1. DOM操作 —— 如何添加、移除、移动、复制、创建和查找节点等

1.创建元素节点

document.createElement() 方法 用于创建元素，接受一个参数，即要创建元素的标签名，返回创建的元素节点

1 var div = document.createElement("div"); //创建一个div元素

2 div.id = "myDiv"; //设置div的id

3 div.className = "box"; //设置div的class

创建元素后还要把元素添加到文档树中

2.添加元素节点

appendChild() 方法 用于向childNodes列表的末尾添加一个节点，

返回要添加的元素节点

1 var ul = document.getElementByIdx("myList"); //获得ul

2 var li = document.createElement("li"); //创建li

3 li.innerHTML = "项目四"; //向li内添加文本

4 ul.appendChild(li); //把li 添加到ul子节点的末尾

appendChild() 方法还可以添加已经存在的元素，

会将元素从原来的位置移到新的位置

1 var ul = document.getElementById("myList"); //获得ul

2 ul.appendChild(ul.firstChild);

//把ul的第一个元素节点移到ul子节点的末尾

insertBefore() 方法，如果不是在末尾插入节点，

而是想放在特定的位置上，用这个方法，该方法接受2个参数，

第一个是要插入的节点，第二个是参照节点，返回要添加的元素节点

1 var ul = document.getElementById("myList"); //获得ul

2 var li = document.createElement("li"); //创建li

3 li.innerHTML= "项目四"; //向li内添加文本

4 ul.insertBefore(li,ul.firstChild); //把li添加到ul的第一个子节点前

1 var ul = document.getElementById("myList"); //获得ul

2 var li = document.createElement("li"); //创建li

3 li.innerHTML= "项目四"; //向li内添加文本

4 ul.insertBefore(li,ul.lastChild); //把li添加到ul的最后一个子节点（包括文本节点）之前

1 var ul = document.getElementById("myList"); //获得ul

2 var li = document.createElement("li"); //创建li

3 li.innerHTML= "项目四"; //向li内添加文本

4 var lis = ul.getElementsByTagName("li") //获取ul中所有li的集合

5 ul.insertBefore(li,lis[1]); 　　　　//把li添加到ul中的第二个li节点前

添加后:

3.移除元素节点

removeChild() 方法，用于移除节点，接受一个参数，即要移除的节点，返回被移除的节点，

注意被移除的节点仍然在文档中，不过文档中已没有其位置了

1 var ul = document.getElementById("myList"); //获得ul

2 var fromFirstChild = ul.removeChild(ul.firstChild); //移除ul第一个子节点

1 var ul = document.getElementById("myList"); //获得ul

2 var lis = ul.getElementsByTagName("li") //获取ul中所有li的集合

3 ul.removeChild(lis[0]); 　　　　　　//移除第一个li，与上面不同，要考虑浏览器之间的差异

4.替换元素节点

replaceChild() 方法，用于替换节点，接受两个参数，第一参数是要插入的节点，第二个是要替换的节点，返回被替换的节点

1 var ul = document.getElementById("myList"); //获得ul

2 var fromFirstChild = ul.replaceChild(ul.firstChild); //替换ul第一个子节点

1 var ul = document.getElementById("myList"); //获得ul;

2 var li = document.createElement("li"); //创建li

3 li.innerHTML= "项目四"; //向li内添加文本

4 var lis = ul.getElementsByTagName("li") //获取ul中所有li的集合

5 var returnNode = ul.replaceChild(li,lis[1]); //用创建的li替换原来的第二个li

5.复制节点

cloneNode() 方法，用于复制节点， 接受一个布尔值参数，

true 表示深复制（复制节点及其所有子节点）， false 表示浅复制（复制节点本身，不复制子节点）

1 var ul = document.getElementById("myList"); //获得ul

2 var deepList = ul.cloneNode(true); //深复制

3 var shallowList = ul.cloneNode(false); //浅复制

1. 如何居中div？

父元素设置为：position: relative;

子元素设置为：position: absolute;

left:50%;top:50%，

teansform：translate(50%,50%)

display: flex;//flex布局

justify-content: center;//使子项目水平居中

align-items: center;//使子项目垂直居中

display:flex//父元素

margin:auto//子元素

display: table-cell;//父元素

vertical-align: middle;//父元素

margin:auto;//子元素

1. 数组和对象有哪些原生方法，列举一下？

赋值方法：

pop 和 push

shift 和 unshift

splice

reverse

sort

访问方法

concat

join

slice

toString

indexOf

lastIndexOf

迭代方法

forEach

map

filter

some

every

reduce

reduceRight

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')) falst

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

a是局部变量 b是全局变量

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();

bar

bar

undefined

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());

不一样 ，js中 return不能跟回车

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);

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

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

console.log(4);

})();

1432

1. 请自选一种方法来实现 fn(1)(2) == 3 的方法

var fn = function (a) {

var add = function (b) {

return fn(a + b);

}

add.toString = function () {

return a;

}

return add;

};

console.log(add(1)(2))

主要是了解toString的这个函数的意思

1. 什么情况下会遇到跨域，怎么解决？

在浏览器同源策略限制下，向不同源（不同协议、不同域名或者不同端口）发送XHR请求，浏览器认为该请求不受信任，禁止请求，具体表现为请求后不正常响应

意思就是只要页面的协议、域名和端口与请求地址的没有完全一样，就认为你是不同源



jsonp

jsonp就是利用了script标签的src属性是没有跨域的限制的，从而达到跨域访问的目的。因此它的最基本原理就是：动态添加一个<script>标签来实现

JSONP的不足之处：

　　1、只能使用get方法，不能使用post方法：

我们知道 script，link, img 等等标签引入外部资源，都是 get 请求的，那么就决定了 jsonp 一定是 get 的。但有时候我们使用的 post 请求也成功，为啥呢？这是因为当我们指定dataType:'jsonp',不论你指定：type:"post" 或者type:"get"，其实质上进行的都是 get 请求！

2、没有关于 JSONP 调用的错误处理。如果动态脚本插入有效，就执行调用；如果无效，就静默失败。失败是没有任何提示的。例如，不能从服务器捕捉到 404 错误，也不能取消或重新开始请求。不过，等待一段时间还没有响应的话，就不用理它了。

cors

优点：

　　　　1、CORS支持所有类型的HTTP请求。

2、 使用CORS，开发者可以使用普通的XMLHttpRequest发起请求和获得数据，比起JSONP有更好的错误处理。

　　缺点： 兼容性方面相对差一点，ie10或以上才支持

1. 简单说一下 vue 的双向绑定原理

vue.js 是采用数据劫持结合发布者-订阅者模式的方式，通过Object.defineProperty()来劫持各个属性的setter，getter，在数据变动时发布消息给订阅者，触发相应的监听回调。

具体步骤：

第一步：需要observe的数据对象进行递归遍历，包括子属性对象的属性，都加上 setter和getter

这样的话，给这个对象的某个值赋值，就会触发setter，那么就能监听到了数据变化

第二步：compile解析模板指令，将模板中的变量替换成数据，然后初始化渲染页面视图，并将每个指令对应的节点绑定更新函数，添加监听数据的订阅者，一旦数据有变动，收到通知，更新视图

第三步：Watcher订阅者是Observer和Compile之间通信的桥梁，主要做的事情是:

1、在自身实例化时往属性订阅器(dep)里面添加自己

2、自身必须有一个update()方法

3、待属性变动dep.notice()通知时，能调用自身的update()方法，并触发Compile中绑定的回调，则功成身退。

第四步：MVVM作为数据绑定的入口，整合Observer、Compile和Watcher三者，通过Observer来监听自己的model数据变化，通过Compile来解析编译模板指令，最终利用Watcher搭起Observer和Compile之间的通信桥梁，达到数据变化 -> 视图更新；视图交互变化(input) -> 数据model变更的双向绑定效果

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？

如果是模块化机制，需要调用 Vue.use(VueRouter)

创建 router 实例

const router = new VueRouter({

routes

})

定义路由组件

const Foo = {

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

};

定义路由（数组）：

const router = new VueRouter({

routes

});

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

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

this.$router.push({

name: 'aaa',或者写 path: 'aaa',

params：{ 或者用query: { id: name1,}

id: name1,

id2: name2

}

})

获取 this.$route.params.id

this.$route.query.id

1. <keep-alive></keep-alive>的作用的是什么？

什么是keep-alive

keep-alive用来缓存组件,避免多次加载相应的组件,减少性能消耗，简单一点来说就是从页面1链接到其他页面后回退到页面1不用在重新执行页面1的代码，只会从缓存中加载之前已经缓存的页面1，这样可以减少加载时间及性能消耗，提高用户体验性。

通过设置了keep-alive，可以简单理解为从页面1跳转到页面2后，然后后退到页面1，只会加载缓存中之前已经渲染好的页面1，而不会再次重新加载页面1，及不会再触发页面一种的created等类似的钩子函数，除非自己重新刷新该页面1。

keep-alive缓存，而其中一个地图路由不能有缓存。

所以我用到以下结构。需要缓存的路由，在keep-alive中加载，而不需要缓存的路由，单独用一个有name属性的router-view盛放。name属性指向不需缓存的路由名。

<template>

<div id="app">

<headpage class="header"></headpage>

<keep-alive>

<router-view></router-view>

</keep-alive>

<router-view name="history"></router-view>

</div>

</template>

在路由引入时，其中history路由即是不需要缓存的路由。

// 引入我的客户页面

import Content from '@/pages/myclientsubPages/content'

// 引入监控页面

import monitoring from '@/pages/monitoring/content'

// 引入数据报表

import datarebort from '@/pages/datarebort/content'

// 引入历史轨迹

import historyTrail from '@/pages/monitoring/history'

export default new Router({

mode: 'history',

routes: [{

path: '/',

component: Content

},

{

path: '/myclient',

name: 'myclient',

component: Content

},

{

path: '/monitoring',

name: 'monitoring',

component: monitoring

},

{

path: '/history',

name: 'history',

components: {

history: historyTrail

}

}]

})

至此，每次进入该路由，都会重新执行一遍生命周期。

keep-alive 简介

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。

用法也很简单：

<keep-alive>

<component>

<!-- 该组件将被缓存！ -->

</component>

</keep-alive>

include - 字符串或正则表达，只有匹配的组件会被缓存

exclude - 字符串或正则表达式，任何匹配的组件都不会被缓存

// 组件 a

export default {

name: 'a',

data () {

return {}

}

}

<keep-alive include="a">

<component>

<!-- name 为 a 的组件将被缓存！ -->

</component>

</keep-alive>可以保留它的状态或避免重新渲染

<keep-alive exclude="a">

<component>

<!-- 除了 name 为 a 的组件都将被缓存！ -->

</component>

</keep-alive>可以保留它的状态或避免重新渲染

<keep-alive include="test-keep-alive">

<!-- 将缓存name为test-keep-alive的组件 -->

<component></component>

</keep-alive>

<keep-alive include="a,b">

<!-- 将缓存name为a或者b的组件，结合动态组件使用 -->

<component :is="view"></component>

</keep-alive>

<!-- 使用正则表达式，需使用v-bind -->

<keep-alive :include="/a|b/">

<component :is="view"></component>

</keep-alive>

<!-- 动态判断 -->

<keep-alive :include="includedComponents">

<router-view></router-view>

</keep-alive>

<keep-alive exclude="test-keep-alive">

<!-- 将不缓存name为test-keep-alive的组件 -->

<component></component>

</keep-alive>

遇见 vue-router

router-view 也是一个组件，如果直接被包在 keep-alive 里面，所有路径匹配到的视图组件都会被缓存：

<keep-alive>

<router-view>

<!-- 所有路径匹配到的视图组件都会被缓存！ -->

</router-view>

</keep-alive>

然而产品汪总是要改需求，拦都拦不住…

问题

如果只想 router-view 里面某个组件被缓存，怎么办？

使用 include/exclude

增加 router.meta 属性

使用 include/exclude

// 组件 a

export default {

name: 'a',

data () {

return {}

}

}

<keep-alive include="a">

<router-view>

<!-- 只有路径匹配到的视图 a 组件会被缓存！ -->

</router-view>

</keep-alive>

exclude 例子类似。

缺点：需要知道组件的 name，项目复杂的时候不是很好的选择

增加 router.meta 属性

// routes 配置

export default [

{

path: '/',

name: 'home',

component: Home,

meta: {

keepAlive: true // 需要被缓存

}

}, {

path: '/:id',

name: 'edit',

component: Edit,

meta: {

keepAlive: false // 不需要被缓存

}

}

<keep-alive>

<router-view v-if="$route.meta.keepAlive">

<!-- 这里是会被缓存的视图组件，比如 Home！ -->

</router-view>

</keep-alive>

<router-view v-if="!$route.meta.keepAlive">

<!-- 这里是不被缓存的视图组件，比如 Edit！ -->

</router-view>

优点：不需要例举出需要被缓存组件名称

【加盐】使用 router.meta 拓展

假设这里有 3 个路由： A、B、C。

需求：

默认显示 A

B 跳到 A，A 不刷新

C 跳到 A，A 刷新

实现方式

在 A 路由里面设置 meta 属性：

{

path: '/',

name: 'A',

component: A,

meta: {

keepAlive: true // 需要被缓存

}

}

在 B 组件里面设置 beforeRouteLeave：

export default {

data() {

return {};

},

methods: {},

beforeRouteLeave(to, from, next) {

// 设置下一个路由的 meta

to.meta.keepAlive = true; // 让 A 缓存，即不刷新

next();

}

};

在 C 组件里面设置 beforeRouteLeave：

export default {

data() {

return {};

},

methods: {},

beforeRouteLeave(to, from, next) {

// 设置下一个路由的 meta

to.meta.keepAlive = false; // 让 A 不缓存，即刷新

next();

}

};

这样便能实现 B 回到 A，A 不刷新；而 C 回到 A 则刷新。

总结

路由大法不错，不需要关心哪个页面跳转过来的，只要 router.go(-1) 就能回去，不需要额外参数。

然而在非单页应用的时候，keep-alive 并不能有效的缓存了= =

1. 列举Vue的生命周期几个关键钩子函数？

Beforecreate

Created

BoforeMount

Mounted

BeforeUpdate

Updated

Beforedestroy

dastroyed

创建 挂载 更新 销毁

1. 手写一个vue父子互相通讯的例子(写出关键步骤即可)

父--子

父级

<div>

  <child :child-str="data"></child>

</div>

import child from './child';

export default {

    name: "parent",

    components:{child},

    data(){

      return {

        data:'我要向子组件传递数据'

      }

    },

}

子集

<div>我是子组件，接收父组件传递过来的值-{{childStr}}</div>

export default {

    name: "child",

    props:{

      childStr:{

        type:String,

        default:'我是空的'//如果不传值，显示的值

      }

    }

  }