Vscode安装：<https://code.visualstudio.com/>

****Open in Browser 插件 可以右键直接打开浏览器****

./表示当前目录 ../表示父级目录

****div中图片文字并列，处于同一水平线对齐，图片属性：vertical-align:middle****

**1.用 var app = new Vue({.....})**

**在app外面修改app中的data数据 --> app.$data.content = “ ”**

**在app方法里面修改: this.content**

**2.v-for指令：数组可以放在data的List中 : List : [ 1 , 2 ,3 ]**

**< li v-for = “(item,index) of List”:key=”写个唯一值”> {{item}} </li>**

**// ：key最好不写index，写item.index或者item.id等**

1. **v-on:click简写成 @click 例: @click=”fun1”**

**方法写在app实例methods中 例 methods:{**

**fun1:function(){**

**......**

**}**

**}**

1. **v-model 双向数据绑定 例: v-model = “****inputVal”**

**然后将inputVal放入app实例的data中**

**v-model原理：**

## **一、访问器属性**

[Object.defineProperty()函数](http://javascript.ruanyifeng.com/stdlib/attributes.html" \l "toc2" \t "https://www.cnblogs.com/zhuzhenwei918/p/_blank)可以定义对象的属性相关描述符， 其中的set和get函数对于完成数据双向绑定起到了至关重要的作用，

var obj = {

foo: 'foo'

}

Object.defineProperty(obj, 'foo', {

get: function () {

console.log('将要读取obj.foo属性');

},

set: function (newVal) {

console.log('当前值为', newVal);

}

});

obj.foo; // 将要读取obj.foo属性

obj.foo = 'name'; // 当前值为 name

## **二、简单的数据双向绑定实现方法**

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

输入：<span id="textSpan"></span>

<script>

var obj = {},

textInput = document.querySelector('#textInput'),

textSpan = document.querySelector('#textSpan');

Object.defineProperty(obj, 'foo', {

set: function (newValue) {

textInput.value = newValue;

textSpan.innerHTML = newValue;

}

});

textInput.addEventListener('keyup', function (e) {

obj.foo = e.target.value;

});

</script>

**MVVM模式 ：model数据层，页面中所需的数据，view界面层，viewModel逻辑层**

**创建全局组件： Vue.component(“TodoItem”,{**

**//这里的component不加s**

**data: function(){**

**return {**

**val: 123 //data是一个函数**

**}**

**}**

**props:[“item”],**

**template:”<li>{{item}}</li>”**

**})**

**使用组件的时候 TodoTiem要写成：todo-item**

**创建局部组件: var todoItem = {**

**data : function(){**

**return {**

**val : 456**

**}**

**}**

**props: [“item”],**

**template:”<li>{{item}}</li>”**

**}**

**还需要将局部组件注册到app实例中:**

**var app = new Vue({**

**......**

**component**s**:{ //注意要写s**

**todoItem**

**},**

**....**

**})**

****组件传值问题：****

**父组件向子组件传值：在父组件上写 :item=”item” 子组件中：props:[“item”]**

**子组件向父组件传值：子组件上定义一个事件如@click，然后在子组件的方法中写: this.$emit(“fufun”,this.index) 父组件上写事件: @fufun = “shijian” 然后在父组件的方法中写: shijian:function(index){......}**

**销毁实例：如：app.$destroy() 就会销毁app这个之前创建出来的实例**

**父组件上的方法：<todolist @click.native=”todofun”></todolist>**

**然后将todofun写在父组件的methods中**

**生命周期函数：直接写在实例中，不写在methods里，**

**如：mounted:function(){........}**

**beforeCreate：表示实例完全创建出来之前，会执行，data和methods还未初始化**

**created：data和methods都已经初始化好了，如果要调用methods中的方法，或者操作data中的数据，最早，只能在created中操作**

**beforeMount：模板已经在内存中编辑完成，未渲染到页面中，页面中的元素还未被替换过来，只是之前写的一些模板字符串**

**mounted：内存中的模板已经真实的挂载到了页面中，用户可以看到渲染好的页面了**

**beforeUpdate：改变data****中的数据的时候触发（app.shuju = “shuju2”），页面显示的数据是旧的，此时data的数据是新的，页面尚未和最新的数据保持同步**

**updated：改变data中的数据的时候触发（app.shuju = “shuju2”），页面和data数据已经保持同步了，都是最新的**

**beforeDestroy：实例销毁之前的时候触发( app.$destroy( ) )**

**destroyed：实例销毁后触发**

**v-text = “val” v-html = “val2”，v-html可以写标签，和{{val3}}一样都可以写js表达式: {{val + ‘3’}}**

****计算属性：要计算的属性，不能定义在data中，否则会报错****

**computed:{**

**val : function(){ //val是属性值：{{val}}，但是不能写在data中，不是方法**

**return this.val1 + this.val2**

**}**

**}**

**computed和 methods中的方法相比 ，**computed是定义属性**，**methods是定义方法**。计算属性存在缓存，只要需要的data值不变，就不会重新计算**

**升级：**

**computed:{**

**val:{**

**get:function(){**

**return this.val1 + this.val2**

**},**

**set:function(val3){ // 监视当前属性值的变化，当属性值发生变化时执行，更新相关的属性数据**

**.........**

**}**

**}**

**}**

**computed中的get和set方法：当我们改变了val的值就可以在set中操作改变后的值val3**

**watch监听**：需要监听的数据，要在data中定义****

**watch:{**

**val: function(){**

**console.log(this.val)**

**}**

**}**

**当val值发生改变的时候，watch监听的方法就会执行,优先使用computed,**

**watch:**

无缓存性，页面重新渲染时值不变化也会执行

应用：监听props，$emit或本组件的值执行异步操作

**样式绑定：**

1. **class的对象绑定 :class = “{divStyle: isdivStyle}”将divStyle作为类写在样式里: .divStyle{......} isdivStyle写在data中值为true或false**
2. **class数组的方式 :class = “[divStyle]”，data中divStyle :“divStyle”，将类divStyle的样式写出来**

**内联样式：**

**:style = “styleObj”，将styleObj写在data中: styleObj:{fontSize:”200px”}**

**条件渲染：**

**v-if和v-show: v-show是通过display：none来显示隐藏的，v-if是通过添加删除dom元素来显示隐藏的**

**v-if和v-else的使用：如：**

**<div v-if=”show”>{{....}}</div>**

**<div v-else>{{....}}</div> //标签要紧接着下个标签**

**标签上加一个 key=”随意取名字” 可以表示该标签的唯一性，就不会存在标签复用的问题。**

**template标签：做占位符使用，包裹其它标签使用，渲染页面的时候不会渲染出来**

**往实例data的一个对象(对象名obj)上添加数据**

**Vue.set(app.obj,”user”,”yang”)或者：**

**app.$set(app.obj,”user”,”yang”)**

**改变数组的方式和上一样，如：Vue.set(app.arr, 1 ,5)将下标1的值改为5**

****is：****

**当我们把结构标签的子标签设置为组件的时候会出现bug，我们需要写明组件的来历：如：li作为组件row，<li is=”row”></li>**

****ref:**  //通过ref操作dom元素**

**比如： <div ref = “divref”>hello</div>**

**可以获取此dom元素的值：this.$refs.divref.innerHTML**

**单向数据流概念：父组件向子组件传递一个数据，子组件只能去使用此数据，当子组件改变该数据的时候，会出警告。原因是父组件传递的数据可能是引用型数据，并且传递给了其他的组件，如果某个子组件改变了此数据，就会影响到别的位置，因此会出警告。解决方式：我在子组件data中存一个值，并将父组件传递过来的值赋值给他，在子组件上显示自己data中的值，并修改他，就不会出现警告,如下：**

props:['listShop'],

**data**(){

**return**{

listShopChild:**this**.listShop

}

},

created(){

**this**.listShopChild=30

}

****克隆的问题：****

**数组深度克隆:**

**适用于对象数组克隆简单方法：**

**扩展运算符：**

**克隆数组a: [...a]**

**克隆对象a: {...a}**

**var** x = [1,2,3];**var** y = [];**for** (**var** i = 0; i < x.length; i++) {

y[i]=x[i];

}console.log(y); *//[1,2,3]*

y.push(4);console.log(y); *//[1,2,3,4]*console.log(x); *//[1,2,3]*

**对象深度克隆:**

**var** x = {a:1,b:2};**var** y = {};**for**(**var** i **in** x){

y[i] = x[i];

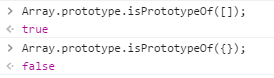
}console.log(y); *//Object {a: 1, b: 2}*

y.c = 3;console.log(y); *//Object {a: 1, b: 2, c: 3}*console.log(x); *//Object {a: 1, b: 2}*

****判断一个对象是不是数组：****

**1、从原型入手，Array.prototype.isPrototypeOf(obj);**

利用isPrototypeOf()方法，判定Array是不是在obj的原型链中，如果是，则返回true,否则false。

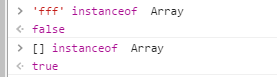


**2、也可以从构造函数入手，obj instanceof Array**

先说说 typeof 和 instanceof 的区别？

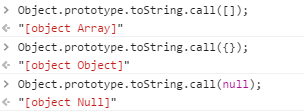
两者都可以用来判断变量，typeof会返回基本类型，如文章开头，我们很简单可以用

typeof a != 'undefined' 判断a变量存在。而instanceof只会返回一个布尔值，那么我们试试，结果如下:



**3、根据对象的class属性(类属性)，跨原型链调用toString()方法。（推荐）**

js中提供了，调用对象原型中的toString方法， Object.prototype.toString.call(obj)；因为很多对象继承的toString（）方法被重写了，为了能够调用正确的toString（）版本，也就是最原始的版本。可以使用Function.call()的方法，其中call可以这么理解，相当于obj去借用这个 Object.prototype.toString();

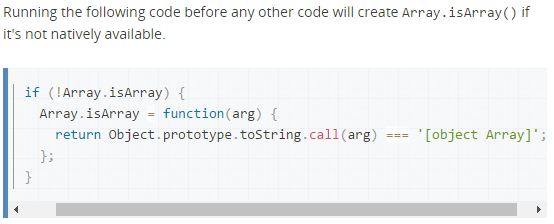


**4、Array.isArray()方法。**

其实js已经提供了判断是不是数组类型的方法，如下：

|  |  |
| --- | --- |
| 1  2  3  4 | Array.isArray([1, 2, 3]);  // true  Array.isArray({foo: 123}); // false  Array.isArray('foobar');   // false  Array.isArray(undefined);  // false |

在MDN中就比较了isArray和instanceof的区别，当Array.isArray()不可用的使用，MDN做了如下的补丁，说明还是比较推荐使用前面讲的第三种方法 Object.prototype.toString.call(obj)。



****for in 和 for of 的区别：****

1. ****for in 遍历的是数组的索引(即键名)，而for of 遍历的是数组元素值****
2. ****for in可以遍历数组和对象，for of用来遍历数组****

****组件校验：** <todu :val = “hello”></todu>**

**props接收数据改写为对象的形式：**

**props:{**

**val : { //接受的属性val 也可以写成 val: [stying , number]**

**type : String, // 类型字符串**

**required : true, // 该值必须传递过来**

**default: “default value”, // 不传递则显示默认值**

**validator:function(value){ //验证器 校验传递过来的值满足某条件**

**return (vaule.length > 5) //验证长度大于5 否则有警告！**

**}**

**}**

**}**

**非父子组件传值（Bus/总线/发布订阅模式/观察者模式）**

**插槽：父组件向子组件传递标签**

**如：<todu>**

**<div>标签</div> //父组件中传递个标签**

**</todu>**

**子组件中： template:”<div><slot></slot></div>”子组件就会将传递过来的div标签放入slot标签中，也可以在传递过来的标签中写入: slot = “header”，在slot标签中写 name = “header”。对应起来。**

**作用域插槽：**

**父组件： <todu>**

**<template slot-scope = “props”>**

**<h1>{{props.item}}</h1>**

**</template>**

**</todu>**

**子组件： Vue.component(“todu”,{**

**data:function(){**

**return {**

**list:[.......]**

**}**

**},**

**template:`<div>**

**<ul>**

**<slot v-for = “item of list” :item=item ></slot>**

**</ul>**

**<div>`**

**})**

**这种写法，显示的标签是由父组件决定的(上述的h1标签)，而不是由子组件决定.**

****动态组件:** 根据组件名显示不同的组件，**让组件名赋值给valTodu****

**<component :is=”valTodu”></component>**

**这里的组件显示是通过判断进行创建/销毁的，会消耗性能，所以可以在各自的组件上加上 v-once 只要在页面上显示了一次，就会放在内存中，方便切换，如：Vue.component(“todu”,{**

**template:”<div v-once></div>“**

**})**

**vue动画：（css动画官网：****Animate.css）**

**1.transition标签 用transition包裹需要做动画的标签，如：**

**<transition name=”fade”> //不写name则style中为 .v-enter**

**<div v-if=”show”>hello<div> //写个控制显示隐藏的按钮**

**</transition>**

**然后在style中写入： .fade-enter{opacity:0}**

**.fade-enter-active{transition:opacity 3s;}**

**.fade-leave-to{...} .fade-leave-active{...}**

****2.使用********Animate.css制作动画，**通过在标签上添加特制的class属性就会有动画效果，以下是动态的添加class属性：**

**示例:<div class=”animated bounce”>123<div> //前者必填 后有对应不同的动画效果**

<div :class="{animated:true,bounce:isShow}">这个div做动画</div>

**我们可以动态的控制isShow为ture或false，来让bounce效果做展示**

**使用方法：**<https://www.cnblogs.com/matd/p/11302014.html>

**第一步：安装：**

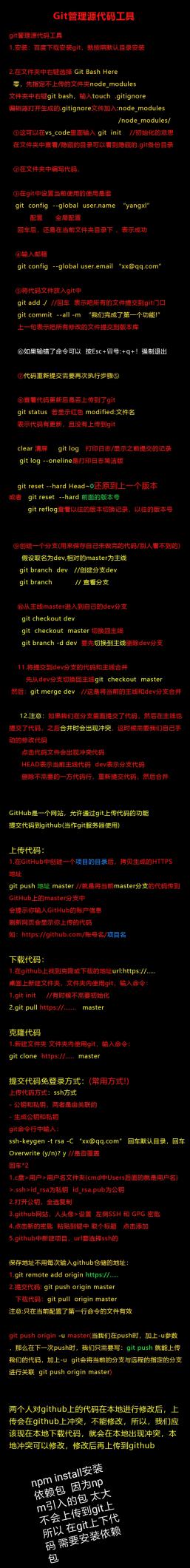
在命令行中执行：npm install animate.css --save

**第二步：引入及使用：**

main.js中：  
import animated from 'animate.css'

Vue.use(animated)

1. **Powershell窗口**

**7.git管理源代码工具**

**8.webpack**

****vue：****

**<template></template>标签中必须有一个标签完整包裹其他标签，如：**

**<template>**

**<div>**

**.........**

**</div>**

**</template>**

**<router-view/>标签显示的是根据路由所对应的组件内容，如”/”改为”list”就会将展示的内容改为list组件里的内容，可以理解为占位符**

**main.js 为项目的入口文件 如：引入全局的css文件:**

**import "../public/index.css"**

**index.js 为组件分配路由**

**import Home from “@/pages/home” @代表list目录**

**<router-link to=”/”>home<router-link> 点击完成单页路由切换**

**移动端页面不可缩放：**

**<meta name=”viewport” content=”width=device-width,initial-scale=1.0,minimum-scale=1.0,maximum-scale=1.0,user-scalable=no”>**

**stylus还不是很懂**

**stylus安装(**stylus 给 CSS 添加了可编程的特性，也就是说，在 stylus 中可以使用变量、函数、判断、循环一系列 CSS 没有的东西来编写样式文件**)：**

**①终端中输入：npm install stylus --save**

**安装stylus-loader:**

**npm install stylus-loader --save**

**②然后在style中<style lang=”stylus”><style>**

**③新建一个styl为后缀的文件，创建变量,如：$logoColor:#111;**

**④在组件style中引入: @import “路径”**

**⑤直接使用这个变量：color：$logoColor;**

****VUE**大的组件通过路由控制显示在APP.vue的<router-view/>标签中，然后把小的组件在大的组件上引用，①import todu from “@/components/todu.vue”**

**②export default{components:todu} ③使用该标签<todu><todu/>**

**<style scoped>**

**</style> 当前的样式只对当前组件有效**

****①**rem适配:(vs-code里面安装cssrem插件)**

****②**重要！：文件-首选项-设置-扩展-cssrem con... Root Font Size改为50**

**750的屏幕下分多少等份，html的字体为多大就改为多少，重启软件生效**

**根据屏幕的宽度来做媒体查询，改变html的字体大小，从而影响内容的大小，内容使用rem为单位。**

**rem的值 = 元素实际的大小 / html的字体大小**

**html文字的大小 = 屏幕的尺寸 / 屏幕分为多少等份**

**(可以根据媒体查询来引入不同的资源:<link rel=”stylesheet” href=”style.css” media=”screen and (min-width:320px)”>)**

****③**自己写媒体查询(根据屏幕尺寸改变html的文字大小，使用rem做适配)：**

**如:新建less文件**

**html{font-size:50px;} //pc端打开手机页面，默认使用750的尺寸**

**@no: 15; //定义一个数字15将屏幕分为15等份，750尺寸对应的文字50px**

**@media screen and (min-width:320px){**

**html{**

**font-size:320px / @no;**

**}**

**}**

**@media screen and (min-width:360px){**

**html{**

**font-size:360px / @no;**

**}**

**}**

**@media screen and (min-width:375px){**

**html{**

**font-size:375px / @no;**

**}**

**}**

**@media screen and (min-width:384px){**

**html{**

**font-size:384px / @no;**

**}**

**}**

**@media screen and (min-width:400px){**

**html{**

**font-size:400px / @no;**

**}**

**}**

**@media screen and (min-width:414px){**

**html{**

**font-size:414px / @no;**

**}**

**}**

**@media screen and (min-width:424px){**

**html{**

**font-size:424px / @no;**

**}**

**}**

**@media screen and (min-width:480px){**

**html{**

**font-size:480px / @no;**

**}**

**}**

**@media screen and (min-width:540px){**

**html{**

**font-size:540px / @no;**

**}**

**}**

**@media screen and (min-width:720px){**

**html{**

**font-size:720px / @no;**

**}**

**}**

**@media screen and (min-width:750px){**

**html{**

**font-size:750px / @no;**

**}**

**}**

**Less：中文网:http://lesscss.cn/**

**在一个less文件中引入另一个less文件: @import “文件名”**

**安装：cmd中npm install -g less**

**安装less-loader：npm install less less-loader --save-dev**

**组件中使用:<style scoped lang="less">**

**@import url("../css/index.less");**

**查看版本：lessc -v**

**新建后缀为less的文件，变量命名规范：必须有@为前缀，大小写敏感，不能数字开头，不含特殊字符，如：定义一个颜色@colorLogo:#444;**

**使用：div{background:@colorLogo;}**

****④**less需要编译为css才能使用，vs-code中安装easy插件，保存less文件就会保存出一个对应的css文件，然后在main.js中引入css文件**

**less中的嵌套：css 中写法：div a{...} less中：div{ a{...} }还有 .clss{ .class2{}}**

**css中a:hove{} less中为：a{ &:hove{...}}**

**less中都可以写成加减乘除的的样式：运算符左右要空格隔开:**

**如: width:200px + 50;(两个数运算，有一个没有单位，则以第一个单位为准)**

**input标签：**

**去除input的默认点击框:outline:none;**

**改颜色:outline-color:red;**

**光标颜色：caret-color:red;**

**光标文本向后偏移:text-indent:.4rem;**

**设置input标签提示信息的文字:placeholder=””修改字体颜色：**

**css中input获取焦点的样式:**input:focus{color:green;}

input::-webkit-input-placeholder { /\* WebKit browsers 适配谷歌 \*/

color: #BDCADA;}

input:-moz-placeholder { /\* Mozilla Firefox 4 to 18 适配火狐 \*/

color: #BDCADA;}

input::-moz-placeholder { /\* Mozilla Firefox 19+ 适配火狐 \*/

color: #BDCADA;}

input:-ms-input-placeholder { /\* Internet Explorer 10+ 适配ie\*/

color: #BDCADA;}

**input标签上的事件：**

**@input=”monitor” @keyup.enter=”monitor”**

**两者的区别：前者是实时监控，每次输出都会调用，而后者是pc上需要回车键，手机上点击确认键才会触发，**

**@change****=”monitor”**

**该事件需要修改input的值并确认才会出触发**

**@blur=”monitor” @focus=”monitor”**

**前者失去焦点 后者获取焦点触发**

****Github的使用: github.com官网****

**常用的版本控制系统：**

**cvs/svn：集中式版本控制系统，所有开发者代码都上传到中央服务器上**

**git：分布式版本控制系统，每个开发者本地都是一个代码管理仓库(包含工作区，暂存区，历史区)，先拉取中央服务器的代码，在本地处理冲突，然后上传到中央服务器**

**环境安装：**

**1.node.js nodejs.org -> DOWNLADS -> LTS 选择对应的系统安装 终端cmd输入 node -v 查看版本号 npm -v查看npm是否安装**

1. **使用git上传代码就需要安装git 终端输入 git --version 查看版本号**

**git下载地址:** <https://www.git-scm.com/download/>

**下载界面化操作(乌龟)：**<https://download.tortoisegit.org/tgit/>

**选择稳定版本下载，还需要下载中文语言包(网站下方有) 先安装乌龟再汉化**

**安装：选择OpenSSH这项 其他默认**

**最后填入github的姓名和邮箱**

**汉化：随便文件夹右键TortoiseGit 设置 选择语言为中文**

**基本步骤：**

1. **组长创建中央仓库（增加协作者）**
2. **小组成员 git clone 仓库地址 把远程仓库及默认的内容克隆到本地一份(解决了三件事：1.初始化了一个本地仓库git init 2.和对应的远程仓库也保持了关联 git remote add 3.吧远程仓库默认内容拉去到本地 git pull)**
3. **每个组员写完自己的程序后，git add 吧自己的代码放在暂存区再放入历史区git commit ，然后git pull 拉取远程代码到本地，然后将本地提交到远程git push**

****界面操作**：**

1. **先创建一个工作目录文件夹，在文件夹内初始化git仓库**

**右键-Git在这里创建版本库-**不勾选**-确定 就创建了一个隐藏文件**

**2.将本地文件提交到**本地git仓库**：**

**框选文件-右键-乌龟-添加 （提交到暂存区）成功有“+”**

**然后选择添加的文件-右键-Git提交 将文件提交到本地仓库 成功有√**

**上述步骤****的简化：**文件-右键-乌龟-提交****

**3.删除版本库文件：**

**直接右键删除文件-选择空白处右键提交-写日志提交 (版本库中就没有删除的文件)**

**将文件保留在本地，不提交到版本库：已提交的文件右键-删除并保留本地副本-点击空白处提交**

**4.**忽略不需要上传到本地仓库的文件**：**

**选中**已放在暂存区的文件夹**-右键删除并添加到忽略列表-根据名称删除和忽略-递归忽略-放在文件夹所在目录-保留本地文件 成功会有灰色的 -**

**然后选中文件夹提交到版本库**

****GitHub上使用SSH上传代码：通过密钥来认证身份****

****生成私钥公钥**：本地文件夹中输入命令行：ssh-keygen -t rsa 回车三四下**

****文件所在位置：**C盘 Users 用户名 .ssh id\_rsa 为私钥 id\_rsa.pub为公钥**

****步骤**：打开公钥文件复制内容- 点击GitHub网站的头像设置栏- 左侧SSH add GPG keys- New SSH key- 随便取个名字然后粘贴公钥**

****上传代码带远程：**文件夹内右键同步 - 远端url 管理- 远端处取个名字区分仓库 - url中填入GitHub上SSH链接 填入密钥 - OK - 推送 -输入用户名密码**

****GitHub上使用Https上传代码：通过用户账号密码认证身份****

****上传代码到远程：**文件夹内右键同步 - 远端url 管理- 远端处取个名字区分仓库 - url中填入GitHub上Http链接 不需要密钥 - OK - 推送 -输入用户名密码**

****拉取代码：**文件夹内右键- 同步 -拉取**

****冲突：**远程的文件修改了的同时本地也修改了该文件，本地无论是拉取远程代码还是同步代码，都会报错**

****解决：**先拉取，报错后文件会出现感叹号，打开文件手动处理冲突，保存文件 - 选中文件右键- 乌龟 -解决冲突 - 提交 -推送**

****创建本地分支：**文件夹内-右键-乌龟-切换检出或者创建分支**

**创建出的新分支dev与master主分支无影响，实验性代码在dev上确认后可以与master主分支合并，并删除原dev分支**

****合并本地分支：**先切换回master主分支-然后右键-合并-选择要合并的分支-**

**合并分支后手动解决冲突 - 右键解决冲突 - 提交master到本地仓库 - 同步到远程**

****删除分支：**右键切换检出选择主分支确定-右键切换检出-点击三个点 - 选择分支右键-删除分支**

****基本设置****

**git安装好后首次使用需要先进行全局配置**

1. **设置****用户名：左面空白地方右键 git bash here**

**git config --global -l 查看全局配置信息**

****git config --global user.name “gitHub上的名字”****

1. **设置用户邮箱**

****git config --global user.email “gitHub邮箱地址”****

****初始化一个git仓库****

1. **先创建一个工作目录文件夹，在文件夹内初始化git仓库**
2. **在文件内初始化git(创建git仓库):**

**进入创建好的文件夹右键 git bash here，输入：git init,（创建隐藏文件.git）**

****将工作区的文件提交到本地仓库****

**在与.git文件夹同层级 输入**git status 查看工作区文件状态** 文件标红说明未提交到暂存区，还在工作区**

1. **在本地编写完代码后，**将文件提交到暂存区**,如：git add 文件1 文件2 ...**

**或者：**git add .** 提交所有文件到暂存区 **.有空格****

1. ****git commit -m”这是注释-提交到历史区”**提交上一步完成的文件到本地仓库，m和字没有空格，**注释一点要写，预防以后回退版本****

**从工作区提交到暂存区，从暂存区提交到本地仓库，是吧内容复制过去了，本区域仍然存在这些信息**

****将本地仓库的文件第一次推送到远程需要执行如下：****

**git remote add origin** [git@github.com:yangxianlin-dada/shiyan2.git](mailto:git@github.com:yangxianlin-dada/shiyan2.git)

**git push -u origin master**

****之后再提交文件就可以直接 git push****

****版本回退****

****git log** 查看历史版本信息**

**git reflog 包含回滚的信息（防止回到过去版本，不能再反悔）**

****git reset --hard** **commit后面的历史版本号** **将工作区回退到历史版本****

****GitHub操作****

****使用Http协议****

**在文件夹中输入命令，**克隆GitHub上的项目**：**git clone 地址链接****

****然后直接git push将本地仓库文件提交到远程会报错，**所以我们需要**修改**本地仓库中.git文件夹中的**config**文件：**

**[remote”origin”]**

**url = https://**github用户名:github密码@**github.com/.....**

**红色区域为添加的内容**

****拉取线上仓库的文件**：**git pull****

****在将本地的代码上传到远程：git push****

****使用ssh协议( 推荐 )****

**与第一种的区别就是改变的鉴定身份的方式**

1. **生成客户端公私钥文件**
2. **将公钥上传到GitHub**

****生成私钥公钥**：本地文件夹中输入命令行：**ssh-keygen -t rsa -C “github邮箱”** 回车四下**

****文件所在位置：**C盘/Users/用户名/ .ssh/id\_rsa 为私钥 /**id\_rsa.pub为公钥****

****步骤**：打开**公钥**文件复制内容- 点击GitHub网站的头像设置栏- 左侧SSH add GPG keys- New SSH key- 随便取个名字然后粘贴公钥**

**操作：克隆：git clone github地址**

**将本地提交暂存区：git add .**

**将工作区提交到本地git：git commit -m”提交注释”**

**提交文件到远程：git push**

****分支管理：分支之间是互不影响的****

**查看本地分支：**git branch****

**创建本地分支并切换到：**git checkout -b 分支名****

**切换本地分支：**git checkout 分支名****

**合并分支：**git merge 分支名（要被删除的分支）先切换到主分支再合并分支****

**删除本地分支：**git branch -d 分支名（需要先切换到其他分支）****

**将本地分支推送到远程：**git push origin 分支名****

**查看远程分支：**git branch -v****

**git push --set-upstream origin 分支名 **来在远程创建一个与本地**分支名**同名的分支并跟踪；****

****利用**git checkout --track origin/分支名**来在本地创建一个与远程**分支**同名并跟踪的本地分支。****

****冲突的产生与解决****

**本地仓库的内容与线上的内容不一致，没有做git pull操作，而是直接修改了本地的内容并提交代码git add -> git commit -m”” -> git push**

**会提示我们在git push之前先拉取代码git pull**

****[解决冲突]****

1. **先git pull,此时git已经将线上与本地仓库的冲突合并到了对应的文件中**
2. **打开冲突的文件，手动解决冲突，再次提交代码**

****忽略文件****

****新建文件名为.gitignore的文件**，该文件用于声明忽略文件或不忽略文件的规则，规则对当前目录及其子目录生效(**注意：**改文件没有文件名，没办法直接在windows目录下直接创建，可以通过命令行git bash来创建)**

**如输入命令：**touch .gitignore****

**常见的规则写法：**

1. ****/mtk/** 过滤整个文件夹**
2. ****\*.zip** 过滤所有.zip文件**
3. ****/mtk/do.c** 过滤某个具体文件**
4. ****index.php** 不过滤具体某个文件**

****如忽略提交文件夹名为js的文件夹：****

****.gitignore文件中写入：/js/****

****图形化的界面操作工具****

1. ****Github for Desktop****
2. ****Source tree****
3. ****TortoiseGit****



****Mint-ui的使用：****

**安装：npm install mint-ui -S**

**完整引入（main.js文件中）：**

**import MintUI form “mint-ui”**

**import “mint-ui/lib/style.css”**

**Vue.use(MintUI)**

**按需引入：**

**import { Button } from “mint-ui”;**

**import “mint-ui/lib/style.css”;**

**Vue.component(Button.name,Button);**

**修改Mint-ui中的样式：可以直接复制一份mint-ui的css文件到静态目录下，然后更改main.js的引入路径，修改css里面的样式即可**

****>>>可以换成/deep/****

**<style scoped>时**

**.banner-wrap**>>>**.mint-swipe-indicator{ //用 >>>连接专有class**

**background:rgb(13, 3, 58);**

**}**

**动态路由：**

**不同路由传值可以用动态路由传值**

**案例：推荐使用下面这种方式传参**

**①news.vue中: <router-link :to=”’/content/’+key”>跳转</router-link> key就是当前页面要传过去的值**

**②main.js中**

**import conent from “./components/content.vue”**

**{ path:’/content/:id’,component:content}**

**③content.vue中就可以通过this.$route.params获取key的值**

**如：mounted:function(){**

**console.log(this.$route.params.id)**

**}**

* 1.$router为VueRouter实例，想要导航到不同URL，则使用$router.push方法
* 2.$route为当前router跳转对象，里面可以获取name、path、query、params等

**使用get传值:**

**①news.vue中: <router-link :to=”’/content/?id=’+key”>跳转</router-link> key就是当前页面要传过去的值**

**②main.js中**

**import conent from “./components/content.vue”**

**{ path:’/content’,component:content}**

**③content.vue中就可以通过this.$route.query获取key的值**

**如：mounted:function(){**

**console.log(this.$route.query)**

**}**

二者还有点区别，直白的来说query相当于get请求，页面跳转的时候，可以在地址栏看到请求参数，而params相当于post请求，参数不会再地址栏中显示。

**axios：**vueRouter已经淘汰，现在主流使用axios发起ajax请求

1.npm install axios安装

2.main.js中

import axios from “axios”

import Qs from “qs” //qs是axios库中自带的，不需要安装

Vue.prototype.axios = axios;

Vue.prototype.qs = Qs;

如:**post请求**

this.axios.post('****www.baidu.com****',this.qs.stringify({'name':'xiaoming','sex':'nan'}),{

headers: {

'Content-Type': 'application/x-www-form-urlencoded'

//常见的表单数据提交数据的编码类型content-type

// 'Content-Type': ‘application/json’ json数据

//如果后台接受的是json数据则去掉上面的this.qs.stringify

//并且使用 'Content-Type': ‘application/json’

}

}) .then(function(res){

console.log(res.data)

//控制台打印请求成功时返回的数据

}.catch(function(err){

if(err.response) {

console.log(err.response)

//控制台打印错误返回的内容

}

}

**get请求：**

axios.get('/user', {

params: {

ID: 12345

}

})

.then(function (response) {

console.log(response);

})

.catch(function (error) {

console.log(error);

});

过滤器看官方文档

vue的两大核心思想：组件化和数据驱动

**vuex：**主要应用于vue.js中管理数据状态的一个库

1.安装: npm i vuex -S

2.main.js中：

import Vuex from “vuex”

Vue.use(Vuex)

1. 在main.js中创建一个数据仓库

const store = new Vuex.Store({

state:{

totalcount:1

},

mutations:{

uptatecount(state,arg){

state.totalcount = arg //arg是传递过来的参数

}，

btnUpdata(state,arg){ //可以自定义一些方法

用新值对旧值做出处理.......

}

}

});

4.注册到实例中

var app = new Vue({

el:”#app”,

data:{},

store:store //或者简写成store

})

1. 使用数据：两种方式都可以

第一种方式①：在组件中获取数据的值{{this.$store.state.totalcount}}

<input :value=”this.$store.state.totalcount”>

第二种方式②：组件文件中按需导入mapState函数 //该函数是vuex自带的

import { mapState } from “vuex”

通过刚才导入的mapState函数，将当前组件需要的全局数据，映射为当前组件的computed计算属性:

computed:{

...mapState([‘数据totalcount’])

}

在页面上直接写该数据{{totalcount}}

1. 修改数据：

第一种方法①：组件方法中this.$store.commit(“uptatecount”,新val)

第二种方法②：组件文件中按需导入mapMutations函数 //该函数是vuex自带的

import { mapMutations} from “vuex”

通过刚才导入的mapMutations函数，将当前组件需要的mapMutations函数，映射为当前组件的methods方法:

methods:{

...mapMutations([’函数uptatecount‘])，

btnfun(){ //这个是在组件上自己定义的一个方法如点击触发事件

this.uptatecount(新val) //这里就可以调用store中的方法了

}

}

Vuex核心概念：state(声明) getter(获得者) mutation(变化)

action(行动) module(组件/模块)

vuex中异步的操作数据：

定义的store数据仓库中，如果通过异步操作变更数据（如写一个定时器），必须通过Action而不能使用mutation，但是在Action中还是要通过触发Mutation的方式间接变更数据如下：

main.js文件中先安装vuex(上面有)

import Vuex from “vuex”

Vue.use(Vuex)

//定义action

const store = new Vuex.Store({

//....省略其他代码

mutations:{

add(state, step){

state.count = step //改变state中的数据count

}

},

actions:{

addAsync(context, step){ //step接收传递过来的值

setTimeout(()=>{ //异步的触发mutations中的方法

context.commit(‘add’, step) //commit用来触发某个mutation函数

},1000)

}

}

})

触发action的第一种方法①：

//组件中触发action

methods:{

handle(){ //组件上定义函数用来触发该事件

//触发actions的第一种方式

this.$store.dispatch(‘addAsync’,5 ) //dispatch函数专门用来触发action 5为传递过去的值

}

}

触发action的第二种方法②：

组件中引入mapActions函数

import { mapActions } from ‘vuex’

通过刚才导入的mapActions函数，将需要的actions函数映射为当前组件的methods方法

methods:{ //将指定的actions函数，映射为当前组件的methods函数

...mapActions([‘addAsync’])，

handle(){ //组件上定义函数用来触发该事件

this.addAsync(新val) //调用函数

}

}

getter：

getter可以对store中已有的数据加工处理之后形成新的数据，不会改变store中的数据，类似vue的计算属性

store中的数据发生变化，getter的数据也会跟着变化

main.js定义：

const store = new Vuex.Store({

state:{

count:0

},

getters:{

showNum:state=>{

return “当前最新的数据是‘+state.count’”//只是对原数据进行包装

}

}

})

组件中获取到gerters的值：

第一种方法①：

this.$store.getters.名称

如：this.$store.getters.showNum

第二种方法②：映射为自己组件中的属性

组件中引入mapGetters

import { mapGetters } from ‘vuex’

computed:{

...mapGetters([‘showNum’])

}

组件中直接使用：{{showNum}}

综上：组件中引入的有以下方法:

import { mapState, mapMutations, mapActions, mapGetters } from ‘vuex’

**webpack：**

1. 可以做到：代码转换、文件优化、代码分割、模块合并、自动刷新、代码校验、自动发布
2. 需要掌握的内容：需要node基础、以及npm的使用、需要掌握es6语法
3. webpack需要掌握的知识：常见配置、高级配置、优化策略、ast抽象语法树、webpack中的Tapable、webpack中的loader、webpack中的plugin

**安装本地的webpack**

1. 新建项目空白目录，并运行npm init -y命令，初始化包管理配置文件package.json

**项目中安装和配置webpack**

1. 运行npm i webpack webpack-cli -D 命令，安装webpack相关的包
2. 在项目根目录中，创建名为webpack.config.js的webpack配置文件
3. 在webpack的配置文件webpack.config.js中，初始化如下基本配置：

module.exports = { //编译模式

mode:”development” //mode用来指定构建模式 值为：开发阶段

//值为production是发布阶段

}

1. 在package.json配置文件中的scripts节点下，新增dev脚本如下：

“scripts”:{

“dev”:”webpack”//script节点下的脚本，可以通过npm run运行

}//这里就是终端中运行npm run dev命令，启动webpack进行项目打包

就会创建一个dist文件夹

1. 如果我们在index.html引用了的文件，已被打包，我们需要重新引用一下dist下的打包文件main.js

**webpack的基本使用**

1. **配置打包的入口与出口**

webpack的4.x版本中默认约定：

打包的入口文件为src -> index.js

打包的输出文件为dist -> main.js

如果要修改打包的入口与出口，可以在webpack.config.js中新增如下配置信息：

const path = require(“path”) //导入node.js中专门操作路径的模块

module.exports = {

entry: path.join(\_\_dirname,”./src/index.js”), //打包入口文件的路径

output: {

path: path.join(\_\_dirname,”./dist”), //输出文件的存放路径 双下划线

filename: “bundle.js”//输出文件的名称 main.js -> bundle.js

}

}

1. **配置自动打包功能**

为什么：避免每次修改了代码，页面不能同步，需要频繁打包才能同步

1. 运行npm install webpack-dev-server -D，安装支持项目自动打包的工具
2. 修改package.json -> scripts中的dev命令如下：

“scripts”:{

“dev”:”webpack-dev-server”//script节点下的脚本，可以通过 npm run 执行

}

1. 将src -> index.heml中，script脚本的引用路径，修改为”/bundle.js”，这里的根目录下的bundle.js文件是在内存中存在的，看不到的虚拟文件
2. 运行npm run dev命令，重新打包
3. **配置html-webpack-plugin生成预览页面**

为什么：使用npm run dev打包可以直接访问index.html页面

原理：将index.html文件复制到根目录下，访问根目录文件就会直接访问index.html文件

1. 运行npm install html-webpack-plugin -D命令，安装生成预览页面的插件
2. 修改webpack.config.js文件头部区域，添加如下配置信息

//导入生成预览页面的插件，得到一个构造函数

const HtmlWebpackPlugin = require(“html-webpack-plugin”)

const htmlPlugin = new HtmlWebpackPlugin({//创建插件的实例对象

template:”./src/index.html”,//指定要用到的模板文件

filename:”index.html”//指定生成的文件的名称，改文件存在于内存中，在目录中不显示

})

1. 修改webpack.config.js文件中向外暴露的配置对象，新增如下配置节点

module.exports = {

plugins:[htmlPlugin] //plugin数组是webpack打包期间会用到的一些插件列表

}

1. **配置自动打包相关的参数**

目的：npm run dev后，可以自动打开浏览器访问页面

//package.json中的配置

//--open打包完成后自动打开浏览器页面

//--host配置IP地址

//-port配置端口

“scripts”:{

“dev”:”webpack-dev-server --open --host 127.0.0.1 --port 8888”//蓝色部分是修改地址和端口

}

1. **webpack中的加载器**

目的：通过loader**打包非js模块**

原因：在实际开发的过程中，webpack默认只能打包处理以.js后缀名结尾的模块，其他非.js后缀名结尾的模块，webpack默认处理不了，需要调用loader加载器才可以正常打包，否则会报错！

loader加载器可以协助webpack打包处理特定的文件模块，比如：

less-loader可以打包处理.less相关的文件

sass-loader可以打包处理.sass相关的文件

url-loader可以打包处理css中与url路径相关的文件

1. 打包处理css文件

①运行npm i style-loader css-loader -D命令，安装处理css文件的loader

②在webpack.config.js的module -> rules数组中，添加loader规则如 下：

//所有第三方文件模块的匹配规则 module.exports中

module: {

rules: [

{test:/\.css$/,use: [“style-loader”,”css-loader”]}

] //test表示匹配文件类型，use表示对应要调用的loader，use中顺序不能错

}

1. 打包处理less,scss文件

①运行npm i less-loader less -D命令，//less

运行npm i sass-loader node-sass -D命令，//sass

②在webpack.config.js的module -> rules数组中，添加loader规则如 下：

//所有第三方文件模块的匹配规则 module.exports中

module: {

rules: [

{test:/\.css$/,use: [“style-loader”,”css-loader”]},

{test:/\.less$/,use: [“style-loader”,”css-loader”,”less-loader”]},

{test:/\.s**c**ss$/,use: [“style-loader”,”css-loader”,”sass-loader”]}

] //scss没有写错

}

1. **配置postCSS自动添加css的兼容前缀**

目的：有些css语法在IE浏览器中不能很好的兼容

1. 运行npm i postcss-loader autoprefixer -D命令
2. 在项目根目录中创建postcss的配置文件postcss.config.js，并初始化如下配置：

const autoprefixer = require(“autoprefixer”)//导入自动添加前缀的插件

module.exports = {

plugins: [autorefixer] //挂在插件

}

1. 在webpack.config.js的module -> rules数组中，修改css的loader规则如下：

module:{

rules:[

{test:/\.css$/,use:[“style-loader”,”css-loader”,”postcss-loader”]}

]

}

1. **打包样式表中的图片和字体文件**
2. 运行npm i url-loader file-loader -D
3. 在webpack.config.js的module -> rules数组中，添加loader规则如下：

module:{

rules:[

{

test:/\.jpg|png|gif|bmp|ttf|eot|svg|woff|woff2$/,

use:”url-loader?limit=16940”

}//?之后的是loader参数，limit指定图片的大小单位是字节，小于值才会被转为base64图片

]

}

**8.打包处理js文件中的高级语法**

1. 安装babel转换器相关的包: npm i babel-loader @babel/core @babel/runtime -D

2. 安装babel语法插件相关的包: npm i @babel/preset-env @babel/plugin-transform-runtime @babel/plugin-proposal-class-properties -D

3. 在项目根目录中，创建babel配置文件babel.config.js并初始化基本配置如下：

module.exports = {

presets: [“@babel/preset-env”],

piugins: [“@babel/plugin-transform-runtime”,”@babel/plugin-proposal-class-properties”]

}

1. 在webpack.config.js的module -> rules数组中，添加loader规则如下：

//exclude为排除项，表示babel-loader不需要处理node\_modules中的js文件

{test:/\.js$/,use:”babel-loader”,exclude:/node\_modules/}

**9.webpack打包发布**

1. 上线之前需要通过webpack将应用进行整体打包，可以通过package.json文件配置打包命令

//在package.json文件中配置webpack打包命令

//该命令默认加载项目根目录中的webpack.config.js配置文件

“scripts” :{

//用于打包的命令

“build”:”webpack -p”,

//用于开发调试的命令

“dev”:”webpack-dev-server --open --host 127.0.0.1 --post 3000”,

}

**postman**

安装下载：[www.getpostman.com](http://www.getpostman.com)

页面详解：

左上角：New 新建请求，新建集合

Import 导入脚本

Runner 运行集合请求

正方形加号： 新增窗口

右上角：同步状态 同步到多台电脑

卫星：抓取api，抓取请求

扳手：设置栏

通知栏

爱心：联系postman

个人信息

登录状态

左侧：搜索框 过滤信息

History 记录信息

Collections 集合 不同的项目可以设置不同的集合

中间：Get:请求的方法 填入请求的地址（整个地址，不是单纯的接口地址） Params请求参数 Send发送 Save保存到某个集合下面

Authorization认证信息 Headers请求头 Body请求体 Pre-request-Script请求前置脚本 Tests请求后的脚本 Cookies cookies管理器 Code根据需求生成代码 右侧为脚本

下左方 隐藏左边 搜索关键字 提起控制台

下右方 返回上次请求 显示改为左右布局 快捷键设置 帮助功能

接口测试必要条件：

请求地址 请求协议 请求方式 请求头 参数

请求方式：get请求 post请求

传参格式：表单提交 请求体提交

举例：**post请求**

1. 选择post请求 填入请求地址：<http://.....> 请求格式是json的话 我们需要在Headers中 key : content-Type Value : application/json
2. Body中：{**“**name**”: ”**lemonyameng**”,”**pwd**”:”**123456**”**}
3. Send发起请求
4. 界面的下半部分就是返回结果