Day01笔记

|  |
| --- |
| JavaScript  1.初识JavaScript  1.1什么是JavaScript  JavaScript(JS)是一种具有函数优先的轻量级、解释型或即时编译型的编程语言。虽然它因作为开发网页的脚本语言而出名，但它也被用于很多非浏览器环境中，例如Node.js、ApacheCouchDB和AdobeAcrobat。JavaScript是一种基于原型编程、多范式的动态脚本语言，并且支持面向对象、命令式和声明式（如函数式编程）风格。  1.2JavaScript的组成部分  ECMAScript：描述了该语言的语法和基本对象。  文档对象模型（DOM）：描述处理网页内容的方法和接口。  浏览器对象模型（BOM）：描述与浏览器进行交互的方法和接口。  1.3JavaScript的历史  -1994年，网景公司（Netscape）发布了Navigator浏览器0.9版，这是历史上第一个比较成熟的网络浏览器。  -1995年，网景公司决定开发一种网页脚本语言，使浏览器可以与网页互动。  -1995年，BrendanEich设计了JavaScript，最初命名为Mocha，后改名为LiveScript，再改名为JavaScript。  -1996年，微软发布了支持JScript的IE3。  -1997年，第一个ECMAScript标准发布。  2.JavaScript的特点  1.\*\*解释型语言\*\*：被内置于浏览器或Node.js平台中的JS解析器解析执行，无需编译。  2.\*\*弱类型语言\*\*：变量的数据类型取决于值的数据类型。  3.\*\*顺序执行\*\*：从上到下顺序解析执行。  4.\*\*跨平台\*\*：既可以作为前端脚本语言，也可以作为后端语言。  3.JavaScript的组成  1.ECMAScript：JS的标准，类似于核心Java，制定了基础的语法。  2.DOM：文档对象模型，JS操作HTML的API。  3.BOM：浏览器对象模型，JS操作浏览器的API。  4.在网页中使用JavaScript  4.1内部JavaScript  在<head>标签中添加<script>元素，然后将JS代码填写进来即可。  <!DOCTYPEhtml>  <htmllang="en">  <head>  <metacharset="UTF-8">  <title>Document</title>  <scripttype="text/javascript">  alert('HelloJavaScript');  </script>  </head>  <body>  </body>  </html>  4.2外部JavaScript  单独新建一个后缀名为.js的JS文件，在HTML文件中使用<script>标签的src属性将JS文件导入进来。  <!DOCTYPEhtml>  <htmllang="en">  <head>  <metacharset="UTF-8">  <title>Document</title>  <scripttype="text/javascript"src="hello.js"></script>  </head>  <body>  </body>  </html>  5.编辑器与运行环境  -编辑器：vi、vim、vscode、sublime。  -运行环境：浏览器控制台、Node.js。  6.JavaScript关键字与保留字  6.1关键字  在JS中有特殊功能的保留字，如break、case、catch、continue、debugger、default、delete、do、else、finally、for、function、if、in、instanceof、new、return、switch、this、throw、try、typeof、var、void、while、with。6.2保留字  将来可能成为关键字的保留字，如abstract、boolean、byte、char、class、const、debugger、double、enum、export、extends、final、float、goto、implements、import、int、interface、long、native、package、private、protected、public、short、static、super、synchronized、throws、transient、volatile。  7.注释  单行注释：//aaa  多行注释：/\*aaa\*/  日期2024.7.18 |

Day02笔记

|  |
| --- |
| undefined继承null，所以undefined == null结果为true，但是null表示空对象，undefined表示未定义；null与undefined⽤途不同，null可以⽤来表示⼀个空对象但是没有必要把⼀个变量的值显式设置为undefined  深拷⻉与浅拷⻉主要针对于引⽤数据类型参数说的，浅拷⻉表示仅拷⻉引⽤地址，深拷⻉ 表示对于对象的克隆。  实现深拷⻉的⽅法 通过json对象实现深拷⻉（JSON.stringify，JSON.parse） Object.assign()拷⻉ lodash函数库实现深拷⻉ 递归的⽅式实现深拷⻉  使⽤typeof判断数据类型  返回该变量名所指向数据的类型 语法：  typeof 变量名 返回值如下：  "undefined"未定义  "boolean"布尔类型  "string" 字符串  "number"数值  "object" 对象或者null或者数组  "function" 函数  日期2024.7.19 |

Day03笔记

|  |
| --- |
| **1. 操作符**  1.1 算术运算符   * **加法 (+)**: 数字相加，字符串拼接。   var num = 1 + 2; // 3  var str = "Hello" + " " + "World"; // "Hello World"   * **减法 (-)**: 数字相减，可以将字符串转换成数字后相减。   var num = 4 - 3; // 1  var num = "8" - "4"; // 4   * **乘法 (\*)**: 数字相乘，可以将字符串转换成数字后相乘。   var num = 4 \* 3; // 12   * **除法 (/)**: 数字相除，可以将字符串转换成数字后相除。   var num = 6 / 3; // 2   * **取余 (%)**: 取余数操作。   var num = 7 % 3; // 1  1.2 一元运算符   * **正号 (+)**: 将操作数转换成数字。   var num = +"123"; // 123   * **负号 (-)**: 将操作数转换成负数。   var num = -"123"; // -123   * **逻辑取反 (!)**: 将布尔值取反。   var bool = !true; // false   * **递增 (++)**: 变量自增1。   var num = 1;  num++; // 2   * **递减 (--)**: 变量自减1。   var num = 1;  num--; // 0   * **删除 (delete)**: 删除对象或数组的属性。   var obj = { name: 'John', age: 30 };  delete obj.age; // { name: 'John' }   * **类型判断 (typeof)**: 返回操作数的类型。   var type = typeof 123; // "number"  **2. 赋值运算符**   * **赋值 (=)**: 赋值操作。   var x = 5;   * **加等 (+=)**: 相加后赋值。   var x = 5;  x += 2; // 7   * **减等 (-=)**: 相减后赋值。   var x = 5;  x -= 2; // 3   * **乘等 (\*=)**: 相乘后赋值。   var x = 5;  x \*= 2; // 10   * **除等 (/=)**: 相除后赋值。   var x = 5;  x /= 2; // 2.5   * **取余等 (%=)**: 取余后赋值。   var x = 5;  x %= 2; // 1  **3. 比较运算符**   * **等于 (==)**: 仅值相等即可。   var result = (5 == "5"); // true   * **全等 (===)**: 值和类型都要相等。   var result = (5 === "5"); // false   * **不等于 (!=)**: 值不等即可。   var result = (5 != "5"); // false   * **不全等于 (!==)**: 值或类型不等。   var result = (5 !== "5"); // true   * **大于 (>)**: 大于。   var result = (5 > 3); // true   * **小于 (<)**: 小于。   var result = (5 < 3); // false   * **大于等于 (>=)**: 大于或等于。   var result = (5 >= 3); // true   * **小于等于 (<=)**: 小于或等于。   var result = (5 <= 3); // false  **4. 隐式转换**   * **ToPrimitive**: 将值转换为原始值。 * **ToNumber**: 将值转换为数字。 * **ToString**: 将值转换为字符串。   **隐式转换规则**   1. 字符串与数字相加时，数字会被转换为字符串。   var result = 1 + "2"; // "12"   1. 字符串与数字相减时，字符串会被转换为数字。   var result = "5" - 3; // 2  **5. 流程控制语句**   1. **if 语句**: 条件判断。   if (condition) {  // TODO  }   1. **switch 语句**: 多条件判断。   switch (expression) {  case value1:  // TODO  break;  case value2:  // TODO  break;  default:  // TODO  }   1. **for 循环**: 计数器控制的循环。   for (var i = 0; i < 5; i++) {  // TODO  }   1. **while 循环**: 条件控制的循环。   j  while (condition) {  // TODO  }   1. **do...while 循环**: 至少执行一次的循环。   do {  // TODO  } while (condition);  **使用注意**   * 尽量使用严格比较运算符（===），避免隐式转换带来的困惑。 * 注意类型转换的规则和顺序，避免意外的结果。   日期2024.7.22 |

Day04笔记

|  |
| --- |
| **1. JavaScript 对象基础**   * **对象定义**: JavaScript中的对象是无序属性的集合，其属性可以包含基本值、对象或函数。对象是由键值对（key   ）组成，值可以是数据或函数。   * **现实中的对象类比**: 每个人都是一个对象，具有属性（如身高和体重）和方法（如走路和跑步）。每个人的属性和方法虽相似，但具体值和执行时间各不相同。 * **对象创建**:   + **字面量模式**: 使用 {} 作为对象的边界，属性之间用 , 隔开，属性名和属性值用 : 隔开。   + **构造函数模式**: 使用 Object 或自定义构造函数来初始化对象。   var person = {name:"张三", height:1.71, gender: 'male'};  var obj = new Object();  obj.name = "terry";  obj.age = 12;  obj.sayName = function() {  console.log("my name is", this.name);  };  **2. 对象的访问**   * **属性访问**: 有点访问（.）和中括号访问（[]）两种方式。   console.log(obj.name);  console.log(obj['name']);   * **方法的访问**: 需要按照函数调用的方式使用对象的方法。   obj.sayName(); // 调用对象的方法  **3. 遍历对象的属性**   * 使用 for..in 循环遍历对象的属性。   for (var key in obj) {  console.log(key + "--" + obj[key]);  }  **4. 新增和删除对象中的属性**   * 新增属性可以直接赋值。   obj.newProperty = "newValue";   * 删除属性使用 delete 关键字。   delete obj.propertyName;  **5. 对象的强制类型转换**   * **转换为 Boolean**:   var obj = { name: "briup", age: 12 };  console.log(Boolean(obj)); // true   * **转换为 String**: 调用 toString 方法。   console.log(obj.toString());   * **转换为 Number**: 调用 valueOf 方法。   var obj = {  valueOf: function() { return 10; }  };  console.log(Number(obj)); // 10  日期2024.7.23 |

Day05笔记

|  |
| --- |
| **开题报告、开题PPT，任务书，选题审批表**  **1. 检测属性**   * **in 运算符**: 检测属性是否存在于对象中，无论是自有属性还是继承属性。   console.log('name' in obj); // true   * **hasOwnProperty**: 检测属性是否为对象的自有属性。   console.log(obj.hasOwnProperty('name')); // true   * **propertyIsEnumerable**: 检测属性是否为对象的可枚举自有属性。   console.log(obj.propertyIsEnumerable('name')); // true  **2. 对象原型属性及方法**   * **Object 构造函数的原型属性和方法**: 包括 constructor、hasOwnProperty、propertyIsEnumerable、valueOf、toLocaleString、toString 和 isPrototypeOf。   var obj = new Object();  console.log(obj.toString());  console.log(Object.prototype);  **3. 深入理解对象-定义属性**   * **数据属性**: 包含 [[Configurable]]、[[Enumerable]]、[[Writable]] 和 [[Value]] 特性。   Object.defineProperty(obj, 'name', {  configurable: true,  enumerable: true,  writable: true,  value: 'terry'  });   * **访问器属性**: 包含 [[Configurable]]、[[Enumerable]]、[[Get]] 和 [[Set]] 特性。   Object.defineProperty(obj, 'fullName', {  get: function() { return this.firstName + ' ' + this.lastName; },  set: function(value) {  var parts = value.split(' ');  this.firstName = parts[0];  this.lastName = parts[1];  }  });  日期2024.7.24 |

Day06笔记

|  |
| --- |
| **函数概述与声明**  **1. 函数介绍**   * **定义**：函数是封装一系列代码以完成特定任务的结构。 * **用途**：   + 功能封装，提高代码复用率   + 构建对象的模板（构造函数） * **函数类型**：每个函数都是 Function 类型的实例，具有属性和方法。   **2. 函数声明与表达式**   * **函数声明**：   function 函数名(形参列表) {  // 函数体  }  示例：  function add(a, b) {  var result = a + b;  return result;  }  var total = add(1, 2); // total = 3   * **函数表达式**：   var 函数名 = function(形参列表) {  // 函数体  }  示例：  var add = function(a, b) {  var result = a + b;  return result;  }  **3. 函数提升**   * 函数声明会被提升到代码的最前面，可以在声明前调用。   foo(); // 可以调用  function foo() {  console.log("Hello World");  }   * 函数表达式不会提升，必须在赋值后调用。   **4. 函数内部属性**   * **arguments**：类数组对象，包含传入函数的所有参数。   + arguments.callee：指向拥有该 arguments 对象的函数。   function add(a, b) {  console.log(arguments[0], arguments[1]);  return a + b;  }  add(10, 20); // 输出：10 20  **5. this 关键字**   * **用法**：   + 方法中：this 指向方法所属的对象。   + 函数中：默认指向全局对象（浏览器中为 window，Node.js 中为 global）。   + 事件中：指向接收事件的元素。   + 显式绑定：使用 call 和 apply 切换 this 的上下文。   var person = {  firstName: "张",  lastName: "建安",  fullName: function() {  return this.firstName + " " + this.lastName;  }  };  console.log(person.fullName()); // 输出  **1. IIFE（立即调用函数表达式）**   * **定义**：声明函数的同时立即调用该函数。   (function() {  console.log("IIFE called");  })();   * **用途**：   + 只执行一次的初始化代码   + 创建局部作用域，防止变量污染全局作用域   **2. 作用域**   * **全局作用域**：函数外声明的变量，全局可访问。 * **局部作用域**：函数内部声明的变量，仅在函数内可访问。   **3. 作用域链**   * **定义**：嵌套函数在当前作用域找不到变量时，会沿着作用域链向上查找直至全局作用域。   var a = 100;  function outer() {  var b = 200;  function inner() {  var c = 300;  console.log(a); // 100  console.log(b); // 200  console.log(c); // 300  }  inner();  }  outer();  **4. 闭包**   * **定义**：函数能够记住并访问其词法作用域，即使函数在其词法作用域之外执行。   function outer() {  var a = 100;  return function inner() {  console.log(a); // 100  }  }  var fn = outer();  fn();  **5. 经典面试题 - IIFE 和作用域**   * **IIFE 用于循环中的变量隔离**：   for (var i = 0; i < 5; i++) {  (function(j) {  console.log(j); // 输出：0, 1, 2, 3, 4  })(i);  }   * **解释**：IIFE 创建了一个新的作用域，使得每次循环的 i 值作为参数传递给 IIFE，从而避免变量提升导致的错误。   日期2024.7.25 |

Day07笔记

|  |
| --- |
| 什么是闭包？  简单讲，闭包就是指有权访问另⼀个函数作⽤域中的变量的函数。  MDN 上⾯这么说：闭包是⼀种特殊的对象。它由两部分构成：函数，以及创建该函数的环境。环境由闭包创建时在作⽤域中的任何局部变量组 成。  闭包的⽣成有三个必要条件 函数嵌套函数 内部函数引⽤了外部函数中的数据（属性、函数） 参数和变量不会被回收 这样就形成了⼀个不会销毁的函数空间 产⽣⼀个闭包  创建闭包最常⻅⽅式，就是在⼀个函数内部创建另⼀个函数。下⾯例⼦中 的 closure 就是⼀个闭包：  function func() {  var a = 1, b = 2;  function closure() {  return a + b;  }  return closure;  }  console.log(func()());  闭包的作⽤域链包含着它⾃⼰的作⽤域，以及包含它的函数的作⽤域和全局作⽤域。  在Javascript语⾔中，只有函数内部的⼦函数才能读取局部变量，因此可以把闭包简单理解成"定义在⼀个函数内部的函数"。所以，在本质上，闭包就是将函数内部和函数外部连接起来的⼀座桥梁。  闭包的⽤途，闭包可以⽤在许多地⽅。它的最⼤⽤处有两个，⼀个是前⾯提到的可以读 取函数内部的变量，另⼀个就是让这些变量的值始终保持在内存中。  function f1() {  var n = 999;  nAdd = function () {  n += 1  }  function f2() {  console.log(n);  }  return f2;  }  var result = f1();  result(); // 999  nAdd();  result(); // 1000  =使⽤闭包的注意点  1）由于闭包会使得函数中的变量都被保存在内存中，内存消耗很⼤，所以 不能滥⽤闭包，否则会造成⽹⻚的性能问题，在IE中可能导致内存泄露， 这是IE的BUG。解决⽅法是，在退出函数之前，将不使⽤的局部变量全部 删除。  2）闭包会在⽗函数外部，改变⽗函数内部变量的值。所以，如果你把⽗函 数当作对象（object）使⽤，把闭包当作它的公⽤⽅法（Public Method），把内部变量当作它的私有属性（private value），这时⼀定要 ⼩⼼，不要随便改变⽗函数内部变量的值。多个⼦函数的scope都是同时指 向⽗级，是完全共享的。因此当⽗级的变量对象被修改时，所有⼦函数都 受到影响。  **字⾯量创建数组**  由⼀对包括元素的⽅括号"[]"表示，元素之间以逗号","隔开  var names = ["terry","larry","tom"]  var name = "name"  // 可以接收任何类型的数据当做数组的元素  var arr = [12,name,true,"larry",{},function(){},[],null];  // 通过数组下标获取对应的数据  console.log(arr,arr[2]);  // 获取数组的⻓度  console.log(arr.length);//arr.length  构造函数创建数组  通过Array构造函数来创建数组  var names = new Array(); // 等价于 var names = [];  // 如果⼀个参数，并且是number类型的整数，则代表的是数组的⻓度。如果是  number类型的⼩数，则报错。如果是其他类型，则当做数组元素放进去。  //var arr = new Array(length); 创建⼀个⻓度为length的数组  var names = new Array(3);  // 创建⼀个包含3个元素的数组  var arr = [undefined,undefined,undefined];  //var ages = new Array(2.4);//Error: Invalid array length  var ages = new Array("2.4");  var names = new Array('terry') //创建⼀个包含1个元素的数组，该元素的值为'terry'  // 两个参数或者多个参数,当做数组元素放进去  var names = new Array('terry','robin')//创建⼀个数组，数组中的元素使⽤实参初始化  通过索引访问数组，数组的索引从0开始，数组的索引超过数组⻓度会访问  到undefined值⽽不会报错。数组的⻓度通过length属性获取  a) [index] 直接访问,索引可以超过索引范围，只不过访问的值为undefined  b) length-1=Max(index)  c) length+N 或 length-N 开辟新的内存空间 或 数组元素的删除  日期2024.7.26 |

Day08笔记

|  |
| --- |
| **数组 API**  **1. 数组的创建与访问**  var array1 = new Array(10);  var array2 = [1, 2, 3];  console.log(array2[0]); // 1  console.log(array1.length); // 10  通过数组字面量和构造函数创建数组是JavaScript中的基础操作。特别是在处理动态数据时，`length` 属性的灵活性非常有用，可以控制数组的大小。但是要注意越界访问会返回 `undefined`，这在调试时可能导致意料之外的错误。  **2. 数组方法示例**  var stack = [];  stack.push(1);  stack.push(2);  console.log(stack.pop()); // 2  `push` 和 `pop` 方法模拟了栈的行为，在需要后进先出（LIFO）数据结构时非常有用。这些方法非常直观且易于使用。  var queue = [];  queue.unshift(1);  queue.unshift(2);  console.log(queue.shift()); // 2  `shift` 和 `unshift` 方法适用于队列操作，模拟了先进先出（FIFO）的数据结构。值得注意的是，这些方法在大数组上可能不如 `push` 和 `pop` 高效，因为它们涉及数组的重排。  var arr = [1, 2, 3, 4, 5];  arr.splice(2, 1, 'a', 'b');  console.log(arr); // [1, 2, 'a', 'b', 4, 5]  `splice` 是非常强大的数组方法，可以在数组中添加、删除或替换元素。灵活性很高，但也容易导致代码复杂性增加，因此在使用时应注意维护代码的可读性。  **正则表达式**  **1. 创建和匹配正则表达式**  var regex = /ab+c/;  console.log(regex.test('abc')); // true  var regex = new RegExp('ab+c');  console.log(regex.test('abc')); // true  正则表达式是字符串处理的强大工具。无论是字面量方式还是构造函数方式，正则表达式都能让我们高效地进行模式匹配。理解和掌握正则表达式语法对于处理复杂字符串匹配场景非常重要。  2. 常用方法  var regex = /a(bc)d/;  var result = regex.exec('abcd');  console.log(result); // ['abcd', 'bc']  `exec` 方法返回匹配结果的数组，并包含捕获组的信息，非常适合需要详细匹配结果的情况。需要注意的是，当使用全局标志 `g` 时，`exec` 会在多次调用之间记录匹配位置。  var regex = /hello/;  console.log(regex.test('hello world')); // true  `test` 方法用于检测字符串是否匹配正则表达式，简单直观。它返回布尔值，是快速检查字符串内容的首选方法，适合需要快速验证的场景。  **3. 边界符**  匹配字符串的开始和结束  var regexStart = /^abc/;  var regexEnd = /abc$/;  console.log(regexStart.test('abcde')); // true  console.log(regexEnd.test('deabc')); // true  边界符 `^` 和 `$` 非常适合匹配字符串的开头和结尾，尤其在验证格式和模式时，如验证特定前缀或后缀。使用边界符可以提高正则匹配的准确性，但需要小心使用，以确保匹配的灵活性。  **总结**  通过对数组 API 和正则表达式的学习，我们可以看到它们在处理数据和字符串操作时的强大之处。数组 API 提供了丰富的操作方法，让我们能够灵活地操控数据集合；正则表达式则是处理字符串匹配和提取的利器。了解这些 API 的边界情况，能够帮助我们在编写代码时避免常见错误，并写出更加健壮的程序。  日期2024.8.19 |

Day09笔记

|  |
| --- |
| JavaScript中的原型与对象  一、JavaScript 原型与原型链  1.1 函数  function add(a, b) {  return a + b;  }  function Person(name, age) {  this.name = name;  this.age = age;  }  var p1 = new Person('张三', 18);  函数不仅是基本的操作单元，它还可以作为构造函数来创建对象。理解函数的双重角色（普通函数和构造函数）对于掌握原型链至关重要。`new` 操作符在创建对象时，背后会涉及到原型链的构建。  1.2 属性  \_\_proto\_\_: 对象的私有属性，指向其原型对象。  prototype: 函数独有属性，指向实例对象的原型对象。  constructor: 对象的构造函数属性，指向创建该对象的函数。  console.log(p1.\_\_proto\_\_ === Person.prototype); // true  console.log(p1.constructor); // Person  这些属性揭示了对象与其原型对象之间的关系。`\_\_proto\_\_` 和 `prototype` 在继承和属性查找过程中起着核心作用，而 `constructor` 则指明了对象的来源。  1.2.1 prototype  Person.prototype.sayHi = function() {  console.log(this.name, '说hi');  };  p1.sayHi(); // 张三 说hi  p2.sayHi(); // 李四 说hi  `prototype` 是实现对象共享属性和方法的关键。将方法定义在 `prototype` 上可以让所有实例共享该方法，避免了重复定义。但需要注意的是，原型方法的共享性意味着修改时需要考虑所有实例的影响。  二、深入理解对象  2.1 对象的创建  function createStudent(name, age) {  var stu = new Object();  stu.name = name;  stu.age = age;  stu.sayHi = function() {  console.log(this.name + '说Hi');  };  return stu;  }  var stu1 = createStudent('张三', 18);  ```  手动创建对象虽然简单，但在大量创建时效率低且容易出错。构造函数或工厂模式封装了对象的创建逻辑，可以大大提高代码的复用性和维护性。  2.2 instanceof 操作符  console.log(stu1 instanceof Student); // true  `instanceof` 检查对象是否是某个构造函数的实例，其本质是通过原型链判断构造函数的 `prototype` 是否出现在对象的原型链上。这个操作符在类型检查和多态性实现中非常重要。  2.3 组合模式  Student.prototype.sayHi = function() {  console.log(this.name + '说Hi');  };  var stu1 = new Student('张三', 18);  var stu2 = new Student('李四', 20);  console.log(stu1.sayHi === stu2.sayHi); // true  将方法定义在原型上是提高性能和代码可维护性的常见策略，避免了每个实例重复创建方法。但要注意的是，所有实例共享同一个方法，如果需要特定实例有不同的行为，需要对方法进行适当扩展或覆盖。  2.4 in 操作符  console.log('name' in stu1); // true  `in` 操作符不仅检查对象自身是否有某个属性，还会沿着原型链查找。这在调试和验证对象属性时非常有用，但也需要注意避免意外地访问到继承自原型的属性。  总结：  在 JavaScript 中，原型链和对象的关系决定了属性和方法的查找规则。理解 `\_\_proto\_\_`、`prototype` 和 `constructor` 等属性，以及如何使用 `instanceof`、`in` 等操作符来操控对象，能够帮助我们更深入地掌握 JavaScript 的面向对象编程思想。这些知识点不仅在原生 JS 中重要，在使用如 Vue 这样依赖原型机制的框架时也尤为关键。  日期2024.8.20 |

Day10笔记

|  |
| --- |
| JavaScript事件与继承  一、JavaScript事件  JavaScript事件是浏览器和用户之间互动的核心机制。事件由三部分组成：事件源、事件类型、事件处理程序，这三者共同构成了事件的完整过程。  1.事件的三要素  事件源：触发事件的HTML元素，如按钮或文本框。  事件类型：事件的类型，如`click`（点击）、`mouseover`（鼠标悬停）等。  事件处理程序：响应事件的JavaScript函数。  2.事件流  事件捕获：事件从最不具体的元素向最具体的元素传播。类似将点击事件从`document`传递到目标元素。  事件冒泡：事件从最具体的元素向外层元素传播，直到`document`为止。它是事件的默认传播方式。  3.事件绑定示例  <button id="btn">点我啊</button>  <script>  var btn = document.getElementById("btn");  btn.onclick = function() {  alert("按钮被点击了！");  }  </script>  这段代码展示了如何为一个按钮绑定点击事件。当按钮被点击时，会弹出一个提示框。绑定事件的方式非常直观，直接通过`onclick`属性将函数赋值给事件处理程序。  4.事件冒泡的应用  在复杂的网页中，利用事件冒泡机制，可以在父元素上统一管理子元素的事件处理，而不需要为每个子元素单独添加事件。这不仅减少了代码的冗余，还提高了性能。  二、JavaScript继承  JavaScript中的继承是面向对象编程中的一个重要概念。主要通过\*\*原型链\*\*实现。  1.原型链  概念：每个构造函数都有一个原型对象，这个原型对象又指向另一个构造函数的原型对象，形成了一条链。实例对象可以通过这条链继承属性和方法。  示例：  function Animal() {  this.type = "animal";  }  Animal.prototype.getType = function() {  console.log(this.type);  }  function Dog() {  this.name = "dog";  }  Dog.prototype = new Animal();  var myDog = new Dog();  myDog.getType(); // 输出 "animal"  `Dog`继承了`Animal`的属性和方法。当调用`myDog.getType()`时，会输出`Animal`中的`type`属性。  2.组合继承  概念：结合原型链和构造函数两者的优点，使用原型链继承方法，使用构造函数继承属性。  function Animal(name) {  this.name = name;  }  Animal.prototype.sayName = function() {  console.log(this.name);  }  function Dog(name, age) {  Animal.call(this, name);  this.age = age;  }  Dog.prototype = new Animal();  Dog.prototype.sayAge = function() {  console.log(this.age);  }  var myDog = new Dog("Buddy", 5);  myDog.sayName(); // 输出 "Buddy"  myDog.sayAge(); // 输出 "5"  `Dog`不仅继承了`Animal`的属性`name`，还添加了自己的属性`age`和方法`sayAge`。这种继承方式能够最大化代码复用，并确保每个实例都拥有独立的属性。  3.继承的注意事项  当使用原型链继承时，需要注意所有实例共享同一个原型对象。这意味着一个实例的改动会影响其他实例。因此，在构造函数中定义实例属性，而在原型中定义共享方法是更合理的选择。  日期2024.8.21 |

Day11笔记

|  |
| --- |
| BOM  1. 什么是BOM  BOM（浏览器对象模型）是用于控制浏览器窗口及其各项功能的JavaScript API集合。它包含了一系列对象，能够访问和操作浏览器的功能，如窗口的大小、位置、导航历史、用户屏幕信息以及网络请求等。  alert("Hello, World!"); // 弹出提示框  var width = window.innerWidth; // 获取窗口的内部宽度  var location = window.location.href; // 获取当前页面的URL  BOM的核心是`window`对象，它在浏览器中既是全局对象，又是浏览器窗口的接口。通过BOM，我们可以实现许多与浏览器交互的操作，如弹出窗口、页面重定向、获取屏幕分辨率等。BOM虽然功能强大，但由于其各浏览器实现存在差异性，使用时需注意兼容性问题。  2. 常见的BOM对象与方法  2.1 window 对象  `window`对象是BOM的核心，表示浏览器窗口或框架。它提供了浏览器窗口相关的属性和方法。  console.log(window.name); // 获取窗口的名称  window.open('https://www.example.com', '\_blank'); // 在新窗口打开链接  通过`window`对象，我们可以获取和操作浏览器窗口的属性，如名称、大小、位置等。`window`对象还与ECMAScript中的全局对象紧密关联，因此全局变量和函数也是`window`对象的属性和方法。  2.2 navigator 对象  `navigator`对象包含了浏览器的信息，如应用程序的名称、版本号、用户代理等。  console.log(navigator.userAgent); // 输出浏览器的用户代理字符串  `navigator`对象让我们能够检测浏览器的信息，对于实现不同浏览器的兼容性处理非常有用。特别是在处理用户代理（user agent）时，可以根据不同浏览器的特性加载不同的脚本或样式。  2.3 location 对象  `location`对象提供了当前文档的位置信息（URL），并允许重定向页面。  console.log(location.href); // 输出当前页面的URL  location.href = "https://www.newpage.com"; // 重定向到新的URL  `location`对象是处理页面导航的核心对象，通过它可以轻松获取当前URL信息并执行页面跳转操作。它在实现动态导航和单页应用程序（SPA）的路由管理时尤为重要。  2.4 screen 对象  `screen`对象包含有关用户屏幕的信息，如分辨率、颜色深度等。  console.log(screen.width); // 获取屏幕的宽度  console.log(screen.height); // 获取屏幕的高度  `screen`对象主要用于处理与用户设备屏幕相关的操作，例如根据屏幕分辨率调整页面布局。虽然它的使用场景较为有限，但在响应式设计中非常实用。  2.5 history 对象  `history`对象用于操作浏览器的会话历史，可以实现页面的前进、后退功能。  history.back(); // 返回前一个页面  history.forward(); // 前进到下一个页面  `history`对象的前进、后退功能使得我们能够更好地控制用户的浏览体验，特别是在单页应用中，通过`pushState`和`replaceState`等方法还能实现浏览器地址栏的动态更新而不刷新页面。  3. 兼容性与BOM的注意事项  虽然BOM提供了强大的功能，但由于各浏览器厂商在实现上存在差异，部分属性和方法在不同浏览器中的支持情况并不完全一致。  在使用BOM时，应始终考虑到兼容性问题。可以通过检测特性是否存在来确保代码的健壮性，例如使用`if`语句检测`window.screen`对象是否存在。了解并应对不同浏览器之间的差异，对于编写跨平台兼容性好的代码至关重要。  通过这次学习，我对BOM的功能和应用有了更深入的理解。BOM在JavaScript编程中提供了非常重要的功能，使得我们能够更好地与浏览器交互。在实际开发中，合理使用BOM能够极大提升用户体验，但也需要注意浏览器兼容性问题，并在必要时进行相应的处理。  日期2024.8.22 |

Day12笔记

|  |
| --- |
| 1. 什么是AJAX？  AJAX（Asynchronous JavaScript And XML）是一种用于创建交互式网页应用的开发技术，旨在通过异步数据传输（HTTP请求）来增强网页与服务器之间的数据交换。其特点是能够在不重新加载整个页面的情况下，对页面的部分内容进行更新。  AJAX技术是现代Web开发中的核心部分。它的异步特性使得用户能够在不感知页面刷新的情况下进行操作，提高了用户体验。尽管名称中包含XML，但实际应用中JSON更为普遍，因为它更轻量且易于解析。  2. HTTP请求与响应  HTTP（超文本传输协议）是Web浏览器与服务器之间通信的基础。AJAX通过JavaScript对HTTP的请求和响应进行操作，从而实现数据的动态加载。  3. XMLHttpRequest对象  `XMLHttpRequest`对象是实现AJAX功能的关键，它用于在后台与服务器交换数据。常用的方法包括：  `open(method, url, async)`：设置请求类型（如GET或POST）、目标URL以及是否异步。  `setRequestHeader(header, value)`：设置HTTP请求头。  `send(data)`：发送请求数据。  var request = new XMLHttpRequest();  request.open('GET', '请求地址', true);  request.send();  request.onreadystatechange = function() {  if (request.readyState === 4 && request.status === 200) {  console.log(request.responseText);  }  };  `XMLHttpRequest`提供了一个简洁的接口用于与服务器进行通信，尤其是在处理异步请求时，它避免了页面的阻塞。然而，随着Fetch API的引入，`XMLHttpRequest`的使用逐渐减少，但它仍然是理解AJAX的基础。  4. AJAX的封装  为了简化AJAX的使用，通常会对其进行封装。  var myAjax = {  get: function(url, params, success, error) {  var httpRequest = new XMLHttpRequest();  httpRequest.open('GET', url + "?" + new URLSearchParams(params).toString(), true);  httpRequest.send();  httpRequest.onreadystatechange = function() {  if (httpRequest.readyState === 4 && httpRequest.status === 200) {  success(JSON.parse(httpRequest.responseText));  } else if (httpRequest.readyState === 4) {  error(httpRequest.responseText);  }  };  }  };  封装AJAX有助于提高代码的可复用性和可维护性。通过封装，我们可以统一处理请求的细节，减少冗余代码，同时也可以更容易地应对跨浏览器的兼容性问题。  5. 实战操作：GET与POST请求  GET请求  var request = new XMLHttpRequest();  request.open('GET', 'https:// 请求地址?param=value', true);  request.send();  POST请求  var request = new XMLHttpRequest();  request.open('POST', '请求地址', true);  request.setRequestHeader('Content-type', 'application/json;charset=UTF-8');  request.send(JSON.stringify({key: 'value'}));  GET请求适用于获取数据且不会改变服务器状态的操作，POST则更适合提交数据。理解两者的区别并合理选择，可以提高请求的效率与安全性。  总结  AJAX通过异步数据传输大大提升了Web应用的交互性，理解并熟练掌握其基本原理和常用操作是前端开发的重要技能。通过对`XMLHttpRequest`的深入理解和灵活应用，可以实现更加高效和友好的用户体验。  日期2024.8.23 |