

js基础

break,continue,return区别

- 跳出整个循环，执行循环外的代码 —— break
- 跳出本次循环，进入下次循环 —— continue
- 结束整个函数 —— return

堆和栈

- 内存一般储存基础数据和函数类型，后进先出的原则。
 - stack是有结构的，每个区块按照一定次序存放，可以明确知道每个区块的大小；
 - stack创建的时候，大小是确定的，数据超过这个大小，就发生stack overflow错误
 - 每个线程分配一个stack，线程独占
- 内存一般储存引用数据类型
 - JavaScript不允许直接访问堆内存中的位置
 - heap是没有结构的，数据可以任意存放。因此，stack的寻址速度要快于heap。
 - heap的大小是不确定的，需要的话可以不断增加。
 - 每个进程分配一个heap，线程公用

i++和++i

- 在for循环语句中，前++和后++没什么区别。因为他们是单独成语句
 - 如果自加语句单独成句，则前++和后++没有区别
- i++是先进性赋值或比较后，自加
 - 如果自加语句和赋值，比较等形成一个语句，就有区别了
- ++i是先进性自加，再进行赋值和比较

null和undefined

- 转换为数字是0 —— null
- 变量值为null可以被垃圾回收
- 是一个特殊对象，可以作为参数传递
- 转换为数字是NaN —— undefined
- 只代表一个变量未被赋值

闭包

- 概念
 - 闭包是指一个有权访问另一个函数作用域中的变量的函数
 - 或者子函数在声明之外的地方被调用，子函数所在的父函数的作用域不会被释放
- 形成原因
 - 函数的内部存在对外部作用域的引用就会导致闭包。
 - var a = 0; function foo(){ var b = 14; function fo(){ console.log(a, b); } fo(); }
 - 这里的子函数 fo 内存就存在外部作用域的引用 a, b, 所以这就会产生闭包
- 闭包变量存储的位置
 - 闭包中的变量存储的位置是堆内存。
 - 假如闭包中的变量存储在栈内存中，那么栈的回收 会把处于栈顶的变量自动回收。所以闭包中的变量如果处于栈中那么变量被销毁后，闭包中的变量就没有了。所以闭包引用的变量是出于堆内存中的。
- 原理
 - 闭包的实现原理，根本上来说是作用域链。
 - 每一个函数(包括全局和自定义函数)，在词法解析阶段，都会有自己的词法作用域。当我们调用一个函数的时候，若该环境没有被js回收机制回收时，则我们仍可以通过其来引用它原始的作用域链。
- 应用场景
 - 手动延长某些局部变量的寿命
 - 函数作为参数
 - 所有的回调函数
 - 防抖，节流
 - 自执行函数
- 优缺点
 - 缺点：
 - 会引起内存泄漏
 - 我们在闭包中引用的变量，JS的回收机制不会主动的进行释放。当达到一定量后，会引起内存泄漏
 - 但其实闭包的内存泄露并不是因为闭包自身的机制，而是来源于某些浏览器针对DOM和BOM对象使用的是引用计数回收。当两个对象相互引用的时候，自然就发生了计数永不为零，而永远占用空间的情况。
 - 优点：
 - 保护函数的私有变量不受外部的干扰。形成不销毁的栈内存。
 - 保存，把一些函数内的值保存下来。闭包可以实现方法和属性的私有化
- 参考链接
 - https://github.com/HXWfromDJTU/blog/issues/43

for...in...和for...of..

- for...in...
 - 可以遍历对象，数组
 - 可以遍历对象，数组，但更适合遍历对象
 - for...in...取得是key/index
 - for...in... 中的index是string类型， 不是num类型
 - for...in... 会把原型链上的方法打印出来 —— 可以使用hasOwnProperty来避免遍历原型
 - 遍历顺序可能是乱序的
- for...of..
 - 能遍历可迭代对象
 - 遍历可迭代对象 —— for-of循环每执行一次都会调用可迭代对象的next()方法 —— 只有实现了Iterator接口的对象才能够使用 for of 来进行遍历取值。
 - 可以简单，正确的遍历数组
 - 不遍历原型
 - 取的是value
 - break,continue,return都能正常使用

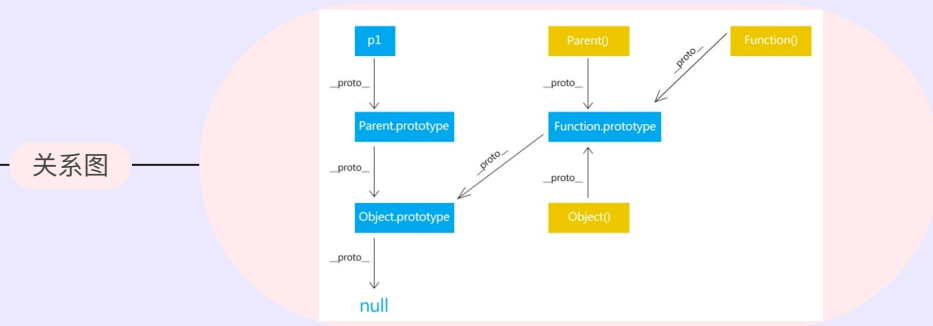
箭头函数和普通函数的区别

- 箭头函数只有函数表达式，没有函数声明式
- 箭头函数没有constructor，没有原型链(有__proto__，但是没有prototype)
- 箭头函数没有arguments属性
- 箭头函数的this指向最近一层包裹它的普通函数的this。这个this有需要的属性值就用，没有拉倒，也不往上继续找。如果没有普通函数包裹，那就指向全局this(在浏览器中全局this是window，在node中全局this是个空对象{}).
- call,apply,bind无法改变箭头函数中的this指向
 - let obj = { school: "sxau", func: (a,b) => { console.log(this.school,a,b); } }; let func = obj.func; func.call(obj,1,2); // undefined 1 2
- 箭头函数为什么不能作为构造函数?
 - 因为构造函数需要用new创建一个空对象，将空对象.__proto__ = 构造函数.prototype。但是箭头函数没有prototype

数组浅拷贝

- concat
- slice()
- Array.from()
- ...

原型链



继承

- 原型链继承
 - Student.prototype = new Person()
 - 父类型所有内容子类共享，子类可以修改掉父类的属性和方法
 - 缺点：
 - 每新增一个子类就要都要操作一下它的原型链
 - 没有办法在子类自己的原型链上添加方法
 - 没有办法给父类传参
- 构造函数继承
 - Person.call(this, name, age) —— 在子类型的构造函数内部执行
 - 缺点：
 - 没办法继承父类型原型链上的内容
 - 每个子类实例都拷贝了一份父类方法和属性
 - 优点：
 - 创建子类时，可以给父类传参
 - 可以实现多继承，可以call多个父类
- 混合继承
 - Person.call(this,name,age) —— 在子类型的构造函数内部执行
 - Student.prototype = new Person()
 - Student.prototype.constructor = Student
 - 缺点：
 - 调用了两次父类构造函数，生成了两份实例
- 混合继承优化
 - Person.call(this,name,age) —— 在子类型的构造函数内部执行
 - Student.prototype = Object.create(Person.prototype)
 - Student.prototype.constructor = Student
- ES6继承
 - extends
 - super

判断数据类型

- typeof
 - typeof 'blubber' // string
 - 返回消除类型字符串
 - 能判断出: number, string, undefined, boolean, function
 - 无法区分 (都返回object): object, array, null, new创建出来的
 - 判断后者是否在前者的原型链上，比较适合判断引用类型
- instanceof
 - eg: 234 instanceof Number // true
 - 弊端：
 - 1.不能区分null和undefined
 - 2.基本类型的，不是实例的判断不了 —— 比如: "ggg" instanceof String // false
 - 3.arr function obj instanceof Object 都返回true
- constructor
 - eg: let num = 233; num.constructor === Number // true
 - 缺点：
 - 1.不能区分null和undefined (null,undefined没有constructor属性)
 - 2.这种判断不安全 因为constructor指向可变
- Object.prototype.toString.call()
 - eg: Object.prototype.toString.call(null) // "[object Null]"
 - 优点: 能区分所有类型
 - 缺点: —— 非原生实例检测不出构造函数名
- isArray
 - eg: Array.isArray([1, 2, 3]); // true

eventloop

- https://juejin.cn/post/6844904056159223816
- 微任务有哪些
 - Promise.then
 - MutationObserver —— MutationObserver: MO是HTML5中的API，是一个用于监听DOM变动的接口。它可以监听一个DOM对象上发生的子节点删除、属性修改、文本内容修改等。 —— 调用过程是要先给它绑定回调，得到MO实例，这个回调会在MO实例监听到变动时触发。这里MO的回调是放在microtask (微任务) 中执行的。
 - process.nextTick
- 如果在微任务执行期间微任务队列加入了新的微任务，会将新的微任务加入队列尾部，之后也会被执行。

JS模块化

- commonJS
 - require/module.exports
 - node的模块方案
 - 同步加载 —— 因为node是服务端，所以服务端开发请求没什么问题，但是客户端需要异步加载，同步加载文件会需要很长时间
 - 输出的是值的拷贝
 - 运行时加载
- AMD
 - require/define
 - 依赖前置
 - 异步模块加载
 - 是RequireJS 在的推广和普及过程中被创造出来。
- CMD
 - 异步加载
 - 是SeaJS 在的推广和普及过程中被创造出来。
- ES6模块化
 - import/export
 - 输出的是值的引用
 - 编译时加载
 - 前面几种都是前端社区实现的，ES6才是真正的官方出品
 - 既可以服务器端使用，也可以在浏览器中使用
 - 可以识别 type="module" 的浏览器会忽略带有 nomodule 属性的 script 标签，可以通过这种方式为不支持 ESM 的浏览器做降级处理。
 - 如果浏览器支持 ESM，这也意味着浏览器支持其他的 ES6 语法，像 Promise, async-await, 可以通过这种方法减少代码中 ES6 等高级语法的编译，减少代码打包体积，提升加载速度和 JS 代码的执行速度，提高页面性能，然后对不支持 ESM 的浏览器提供编译后的版本，做降级处理
 - 默认的 <script> 标签加载资源会阻塞 HTML 解析，可以通过 defer 和 async 属性来 —— defer 与 async 的区别是: defer 要等到整个页面在内存中正常渲染结束 (DOM 结构完全生成，以及其他脚本执行完成)，才会执行；async 一旦下载完，渲染引擎就会中断渲染，执行这个脚本以后，再继续渲染。一句话，defer 是“渲染完再执行”，async 是“下载完就执行”下载是异步的。另外，如果有多个 defer 脚本，会按照它们页面出现的顺序加载。而多个 async 脚本是不能保证加载顺序的。
 - ESM 在浏览器中会有 CORS 跨域问题，所有跨域的 ESM 资源加载都需要在资源响应头上加 Access-Control-Allow-Origin 的响应头，而在之前的 js 资源加载上是不需要的。

websocket

- 区别是:
 - HTTP协议只能由客户端主动发起通信
 - websocket协议可以让服务器主动给客户端推送消息。双向通信。实现客户端与服务器的平等对话
- 是基于HTTP协议的websocket协议
- 特点
 - 建立在TCP协议上，比较容易实现
 - 与HTTP协议兼容
 - 数据格式轻量，性能开销小，通信效率高
 - 可以发送文本和二进制数据
- 语法
 - 协议标识符是ws
 - eg: ws://example.com:80/some/path
 - 没有同源限制，客户端和服务端可以任意通信
 - 与服务端连接: var ws = new WebSocket('ws://localhost:8080');
 - ws.onopen连接成功后的回调，发送数据ws.send()
 - ws.onmessage收到数据后的回调，不需要连接的远关闭连接ws.close
 - ws.onclose连接关闭后的回调
 - Connection: Upgrade
 - Upgrade: websocket
- 请求头变化
 - sec-websocket-key: 客户端随机生成的字符串。
 - sec-websocket-accpet: 服务端加密sec-websocket-key后生成的字符串
 - Sec-WebSocket-Protocol: 客户端和服务端商量用哪个协议
- 断线问题
 - 如果是超时断线，可以发送心跳包保活
 - 下一个定时器，在一段时间间隔下发一个空包给客户端，然后客户端反馈一个同样的空包回来，服务器如果在一定时间内收不到客户端发送过来的反馈包，那就只有认定说掉线了。