数据对象。

我们今天主要学会的就是：抽象props参数，传入什么就显示什么

* **一级分类-整体认识和路由配置**

 <RouterLink :to="`/category/${item.id}`">{{ item.name }}</RouterLink>   <!--导航目录-->

在这条语句中是设置跳转路由的语句，to则是跳转到的路径，在这里用单号符一直不可以 只有在英文输入法下用这个符号才可以：`。

**8月2日**

* **一级分类-面包屑导航渲染**

基本思想：

准备组件模版 封装接口函数 调用接口获取数据 渲染模板

（使用路由参数）

不得不说vue真方便！面包屑导航都是自己帮写好的！

<div class="bread-container">

        <el-breadcrumb separato=">">

          <el-breadcrumb-item :to="{path:'/'}">首页</el-breadcrumb-item>

          <el-breadcrumb-item>居家</el-breadcrumb-item>

        </el-breadcrumb>

      </div>

**8月8日（感觉挺无助的整个项目都坏了）**

* **不知道弄了什么狗逼东西导致运行错误想着我使用git，那应该是可以回退到某一个版本的呀**

a.git log查看所有的提交信息

b. git checkout <commit-id>回退到某一个版本

可恶呀到这里再运行是发生错误的，问chat说是我的

**8月11日（继续完成8月8日没完成的）**

* 今天再试了回退到某个版本居然成功了，果然git好好用。我上次没成功不知道是什么原因，但是我都应该验证一下回退是否成功！

M 和 A 表示以下含义：

* M (Modified): 这些文件相对于你当前查看的提交来说已经被修改了，但修改内容还没有提交。
* A (Added): 这些文件已经被添加到暂存区（staging area），但还没有被提交。

如果你想永久回退到这个版本，可以使用 git reset --hard 命令

git reset --hard 385fadc0f846b2208d07ec0970af58636fc3ff5c

* 现在又来完成面包屑导航啦（由于8月2日的没完成，请往上看8月2日的相关内容）

**8月12日（小兔鲜项目彻底崩，昨天根本没完成，面包屑导航我就先跳过）**

* **获取id那个可恶啊！怎么就成功了呢，和之前写的是一样的呀**

**来感受一下带参的接口函数吧**

export function getCategoryId (id) {

    return request({

        url:'/category' ,  //接口发起 这个接口怎么就设定成这样了？md在那个面包屑导航直接用这个接口就好

        params: {

            id // 通过 params 传递查询参数 id

        }

    })

}

获取数据情况

//获取数据

const categoryData = ref({})

const route = useRoute()

const categoryId = async () =>{

  const res =await getCategoryId(route.params.id)

  categoryData.value = res.result

}

onMounted(()=>categoryId())

以下实例对象专门来获取id参数

const route = useRoute()

* **一级分类-轮播图功能实现**

思想：

改造先前的接口 迁移首页轮播图逻辑

（适配参数）

①改造为传参接口

export function getBannerAPI(params={}){

  const {distributionSite='1'} = params;

    return httpInstance({

        url:'/home/banner',

        params:{

          distributionSite

        }

    })

}

②将改造的接口在分类中进行应用

const bannerList = ref([])

const getBanner = async () => {

    const res = await getBannerAPI({

      distributionSite:'2'

    })

    console.log(res)

    bannerList.value = res.result

}

onMounted(() => getBanner())

说实话我是有点不理解为啥获取的数据都一样还要为接口设置不同的参数呀：

**理解 distributionSite 参数的作用**

* **相同接口，不同数据**：在很多 API 设计中，一个接口可以根据传递的不同参数返回不同的数据。distributionSite 参数可能被用于在服务器端区分不同页面所需的内容。例如，distributionSite='1' 可能返回首页的轮播图，而 distributionSite='2' 可能返回“居家”页面的轮播图。
* **灵活性**：这种设计方式的好处是，前端开发者只需要记住一个接口，而不是多个接口，参数的不同会自动决定返回的数据内容。这种方法也使得 API 设计更具扩展性，可以通过增加参数值来支持更多页面或场景。

**举个例子**

假设服务器端根据 distributionSite 的值返回不同的轮播图数据：

* 当 distributionSite='1' 时，返回首页的轮播图数据，例如广告、促销信息等。
* 当 distributionSite='2' 时，返回居家页面的轮播图数据，可能是家居用品的推荐。

**为什么设置不同参数**

通过设置不同的 distributionSite 参数，你能够复用同一个接口获取不同页面的定制化数据，而不是硬编码或为每个页面设计不同的接口。这种方式不仅提高了代码的可复用性，还保持了 API 的一致性。

**8月13日**

* **那个一级分类列表有点垃圾那每次点击要刷新一下才可以（我要进行一下优化）**

原因：

可能是因为数据获取和渲染的时间不同步，导致 UI 没有及时更新

//获取数据

const categoryData = ref({})

const route = useRoute()

const categoryId = async () =>{

  const res =await getCategoryId(route.params.id)

  categoryData.value = res.result

}

onMounted(()=>categoryId())

// 监听路由参数 id 的变化

watch(() => route.params.id, (newId) => {

  if (newId) {

    categoryId() // 直接调用获取数据的函数

  }

})

**watch 函数**：监听 route.params.id 的变化，当分类 ID 变化时，重新调用 categoryId 函数，以获取新的分类数据并更新 categoryData。

* **正式开始一级分类-激活状态显示**

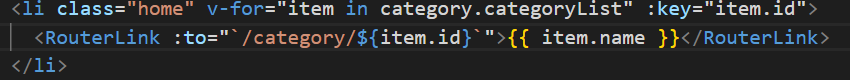
****

**在点击的部分能显示激活**

**以便用户知道前所处位置**

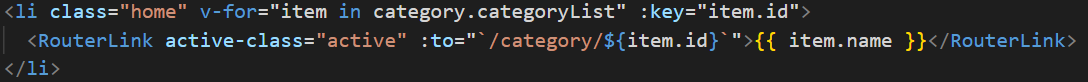
****

**方法：**

****

**原本 <RouterLink自带激活功能**

**但是不够长久，干脆写个属性在样式中定义**

****

* **一级分类-分类列表的渲染**

其实在这一部分就没有什么困难的，在这里我觉得最重要的就是重用之前封装的组件

之前封装的组件：

在 '@/views/Home/components/Goodsitem.vue'中Goodsitem文件有这样的组件：

<template>

    <RouterLink to="/" class="goods-item">

            <img v-img-lazy="good.picture" alt="" />

            <p class="name ellipsis">{{ good.name }}</p>

            <p class="desc ellipsis">{{ good.desc }}</p>

            <p class="price">￥{{ good.price }}</p>

    </RouterLink>

</template>

我们在别的页面也想要这个组件直接引用就好

import Goodsitem from '@/views/Home/components/Goodsitem.vue'

 <div class="sub-list">

      <h3>全部分类</h3>

      <ul>

        <li v-for="i in categoryData.children" :key="i.id">

          <RouterLink to="/">

            <img :src="i.picture"/>

            <p>{{ i.name }}</p>

          </RouterLink>

        </li>

      </ul>

    </div>

    <div class="ref-goods" v-for="item in categoryData.children" :key="item.id">

      <div class="head">

       <h3>-{{ item.name}}-</h3>

      </div>

      <div class="body">

        <Goodsitem v-for="good in

item.goods" :good="good" :key="good.id"/><!--之前封装的主键-->

      </div>

    </div>

**8月14日**

* **我又产生了这样一个迷惑就是这些页面怎么不像jsp学习的那样如果是页面展示都要自己创建一个网页来，但这个vue好神奇它怎么页面衔接展示的**

**首先咱先看看路由**

import { createRouter, createWebHistory } from 'vue-router'

import Login from '@/views/Login/index.vue'

import Layout from '@/views/Layout/index.vue'

import Category from '@/views/Category/index.vue'

import Home from '@/views/Home/index.vue'

const router = createRouter({

  history: createWebHistory(import.meta.env.BASE\_URL),

  //path和componet对应关系的位置

  routes: [

    {

      path: '/',

      name: 'Layout',

      component: Layout,

      children:[

        {

          path: '', //默认二级路由为空？

          name:'Home',

          component: Home

        },

        {

          path:'Category/:id',

          name:'Category',

          component: Category

        }

      ]

    },

    {

      path: '/login',

      name: 'Login',

      component: Login

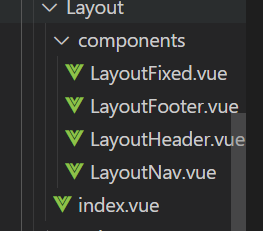
    }

  ]

})

export default router

Layout是一级路由然后Home和Category（你看那个Category后有个动态id它有许多个路由）是二级路由



这个是Layout的布局，其中index.vue这个引索是决定谁可以展现在页面中的：

<template>

  <LayoutFixe />

  <LayoutNav />

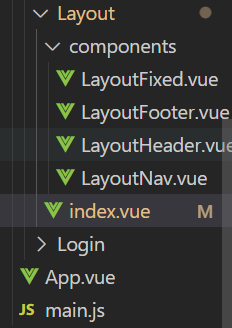
  <LayoutHeader />

  <RouterView /><!--以及路由入口吧-->

  <LayoutFooter />

</template>

你在一级路由中点击并且可以跳转的到其他页面的“点击”这个就是二级路由



这里的APP.vue是掌管了总页面谁可以展示

<template>

 <!--一级路由出口-->

  <RouterView />

</template>

直接这样简单明了

这个是Layout的布局，其中index.vue这个引索是决定谁可以展现在Layout页面中的：

<template>

  <LayoutFixe />

  <LayoutNav />

  <LayoutHeader />

  <RouterView /><!—二及路由出口吧-->

  <LayoutFooter />

</template>

* **一级分类-路由缓存问题**

了解一些小知识：

* **生命周期钩子**：

在 Vue.js 中，生命周期钩子是指在组件的不同阶段自动调用的一组函数。这些钩子允许你在组件的创建、更新和销毁过程的特定时刻执行代码。通过这些钩子，你可以更好地控制组件的行为。

以下是 Vue.js 中常用的生命周期钩子及其触发时机：

1. **beforeCreate**：
   * 在实例初始化之后，数据观测 (data observer) 和事件配置 (event/watcher) 之前被调用。
   * 此时，组件实例还没有被完全初始化，数据、计算属性、方法都还不可用。
2. **created**：
   * 在实例创建完成后被调用。
   * 此时，实例已经完成了数据观测、属性和方法的初始化，但还未开始编译模板。
   * 如果需要在组件实例被完全初始化后立即执行一些逻辑，可以在这个钩子中处理。
3. **beforeMount**：
   * 在挂载开始之前被调用。
   * 此时模板已经编译好了，但还没有挂载到 DOM 上。
   * 你可以在这里做一些在 DOM 挂载前的最后准备工作。
4. **mounted**：
   * 在挂载完成后调用。
   * 此时组件的 DOM 元素已经被插入文档中，可以在这里进行依赖 DOM 操作的逻辑。
5. **beforeUpdate**：
   * 当响应式数据更新，且在虚拟 DOM 重新渲染和打补丁之前被调用。
   * 你可以在这个钩子中读取旧的状态和即将更新的状态。
6. **updated**：
   * 在由于数据变化导致的虚拟 DOM 重新渲染和打补丁之后调用。
   * 此时组件的 DOM 已经更新，你可以在这里进行依赖更新后 DOM 状态的操作。
7. **beforeDestroy**：
   * 在实例销毁之前调用。
   * 此时实例仍然完全可用，你可以在这里进行清理工作，例如移除事件监听器或清除计时器。

**什么是路由缓存问题：**

使用带参数的路由时需要注意的是，当用户从/use/category?id=01导航到

**destroyed**：

* 在实例销毁后调用。
* 调用后，所有的事件监听器会被移除，子实例也会被销毁，组件从 DOM 中移除

（感觉上面的看一眼了解一个大概就可以了）

onMounted(() => getBanner())

在 Vue 3 中，onMounted 是一种组合式 API（Composition API）中的生命周期钩子，它的功能与 Vue 2 中的 mounted 钩子相同。onMounted 用于在组件挂载到 DOM 之后执行某些操作。具体来说，当组件的 DOM 元素已经被插入文档后，onMounted 钩子会被调用。

/use/category?id=02时，相同的组件实例将被重复使用。因为两个路由都渲染同个组件，比起销毁再创建，复用则显得更加高效。不过，这也意味着组件的生命周期钩子不会被调用。简单来说就是一级分类的切换正好满足上面的条件，组件实例复用，导致分类数据无法更新。

**解决问题：**

**解决这个问题有两种方法，1.让组件实例不复用，强制销毁重建2.、**

**2.监听路由变化，变化之后执行数据更新操作（这个我们做过了就是用**

**Watch进行监听）**

**使用方法一：**

**给<Router-view添加key**

 <!--添加key 破坏复用机制 强制销毁重建-->

  <RouterView :key="$route.fullPath" />

在Layout中的index.vue添加，但是这个方法回导致同样的资源重复申请（它的申请是整个页面的申请）

**8月15日**

* **一级分类-使用逻辑函数拆分业务**

小理解：

基于逻辑函数拆分业务是指把同一个组件中独立的业务代码通过函数做封装处理，提升代码的可维护性

实施步骤：

1. 按照业务声明以`use`打头的逻辑函数
2. 把独立的业务逻辑封装到各个函数内部
3. 函数内部把组件中需要用到的数据或方法return出去
4. 在组件中调用函数把数据或者方法组合回来使用

* **二级分类-整体认识和路由配置**

思路：

创建路由组件 配置路由关系 修改模板实现跳转

有一个需要注意的点就是在配置路由结束后需要重启项目才能达到想要的效果。

* **二级分类-面包屑导航实现（感觉蛮简单的照之前做的就好了）**

思想：

**（很easy自己操作一下就可以！可惜今天没做完，明天一定做完）**

封装接口调用接口渲染 测试跳转

**8月16日（现在是8:16一定要在今天完成小兔鲜项目）**

* 二级分类基础商品列表展示
* 基础列表渲染
* 实现筛选功能
* 实现无线加载功能

思路： 使用elementPlus提供的 v-infinite-scroll指令监听是否满足触底条件，满足加载条件时让页面参数加一获取下一页数据，做新老数据拼接渲染。

配置指令（v-infinite-scroll）——>页数加-获取下一页数据

——>老数据和新数据并接——>加载数据完毕

发现个问题这个后端有问题在实现这个功能的时候是展示出了所有的商品并不按分类来（我觉得是接口有问题你看那个接口哪里有分类数据的id啊）

* 二级分类-定制路由scrollBehavior

直接使用这个：scrollBehavior

当跳级到上一级路由页面会回到顶部

* 详情页-整体认识和路由配置

创建详情组件——>绑定路由关系——>绑定模板测试跳转

* 详情页-基础数据渲染

封装接口——>调用获取数据——>渲染模板

主要需要注意的就是

渲染模板时遇到对象的多层属性访问可能出现什么问题？

访问的属性为空！

Uncaught (in promise) TypeError: Cannot read properties of undefined (reading 'name')

解决方法：1.可选链 2.v-if 控制渲染

* 详情页-热榜区域的实现

封装Hot热榜组件 ——> 获取渲染基础数据 ——> 适配不同标题Title

——>适配不同列表内容

**8月18日**

* 适配不同的标题
* 设计props参数 适配不同的title和数据

const props=defineProps({

    hotType:{

        type:Number

    }

})

* 编写标题对象

const Typemap={

  1:'24小时热榜',

  2:'周榜热榜',

}

* 编写可进行判断的标题对象

const title = computed(()=>Typemap[props.hotType])

* 详情页-图片预览组件封装
* 通过小图切换大图实现

思路：维护一个数组图片列表，鼠标划入小图记录当前小图标下值，通过下标在数组中取对应图片，显示到大图位置

准备组件静态模版——>为小图绑定事件——>通过下标切换到大图显示

记录当前激活下标值

——>通过下标实现激活状态显示

* 为小图绑定事件记录当前激活下标值：

<!-- 小图列表 -->

    <ul class="small">

      <li v-for="(img, i) in imageList" :key="i" @mouseenter="enterhandler(i)">   <!--给小图绑定鼠标事件-->

        <img :src="img" alt="" />

      </li>

    </ul>

* 对激活事件进行编写

//小图切换大图显示

const activeIndex=ref(0)

const enterhandler =(i) =>{

  activeIndex.value = i

}

* 通过小图下标显示大图

 <!-- 左侧大图-->

    <div class="middle" ref="target">

      <img :src="imageList[activeIndex]" alt="" />

* 激活状态一直显示

.small {

    width: 80px;

    li {

      width: 68px;

      height: 68px;

      margin-left: 12px;

      margin-bottom: 15px;

      cursor: pointer;

      &:hover,

      &.active {

        border: 2px solid $xtxColor;

      }

    }

  }

主要是鼠标点击哪个小图就显示哪个小图为激活状态，这时候需要动态类名：

<li v-for="(img, i) in imageList" :key="i" @mouseenter="enterhandler(i)" :class="{active: i===activeIndex}">

**8月19日**

* 图片放大功能
* 左侧滑块跟随鼠标移动

思路：

获取到当前的鼠标在盒子内的相对位置（useMouselnElement）,控制滑块跟随鼠标移动（left/top）

* 鼠标的距离盒子相对位置

[useMouseInElement | VueUse 中文网 (nodejs.cn)](https://vueuse.nodejs.cn/core/useMouseInElement/)

* 控制滑块的跟随移动

1. 有效移动范围内的计算逻辑

横向：

100<elementX<300,left=elementX-小滑块宽度的一半

纵向：

100<elementY<300,top=elementY-小滑块高度的一半

1. 边界值控制距离(这些信息都是教程里面根据自己盒子来进行计算的)

横向：

elementX>300 left=200 elmenttX<100 left=0

纵向：

elementY >300 top=200 elementY<100 top=0

* 大图效果

<div class="large" :style="[

      {

        backgroundImage: `url(${imageList[activeIndex]})`,

        backgroundPositionX: `${positionX}px`,

        backgroundPositionY: `${positionY}px`,

      },

    ]" v-show ="!isOutside" ></div>

 background-repeat: no-repeat;

    // 背景图:盒子的大小 = 2:1  将来控制背景图的移动来实现放大的效果查看 background-position

    background-size: 800px 800px;

这里\*2是因为图片放大两倍了 它的相对位置也放大两倍

//控制大图显示

  positionX.value=-left.value\*2

  positionY.value=-top.value\*2

**8月20日**

* **组件props适配**

思路：在<ImageView/>这个组件中图片的url不能够写死，把它设置为一个可以进行传参的组件：

//props适配图片列表

defineProps({//利用defineProps进行props适配

   imageList:{//props名称

    type:Array,//类型

    required:true,//是否必传

    default:()=>[]//默认值，工厂函数将其导出为数组。

   }

})

利用组件的时候可以这样写：

  <ImageView :image-list="goods.mainPictures"/>

思路总结：

1. 封装复杂交互组件的通用思路：

功能拆解 ——> 寻找核心实现思路 ——> 寻找关键技术 ——>

逐个实现，逐个验证，最后优化

1. 图片预览组件的封装逻辑

小图切换大图显示 ——> 获取鼠标相对位移 ——> useMouselnElement

(left ,top)

——>验证优化

* **详情页-SKU组件（8月21日完善的）**

**问：**在实际工作中，经常会经常遇到别人写好的组件，熟系一个三方组件，首先重点看什么？

**答：** props（入参）和emlit（出参） ,props决定了当前组件接收什么数据，emit决定了会产出什么数据

**验证组件是否成功使用：**

传入必要数据是否交互功能正常 ——> 点击选择规格，是否正常产出数据

* 封装SKU组件（没学完有点难，还是下次再学吧）

初始化规格渲染——> 点击规格更新选中状态——>点击规格更新禁用状态

——> 产出选择的SKU数据

**8月21日**

* 详情页-通用组件统一注册全局

背景：components 目录下有可能还会有很多其他通用型组件,有可能在多个业务模块中共享，所有统一进行全局组件注册比较好

思路：

Components插件

把components目录下的所有组件进行全局注册

Main.js(注册插件)

//把component中的所有组件都进行全局化注册

//通过插件的方式

import ImageView from './ImageView.vue'

import SKU from './SKU.vue'

export const componentPlugin = {

    install(app)//固定方法，固定参数

    {

        //app是createApp返回的实例

        //app.component('组件名','组件配置参数')

        app.component('ImageView', ImageView)

        app.component('SKU', SKU)

    }

}

//引入全局组件插件并注册

import { componentPlugin } from './components'

 app.use(componentPlugin)

**8月22日**

* 登录-整体认识和路由配置

这个是其他的路由形式 好像使用<Routerlin也是可以的

<li><a href="javascript:;" @click="$router.push('/login')">请先登录</a></li><!--编写登录事件-->

**8月24日**

* 表单校验的实现

作用：前端提前校验可以省省去一些错误的请求提交，为后端节省压力

表单数据——>前端校验（过滤错误请求）——>后端查询是否匹配

ElementPlus 表单组件内配置了表单校验功能，只需要按照组件要求配置必要参数即可 ：[Element - 网站快速成型工具](https://element.eleme.cn/#/zh-CN)

el-input(双向绑定表单数据)

el-form-item（绑定使用规则字段）

el-form (绑定表单对象和规则对象)

当功能很复杂时，通过多个组件各自负责某个小功能，再组合成一个大功能是组件设计中常用的方法

表单校验步骤：

1.按照接口字段准备表单对象并绑定 2.按照产品要求准备规则对象并绑定

3.指定表单域的校验字段名 4.把表单对象进行双向绑定

我们以下面的要求为例：

用户名：不能为空 字段名为account

密码：不能为空且为6-14个字符，字段名为 password

同意协议：必选，字段名为 agree（还没做完，我出错了出错是因为required写错啦）

**8月28日**

* **自定义校验规则（是否同意协议的校验）**

ElementPlus表单组件内置了初始的校验配置，应付简单的校验只需要通过配置即可，如果想要定制一些特殊的校验需求，可以使用自定义校验规则，格式如下：

validator:(rule,val,callback)=>{

  //自定义校验逻辑

  //val:当前输入的数据

  //callback:校验处理函数 校验通过调用

}

校验逻辑：

如果勾选了协议框，通过逻辑，如果没有勾选，不通过校验

（我觉得这个自定定义校验有问题，为啥一来就同意勾选）

* **整个表单的内容验证**

点击登入时需要对所有需要校验的表单进行统一校验

formEl.validate((valid) => {

     if(valid){

      console.log('登录成功')

     }else{

      console.log('登录失败')

      return false

     }

})

1.获取form组件实例 ——> 2.调用实例方法

* 登录-基础登录业务实现

表单校验——>封装登录接口——>调用登录接口

登录成功后逻辑处理 登录失败后业务逻辑（注册页面）

md接口坏了，为了进度我只可以自己写接口了（md我来搞后端了）

（但是这个又来学后端感觉非常不好我们应该先学完这个前端再学后端吧）

**8月31日（最终还是来学后端了，感觉自己是一个全栈技术人了）**

[gRPC[Nodejs]构建现代API和微服务gRPC[Nodejs] MasterClass: Build Modern API&Microservices\_哔哩哔哩\_bilibili](https://www.bilibili.com/video/BV1Eq4y1z7zB/?spm_id_from=333.337.search-card.all.click&vd_source=65efdae426f89a7b507fd7a3d43f4065)

**9月10日**

**Mysql安装以及可视化配置**

**注意：不要安装在c盘**

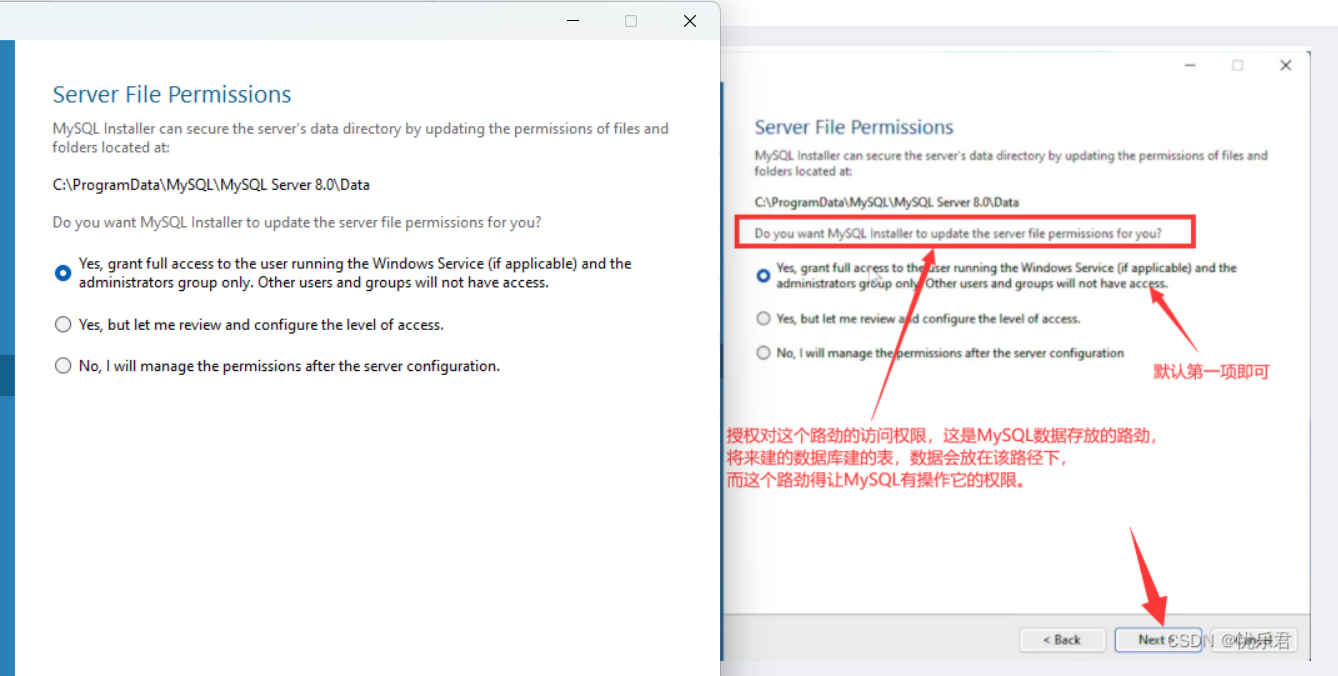
[**https://class.m.imooc.com/qadetail?qid=296837**](https://class.m.imooc.com/qadetail?qid=296837)

[MySQL安装教程（详细版）\_mysql安装教程8.0.36-CSDN博客](https://blog.csdn.net/m0_71422677/article/details/136007088)

**9月20日**

1. **删除并重装mysql**

[**教你彻底卸载MySQL 并重装（保姆级教程 ）\_mysql怎么卸载干净重装-CSDN博客**](https://blog.csdn.net/weixin_56952690/article/details/129678685)

****