Javascript

1.闭包

- 闭包就是能够读取其他函数内部变量的函数。最常见的创建方式就是在一个函数内创建另一个函数，通过另一个函数访问这个函数的局部变量，突破作用域链。

特性

1. 函数内嵌套函数
2. 内部函数可以引用外层的参数和变量
3. 参数和变量不会被垃圾回收机制回收

对闭包的理解

- 使用闭包主要是为了设计私有方法和变量。使用闭包的优点是可以避免全局变量的污染，缺点是常驻内存，增加内存的使用量。在JS中，函数即闭包，只有函数才会产生作用域的概念。

好处：实现封装和缓存
坏处：消耗内存，可能会造成内存溢出(解决办法：在退出函数前，将不使用的局部变量全部删除)
用处：

1. 实现公有变量（累加器）

```
var add = function(){
    var num = 0;
    return function (a){
        return num = num + a;
    }
}
add()(1)
add()(2)
```

2. 可以做缓存（外部不可见）

- cache不可以是Map数据结构，因为Map的键是通过===比较的，即[1]!==[1]，因此即使传入相同的数组或者对象，还是会被存为不同的键

```
const memorize = function(fn){
    const = cache= {} // 存储缓存数据的对象
    return function(...args){
        const _args = JSON.stringify(args) // 将参数作为cache的key
        return cache[_args] || cache[_args] = fn.apply(fn,args)
        // 如果已经缓存过，直接取值。否则重新计算并且缓存
    }
}
const add(a,b){
    console.log('开始缓存')
    return a + b
}
const adder = memorize(add)
console.log(adder(2, 6))    // 输出结果: 开始缓存 8 // cache: {[2,6]': 8 }
console.log(adder(2, 6))    // 输出结果: 8   //cache: { '[2, 6]': 8 }
console.log(adder(10, 10))  // 输出结果: 开始缓存 20 // cache: { '[2, 6]': 8, '[10, 10]': 20 }
```

3.封装对象的私有属性和私有方法

```
//圣杯模式继承
var inherit = (function(){
    var F = function(){}
    return function(origin,target){
        F.prototype = origin.prototype
        target.prototype = new F()
        target.prototype.constructor = target
    }
}())
```

4.模块化开发，防止污染变量

11.如何实现模块化开发

- 使用立即执行函数，不暴露私有成员

```
var module1 = (function(){
       var _count = 0;
       var m1 = function(){
           //...
       };
       var m2 = function(){
           //...
       };
       return {
           m1 : m1,
           m2 : m2
       };
})();
```

2.说说你对作用域链的理解

- 作用域链的作用就是保证执行环境里有权访问的变量和函数是有序的，作用域链的变量只能向上访问，最终到window对象为止，不可以向下访问。
- 简单来说，作用域就是变量和函数的可访问范围，即作用域控制着函数与变量的可见性和生命周期。

3.原型、原型链

- 原型：
  - JavaScript的所有对象中都包含了一个 [__proto__] 内部属性，这个属性所对应的就是该对象的原型
  - JavaScript的函数对象，除了原型[__proto__] 之外，还预置了 prototype 属性
  - 当函数对象作为构造函数创建实例时，该 prototype 属性值将被作为实例对象的原型 [__proto__]。
- 原型链
  - 当一个对象调用的属性/方法自身不存在时，就会去自己 [__proto__] 关联的前辈 prototype 对象上去找如果没找到，就会去该 prototype 原型 [__proto__] 关联的前辈 prototype 去找。依次类推，直到找到属性/方法或 undefined 为止。从而形成了所谓的“原型链”
- 原型特点
  - JavaScript对象是通过引用来传递的，当修改原型时，与之相关的对象也会继承这一改变

4.请解释什么是事件代理

- 又称事件委托，即把原本需要绑定的事件委托给父元素，让父元素担当事件监听的职务。**原理是DOM元素的事件冒泡**。使用事件代理可以提高性能
  - 节省内存占用，减少事件注册（比如在table上代理所有td的click事件）
  - 可以实现当新增子元素时无需再次对其绑定

5.JavaScript如何实现继承

* 构造函数绑定：使用call或apply将父对象的构造函数绑定在子对象上

```
function Cat(name,color){
   Animal.apply(this, arguments);
   this.name = name;
   this.color = color;
}
```

- 拷贝继承

- 原型继承（圣杯模式）

```
function extend(Child, Parent) {
    var F = function(){};
    F.prototype = Parent.prototype;
    Child.prototype = new F();
    Child.prototype.constructor = Child;
    Child.uber = Parent.prototype;
}
```

- class语法糖：extends

7.事件模型

W3C中定义事件的发生经历有三个阶段：捕获，目标阶段，冒泡

- 冒泡型事件：子元素先触发，父级元素后触发(stopPropagation()方法进行阻止冒泡)
- 捕获型事件：父元素先触发，子元素后触发(preventDefault()方法阻止捕获)
- DOM流事件：同时支持两种事件模型

6.谈谈this的理解

- this总是指向**直接调用者**
- 在事件中，this指向触发这个事件的对象，特殊的是IE中attachEvent中的this指向window对象

8.new操作符具体干了什么

- 创建一个空对象，this变量引用该对象，同时继承了该函数的原型(object.create())
- 属性和方法被加入到this引用的对象中
- 隐式的返回this

9.AJAX原理

- AJAX是在客户端与服务器之间增加了一个中间层(AJAX引擎)，通过XmlHttpRequest对象来向服务器发送异步请求，获得数据后，JavaScript操作DOM进行更新页面。
- AJAX的过程只包含JavaScript、DOM和XmlHttpRequest(和新机制)，使用户操作和服务器响应异步化，其中最关键的一步就是从服务器获得请求数据。

优点

- 通过异步模式，提升了用户体验
- 优化了浏览器与服务器之间的传输，减少了不必要的数据往返，减少了带宽占用

- AJAX在客户端运行，承担了一部分服务器承担的工作，减少了大用户量下的服务器负载
- 可以实现动态不刷新，即局部刷新

缺点

- 暴露了与服务器交互的细节
- 对搜索引擎支持比较弱
- 不易调试

10.如何解决跨域问题

- 其实我们通常所说的跨域是狭义的，是由浏览器同源策略限制的一类请求场景

> 同源策略：SOP(Same Origin Policy)是一种约定，指**协议+域名+端口三者相同**由Netscape引入浏览器，它是浏览器最核心也最基本的安全功能，若缺少了同源策略，浏览器易受到XSS和CSFR等攻击。

怎么解决跨域

- 通过JSONP跨域
  - 所有具有src属性的HTML标签都是可以跨域的，包括<scrip><img><iframe>,所以我们通常会把一些图片资源放到第三方服务器上，然后可以通过HTML标签的src属性引用。
- 用jsonp请求数据的基本流程。

> 首先在客户端注册一个callback, 然后把callback的名字传给服务器。服务器先生成 json 数据。然后以 javascript 语法的方式，生成一个function , function 名字就是传递上来的参数 jsonp.将 json 数据直接以入参的方式，放置到 function 中，这样就生成了一段 js 语法的文档，返回给客户端。客户端浏览器，解析script标签，并执行返回的 javascript 文档，此时数据作为参数，传入到了客户端预先定义好的 callback 函数里.（动态执行回调函数）

- 缺点：没有关于JSONP调用的错误处理，也不能取消或重新开始请求。另外不被信任的服务使用时会很危险。因为JSONP服务返回打包在函数调用中的JSON响应，而函数调用是由浏览器执行的，这使宿主Web应用程序更容易受到各类攻击

  1. 原生实现

```
    var script = document.createElement('script');
    script.type = 'text/javascript';

    // 传参一个回调函数名给后端，方便后端返回时执行这个在前端定义的回调函数
    script.src = 'http://www.domain2.com:8080/login?user=admin&callback=handleCallback';
    document.head.appendChild(script);

    // 回调执行函数
    function handleCallback(res) {
        alert(JSON.stringify(res));
    }
    //服务端返回如下（返回时即执行全局函数）：
    handleCallback({"status": true, "user": "admin"})
```

2. vue实现

```
this.$http.jsonp('http://www.domain2.com:8080/login', {
    params: {},
    jsonp: 'handleCallback'
}).then((res) => {
    console.log(res);
})
```

- document.domain+iframe
- CORS跨域资源共享

> CORS是HTML5标准中定义一种跨域访问的机制，可以让AJAX实现跨域访问。CORS 允许一个域上的网络应用向另一个域提交跨域 AJAX 请求。实现此功能非常简单，只需由服务器发送一个响应标头即可。

- 普通跨域请求，只服务端设置Access-Control-Allow-Origin即可，前端无须设置，若要带cookie请求：前后端都需要设置。目前所有浏览器都支持该功能，**CORS也已经成为主流的跨域解决方案**

1. 原生AJAX实现

```
var xhr = new XMLHttpRequest(); // IE8/9需用window.XDomainRequest兼容

// 前端设置是否带cookie
xhr.withCredentials = true;

xhr.open('post', 'http://www.domain2.com:8080/login', true);
xhr.setRequestHeader('Content-Type', 'application/x-www-form-urlencoded');
xhr.send('user=admin');

xhr.onreadystatechange = function() {
    if (xhr.readyState == 4 && xhr.status == 200) {
        alert(xhr.responseText);
    }
};
```

- nginx代理跨域
  *
- nodejs中间件代理跨域
- WebSocket协议跨域

13.哪些操作会造成内存泄露

> JS内存泄露是指对象在不需要使用它时仍然存在，导致占用的内存不能使用或回收

- 未使用var声明的全局变量
- 闭包
- 循环引用（互相引用）
- 控制台日志
- 移除存在绑定事件的DOM
- setTimeout的第一个参数使用字符串而非函数

14.XML和KSON的区别

- 数据体积：JSON比XML小，传递速度快
- 数据交互：JSON与JS交互方便
- 数据描述：XML描述性更好
- 传输速度：JSON要远快于XML

15.谈谈你对webpack的看法

- webpack是一个模块打包工具，你可以使用webpack管理你的模块依赖，并编译输出模块模块们所需的静态文件。对于不同类型的资源，webpack有对应的模块加载器。它的模块打包器会分析模块间的依赖关系，最后生成了优化且合并后的静态资源

16. 说说你对AMD和CommonJS的理解

- CommonJS是服务器端模块的规范，Node.js采用了这个规范。它的模块加载是同步的，只有加载完成，才能进行后面的操作。
- AMD规范是非同步加载模块，允许指定回调函数。它推荐的风格是**通过返回一个对象作为模块对象，而CommonJS的风格是通过module.exports或exports的属性赋值来达到暴露模块对象的目的**

27.es6模块，CommonJS和AMD

- CommonJS中，每个JS文件就是一个独立的模块上下文，创建的属性是私有的，对其他文件不可见。且它是同步的，在浏览器中会阻塞，所以不适用。
- AMD是异步，需要定义回调方式
- ES6中一个模块就是一个独立的文件，同样内部变量外部无法获取。但可通过export输出变量或类和方法。

17.常见web安全及防护原理

26.Node的应用场景

- 特点：
  - 它是一个JS运行环境
  - 依赖于V8引擎进行代码解释
  - 事件驱动
  - 非阻塞I/O
  - 单进程，单线程
- 优点：
  - 高并发（最重要的优点）
- 缺点：
  - 不能充分利用CPU

57.异步变成的实现方式

- 回调函数
  - 优点：简单，容易理解
  - 缺点：不利于维护，代码耦合高
- 事件监听
  - 优点：可以绑定多个事件
  - 缺点：流程不够清晰
- 发布/订阅（观察者模式）

    - 类似于事件监听，但可通过消息中心，了解存在多少发布者与订阅者
- Promise对象
    - 优点：有then方法，并且可以处理错误，可链式书写
    - 缺点：编写和理解相对比较难
- Generator函数
    - 优点：函数体内外的数据交换、错误处理机制
    - 缺点：流程管理不方便
- async函数
    - 优点：内置执行器、更好的语义、更广的适用性、返回的是Promise、结构清晰
    - 缺点：错误处理机制

66.如何渲染几万条数据并不卡住页面

> 即每次渲染一部分DOM，就可以通过requestAnimationFrame来每16ms刷新一次

```
<body>
  <ul>控件</ul>
  <script>
    setTimeout(() => {
      // 插入十万条数据
      const total = 100000
      // 一次插入 20 条，如果觉得性能不好就减少
      const once = 20
      // 渲染数据总共需要几次
      const loopCount = total / once
      let countOfRender = 0
      let ul = document.querySelector("ul");
      function add() {
        // 优化性能，插入不会造成回流
        const fragment = document.createDocumentFragment();
        for (let i = 0; i < once; i++) {
          const li = document.createElement("li");
          li.innerText = Math.floor(Math.random() * total);
          fragment.appendChild(li);
        }
        ul.appendChild(fragment);
        countOfRender += 1;
        loop();
      }
      function loop() {
        if (countOfRender < loopCount) {
          window.requestAnimationFrame(add);
        }
      }
      loop();
    }, 0);
  </script>
</body>
```

70. callee和caller有什么作用?

- caller是返回一个函数的引用，该函数调用了当前函数
- callee是返回正在被执行的function函数，即所指定的function对象的正文

80.项目做过哪些性能优化

> 减少 HTTP 请求数，减少 DNS 查询，使用 CDN，避免重定向，图片懒加载，减少 DOM 元素数量，减少DOM 操作，使用外部 JavaScript 和 CSS，压缩 JavaScript 、 CSS 、字体、图片等，优化 CSS Sprite，使用 iconfont，字体裁剪，多域名分发划分内容到不同域名，尽量减少 iframe 使用，避免图片 src 为空，把样式表放在link 中，把JavaScript放在页面底部，

82 WebSocket

> 由于HTTP存在一个明显的缺陷，即消息只能从客户端推送到服务端，而使用轮询解决服务端连续变化的问题时，效率过低，因此使用WebSocket

- 支持双向通信，实时性更强
- 可以发送文本，也可以发送二进制文件
- 较少的控制开销
- 支持扩展
- 无跨域问题

84.深浅拷贝

- 浅拷贝：Object.assign或者展开运算符
- 深拷贝：

```
//对于非symbol、funtion和undefined对象时，可以使用JSON.stringify()和JSON.parse进行深拷贝
//其余情况
function deepClone (obj) {
    const targetObj = obj.constructor === Array ? [] : {}
    for (let keys in obj) {
        if (obj.hasOwnProperty(keys)) {
            if (obj[keys] && typeof obj[keys] === "object") {
                targetObj[keys] = obj[keys].constructor === Array ? [] : {}
                targetObj[keys] = deepClone(obj[keys])
            } else {
                targetObj[keys] = obj[keys]
            }
        }
    }
    return targetObj
}
//数组方法slice和concat都只对第一层进行深拷贝
//对象的...展开实现的是对象带第一层的深拷贝，后面的拷贝都是引用值
```

85.防抖/节流

- 防抖

> 在滚动事件中需要做个复杂计算或者实现一个按钮的防二次点击操作。可以通过函数防抖动来实现

对于按钮防点击来说的实现：

- 开始一个定时器，只要定时器还在，不论如何点击按钮，都不会执行回调函数。一旦定时器结束并设置为null，就可以再次点击了

对于延时执行函数来说的数显：

- 每次调用防抖动函数都会判断本次调用和之前的时间间隔，如果小于需要的时间间隔，就会重新创建一个定时器，并且定时器的延时为设定时间减去之前的时间间隔。一旦时间到了，就会执行相应的回调函数

```
function debounce(fn,delay) {
    var timer;
    return function() {
        var _this=this,
        var args = arguments;
        if(timer){
            clearTimeout(timer)
        }else{
            timer = setTimeout(function() {
                fn.apply(_this,args)
            },delay)
        }
    }
}
```

```
// 使用 underscore 的源码来解释防抖动
/**
 * underscore 防抖函数，返回函数连续调用时，空闲时间必须大于或等于 wait，func 才会执行
 *
 * @param  {function} func        回调函数
 * @param  {number}   wait        表示时间窗口的间隔
 * @param  {boolean}  immediate   设置为ture时，是否立即调用函数
 * @return {function}             返回客户调用函数
 */
_.debounce = function(func, wait, immediate) {
    var timeout, args, context, timestamp, result;

    var later = function() {
      // 现在和上一次时间戳比较
      var last = _.now() - timestamp;
      // 如果当前间隔时间少于设定时间且大于0就重新设置定时器
      if (last < wait && last >= 0) {
        timeout = setTimeout(later, wait - last);
      } else {
        // 否则的话就是时间到了执行回调函数
        timeout = null;
        if (!immediate) {
          result = func.apply(context, args);
          if (!timeout) context = args = null;
        }
      }
    };

    return function() {
      context = this;
      args = arguments;
      // 获得时间戳
      timestamp = _.now();
      // 如果定时器不存在且立即执行函数
      var callNow = immediate && !timeout;
      // 如果定时器不存在就创建一个
      if (!timeout) timeout = setTimeout(later, wait);
      if (callNow) {
        // 如果需要立即执行函数的话 通过 apply 执行
        result = func.apply(context, args);
        context = args = null;
      }

      return result;
    };
  };
```

- 节流

> 二者本质不一样，防抖动是将多次执行变为一次执行，而节流是将多次执行变为每隔一段时间执行

```
function throttle(fn,delay){
    var timer;
    return function() {
        var _this = this;
        var _args = arguments;
        if(timer){
            return
        }else{
            timer = setTimeout(function(){
                fn.apply(_this,_args)
                timer = null;
            },delay);
        }
    }
}
function throttle2(func,delay){
    var previous;
    return function(){
        var _this = this
        var _args = arguments
        var now = new Date()
        if(now - previous > delay){
            func.apply(_this,_args)
            previous = now;
        }
    }
}
```

```
/**
 * underscore 节流函数，返回函数连续调用时，func 执行频率限定为 次 / wait
 *
 * @param  {function}   func      回调函数
 * @param  {number}     wait      表示时间窗口的间隔
 * @param  {object}     options   如果想忽略开始函数的的调用，传入{leading: false}。
 *                                如果想忽略结尾函数的调用，传入{trailing: false}
 *                                两者不能共存，否则函数不能执行
 * @return {function}             返回客户调用函数
 */
_.throttle = function(func, wait, options) {
    var context, args, result;
    var timeout = null;
    // 之前的时间戳
    var previous = 0;
    // 如果 options 没传则设为空对象
    if (!options) options = {};
    // 定时器回调函数
    var later = function() {
      // 如果设置了 leading，就将 previous 设为 0
      // 用于下面函数的第一个 if 判断
      previous = options.leading === false ? 0 : _.now();
      // 置空一是为了防止内存泄漏，二是为了下面的定时器判断
      timeout = null;
      result = func.apply(context, args);
      if (!timeout) context = args = null;
    };
    return function() {
      // 获得当前时间戳
      var now = _.now();
      // 首次进入前者肯定为 true
          // 如果需要第一次不执行函数
          // 就将上次时间戳设为当前的
      // 这样在接下来计算 remaining 的值时会大于0
      if (!previous && options.leading === false) previous = now;
      // 计算剩余时间
      var remaining = wait - (now - previous);
      context = this;
      args = arguments;
      // 如果当前调用已经大于上次调用时间 + wait
      // 或者用户手动调了时间
          // 如果设置了 trailing，只会进入这个条件
          // 如果没有设置 leading，那么第一次会进入这个条件
          // 还有一点，你可能会觉得开启了定时器那么应该不会进入这个 if 条件了
          // 其实还是会进入的，因为定时器的延时
          // 并不是准确的时间，很可能你设置了2秒
          // 但是他需要2.2秒才触发，这时候就会进入这个条件
      if (remaining <= 0 || remaining > wait) {
        // 如果存在定时器就清理掉否则会调用二次回调
        if (timeout) {
          clearTimeout(timeout);
```

```
        timeout = null;
      }
      previous = now;
      result = func.apply(context, args);
      if (!timeout) context = args = null;
    } else if (!timeout && options.trailing !== false) {
      // 判断是否设置了定时器和 trailing
          // 没有的话就开启一个定时器
      // 并且不能不能同时设置 leading 和 trailing
      timeout = setTimeout(later, remaining);
    }
    return result;
  };
};
```

87.什么是单线程，和异步的关系

- 单线程：只有一个线程，只能做一件事
- 原因：避免DOM渲染的冲突
- 解决方案：异步

93.请简单实现双向数据绑定

```
const data = {}
const input = document.getElementById('input')
Object.defineProperty(data,'text',{
    set(value){
        this.value = value ;
        input.value = value;
    }
});
input.onChange = function(e){
    data.text = e.target.value;
}
```

94.实现Storage，使得该对象为单例，并对localStorage进行封装设置值SetItem(key,value)和getItem(key)

```
var instance = null;
class Storage {
  static getInstance() {
    if (!instance) {
      instance = new Storage();
    }
    return instance;
  }
  setItem = (key, value) => localStorage.setItem(key, value),
  getItem = key => localStorage.getItem(key)
}
```

95.说说event loop

微任务与宏任务
宏任务主要包含：script(整体代码)、setTimeout、setInterval、I/O、UI交互事件、setImmediate(Node.js 环境)

微任务主要包含：Promise、MutaionObserver、process.nextTick(Node.js 环境)
所以正确的一次 Event loop 顺序是这样的

- 执行同步代码，这属于宏任务
- 构造函数执行
- 执行栈为空，查询是否有微任务
- 执行所有微任务
- 必要的话渲染UI
- 开始下一轮Event loop

```
console.log('script start');

setTimeout(function () {
    console.log('timeout1');
},10);
//Promise.resolve 方法允许调用时不带参数，直接返回一个resolved 状态的 Promise 对象。立即 resolved 的
Promise.resolve().then(console.log('then2'))
new Promise(resolve => {
    console.log('promise1');
    resolve();
    setTimeout(() => console.log('timeout2'), 0);
}).then(function () {
    console.log('then1')
})

console.log('script end');
//script start - then2 - promise1 - script end- then1  - timeout2 -timeout1
```