1、vue基础知识：

    1.1 实例化vue：

方法1：

new Vue({

    el: '#app',        //html中被替换的的标签，即挂载到html中#app标签里面

    components: { App },

    template: '<App/>'

})

方法2：

const root=new Vue({

    components: { App },

    template: '<App/>',

    data:{

          text:0,

          obj:{email:"123"}

    }

})

root.$mount("#app");    //html中被替换的的标签，即挂载到html中#app标签里面

实例化vue属性：

root.text=12;          //获取实例化vue的data的数据，并改变其值，也可用root.$data.text=12一样。

root.$data;          // 实例化vue的data数据组成的对象；

root.$props;            // 实例化vue的prop自定义属性传值的对象，前提是要先声明自定义属性传值。

root.$el;                 // 实例化vue在html中#app标签节点下所有的标签及内容。

root.$options;       //实例化vue里面传入的所有属性及值，以及其实例化默认的属性及值。注意：通过

                                root.$options.$data.text给text赋值，需要等到下一次数据更新时才能其作用。

root.$root===root；   //当前组件树的根 Vue 实例。如果当前实例没有父实例，此实例将会是其自己。

root.$children ;         //root组件下的直接子标签/组件；

root.$parent;         //父实例，如果有的话。

root.$slots;

root.$scopedSlots;

root.$refs;      //快速的查找到html节点或组件实例，<div ref="div"></div> ,<v-item ref="item"></v-item>

root.$isServer;     //在服务端渲染时，才用，不常用

实例vue方法、数据

root.$watch;   //监听数据变化,const unWatch=root.$watch("text",(newText,oldText)=>{

                                console.log(`${newText}:${oldText}`)

                        })     然后调用unWatch()即可注销watch监听事件，避免内存溢出，在组件里面watch:{

                                text (newText,oldText){

                                     console.log(`${newText}:${oldText}`)

                                }

                        }    监听完自动注销watch监听

root.$emit("text",val);   //绑定一个自定义事件(函数)名为text的事件，常用于子组件向父组件传递方法

root.$on("text",(val)={      //监听了这个事件并触发了，root.$once("text")，表示只监听触发一次，如定时器中仅执行一次。

    console.log("text事件被触发了"+val+“是自定义事件传递过来的参数”)

})

root.$off("text");     // 如果没有提供参数，则移除所有的事件监听器；如果只提供了事件，则移除该事件所有的监听器；如果同时提供了事件与回调，则只移除这个回调的监听器。

root.$set(root.obj,"name","Tom")；   //为obj对象新添加属性name，并为其赋值Tom,然后obj.name的值就可以直接渲染在

 视图中。

root.$delete(root.obj,"email")；    //删除obj对象的email属性，并且同步通知到视图中。

root.$nextTick(()=>{

            // 这是DOM渲染完，数据加载完之后执行的函数体，如重新添加了dom标签，从后台获取数据完成之后。

})

实例vue生命周期：在某个时间点，会自动执行的函数，执行顺序如下

beforeCreate(){}    //new Vue()实例化的时候即创建前后，this.$el在创建生命周期前后（beforeCreate，created）是underfined

created(){}

beforeMount(){}   //el:"#app"或root.$mount("#app")即挂载前后的生命周期，如果没有el:"#app"或root.$mount("#app")则不执行挂载前后的生命周期,this.$el在挂载生命周期前beforeMount是<div id="app"></div>,在挂载生命周期后mounted是替换掉<div id="app"></div>的部分。

mounted(){}    //所以常对dom有关的操作在mounted里面，对数据有关的操作可以在created或者mounted里面

beforeUpdate(){}    //数据更新时执行，如定时器里data里面的text+=1时执行数据更新前后的生命周期

updated(){}

activated(){}

deactivated(){}

beforeDestroy(){}      //销毁实例化vue前后执行的生命周期

defstroyed(){}

errorCaptured((h,err)=>{     //捕获到该组件及其子组件下的所有错误,可以阻止子组件错误向父组件冒泡传递。

    return h("div",{},err.stack)

}){}

vue的数据绑定：

<any v-bind:属性名="表达式"></any>

<any :属性名="表达式"></any>

<any v-bind:class="{类名:true/false条件表达式}" class="类名"></any>

<any :style="style"></any>

vue的数据计算属性computed：对于任何复杂逻辑，都应当使用计算属性。计算属性只有在它的相关数据变量发生改变时才会重新求值。这就意味着只要 message 还没有发生改变，则计算属性会缓存message的值，多次访问算属性会立即返回之前的计算结果，而不必再次执行函数。

computed:{          //计算属性写法1：计算数据的改变，常用于数据需要计算后渲染到视图中。

    message(){

        return "计算属性message值改变时，才会执行该函数"

    }

}，

computed:{          //计算属性写法2：计算数据的改变，常用于数据需要计算后渲染到视图中。

    message：{

         get:function(){},      //获取计算数据message时执行get()

         set:function(){}        //修改计算数据message时执行set()

    }

}，

methods:{

    handle(){

        return "方法函数每次重新渲染时无论message值是否改变，都会执行该函数"

    }

}

vue的监听属性watch： 监听数据改变前后的值，常用于向后台发生数据。

watch:{                  //方法1：该写法，在初始化时，不会执行，只有数据改变时，才会执行里面的函数。

    val:function(newVal,oldVal){

         console.log(newVal,oldVal)

    }

}

watch:{                  //方法2：该写法，在初始化及数据改变时，都会执行里面的函数。

    val:{

          handler(newVal,oldVal){

                  console.log(newVal,oldVal)

            },

            immediate:true,

            deep:true   // 是否深入观察，即如果val是对象，当不需要深入观察时，handler只是当val变化时才执行，当需要

                            深入观察时，当val里面的属性只，或者val对象里面属性对象里面的属性值...改变时都会触发handler执行。

    }

}

注意：computed和watch里尽量不要修改原数据值，避免出现无限循环，如data(){return {a:0 }},不要在computed和watch里面做a+=1之类的，会出现无限循环，computed里面是通过原数据计算结果并赋值给新的变量，watch里面监听数据改变前后的值。

vue的指令：

v-text      ：当前标签下的文本内容，和{{}}功能一样。

v-html     ：当前标签下的html内容。能识别字符串里面的标签。

v-show    ：是否显示当前标签，即display:none/block;

v-if          ： 是否挂载当前标签，即动态增删标签节点。

v-else

v-else-if

v-for       ：遍历循环对象、数组，渲染在视图。

v-on       ：给当前标签绑定事件。

v-bind    ：给当前标签绑定属性。:value="1",与value="1",后者1是字符串，前者1是数字。

v-model  ：在input、select、textarea里面双向绑定数据，与value可以用于单选多选框中。

<input v-model="message" >

<textarea v-model="message" ></textarea>

<input type="checkbox" id="jack" value="Jack" v-model="checkedNames">

<input type="checkbox" id="john" value="John" v-model="checkedNames">

<input type="checkbox" id="mike" value="Mike" v-model="checkedNames">

data: {

checkedNames: []

}

<input type="radio" id="one" value="One" v-model="picked">

<input type="radio" id="two" value="Two" v-model="picked">

data: {

picked: ''

}

<select v-model="selected">

<option disabled value="">请选择</option>

<option>A</option>

<option>B</option>

<option>C</option>

</select>

data: {

selected: '' //selected:[]

}

<!-- 当选中时，`picked` 为字符串 "a" -->

<input type="radio" v-model="picked" value="a">

<!-- `toggle` 为 true 或 false -->

<input type="checkbox" v-model="toggle">

<!-- 当选中第一个选项时，`selected` 为字符串 "abc" -->

<select v-model="selected">

<option value="abc">ABC</option>

</select>

在默认情况下，v-model 在每次 input 事件触发后将输入框的值与数据进行同步 (除了上述输入法组合文字时)。你可以添加 lazy修饰符，

从而转变为使用 change事件进行同步：

<!-- 在“change”时而非“input”时更新 -->

<input v-model.lazy="msg" >

在 type="number"时，HTML 输入元素的值也总会返回字符串 如果想自动将用户的输入值转为数值类

<input v-model.number="age" type="number">

如果要自动过滤用户输入的首尾空白字符

<input v-model.trim="msg">

v-once   ：该标签只渲染一次，后面重新渲染会跳过该标签。

vue的模板语法：{{}}，v-text，v-html，其中{{}}和v-text一样，不会识别标签。

注意：以v-开始的指令，其值如name为js表达式(如js变量)；{{fun()}}仍然可以执行函数方法渲染函数的结果在页面，但是不推荐，推荐放在计算属性中。

vue的组件的定义：

1. 局部组件：

    const com={

        template:` <div></div>`，

        data(){

            return {}

        }

    }

    Vue.component("CompOne",com)      //定义局部组件

   <comp-one></comp-one>                //调用局部组件

2. 全局组件：

      new Vue({

        components:{CompOne},                   //调用全局、局部组件

        template:`   <comp-one></comp-one>  `,     //模板

        data(){

            return {}

        }

    })

vue的props属性的父向子组件传值：可以是数组或对象，用于接收来自父组件的数据。单向下行绑定：父级 prop 的更新会向下流动到子组件中，但是反过来则不行。props接受的父组件的数据可以和data里面的一样，用于直接渲染在视图，用this.props里面的数据名直接进行计算属性，在vue的方法，生命周期中调用。但是不能直接修改props里面的值，可以在data里面定义一个变量初始化为props里面传递过来的数据值，然后在直接修改data里面这个变量的值。

    子修改父组件props传递过来的值：通过子组件触发的事件告诉父组件，要修改props传递过来的值。

方法1：通过父组件传递过来一个方法事件onChange，子组件通过触发该方法事件，告诉父组件，在父组件中修改propOne的值。

    const com={              //子组件

        props:{              //父组件传递过来的propOne数据

               propOne:String，

               name:{

                    type: Number,    //传递过来的数据类型

                    required:true,    //该数据必须传递过来，不然报错

                    default:0              //默认值，可以是个对象default(){return},通常和required不同时出现

               },

               onChange:Function

        },

        template:` <div>

                        <span @click="sonHandle">{{propOne}}</span>

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

                    </div>`，

        data(){

            return {}

        },

        methods:{

            sonHandle(){

                this.onChange();    //子组件点击事件触发父组件传递过来的onChange事件，在父组件中执行该onChange事件的函数，即修改传递给子组件的数据

               //或者通过 this.$emit("change");  子组件自定义change事件，不需要在props接受 父组件传递的

                onChange:Function，父组件 <comp-one  @change="handleChange" :prop-one="prop1"></comp-one>

            }

        }

    }

    Vue.component("CompOne",com)

     new Vue({                        //父组件

        template:`   <comp-one  @on-change="handleChange" :prop-one="prop1"></comp-one>  `,     //模板

        data(){

            return {

                prop1:"这是要传递给子组件的数据"

            }

        }，

        methods:{

            handleChange(){

                this.prop1="这是子组件触发了父组件传递给子组件的onChange事件执行父组件的handleChange函数，在父组件

                                       修改传递给子组件的propOne数据值"

            }

        }

    })

vue的组件的继承：继承的时候，需要propsData{}接受父组件传递的参数，新定义的data数据会覆盖原com组件数据，方法周期等会先执行com组件的，再执行compVue里的。

   const com={

        props:{

               propOne:String，

        },

        template:` <div>

                        <span >{{propOne}}</span>

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

                    </div>`，

        data(){

            return {

                text:0

            }

        },

    }

继承的方法1：

const compVue=Vue.extend(com);

new compVue({

    el:"#root",

    propsData:{

           propOne:'这是继承com组件的新compVue组件'

    }

})

继承的方法2：

const compVue2={

    extends:com,

    data(){return{}}

}

vue的props属性的子向父组件传值：

        在父组件调用子组件时<son @自定义事件="自定义事件处理函数"></son>

        在父组件中定义一个methods方法，处理自定义事件处理函数接受到的子组件的数据

        在子组件中定义一个方法触发this.$emit("自定义事件名",要传递的数据)

2、vuex：

简化版vuex的应用：

    步骤1：在项目文件目录命令行下npm i vuex -S;

    步骤2：新建store文件夹->store.js文件

    步骤3：在store.js里面：

    步骤4：在vue的入口文件，如main.js里面：全局store

    步骤5：在.vue组件文件中通过this.$store获取全局store对象，即可使用store其属性方法API；

模块化vuex的应用：

    步骤1：在项目文件目录命令行下npm i vuex -S;

    步骤2：新建store文件夹->store.js文件