1. DOM操作 —— 如何添加、移除、移动、复制、创建和查找节点等

（1）创建新节点

       createDocumentFragment()    //创建一个DOM片段

       createElement()   //创建一个具体的元素

       createTextNode()   //创建一个文本节点

（2）添加、移除、替换、插入

      appendChild()

      removeChild()

      replaceChild()

      insertBefore() //在已有的子节点前插入一个新的子节点

（3）查找

       getElementsByTagName()    //通过标签名称

       getElementsByName()    //通过元素的Name属性的值(IE容错能力较强，会得到一个数组，其中包括id等于name值的)

       getElementById()    //通过元素Id，唯一性

1. 如何居中div？

答:

1.flex justify-content:center; align-items:center;

2.定位+margin (50%,50%)+调整

3.位移+transform (50%,50%)+调整

1. 数组和对象有哪些原生方法，列举一下？

答:

数组:

1)toString() 作用：将数组中的元素按照逗号分组合成字符串

2)join() 作用：将数组中的元素按照指定字符组合成字符串，默认是逗号。

3)concat(arr2,arr3) 拼接多个数组中的元素

4)slice(star,end) 截取数组中的元素，start开始的下标，end结束的下标，不包含end本身

如果是负数表示倒数，返回一个数组。

5) push()往数组的末尾添加元素，返回数组的长度

6) pop() 删除数组末尾的元素，返回删除的元素

7) unshift() 往数组的开头添加元素，返回数组的长度

8) shift() 删除数组开头的元素，返回删除的元素

对象:

Object Array String Number Boolena Math Date

1. 下面的代码将输出到控制台，为什么？

(function(){

var a = b = 3;

})();

console.log("a defined? " + (typeof a !== 'undefined')); //false

console.log("b defined? " + (typeof b !== 'undefined')); //true

答:

a defined? false

b defined? true

因为a被b覆盖了

1. 下面的代码将输出到控制台，为什么？

var myObject = {

foo: "bar",

func: function() {

var self = this;

console.log("outer func: this.foo = " + this.foo);

console.log("outer func: self.foo = " + self.foo);

(function() {

console.log("inner func: this.foo = " + this.foo);

console.log("inner func: self.foo = " + self.foo);

}());

}

};

myObject.func();

答：outer func: this.foo = bar

outer func: self.foo = bar

inner func: this.foo = undefined

inner func: self.foo = bar

1. 两个函数运行结果一样吗？为什么？

// 函数1

function foo1()

{

return {

bar: "hello"

};

}

// 函数2

function foo2()

{

return

{

bar: "hello"

};

}

console.log(foo1());

console.log(foo2());

答:

{bar: "hello"}

undefined

1. 在下面的代码中，数字 1-4 会以什么顺序输出？为什么会这样输出？

(function() {

console.log(1); //1

setTimeout(function(){console.log(2)}, 1000); //2

setTimeout(function(){console.log(3)}, 0); //3

console.log(4); //4

})();

答:

1

4

3

2

程序的执行顺序

1. 请自选一种方法来实现 fn(1)(2) == 3 的方法

function fn(x){

return function(y){return x+y}

}

fn(1)(2)

1. 什么情况下会遇到跨域，怎么解决？

浏览器最核心，最基本的安全功能是同源策略。限制了一个源中加载文本或者脚本与其他源中资源的交互方式，当浏览器执行一个脚本时会检查是否同源，只有同源的脚本才会执行，如果不同源即为跨域。

1.jsonp：原理就是利用了script标签src属性外联引入文件不受同源策略的限制，在页面中动态插入了script，script标签的src属性就是后端api接口的地址，并且以get的方式将前端回调处理函数名称告诉后端，后端在响应请求时会将回调返还，并且将数据以参数的形式传递回去；

2.document.domin:两个域名必须属于同一基础域名，并且所有的协议端口完全一致，否则无法跨域；例：beijing.58.com

tianjing.58.com ；

3.iframe、hash ：父页面向子页面传输数据：将要传递的数据添加到子页面的url的hash值上，子页面通过location.hash

并添加定时器实时地动态父页面传来的数据；子页面向父页面传输数据：利用window.name的特性，及页面重新加载但当前页的name值不变，即使换了一个页面。需要三个页面配合使用。一个应用页面，一个数据页面，一个代理文件。代理文件一般是一个没有任何内容的html页面，需要和应用页面在同一域下。将数据页面的窗口换成代理页面，代理页面通过window.name获取数据页面留下的数据，应用页面再访问和它同源的代理页面获取数据，就完成了跨域；

4.CORS(Cross-Origin-Resource-Sharing)：在服务器端设置的,不需要客户端进行操作。Cors背后的思想是使用自定义的http头部让浏览器和服务器进行沟通，从而决定请求或响应是否应该成功，还是应该失败。浏览器向服务器发送请求，如果服务器认为这个请求可以接受，就在Access-Control-Allow-Origin头部中回发相同的源信息（如果是公共资源，可以回发\*）;如果没有这个头部，或者有这个头部但信息源不匹配，浏览器就会驳回请求。正常情况下，浏览器会处理请求。（请求和响应都不包含cookie信息）；

5.服务器跨域，服务器中转代理：前端向本地服务器发送请求，本地服务器代替前端再向服务器接口发送请求进行服务器间通信，本地服务器是个中转站的角色，再将响应的数据返回给前端。

6.window.postMessage(h5新增)

7.WebSocket(h5新增)

1. 简单说一下 vue 的双向绑定原理

答:

单向绑定: 只能将程序中的修改更新到页面中。但是无法将页面中的更改，更新回程序中。

双向绑定: 即能将程序中的修改更新到页面中，又能将界面中的修改，反向更新会程序中。

原理: v-model其实就是自动为表单元素绑定了oninput或onchange事件。在这些内容更改事件中，自动修改当前Vue对象中的模型变量。

普通绑定为什么不能更新程序中的数据：因为没有自动添加事件，导致界面上更改，Vue框架不知道，自然就无法更近模型变量。

1. vue-router怎么定义动态路由？写出怎么获取传过来的动态参数？

答:

路由的定义，主要有以下几步：

1.如果是模块化机制，需要调用 Vue.use(VueRouter)

2.定义路由组件，如：

const Foo = {

template: '<div>foo</div>'

};

3.定义路由（数组）：

const routes = [

{

path: '/foo',

component: Foo

}

];

4.创建 router 实例

const router = new VueRouter({

routes

});

5.创建和挂载根实例

const app = new Vue({

routes

}).mount('#app');

嵌套路由主要是通过 children，它同样是一个数组：

{

path: '/user',

component: User,

children: [

{

path: 'file',

component: File

}

]

}

这时访问，/user/file 会映射到 File 组件

动态路由的创建，主要是使用 path 属性过程中，使用动态路径参数，以冒号开头，如：

{

path: /user/:id

component: User

}

这会是访问 user 目录下的所有文件，如 /user/a 和 /user/b，都会映射到 User 组件

当匹配到 /user 下的任意路由时，参数值会被设置到 this.$route.params 下，所以通过这个属性可以获取到动态参数，如：

const User = {

template: '<div>User {{ $route.params.id }}</div>'

}

这里会根据访问的路径动态的呈现，如访问 /user/aaa 会渲染：

<div>

User aaa

</div>

1. <keep-alive></keep-alive>的作用的是什么？

答：keep-alive 是 Vue 内置的一个组件，可以使被包含的组件保留状态，或避免重新渲染。

在vue项目中,难免会有列表页面或者搜索结果列表页面,点击某个结果之后,返回回来时,如果不对结果页面进行缓存,那么返回列表页面的时候会回到初始状态,但是我们想要的结果是返回时这个页面还是之前搜索的结果列表,这时候就需要用到vue的keep-alive技术了.

1. 列举Vue的生命周期几个关键钩子函数？

答：vue生命周期钩子函数的使用并非一成不变的，要根据不同的应用场景而有所改变：

1. beforeCreate：在实例初始化之后，\*\*数据观测(data observer) \*\* 和 **event/watcher事件配置** 之前被调用，注意是 **之前**，此时data、watcher、methods统统滴没有。  
   这个时候的vue实例还什么都没有，但是$route对象是存在的，可以根据路由信息进行重定向之类的操作。
2. created：在实例已经创建完成之后被调用。在这一步，实例已完成以下配置：**数据观测(data observer)** ，**属性和方法的运算**， **watch/event 事件回调**。然而，挂载阶段还没开始，$el属性目前不可见。  
   此时 this.$data 可以访问，watcher、events、methods也出现了，若根据后台接口动态改变data和methods的场景下，可以使用。
3. beforeMount：在挂载开始之前被调用，相关的 **render 函数** 首次被调用。但是render正在执行中，此时DOM还是无法操作的。我打印了此时的vue实例对象，相比于created生命周期，此时只是多了一个$el的属性，然而其值为undefined。  
   使用场景我上文已经提到了，页面渲染时所需要的数据，应尽量在这之前完成赋值。
4. mounted：在挂载之后被调用。在这一步 **创建vm.$el并替换el**，并挂载到实例上。（官方文档中的 “如果root实例挂载了一个文档内元素，当mounted被调用时vm.$el也在文档内” 这句话存疑）  
   此时元素已经渲染完成了，依赖于DOM的代码就放在这里吧~比如监听DOM事件。
5. beforeUpdate：$vm.data更新之后，**虚拟DOM重新渲染** 和打补丁之前被调用。  
   你可以在这个钩子中进一步地修改$vm.data，这不会触发附加的重渲染过程。
6. updated：**虚拟DOM重新渲染** 和打补丁之后被调用。  
   当这个钩子被调用时，组件DOM的data已经更新，所以你现在可以执行依赖于DOM的操作。但是不要在此时修改data，否则会继续触发beforeUpdate、updated这两个生命周期，进入死循环！
7. beforeDestroy：实例被销毁之前调用。在这一步，实例仍然完全可用。  
   实例要被销毁了，赶在被销毁之前搞点事情吧哈哈~
8. destroyed：Vue实例销毁后调用。此时，Vue实例指示的所有东西已经解绑定，所有的事件监听器都已经被移除，所有的子实例也已经被销毁。  
   这时候能做的事情已经不多了，只能加点儿提示toast之类的东西吧。

**注**：beforeMount、mounted、beforeUpdate、updated、beforeDestroy、destroyed这几个钩子函数，在服务器端渲染期间不被调用。

1. 手写一个vue父子互相通讯的例子(写出关键步骤即可)

答: 父组件传递给子组件的值，子组件可以通过props获取。子组件想要改变数据，只能通过触发事件告诉父组件，父组件接收到信号，从而达到改变的目的。

父组件：

<template>

<div>

<parent>

<child :parentToChild="value1"></child>

</parent>

</div>

</template>

<script>

export default {

data () {

return {

value1:'父组件的值传给子组件'

}

},

components: {},

mounted () {},

watch: {},

methods: {}

}

</script>

子组件：

<template>

<div>

<p>这是子组件</p>

</div>

</template>

<script>

export default {

data () {

return {

valueFromParent:'',

}

},

components: {},

props: ['parentToChild'],

mounted () {},

watch: {},

methods: {},

created(){

this.valueFromParent = this.parentToChild

}

}

</script>