Day25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 |

Day26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 |

Day27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 |

Day28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 |

Day29笔记

|  |
| --- |
| AJAX学习笔记  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', 'https://api.example.com/data', 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://api.example.com/data?param=value', true);  request.send();  POST请求  var request = new XMLHttpRequest();  request.open('POST', 'https://api.example.com/data', 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 |