# JavaScript高级

一、目标

理解面向对象开发思想，之前学习的基本是面向过程的编程。

掌握 JavaScript 面向对象开发相关模式。

掌握在 JavaScript 中使用正则表达式。

二、重新介绍 JavaScript

1、解释执行（javascript）：解释一行，执行一行，由浏览器负责解释成cpu能识别的，速度相对较慢，要来切换翻译、执行。它的好处在于灵活性-动态特性，随时可以给一个对象添加属性和方法。java等不行。

2、编译执行（java、c#）：它是一次性将所有的代码翻译成cpu能识别的01010，然后在一行一行去执行。

3、函数是 JavaScript 中的一等重要的，函数能做很多事情，能做参数，返回函数等等。在java中是类，函数是次重要的。

4、执行环境：在宿主环境（host environment）下运行，浏览器是最常见的 JavaScript 宿主环境。

在很多非浏览器环境中也使用 JavaScript ，例如 node.js

三、JavaScript 的组成

1、ECMAScript - 语法规范

- 变量、数据类型、类型转换、操作符

- 流程控制语句：判断、循环语句

- 数组、函数、作用域、预解析

- 对象、属性、方法、简单类型和复杂类型的区别

- 内置对象：Math、Date、Array，基本包装类型String、Number、Boolean

2、Web APIs

（1）BOM

- onload页面加载事件，window顶级对象，所有的内容都在window对象下

- 定时器

- location、history

（2）DOM

- 获取页面元素，注册事件

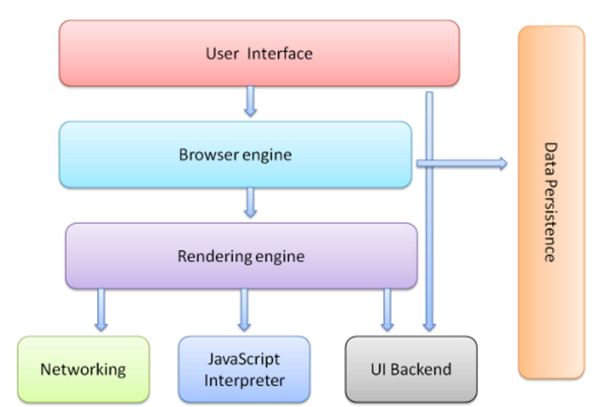
- 属性操作，样式操作

- 节点属性，节点层级

- 动态创建元素

- 事件：注册事件的方式、事件的三个阶段、事件对象

3、浏览器是如何工作的



此时先把域名由运营商的DNS（域名解析为IP地址）服务器转换成IP地址，

DNS再告诉浏览器这个IP地址，

浏览器再根据这个IP地址发送请求，告诉服务器，要去访问你的首页，

服务器把内容准备好之后，返回给浏览器。内容就是一些字符串，就是html文档内容，包涵html代码，css代码，js代码，然后浏览器中的部件去分别解析这些内容

（1）User Interface 用户界面，我们所看到的浏览器，包括工具栏，页面的可视区

（2）Browser engine 浏览器引擎，用来查询和操作渲染引擎

（3）\*Rendering engine 渲染引擎 用来显示请求的内容，负责解析HTML、CSS，并把解析的内容显示出来，并渲染成一棵DOM树，是在内存中存储的

（4）Networking 在浏览器输入www.baidu.com按下回车，它在网络上发送一个请求，负责发送请求和接收来自服务器返回的内容，接收到的基本是字符串

（5）\*JavaScript Interpreter(解析者) JavaScript解析器，负责解析、执行JavaScript的代码，操作内存中的DOM树，然后渲染引擎重新去更新这个DOM树，再告诉浏览器引擎已经更新完毕，把显示在用户界面

（6）UI Backend UI后端，用来绘制类似组合框和弹出窗口，如alert、

（7）Data Persistence(持久化) 数据持久化，数据存储 cookie、HTML5中的sessionStorage

4、JavaScript 执行过程

（1）JavaScript 运行分为两个阶段，不是真正的从上往下去执行：

（2）预解析

- 全局预解析（所有变量和函数声明都会提前；同名的函数和变量，函数的优先级高）

- 函数内部预解析（所有的变量、函数和形参都会参与预解析）

- 函数

- 形参

- 普通变量

（3） 执行

先预解析全局作用域，然后执行全局作用域中的代码，

在执行全局代码的过程中遇到函数调用就会先进行函数预解析，然后再执行函数内代码。

四、JavaScript 面向对象编程

1、面向对象介绍，什么是对象

（1）对象是单个事物的抽象。

一本书、一辆汽车、一个人（具体到个体上去，不能说是某类事物，如汽车是一类事情，并不是指某一辆）都可以是对象，一个数据库、一张网页、一个与远程服务器的连接也可以是对象。当实物被抽象成对象，实物之间的关系就变成了对象之间的关系，从而就可以模拟现实情况，针对对象进行编程。

（2）对象是一个容器，封装了属性（property）和方法（method）。

属性是对象的状态，方法是对象的行为（完成某种任务）。比如，我们可以把动物抽象为animal对象，使用“属性”记录具体是那一种动物，使用“方法”表示动物的某种行为（奔跑、捕猎、休息等等）。

在实际开发中，对象是一个抽象的概念，可以将其简单理解为：**数据集**或**功能集**。

ECMAScript-262 把对象定义为：无序属性的集合，其属性可以包含基本值、对象或者函数（**如果是函数一般称为方法**）。

严格来讲，这就相当于说对象是一组没有特定顺序的值。对象的每个属性或方法都有一个名字，而每个名字都映射到一个值。

提示：每个对象都是基于一个引用类型创建的，这些类型可以是系统内置的原生类型，也可以是开发人员自定义的类型。

2、什么是面向对象

面向对象不是新的东西，它只是过程式代码的一种高度封装，目的在于提高代码的开发效率和可维护性。

面向对象编程 —— Object Oriented Programming，简称 OOP ，是一种编程开发思想。它将真实世界各种复杂的关系，抽象为一个个对象，然后由对象之间的分工与合作，完成对真实世界的模拟。

在面向对象程序开发思想中，每一个对象都是功能中心，具有明确分工，可以完成接受信息、处理数据、发出信息等任务。因此，面向对象编程具有灵活、代码可复用、高度模块化等特点，容易维护和开发，比起由一系列函数或指令组成的传统的过程式编程（procedural programming），更适合多人合作的大型软件项目。

3、面向对象与面向过程：

面向过程就是亲力亲为，事无巨细，面面俱到，步步紧跟，有条不紊

面向对象就是找一个对象，指挥得结果

面向对象将执行者转变成指挥者

面向对象不是面向过程的替代，而是面向过程的封装

4、面向对象的特性：

（1）封装性

（2）继承性

（3）[多态性]抽象，多态性体现不明显，用抽象会更合适

5、程序中面向对象的基本体现

在 JavaScript 中，所有数据类型都可以视为对象，当然也可以自定义对象。自定义的对象数据类型就是面向对象中的类（ **Class ）的概念。ES6中有出现这个概念**

我们以一个例子来说明面向过程和面向对象在程序流程上的不同之处。

假设我们要处理学生的成绩表，为了表示一个学生的成绩，面向过程的程序可以用一个对象表示：

|  |
| --- |
| // 1、面向过程  var stu1 = {  name: 'a',  score: 100  };  var stu2 = {  name: 'b',  score: 80  }  console.log(stu1.name,stu1.score);  console.log(stu2.name,stu2.score);  // 2、面向对象  // 创建一个模板，用于创建对象（实例instance）  // 在javascript中创建对象的模板就是构造函数  // 在其他语言中创建对象的模板是类  function Student(name, score) {  this.name = name;  this.score = score;  this.print = function () {  console.log(this.name, this.score);  }  }  var stu1 = new Student('c',99);  var stu2 = new Student('d',88);  stu1.print();  stu2.print(); |

如果面向过程中出错，是比较难找的，要重新看下整个程序的流程，面向对象，只要看哪个对象出了问题，去找对应的构造函数就ok。

6、面向对象的设计思想

是从自然界中来的，因为在自然界中，类（Class）和实例（Instance）的概念是很自然的。

Class 是一种抽象概念，比如我们定义的 Class----------->Student ，是指学生这个概念，而实例（Instance）则是一个个具体的 Student ，比如， stu1 和 stu2 是两个具体的 Student。

所以，面向对象的设计思想是：

（1）抽象出 Class(构造函数)

（2）根据 Class(构造函数) 创建 Instance

（3）指挥 Instance 得结果

面向对象的抽象程度又比函数要高，因为一个 Class 既包含数据，又包含操作数据的方法。

遇到一个事情的时候，先分析哪些东西可以抽象成为对象

五、创建对象

1、简单方式

我们可以直接通过 new Object() 创建：

|  |
| --- |
| var car = new Object();  **car.name = '丰田';**  car.priace = 1000;  car.running = function () {  console.log('the car running');  } |

2、对象字面量 {} 可以看成是new Object()缩写来创建：

|  |
| --- |
| var car = {  **name:** '丰田',  priace: 1000,  running: function () {  console.log('the car.running');  }  } |

（1）对于上面的写法固然没有问题，但是假如我们要生成两个 person 实例对象呢？

|  |
| --- |
| var person1 = {  name: 'Jack',  age: 18,  sayName: function () {  console.log(this.name)  }  }  var person2 = {  name: 'Mike',  age: 16,  sayName: function () {  console.log(this.name)  }  } |

通过上面的代码我们不难看出，这样写的代码**太过冗余，重复性太高。**

3、工厂函数，解决代码重复问题

|  |
| --- |
| function createCar(name, priace) {  var car = new Object();  car.name = name;  car.priace = priace;  car.running = function () {  console.log('the car.running');  }  return car;  }  var car1 = createCar('本田',9999);  car1.running(); |

通过工厂模式封装，我们解决了创建多个相似对象代码冗余的问题，

但却没有解决对象识别的问题（即怎样知道一个对象的具体类型，只是知道是object，使用instance时候无法知道）。

4、构造函数

与 createCar() 有以下几点不同之处：

- 没有显示的创建对象

- 直接将属性和方法赋给了 this 对象

- 没有 return 语句

- 函数名使用的是大写的 Car

构造函数名字的第一个字母大写，用于区分普通函数

|  |
| --- |
| function Car(name, priace) {  this.name = name;  this.priace = priace;  this.running = function () {  console.log('the car running');  }  }  var car1 = new Car('保时捷', 99999);  console.log(car1 instanceof Car); // output: true  console.log(car1.constructor); |

（1）解析构造函数代码的执行

必须使用 new 操作符。以这种方式调用构造函数会经历以下 4 个步骤：

|  |
| --- |
| 1、会在内存中创建一个空的对象  2、设置构造函数中的this指针，指向刚刚创建好的对象  3、执行构造函数中的代码  4、返回刚刚创建的对象 |

（2）下面是具体的伪代码：

|  |
| --- |
| function Person (name, age) {  // 当使用 new 操作符调用 Person() 的时候，实际上这里会先创建一个对象  // var instance = {}  // 然后让内部的 this 指向 instance 对象  // this = instance  // 接下来所有针对 this 的操作实际上操作的就是 instance  this.name = name  this.age = age  this.sayName = function () {  console.log(this.name)  }  // 在函数的结尾处会将 this 返回，也就是 instance  // return this  } |

（3）构造函数和实例对象的关系

使用构造函数的好处不仅仅在于代码的简洁性，更重要的是我们可以识别对象的具体类型了。

在每一个实例对象中同时有一个 constructor 属性（**并不是直接有这个属性，见构造函数、实例、原型三者之间的关系**），该属性指向创建该实例的构造函数： **constructor 是实例的属性的说法不严谨，之所以能直接访问**



对象的 constructor 属性最初是用来标识对象类型的，但是，如果要检测对象的类型，还是使用 instanceof 操作符更可靠一些。

|  |
| --- |
| console.log(car1.constructor === Car);  var array = [];  console.log(**array.constructor === Array**); |

无法使用typeof获取对象的具体类型，返回的都是object。

获取对象的具体类型，对象有一个constructor属性，不建议使用，它是可以被修改的。

**instanceof 判断某个对象是否是某个构造函数的实例（对象）。**

（4）总结：

构造函数是根据具体的事物抽象出来的抽象模板。

实例对象是根据抽象的构造函数模板得到的具体实例对。

每一个实例对象都具有一个 **constructor 属性，指向创建该实例的构造函数**。

注意： **constructor 是实例的属性的说法不严谨，之所以能直接访问是系统通过\_\_proto\_\_指针指向的原型对象去查找的，原型链相关之知识**。

可以通过实例的 constructor 属性判断实例和构造函数之间的关系。

注意：这种方式不严谨，推荐使用 instanceof 操作符，后面学原型会解释为什么。

5、构造函数的问题

使用构造函数带来的最大的好处就是创建对象更方便了，但是其本身也存在一个**浪费内存的问题**：

|  |
| --- |
| function Student(name, age, sex) {  this.name = name;  this.age = age;  this.sex = sex;  this.sayHi = function () {  console.log('hello world');  }  }  var stu1 = new Student('a', 15, '男');  var stu2 = new Student('b', 33, '女');  console.log(stu1.sayHi === stu2.sayHi);**// => false，它比较的其实是两个函数的地址，它们的内存地址是不一样的** |

在该示例中，从表面上好像没什么问题，但是实际上这样做，有一个很大的弊端。

那就是对于每一个实例对象，sayHi 都是一模一样的内容。

每一次生成一个实例，都必须为重复的内容，多占用一些内存，如果实例对象很多，会造成极大的内存浪费。

（1）对于这种问题我们可以把需要共享的函数定义到构造函数外部：

|  |
| --- |
| function Student(name, age, sex) {  this.name = name;  this.age = age;  this.sex = sex;  this.sayHi = sayHi;  }  function sayHi() {  console.log('hello world ' + this.name);  // 在书写函数的时候，this的指向是无法确认的，只有当它**被调用**，执行的时候，才能确定，书写或者注册的时候暂时无法确定的。  }  var stu1 = new Student('a', 15, '男');  var stu2 = new Student('b', 33, '女');  stu1.sayHi(); |

这样确实可以了，但是如果有多个需要共享的函数的话就会造成全局命名空间冲突的问题。

（2）可以把多个函数放到一个对象中用来避免全局命名空间冲突的问题：

|  |
| --- |
| function Student(name, age, sex) {  this.name = name;  this.age = age;  this.sex = sex;  this.sayHi = fns.sayHi;  }  var fns = {  sayHi: function () {  console.log('hello world');  }  } |

至此，我们利用自己的方式**基本上解决了构造函数的内存浪费问题**。

但是代码看起来还是那么的格格不入，那有没有更好的方式呢？

6、静态成员跟实例成员

1. 静态成员

|  |
| --- |
| 直接通过对象来调用的属性和方法，称为静态成员（一般是字面量方式）。**没有动态成员**  工具中使用静态成员，如Math，可以求出最大值，最小值等等，这个Math工具只有一个。  var car = {  name: '丰田',  priace: 1000,  running: function () {  console.log('the car.running');  }  }  **car.running(); // 静态成员** |

（2）实例成员

|  |
| --- |
| 实例是对象的另外一种说法。当有很多个对象的时候，使用构造函数来创建。  实例成员：构造函数中的成员就是实例成员，它的调用需要创建出实例之后，才能通过实例去调用。  function Car(name, priace) {  this.name = name;  this.priace = priace;  this.running = function () {  console.log('the car running');  }  }  var car1 = new Car('保时捷', 99999);  car1.running(); |

六、原型

1. 更好的解决方案： prototype

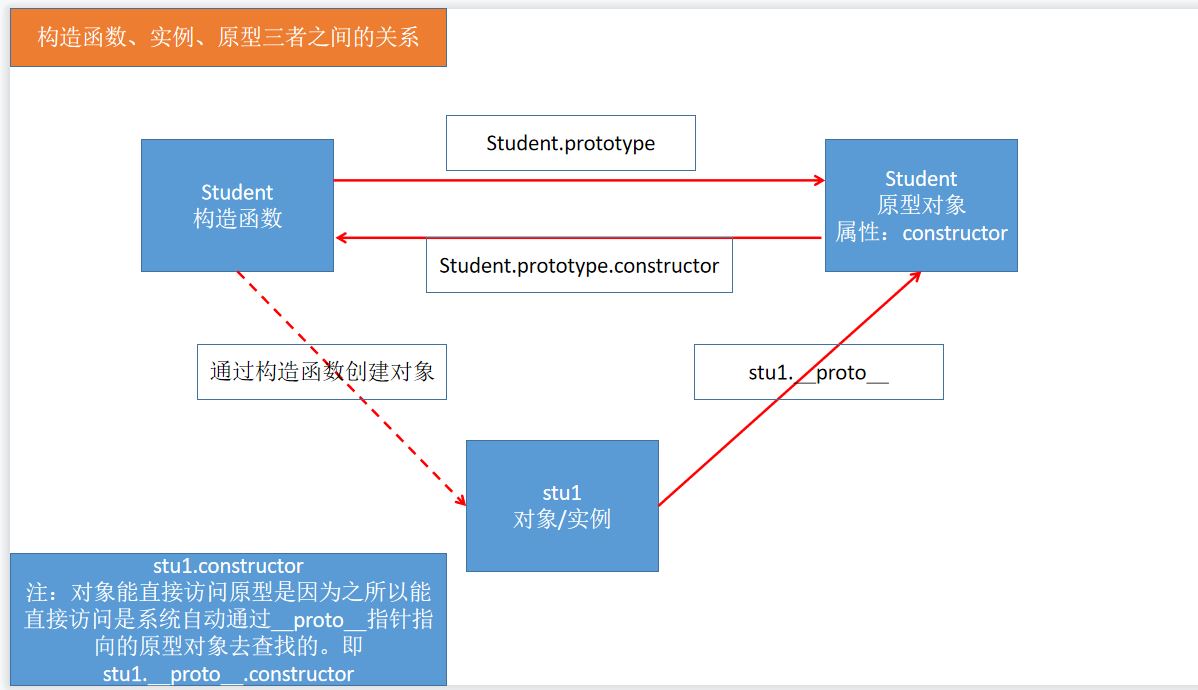
JavaScript 规定，**每一个构造函数（不是实例，容易混淆）都有一个 prototype（原型 / 原型对象） 属性**，它其实是一个对象。这个对象的所有属性和方法，都会被构造函数的所拥有。

这也就意味着，我们可以把所有对象实例需要共享的属性和方法直接定义在 prototype 对象上。

|  |
| --- |
| function Student(name, age, sex) {  this.name = name;  this.age = age;  this.sex = sex;  }  **Student.prototype.sayHi** = function () {  console.log('hello world');  }  **Student.prototype.class** = '三班';  var stu1 = new Student('a', 15, '男');  var stu2 = new Student('b', 33, '女');  stu1.sayHi();// **实例对象可以直接访问原型对象成员。之所以能直接访问是系统通过\_\_proto\_\_指针指向的原型对象去查找的，原型链相关之知识**  console.log(stu1.sayHi === stu2.sayHi); // output true  console.log(Student.prototype);  通过该构造函数创建的对象，可以直接访问构造函数.prototype中的成员（属性、方法），**之所以能直接访问是系统通过\_\_proto\_\_指针指向的原型对象去查找的，原型链相关之知识** |
| 1 |

**Student.prototype.constructor指向了构造函数本身**

2、构造函数、实例、原型三者之间的关系



（1）任何函数都具有一个 prototype 属性，该属性是一个对象。

|  |
| --- |
| function fn() {  console.log('f');  }  console.log(fn.prototype.constructor);  1 |

（2）构造函数的 prototype 对象默认都有一个 constructor 属性，指向 prototype 对象所在函数。

（3）通过构造函数得到的实例对象内部会包含一个指向构造函数的 prototype 对象的指针 **\_\_proto\_\_**。（ \_\_proto\_\_ 是非标准属性。）

|  |
| --- |
| console.log(stu1.constructor);  console.log(stu1.\_\_proto\_\_.constructor);  二者输出的结果是一致的 |

（4）实例对象可以**直接访问原型对象成员**。**之所以能直接访问是系统通过\_\_proto\_\_指针指向的原型对象去查找的，原型链相关之知识**

console.log(stu1.constructor);

（5）总结：

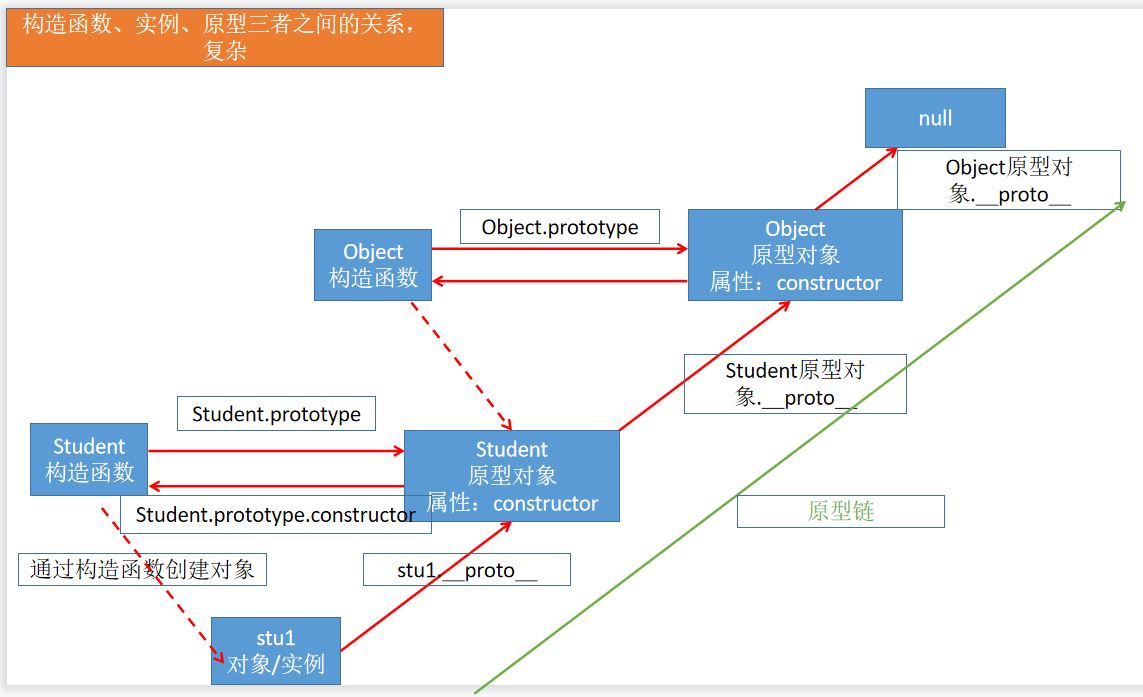
- 任何函数都具有一个 prototype 属性，该属性是一个对象。

- 构造函数的 prototype 对象默认都有一个 constructor 属性，指向 prototype 对象所在函数。

- 通过构造函数得到的实例对象内部会包含一个指向构造函数的 prototype 对象的指针 \_\_proto\_\_。

- 所有实例都直接或间接**继承了**原型对象的成员。

3、属性成员的搜索原则：原型链



构造函数-实例-原型对象 三者之间的关系后，接下来我们来解释一下

（1）为什么实例对象可以**访问原型对象中的成员**。

每当代码读取某个对象的某个属性时，都会执行一次搜索，目标是具有给定名字的属性

|  |
| --- |
| 1、搜索首先从对象实例本身开始  2、如果在实例中找到了具有给定名字的属性，则返回该属性的值  3、如果没有找到，则继续搜索**指针（\_\_proto\_\_）**指向的原型对象，在原型对象中查找具有给定名字的属性  4、如果在原型对象中找到了这个属性，则返回该属性的值 |
| <script type="text/javascript">  function Student(name, age) {  this.name = name;  this.age = age;  this.sayHi = function () {  console.log('hello');  }  }  Student.prototype.sayHi = function () {  console.log('Hi' + this.name);  }  var stu1 = new Student('Mike',22);  stu1.sayHi();  </script>  **结果输出的是 hello** |
| 也就是说，在我们调用 stu1.sayHi() 的时候，会先后执行两次搜索：  - 首先，解析器会问：“实例 stu1有 sayHi 属性吗？”答：“没有。  - ”然后，它继续搜索，再问：“ stu1的原型有 sayHi 属性吗？”答：“有。  - ”于是，它就读取那个保存在原型对象中的函数。  - 当我们调用 stu2.sayHi() 时，将会重现相同的搜索过程，得到相同的结果。 |

而这正是多个对象实例共享原型所保存的属性和方法的基本原理。

（2）toString()函数的查找

所有的对象都有toString的方法，根据原型链查找规则，最终查找到Object这个原型对象的成员

console.log(stu1.toString());自己创建的对象返回的是固定的形式[object Object]，object代表的是一个对象，Object代表的是类型，它获取的不是具体的类型，而是所有对象的祖宗。

|  |
| --- |
| function Student(name, age) {  this.name = name;  this.age = age;  }  Student.prototype.sayHi = function () {  console.log('Hi' + this.name);  }  var stu1 = new Student('Mike',22);    console.log(stu1.toString());  console.log(stu1.\_\_proto\_\_);  console.log(stu1.\_\_proto\_\_.\_\_proto\_\_);  1 |

（3）设置属性

要设置的属性如果在原型对象上，在设置值的时候，是不会去搜索原型链的，而是直接给对象新增这个属性。

|  |
| --- |
| function Student(name, age) {  this.name = name;  this.age = age;  }  Student.prototype.sayHi = function () {  console.log('Hi' + this.name);  }  ***Student.prototype.test = 'test';***  var stu1 = new Student('Mike',22);  var stu2 = new Student('Jon',25);  ***stu1.test = '111';***  console.log(stu1.test);  console.log(stu2.test);  1 |

（4）读取属性

读取属性，如果在对象本身查找属性，如果没有找到的话，就会去原型链上查找。

（5）总结：

先在自己身上找，找到即返回。

自己身上找不到，则沿着原型链向上查找，找到即返回。

如果一直到原型链的末端还没有找到，则返回 undefined。

4、注意点

一般情况下，对象的属性在构造函数中来设置。对象的方法在构造函数的原型对象中来设置。

（1）前面例子中每添加一个属性和方法就要敲一遍 Person.prototype 。

为减少不必要的输入，更常见的做法是用一个包含所有属性和方法的对象字面量来重写整个原型对象：

|  |
| --- |
| function Student(name, age) {  this.name = name;  this.age = age;  }  /\*Student.prototype.sayHi = function () {  console.log('Hi' + this.name);  }  Student.prototype.eat = function () {  console.log('eating');  }\*/  **Student.prototype = {**  **sayHi: function () {**  **console.log('Hi' + this.name);**  **},**  **eat: function () {**  **console.log('eating');**  **}**  **}**  var stu1 = new Student('Mike',22);  console.log(stu1.constructor);  1  结果输出的是 f Object() 系统的对象构造函数。  原因是：我们给Student.prototype属性设置了一个新的对象，这个对象里边并没有construtor属性，所以在原型链上查找，最后在原型对象的原型对象上找到了constructor，就返回了Object。 |

（2）手动将 constructor 指向正确的构造函数，保持 constructor 的指向正确

|  |
| --- |
| Student.prototype = {  **constructor: Student,**  sayHi: function () {  console.log('Hi' + this.name);  },  eat: function () {  console.log('eating');  }  } |

（3）想使用的新的prototype 的方法，必须在其创建完之后，才能使用

|  |
| --- |
| function Student(name, age) {  this.name = name;  this.age = age;  }  **var test = new Student('MM', 20);**  **test.sayHi();**  Student.prototype = {  constructor: Student,  sayHi: function () {  console.log('Hi ' + this.name);  },  eat: function () {  console.log('eating');  }  }  var stu1 = new Student('Mike',22);  stu1.sayHi();  这里会出错，因为此时在原型链上找不到sayHi方法 |

5、内置对象方法拓展

|  |
| --- |
| <script type="text/javascript">  var array = [1,2,3,4,5];  Array.prototype.getSum = function () {  var sum = 0;  console.log(**this**); // 在调用的时候会输出 array，没有调用之前是不知道指向谁的  for (var i = 0, len = this.length; i < len; i++) {  sum += this[i];  }  return sum;  }  /\*Array.prototype = { // 这样写直接报错  getSum: function () {  var sum = 0;  for (var i = 0, len = this.length; i < len; i++) {  sum += this[i];  }  return sum;  }  }\*/  console.log(array.getSum());// output 15  </script> |

Array、Sring等内置对象的prototype属性，**系统不允许修改属性的值**。可以通过给它添加新的方法。

6、函数也是对象，可以有方法bind()

bind()方法是ES5中增加的方法，意思是基于xxx新建一个方法

bind中的第一个参数可以改变函数中this的指向

会返回一个新的函数

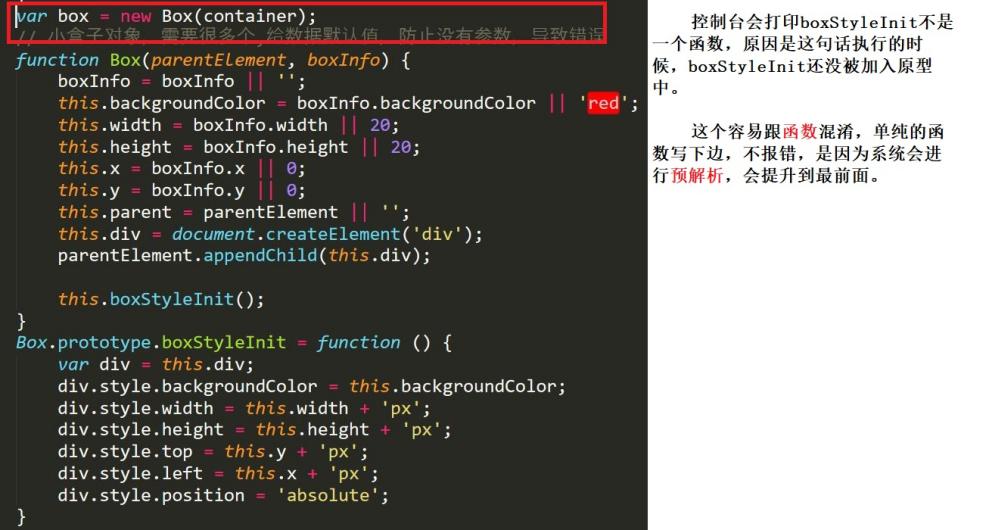
bind方法并不会执行，还需要进行调用

|  |
| --- |
| var a = 123;    function fn() {  console.log(this);  }  fn();  var o = {  a: 'a'  }  var fn1 = fn.bind(o);  fn1();// 相当于是o.fn1  1 |

7、小案例：随机方块

工具对象，目前只有一个，所有使用字面量方式，可以不用构造函数。

引入js文件的顺序也是有讲究的，先引入需要被调入的js文件



1. 贪吃蛇案例
2. 把数据尽量用变量来表示，且提到最前面，后续容易维护。
3. js目录下文件的命名，可以参考github上别人项目的命名。
4. 在render函数中的代码不要写太长，可以再封装函数进行调用，因为删除食物后，需要马上随机生成，所以可以把他们俩放在一起用，没有单独删除食物的需求。
5. 删除数组中的元素，使用splice，如果从0开始删除，那么数组下标会动态变化，此时的i需要跟着减删除的数目，此时可以从数组后面进行删除，这样就不用担心i的值。

splice删除之后，数组的元素会向前移动，如果i没处理的话，从前到后的遍历会出问题。从后往前遍历就ok

1. 不同的js文件中，可能存着相同名字的全局变量，会出错，函数才能开启新的作用域，其他位置是全局作用域。

自调用函数会开启新的作用域，避免命名冲突。

解决方式：把与 全局变量相关的所有代码 放到自调用函数里边。别人基本也是这么操作的

再把需要用到的函数暴露给window，给window设置属性，外部能直接访问。？？？这怎么理解，自调用函数执行完毕之后，内存不是都释放掉吗，怎么还能访问？？？那个私有函数呢，执行一次后还保存内存中？？？？可能跟闭包有关系，或者是常驻于内存中

答：那个构造函数的引用给到了window，由于形成了闭包，它会常驻于内存中。

6、使用对象字面量的时候要记得加分号，因为它是以大括号结尾的，容易被忽略，因为函数也大括号，且不用加分号的。

7、分析蛇对象，就是分析蛇的属性和方法，移动函数和渲染函数可以看成是两个动作。

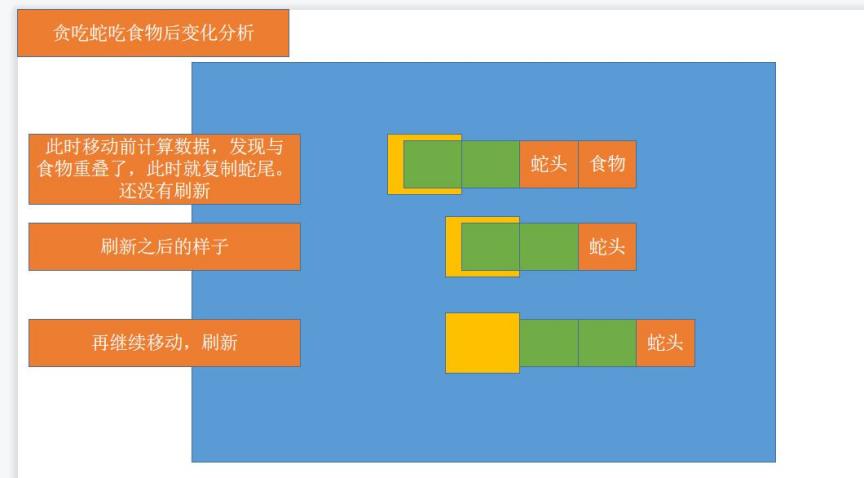
8、蛇的移动分析：就是头部和尾巴，尾巴的部分不管怎么移动，都是移动到上一节的位置就ok，头部就要考虑上下左右。

1. 在单独测试功能的时候，尽量把其他的功能一起注释掉
2. 对外的接口越简单越好
3. 在定时器中的function中的this是指向了window的。因为定时器它是window去进行调用的

12、alert弹出来时会阻止渲染

13、蛇吃食物，它是在蛇移动的过程中发生的，所以把它放在移动过程中来处理

14、移动的时候，是先处理数据，再进行渲染刷新



1. var last = this.body[this.body.length - 1];

this.body.push({

x: last.x,

y: last.y,

color: last.color

});

// this.body.push(last);

解释为什么不能用 this.body.push(last);

这样就导致了最后添加的那个对象last，跟前一个他们两是同一个对象，都指向了同一块地址，等一下移动的时候，他们两的值会一直保持同步的。谁变了就跟着变，知道最后没有变化为止，这就导致了他们的数据一样。渲染出来后就重叠了，看不到蛇的变化。

16、目前的js文件分得比较多出来，如果要发布在服务器上，还要进行一下压缩，把他们合并到一个js文件中去，这样提高网页访问的速度，只要一次就把js代码都下载来，不用多次。

手动合并的话，要注意代码引用顺序的问题

1. 压缩代码---就是除了代码，其他能不要就不要，如空格、注释等等

将来可以使用一些工具进行压缩

百度js在线压缩



自调用函数参数window、undefined目的

1. 传入window是为了代码能被压缩，不然函数体内的window，不会被压缩。。感觉不是更多了吗？？？
2. 在老版本中的浏览器中，undefined 可以被重新赋值？？这又怎么理解

函数参数是undefined，那这个undefined是个变量标识符还是值呢？

未给函数形参传值的时候，该形参的值是undefined。所以上面的形参undefined是一个局部变量标识符，该变量值是undefined。注意，undefined不是JS保留字和关键字。

开始看到很多js函数里都带一个undefined的参数，很是疑惑，后来查了查，原来是这样.  
假如我们定义了一个函数  
function  
a(){  
  if(arg1===undefined)  
alert("arg1未定义");  
}  
有在某一个地方，由于什么原因，存在一个一个变量undefined="xx"(undefined不是保留字)。这样当我们调用函数a();时就不会输出"arg1未定义",但事实上arg1确实未定义。  
但当我们这样定义函数a，  
function  
a(undefined){  
  if(arg1===undefined)  
alert("arg1未定义");  
}  
在调用a();由于未传入参数，所以函数a的参数undefined等价于未定义，再判断arg1===undefined时就会返回true，得到正确结果。  
所以，给函数在原有基础上增加一个undefined的参数，从某种程度上防止的undefined被变量污染。当然这个undefined也可以用其他标示符定义，但那样就不直观，就不容易理解了。

八、继承

继承是类型和类型之间的关系

继承的目的：把子类型中共同的成员提取到父类型中，代码重用。JavaScript中没有提供继承的语法，只是用了一些方法去模拟出来的效果。

1、对象拷贝

以下的这种方式，其实不是继承，是对象的拷贝

真正的继承指的是类型和类型之间的关系

|  |
| --- |
| var Person = {  name: 'aaa',  age: 29,  sex: '男',  sayHi: function () {  console.log('Hi');  }  }  var Mike = {  name: 'Mike',  age: 18  }  /\*for(var key in Person) {  if (Mike[key]) {  continue;  }  Mike[key] = Person[key];  }\*/  function extend(parent, child) {  for (var key in parent) {  if(child[key]) {  continue;  }  child[key] = parent[key];  }  }  extend(Person, Mike);  console.log(Mike);  </script> |
| 1 |

2、原型继承

一般不使用原型继承，缺点是：无法设置构造函数的参数，就是原型的对象设置一次之后，他的属性值基本就是固定。无法去动态的修改，并不实用，看例子

|  |
| --- |
| <script type="text/javascript">  function Person() {  this.name = 'a';  this.age = 18;  this.sex = '男';  }  function Student() {  this.score = 99;  }  **Student.prototype = new Person();**  **Student.prototype.constructor = Student;**  var stu1 = new Student();  console.log(stu1.age);  console.log(stu1);  function Teacher() {  this.salary = 1000;  }  </script>  1 |
| 此时确实能访问到age，但是如果要修改age的值就比较麻烦了，不适用，这样每个student对象的年龄可能不一致。 |

3、call()方法

（1）call()改变函数中this，直接调用函数，可以理解成调用函数的另外一种方式

第一个参数是this的指向

后续参数是给函数的实参。

返回值：使用调用者提供的 this 值和参数调用该函数的返回值。若该方法没有返回值，则返回 undefined。

bind()改变函数的this，并且返回一个新的函数（他不会调用函数）

|  |
| --- |
| function add(x, y) {  console.log(this);  console.log(x + y);  }  // window.add(2,3);实际是window在调用，但是window可以省略  // add(2,3);  var obj = {  name: 'abc'  };  // var addObj = add.bind(obj, 2, 3);  // addObj();  **add.call(obj, 4, 3); // output 7** |

4、借用构造函数

就是把构造函数当成一个普通函数，改变它的this的指向。只能用来借用构造的属性，构造函数上的原型对象的方法，无法借用

|  |
| --- |
| function Person(name, age, sex) {  this.name = name;  this.age = age;  this.sex = sex;  }  Person.prototype.sayHi = function () {  console.log(this.name + ' hi');  }  function Student(name, age, sex, score) {  this.score = score;  **Person.call(this, name, age, sex);**  }  var stu = new Student('Mike', 22, '男', 99);  console.log(stu); |
| 1  没有sayHi |

5、组合继承

借用构造函数继承父类型的属性（实际是模拟继承）+原型继承（原型对象设置为父类型对象）

使用的场景：有大量需要重复使用的属性，这时可以采用这种组合继承的方式，如果只有两三个，那么使用这种方式会变得复杂化，需要看情况来使用。

一般做页面特效的情况是不用的

|  |
| --- |
| function Person(name, age, sex) {  this.name = name;  this.age = age;  this.sex = sex;  }  Person.prototype.sayHi = function () {  console.log(this.name + ' hi');  }  function Student(name, age, sex, score) {  this.score = score;  Person.call(this, name, age, sex);  }  **Student.prototype = Person.prototype;// 这样他们都指向了同个原型，数据会一直保持同步，给学生添加考试方法，那么person也会添加，但是person是不需要的，而且constructor也被改了。这样不适用**  **Student.prototype.constructor = Student;**  **Student.prototype.exam** = function () {  console.log('exam');  }  var stu = new Student('Mike', 22, '男', 99);  stu.sayHi();  console.log(stu);  var person = new Person('peo',22,'男');  console.log(person); |
| 2 |

|  |
| --- |
| Student.prototype = **new Person();**  Student.prototype.constructor = Student; |
| 1  1 |

九、函数

1、函数声明和函数表达式，他们之间的区别

代码的执行会先进行解析，后续才会执行

会进行函数和变量名的提升，发现有，就会提到最前面的去。

（1）函数声明会函数提升，在预解析阶段就已创建，声明前后都可以调用。

|  |
| --- |
| <script type="text/javascript">  fn();  function fn() {  console.log('hello');  }  fn1();  var fn1 = function () {  console.log('hi');  }  </script> |
| 预解析后的代码  function fn() {  console.log('hello');  }  var fn1;  fn();  fn1();  fn1 = function () {  console.log('hi');  }  此时的fn1仅仅是声明，值是undefined，对其进行调用就会出错 |

（2）根据条件声明函数

在现代浏览器中（指的是支持html5的浏览器），对if语句中的函数是不会进行提升的

在老的IE浏览器中，对if语句中的函数是会进行提升的

使用函数表达式就不会出现这样的问题

|  |
| --- |
| if (true) {  function fn() {  console.log('true');  }  } else {  function fn() {  console.log('false');  }  }  fn();  var fn1;  if (true) {  fn1 = function () {  console.log('true');  }  } else {  fn1 = function () {  console.log('false');  }  }  fn1(); |

2、new Function()

执行效率慢，首先要把字符串解析成js代码。一般不会使用这个方式

函数也是对象，它也会给我们提供一系列的方法

|  |
| --- |
| <script type="text/javascript">  var fn = new Function(**'a'**,**'b'**,'console.log(a+b);console.log("hello");');  fn(10, 10);  console.dir(fn);  </script>  1 |

3、函数的调用

（1）普通函数的调用

this 指向window。

（2）方法调用

this指向调用该方法的对象。

（3）作为构造函数调用

构造函数内部的this指向了由该构造函数创建的对象。

（4）作为事a件处理函数

this指向了触发该事件的对象。

（5）作为定时器的参数

this指向了window，最后那个函数的调用是window去执行的。

1. 终极总结：函数内部的this，是由函数调用的时候来确定其指向的，在书写的时候是无法确认的。

|  |
| --- |
| function fn() {  // 书写时候不能确认是指向谁的  console.log(this);  }  **fn();**  var obj = {  name: 'obj',  fn: fn  };  **obj.fn();**  1 |
| var obj = {  name: 'obj',  fn: function () {  console.log(this);  }  };  **obj.fn();**  **var fn1 = obj.fn;// 只是赋值，没有进行调用**  fn1(); |

1. call()、bind()、apply()，函数对象的方法

（1）call()

|  |
| --- |
| 作用：调用参数、改变函数中的this指向。  参数1：设置函数内部的this指向。  参数其他：对应函数的参数。  返回值：函数的返回值。 |
| 应用：  （1）伪数组借用数组中的方法，对象的属性只能是字符串  var guiseArray = {  0: 100,  1: 200,  2: 300,  3: 55,  length: 4  };  /\*guiseArray['4'] = 30;  guiseArray['length']++;  console.log(guiseArray);\*/  **Array.prototype.push.call(guiseArray, 487);**  console.log(guiseArray);  **Array.prototype.splice.call(guiseArray, 0, 2);**  console.log(guiseArray);  1  （2）使用toString()打印类型  var guiseArray = {  0: 100,  1: 200,  2: 300,  3: 55,  length: 4  };  // 未重写过toString  console.log(guiseArray.toString());  var array = [1,2,3,4];  // 数组内部是有重写过toString()的方法  **console.log(array.toString());**  // 使用Object的toString方法，看最原始的打印  **console.log(Object.prototype.toString.call(array));**  var num = 1;  console.log(Object.prototype.toString.call(num));  1 |

（2）bind()

它和 call、apply 最大的区别是：bind 不会调用

bind 支持传递参数，它的传参方式比较特殊，一共有两个位置可以传递

在 bind 的同时，以参数列表的形式进行传递

在调用的时候，以参数列表的形式进行传递

那到底以谁 bind 的时候传递的参数为准呢还是以调用的时候传递的参数为准

两者合并：bind 的时候传递的参数和调用的时候传递的参数会合并到一起，传递到函数内部。

|  |
| --- |
| var obj = {  name: 'abc',  fn: function () {  setInterval(function (){  console.log(this.name);  },1000);  }  }  obj.fn();  此时输入的是空白，因为定时器是window去调用的，所以this指向了window，window原本就有name这个属性，而且为空，所以打印的是空白，不是undefined。 |
| var obj = {  name: 'abc',  fn: function () {  setInterval(function (){  console.log(this.name);  }.**bind(this)**,1000);// 这个this指向的obj对象，是obj去调用的  }  }  obj.fn(); |

（3）apply()

作用：就是把数组展开，然后传给前面的方法。在执行的时候，会将数组内部的元素一个一个拿出来，与形参一一对应进行传递，也可以改变this的指向

apply() 方法调用一个具有给定this值的函数，以及以**一个数组（或类数组对象）的形式**提供的参数。

注意：call()方法的作用和 apply() 方法类似，区别就是call()方法接受的是**参数列表**，而apply()方法接受的是一个**参数数组**。

|  |
| --- |
| Math.max(value1[,value2, ...])  参数value1, value2, ...是一组数值，不是数组  var array = [1,2,3,4];  console.log(array);    console.log(Math.max**(1,2,3,4)**);// 参数是一个数组的形式。  console.log(Math.max.apply(**Math**, array)); // Math可以指定为null  console.log**(1,2,3,4)**;  console.log.apply(console, array);  1 |

5、函数的其它成员

（1）arguments

伪数组，获取函数的实参

（2）caller

函数的调用者，在全局范围内调用的时候caller，是null

（3）length

函数形参的个数

（4）name

函数的名称，字符串类型

|  |
| --- |
| function fn(x,y) {  console.log(fn.arguments);  console.log(fn.caller);  console.log(fn.length);  console.log(fn.name);  }  (function (){  fn(1,2,3,4);  })();  1 |

（5）函数内部的私有变量arguments，是伪数组

它跟（1）是一样的，作用

例如Math.max，它传入的参数的个数是不确定的，此时就可以使用arguments来获取所有的实参，再进行处理。

|  |
| --- |
| function max() {  var max = max.arguments[0]; // 把变量max，重新命名就ok  for (var i = 1, len = arguments.length; i < len; i++) {  if (max < arguments[i]) {  max = arguments[i];  }  }  return max;  }  这里出错的原因是：max跟函数名max命名重复了，局部变量覆盖了。max的值此时的undefined，命名时候还是要注意一下的  1 |
| function max() {  var max = arguments[0];  for (var i = 1, len = arguments.length; i < len; i++) {  if (max < arguments[i]) {  max = arguments[i];  }  }  return max;  }  console.log(max(1,2,3,4,5,666)); |

6、高阶函数

（1）函数作为参数的时候

|  |
| --- |
| 可以通过设置断点的方式来，看代码的调用过程  function eat(doWork) {  setTimeout(function (){  console.log('eat');  doWork();  },2000);  }  function play() {  console.log('playing');  }  eat(play); |

|  |
| --- |
| Array.prototype.mySort = function (**compare**) {  for (var i = 0; i < this.length - 1; i++) {  var isSort = true;  for (var j = 0; j < this.length - 1 - i; j++) {  if (**compare(this[j], this[j + 1])** > 0) {  var temp = this[j];  this[j] = this[j + 1];  this[j + 1] = temp;  isSort = false;  }  }  if (isSort) {  break;  }  }  }  var array = [2,44,22,11,26,666];  array.mySort(function (a, b) {  **return a - b;**  });  console.log(array); |

（2）函数作为返回值的时候

|  |
| --- |
| // 局部变量一直在内存中，跟闭包有关系  function getFixRandom(min ,max) {  var random = Math.floor(Math.random()\*(max - min + 1) + min);  return function () {  return random;  }  }  var temp;  temp = getFixRandom(10,100);  console.log(temp());  console.log(temp());  console.log(temp());    function getSum(n, m) {  return function (m) {  return m + n;  }  }  var fn = getSum(100);  console.log(fn(1));  var fn1 = getSum(1000);  console.log(fn1(22));  1 |

十、闭包

1. 闭包概念的理解

**闭包**就**是**能够读取其他函数内部变量（内部函数）的**函数**。

由于在 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。

|  |
| --- |
| function fn() {  var n = 10;  return n;  }  fn();  fn并没有去读取其他函数内部的变量。 |

|  |
| --- |
| function fn() {  var n = 10;  // 返回的这个函数当前并没有执行，当我们去调用的时候，才会执行  // 在这个返回的函数中,我们要访问变量 n,它需要在上一级的作用域中才有  return function () {  return n;  }  }  var f = fn();// f是一个引用，因为函数是一个对象。  console.log(f());// 它访问到了**函数作用域(局部作用域)**的 n .  // fn返回了一个函数的引用，而这个函数还需要访问内部函数 n,所以当我们调用完fn的时候,fn的作用域并没有销毁,因为 变量n还有人要去访问。（这样解释并不是很好，请看下面的MDN解释）  1 |

|  |
| --- |
| function makeFunc() {  var name = "Mozilla";  function displayName() {  alert(name);  }  return displayName;}  var myFunc = makeFunc();myFunc();  原因在于，JavaScript中的函数会形成了闭包。在本例子中，myFunc 是执行 makeFunc 时创建的 displayName 函数实例的引用。displayName 的实例有人引用，它的内存就不会被系统回收。它是存在于makeFunc作用域中，所以这个作用域也会不会被回收。如果被回收，这个函数也不存在。 |

2、闭包的用途：

（1）可以在函数外部读取函数内部成员

（2）让函数内成员始终存活在内存中

（3）可以用来实现JS模块。

JS模块:具有特定功能的js文件,将所有的数据和功能都封装在一个函数内部(私有的),只向外暴露一个包信n个方法的对象或函数,模块的使用者,只需要通过模块暴露的对象调用方法来实现对应的功能。

例如贪吃蛇中每个js文件中的自调用中，有一些就把方法暴露给window。

3、闭包发生需满足的条件（个人理解）

1. 定义在一个函数内部的函数

（2）读取外部函数的局部变量（局部作用域中的函数）

|  |
| --- |
| function fn() {  var n = 10;  return function () {  var n = 9;  return n;  }  }  var f = fn();  console.log(f());  console.log(f());  console.log(f());  并没有去访问外部局部变量，没有发生闭包，可通过F12断点调试验证。 |

4、闭包小案例

|  |
| --- |
| <ul id="fruit">  <li>西瓜</li>  <li>橙子</li>  <li>葡萄</li>  <li>芒果</li>  </ul>  var fruit = document.getElementById('fruit');  var list = fruit.children;  /\*for (var i = 0, len = list.length; i < len; i++) {  var li = list[i];  li.setAttribute('index', i);  li.onclick = function () {  console.log(this.getAttribute('index'));  }  }\*/  // 采用闭包的方式来解决这个问题,一般不采用这个,因为它的创建的函数内存会一直存在，此时的事件处理函数就处于不同的作用域中。形式了闭包  for (var i = 0, len = list.length; i < len; i++) {  var li = list[i];  (function (i) {  li.onclick = function () {  console.log(i);  }  })(i);  }  1 |

|  |
| --- |
| <body>  你好啊,按钮改变字体  <div id="container">  <input type="button" size="18" value="按钮1">  <input type="button" size="20" value="按钮2">  <input type="button" size="22" value="按钮3">  </div>  <script type="text/javascript">  var container = document.getElementById('container');  var buttons = container.children;  function makeFun(size) {  return function () {  document.body.style.fontSize = size + 'px';  }  }  for (var i = 0, len = buttons.length; i < len; i++) {  var button = buttons[i];  var size = button.getAttribute('size');  console.log(size);  button.onclick = makeFun(size);  }  </script>  </body> |

（1）输出1，2，3

|  |
| --- |
| <script type="text/javascript">  console.log('start');  for (var i = 0; i < 4; i++) {  // 形成了闭包  (function (i) {  setTimeout(function () {  console.log(i);  }, 0);  })(i);    }  console.log('end');  </script>  1 |

（2）思考题

|  |
| --- |
| var name = 'The window';  var obj = {  name: 'obj',  getNameFun: function () {  return function () {  return this.name;  }  }  };  console.log(**obj.getNameFun()**());  // output the window并没有形成闭包 |
| var name = 'The window';  var obj = {  name: 'obj',  getNameFun: function () {  var that = this;  return function () {  return that.name;  }  }  };  console.log(obj.getNameFun()());  //output obj，此时形成了闭包 |

1. 闭包的内存泄漏

栈内存提供一个执行环境，即作用域，包括全局作用域和私有作用域,那他们什么时候释放内存的？

全局作用域----只有当页面关闭的时候全局作用域才会销毁

私有的作用域----只有函数执行才会产生

一般情况下，函数执行会形成一个新的私有的作用域，当私有作用域中的代码执行完成后，我们当前作用域都会主动的进行释放和销毁。但当遇到函数执行返回了一个引用数据类型的值，并且在函数的外面被一个其他的东西给**接收或者说引用**了，这种情况下一般形成的私有作用域都不会销毁。

|  |
| --- |
| function fn(){  var num=100;  return function(){  }  }  var f=fn();//fn执行形成的这个私有的作用域就不能再销毁了  **f = null; // 释放资源** |

也就是像上面这段代码，fn函数内部的私有作用域会被一直占用的，发生了内存泄漏。所谓内存泄漏指任何对象在您不再拥有或需要它之后仍然存在。**闭包不能滥用，否则会导致内存泄露，影响网页的性能。闭包使用完了后，要立即释放资源，将引用变量指向null。**

6、定时器是如何工作的

|  |
| --- |
| <script type="text/javascript">  console.log(**'start'**);  setTimeout(function () {  console.log(**'time'**);  },0);  console.log(**'over'**);  </script>  2 |
| 1 |

十一、递归

递归：函数自己调用自己

在递归的过程中会出错，内存溢出：超过了最大的堆栈大小，每次都会开辟一块新的空间

递归一般要写一个结束的条件

1 + 2 + 3 + ..... + n

|  |
| --- |
| function getSum(n) {  if (n === 1) {  return 1;  }  return n + getSum(n - 1);  }  console.log(getSum(4)); |

1 \* 2 \* 3 \* ...... \* n

|  |
| --- |
| function getSum(n) {  if (n === 1) {  return 1;  }  return n \* getSum(n - 1);  }  console.log(getSum(4)); |

斐波那契数列1、1、2、3、5、8........

|  |
| --- |
| function Fei(n) {  if (n === 1 || n === 2) {  return 1;  }  return Fei(n - 1) + Fei(n - 2);  }  console.log(Fei(4)); |

遍历DOM树

|  |
| --- |
| <body>  <div id="box"><h3 style="color: green;">遍历DOM树</h3></div>  <div id="box1">  <ul>  <li>11</li>  <li>22</li>  <li>33</li>  <li>44</li>  </ul>  </div>  <script type="text/javascript">  // 利用递归遍历所有的节点  function loadTreeNode(element, callback) {  for(var i = 0, len = element.children.length; i < len; i++) {  var childNode = element.children[i];  if (callback) {  callback(childNode, callback);  }  loadTreeNode(childNode, callback);  }  }  loadTreeNode(document.getElementById('box1'), function (element) {  element.onclick = function () {  console.log(this.innerHTML);  }  });  </script>  </body> |

十二、对象拷贝

就是复制一份对象，但是修改后，他们俩之间不会互相影响

1、浅拷贝

简单的数据类型直接复制过去，但是引用类型只是复制了引用

|  |
| --- |
| var obj1 = {  name: 'aaa',  age: 18,  sex: '男',  dog: {  name: 'mike',  age: 2,  color: 'yellow'  }  };  var obj2 = {};  function copy (o1, o2) {  for (var key in o1) {  // o1.key不能这样写,因为key是变量,如果是对象本身具有的属性才能这样写,o1.name,否则就用o1[]这种方式  o2[key] = o1[key];  }  }  copy(obj1, obj2);  **obj1.dog.name = 'Jon';**  console.log(obj2);  1  1 |

2、深拷贝

|  |
| --- |
| var obj1 = {  name: 'aaa',  age: 18,  sex: '男',  dog: {  name: 'mike',  age: 2,  color: 'yellow'  },  friends: ['a','b']  };  var obj2 = {};  function **deepCopy**(o1, o2) {  for (var key in o1) {  var item = o1[key];  if (item instanceof Object) {  o2[key] = {};  deepCopy(item, o2[key]);  } else if (item instanceof Array) {  o2[key] = [];  deepCopy(item, o2[key]);  } else {  o2[key] = o1[key];  }  }  }  deepCopy(obj1, obj2);  **obj1.dog.name = 'jon';**  console.log(obj2);  1  但是数组变成了对象，这个待处理下？？？ |
| 1 |

十三、正则表达式

基本上所有的语言都支持正则表达式。

在sublime中使用 ctrl+f 搜索的时候，实际上整个文档就好比字符串，例如输入 div 就会把div高亮显示。但是当我们想找所有数字的时候，就无法用字符搜索，此时只能用正则表达式来搜索。

正则表通常被用来检索、替换那些符合某个模式(规则)的文本。

在软件中开启正则表达式模式也能使用，来查找相匹配的内容

正则表达式的作用：

（1）给定的字符串是否符合正则表达式的过滤逻辑**(匹配)**

（2）可以通过正则表达式，从字符串中获取我们想要的特定部分**(提取)**

（3）强大的字符串替换能力**(替换)**

通常用来做表单的验证

1、正则表达式的组成

（1）普通字符abc 123。

（2）特殊字符(元字符)：正则表达式中有特殊意义的字符\d \w。

示例演示：

\d 匹配数字。

ab\d 匹配 ab1、ab2。

2、常用元字符串

| **元字符** | **说明** |
| --- | --- |
| \d | 匹配数字（[0-9]） |
| \D | 匹配任意非数字的字符（[^0-9]） |
| \w | 匹配字母或数字或下划线（[a-zA-Z\_0-9]） |
| \W | 匹配任意不是字母，数字，下划线 |
| \s | 匹配任意的空白符 |
| \S | 匹配任意不是空白符的字符 |
| . | 匹配除**换行符**以外的任意单个字符 |
| ^ | 表示匹配行首的文本(以谁开始)，是匹配整段的开头，不是某行 |
| $ | 表示匹配行尾的文本(以谁结束) |

只会看到符合条件的字符，不是一行。

3、限定符

就是限定数量

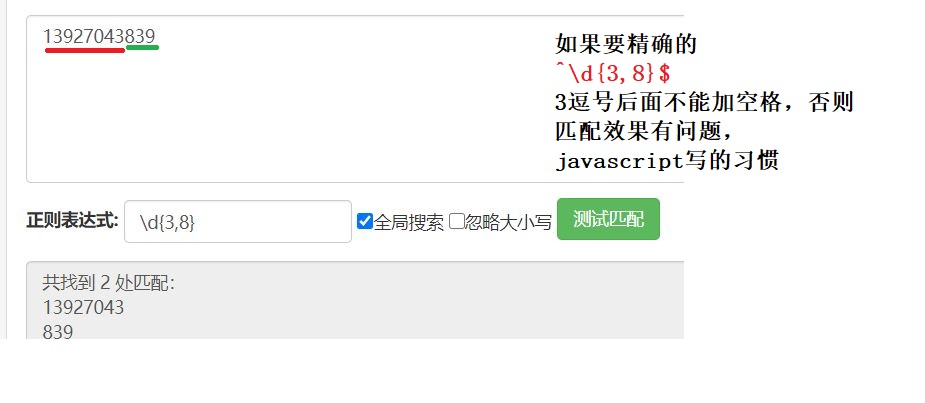
| 限定符，修饰前边的一个字符的次数 | 说明 |
| --- | --- |
| \* | 重复零次或更多次 |
| + | 重复一次或更多次 |
| ? | 重复零次或一次 |
| {n} | 重复n次 |
| {n,} | 重复n次或更多次 |
| {n,m} | 重复n到m次 |







{7}的前一个字符是 \d 就是说要出现连续的7个数字，如果要**精确匹配的话一般写“^\d{7}$”，以他开头跟结尾。**



4、其他特殊字符

|  |
| --- |
| **[]** 字符串用中括号括起来，表示匹配其中的任**一个字符**，相当于或的意思。  **[^]** 匹配**除中括号以内的内容**。  **\** 转义符。如要找字符中的点，就需要加转义符，否则它代表的是元字符  1  **|** 或者，选择两者中的一个。注意|将左右两边分为两部分，而不管左右两边有多长多乱。  2  **()** 从两个直接量中选择一个，**分组的意思**。  eg：gr(a|e)y匹配gray和grey。  **[\u4e00-\u9fa5]** 匹配汉字。 |

只能对字符串进行操作，无法知道它的大小

十四、JavaScript 中使用正则表达式

1、创建正则对象

内置对象RegExp

第一个参数是模式pattern，**字符串**类型，**斜杆要写双斜杆，在字符串中是转义**

第二个参数flag，**字符串**类型（可以忽略，默认是区分大小写）

|  |  |
| --- | --- |
| 标志 | 说明 |
| i | 忽略大小写 |
| g | 全局匹配 |
| gi | 全局匹配+忽略大小写 |

g修饰符用语规定正则表达式执行全局匹配，也就是在找到第一个匹配之后仍然会继续查找。

1. 正则匹配test

当你想要知道一个正则表达式是否与指定的字符串匹配时，就可以使用 **test()。**

正则表达式与指定的字符串匹配 ，返回**true**；否则**false**。

|  |
| --- |
| var regularExpression = new RegExp('lxf**\\**d{2}');  var str = 'lxf12';  console.log(regularExpression.test(str));  如果写成单斜杆，那么就变成了 **lxfd{2}** 这样了 |

|  |
| --- |
| var regularExpression = **/lxf\d{2}/i**;  var str = 'lxF12';  console.log(regularExpression.test(str)); |

onchange当文本框的内容发生改变的时候

触发条件：当文本改变并且失去焦点的时候

也可以在输入的同时进行验证，此时使用onkeydown事件

使用正则表达式的时候，现在工具上测试差不多了，在直接用在代码中

1. input匹配小案例

|  |
| --- |
| <div>  QQ<input type="text" id="txtQQ"><span></span><br />  邮箱<input type="text" id="txtEmail"><span></span><br />  手机号码<input type="text" id="txtTel"><span></span><br />  生日<input type="text" id="txtBirthday"><span></span><br />  姓名<input type="text" id="txtName"><span></span><br />  </div>  <script type="text/javascript">  /\*var txtQQ = document.getElementById('txtQQ');  txtQQ.onblur = function () {  var str = txtQQ.value;  var span = txtQQ.nextElementSibling;  var regEpx = /^\d{5,10}$/;  if (regEpx.test(str)) {  span.innerText = '';  span.style.color = '';  } else {  span.innerText = '输入的QQ号有误';  span.style.color = 'red';  }  }\*/  function addCheckout(elementId, regEpx, tip) {  // 这里是形成了闭包，会常驻于内存  var element = document.getElementById(elementId);  element.onblur = function () {  var str = element.value;  var span = element.nextElementSibling;  if (regEpx.test(str)) {  span.innerText = '';  span.style.color = '';  } else {  span.innerText = tip;  span.style.color = 'red';  }  }  }  addCheckout('txtQQ', **/^\d{5,10}$/**, '请输入正确的QQ格式');  addCheckout('txtEmail', **/^\w+@\w+(\.\w+)+/**, '请输入正确的邮箱格式');  addCheckout('txtTel', **/^\d{11}$/**, '请输入正确手机格式');  addCheckout('txtBirthday', **/^\d{4}-\d{1,2}-\d{1,2}$/**, '请输入正确生日格式');  addCheckout('txtName', **/^[\u4e00-\u9fa5]{2,4}$/**, '请输入2-4个汉字');  </script> |

1. 提取exec()

exec() 方法在一个指定字符串中执行一个搜索匹配。返回一个**结果数组**或 **null。**只会返回一个匹配到的结果。没有匹配的内容就会返回null。

|  |
| --- |
| var salary = 'a:5000, b:6000, c:8000';  var regExp = **/\d+/;**  var content = regExp.exec(salary);  console.log(content);  var content = regExp.exec(salary);  console.log(content);  不加全局搜索标志g的话，结果会一直是第一个匹配的结果  1 |
| **var regExp = /\d+/g;**  1 |

在设置了 [global](https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Reference/Global_Objects/RegExp/global) 或 [sticky](https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Reference/Global_Objects/RegExp/sticky) 标志位的情况下（如 /foo/g or /foo/y），JavaScript [RegExp](https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Reference/Global_Objects/RegExp) 对象是**有状态**的。他们会将上次成功匹配后的位置记录在 [lastIndex](https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Reference/Global_Objects/RegExp/lastIndex) 属性中。使用此特性，exec() 可用来对单个字符串中的多次匹配结果进行逐条的遍历（包括捕获到的匹配）(from MDN)

|  |
| --- |
| exec一次只能返回一个结果，要获得全部匹配的结果，就要使用循环多次进行获取  var salary = 'a:5000, b:6000, c:8000';  var regExp = /\d+/g;  var content;  do {  content = regExp.exec(salary);  if (content) {  console.log(**content[0]**);  }  } while (content);  1 |

1. String对象中的提取match()

（1）返回值

如果使用g标志，则将返回与完整正则表达式匹配的所有结果，但不会返回捕获组。

如果未使用g标志，则仅返回第一个完整匹配及其相关的捕获组（Array）。 在这种情况下，返回的项目将具有如下所述的其他属性。

（2）附加属性

如上所述，匹配的结果包含如下所述的附加特性。

groups: 一个捕获组数组 或 undefined（如果没有定义命名捕获组）。

index: 匹配的结果的开始位置

input: 搜索的字符串.

一个Array，其内容取决于global（g）标志的存在与否，如果未找到匹配则为null。

可以提取多个内容。

字符串中含有多个这样子的模式，所以一般不加开始和结束的特殊字符。

|  |
| --- |
| var salary = 'a:5000, b:6000, c:8000';  var regExp = **/\d+/g;**  var content = salary.match(regExp);  console.log(content);  var str = '123123@xx.com,fangfang@valuedopinions.cn 这是其它内容 286669312@qq.com 2、emailenglish@emailenglish.englishtown.com 286669312@qq.com...';  regExp = **/\w+@\w+(\.\w+)+/g;**  content = str.match(regExp);  console.log(content); |
| 1  2 |

5、分组提取（捕获组）js正则表达式之捕获组

（1）捕获组的定义：

捕获组简单来说就是把正则表达式中用**括号匹配**的内容，保存到内存中，方便调用。js中每一个括号匹配的内容可以被RegExp对象捕获到（最多匹配九个，可以通过返回一个数组索引来取得所有的括号匹配项.），存储在RegExp.$1…中。

需要说明的是正则表达式中（）不能匹配到内容的，就不是捕获组。

|  |
| --- |
| 语法：  RegExp.$1 //第一个括号匹配的内容，以次类推  RegExp.$2  RegExp.$3 |

|  |
| --- |
| RegExp.$1…作为RegExp的只读属性（只有在正确匹配才会改变）  var reg = /([a-z]+)(\d+)/; //匹配一个或多个字母，和一个或多个数字  reg.test('hello123'); //ture  console.log(RegExp.$1) //输出 hello，即匹配的第一个括号里的内容  console.log(RegExp.$2) //输出 123 ，即匹配的第二个括号里的内容 |

**捕获与引用**

被正则表达式匹配（捕获）到的字符串会被暂存起来。其中，由分组捕获的串会从1开始编号，于是我们可以引用这些串：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | **var** reg = /(\d{4})-(\d{2})-(\d{2})/  **var** date = '2010-04-12'  reg.test(date)    RegExp.$1 // 2010  RegExp.$2 // 04  RegExp.$3 // 12 |

$1引用了第一个被捕获的串，$2是第二个，依次类推。

1. MDN中捕获组

组和范围表示表达式字符的“组和范围”。

|  |  |
| --- | --- |
| (x) | **捕获组:** 匹配x并记住匹配项。例如，/(foo)/匹配并记住“**foo** bar”中的“**foo”。**  正则表达式可以有**多个捕获组**。结果，匹配通常在数组中捕获的组，该数组的成员与捕获组中左括号的顺序相同。这通常只是捕获组本身的顺序。 |
| (?:x) | **非捕获组:** 匹配 “x”，但不记得匹配。不能从结果数组的元素中收回匹配的子字符串 |
| \n | 其中n是一个**正整数**。对正则表达式中与n括号匹配的最后一个子字符串的反向引用(计算左括号)。  例如，/apple(,)\sorange\1/ 匹配 “apple，orange，cherry，peach” 中的 "apple，orange，"， 其中 \1 引用了 之前使用 （） 捕获的。  例子：  正则表达式是这样/apple(,)(5)orange\2/那么等同于/apple,5orange5/,那样是可以的。  \n实际上就是反向引用其前面捕获组中捕获的数据,有几个捕获组,就可以引用几个。 |

6、String对象中的提取split()

切连续相同的字符是有问题的？？？未解，暂时无解，只要有连续相同分隔符，只能切一个，其他变成空字符，采用正则表达式解决就行，暂时不纠结

|  |
| --- |
| var date = '2021-1-4';  var content = date.split(**'-'**); // 特定的字符串  console.log(content);  date = '2020/1-2';  content = date.split(/[/-]/); // 正则表达式  console.log(content);  var email = 'xxxx@itcast.com';  content = email.split(/[@\.]/);  console.log(content);  1  date = 'hello 2 hi';  content = date.split(/\d/);  console.log(content);  为什么\d加上括号之后，结果不一致？？？  答：如果分隔符是包含捕获括号的正则表达式，则每次分隔符匹配时，捕获括号的结果（包括任何未定义的结果）将被拼接到输出数组中。但是，并不是所有浏览器都支持此功能。结合捕获组定义来理解。 |

7、String对象中的提取replace()

trim();去除字符串前后的空格。

replace()只能替换掉第一个查找到的内容。

参数一：一个将被 newSubStr 替换的 字符串。其被视为一整个字符串，而不是一个正则表达式。仅**第一个匹配项**会被替换。

返回值：一个部分或全部匹配由替代模式所取代的**新的字符串**

|  |
| --- |
| var str = ' abcdefgh ijklmno pqrst uvwxyz';  var result = str.replace(**' '**, **'x'**);  console.log(result);  1 |

1. 替换字符串中所有的空白（总结）

|  |
| --- |
| 方法一：采用replace，正则表达式，注意全局搜索  var str = ' abcdefgh ijklmno pqrst uvwxyz ';  var result = str.replace(**/\s/g, 'x'**);  console.log(result);  1 |
| 方法二：采用split空字符切割再join空字符组合  var str = ' abcdefgh ijklmno pqrst uvwxyz ';  var result = **str.split(' ').join('');**  console.log(result); |

|  |
| --- |
| 获得去除空白后的数组  方法：先去除前后空格，split正则表达式切割，不能单独切一个空格，要加限定符全部切掉  var str = ' abcdefgh ijklmno pqrst uvwxyz ';  var result = str.trim().split(/\s**+**/g);  console.log(result);  2  引深为如果分隔符是连续的话，最好用split(/分隔符+/g)来直接切割，不加限定符，会出同样的问题。 |

（2）替换多个不同字符

|  |
| --- |
| var str = 'a,b,c,r，d.';  var result = str.replace(/[,，\.]/g,'');  console.log(result);  1 |

（3）当第一个参数是正则表达式，第二个参数是函数的时候。

当匹配的时候，会替换成函数的返回值。

|  |
| --- |
| var str = '1234567897';  var temp = str.replace(/7/g, function () {  return 'hello';  });  console.log(temp);  输出 123456hello89hello |

（4）第二个参数[使用字符串作为参数](https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Reference/Global_Objects/String/replace" \l "%E4%BD%BF%E7%94%A8%E5%AD%97%E7%AC%A6%E4%B8%B2%E4%BD%9C%E4%B8%BA%E5%8F%82%E6%95%B0" \o "Permalink to 使用字符串作为参数)

替换字符串可以插入下面的特殊变量名：

|  |  |
| --- | --- |
| 变量名 | 代表的值 |
| $$ | 插入一个 "$"。 |
| $& | 插入匹配的子串。 |
| $` | 插入当前匹配的子串左边的内容。 |
| $' | 插入当前匹配的子串右边的内容。 |
| $n | 假如第一个参数是 [RegExp](https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Reference/Global_Objects/RegExp)对象，并且 n 是个小于100的非负整数，那么插入第 n 个括号匹配的字符串。提示：索引是从1开始。如果不存在第 n个分组，那么将会把匹配到到内容替换为字面量。比如不存在第3个分组，就会用“$3”替换匹配到的内容。 |
| $<Name> | 这里Name 是一个分组名称。如果在正则表达式中并不存在分组（或者没有匹配），这个变量将被处理为空字符串。只有在支持命名分组捕获的浏览器中才能使用。 |

8、String对象中的提取search()

可以搜索字符串中是否含有指定内容

如果搜索到指定内容，则会返回第一次出现的索引，如果没有搜索到则返回-1

它可以接受一个正则表达式作为参数，然后会根据正则表达式去检索字符串

Search只会查找第一个，即使设置了全局配也只返回第一个

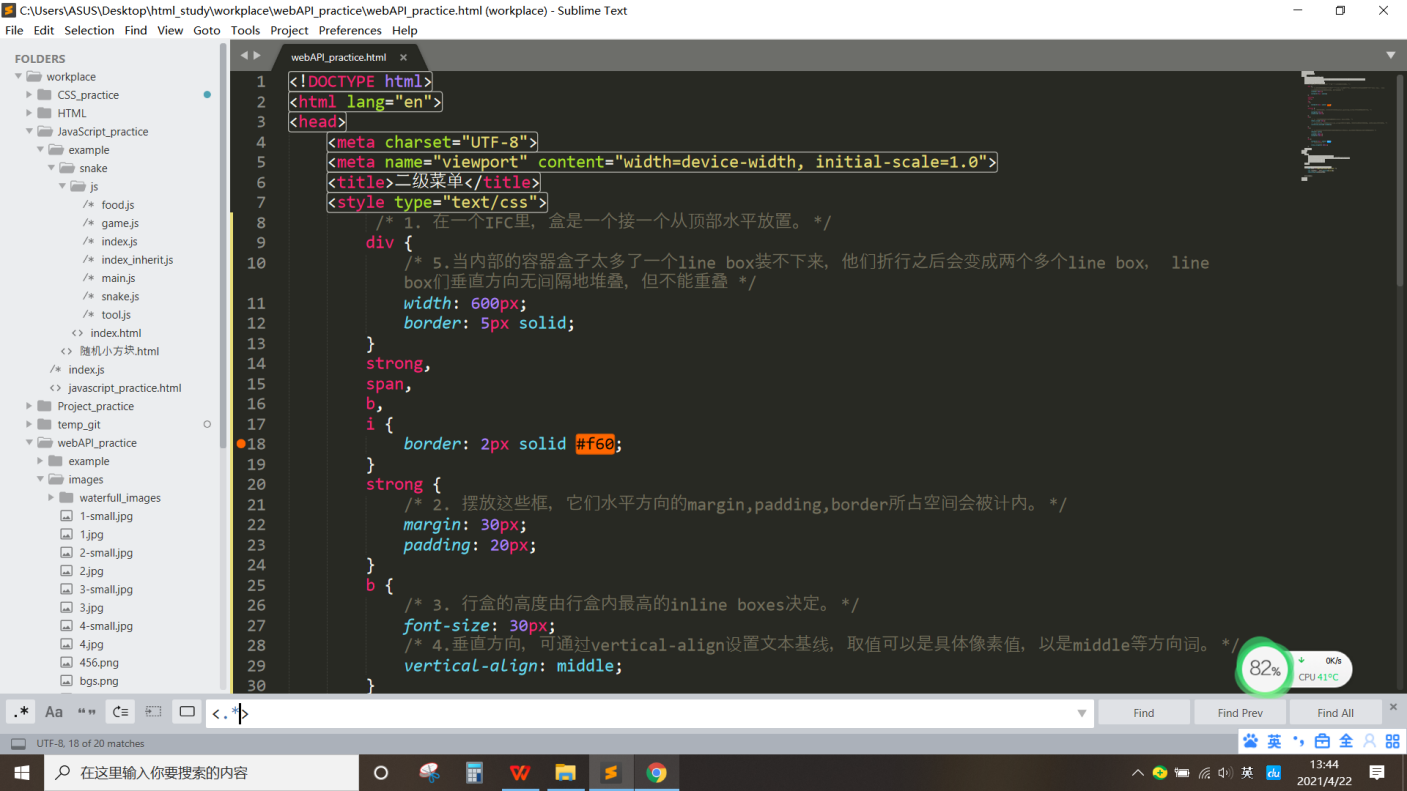
|  |
| --- |
| var str = 'abc ads dsx adc adb ';  var result = str.search(/ad/);  console.log(result);  输出 4 |

9、贪婪模式和非贪婪模式

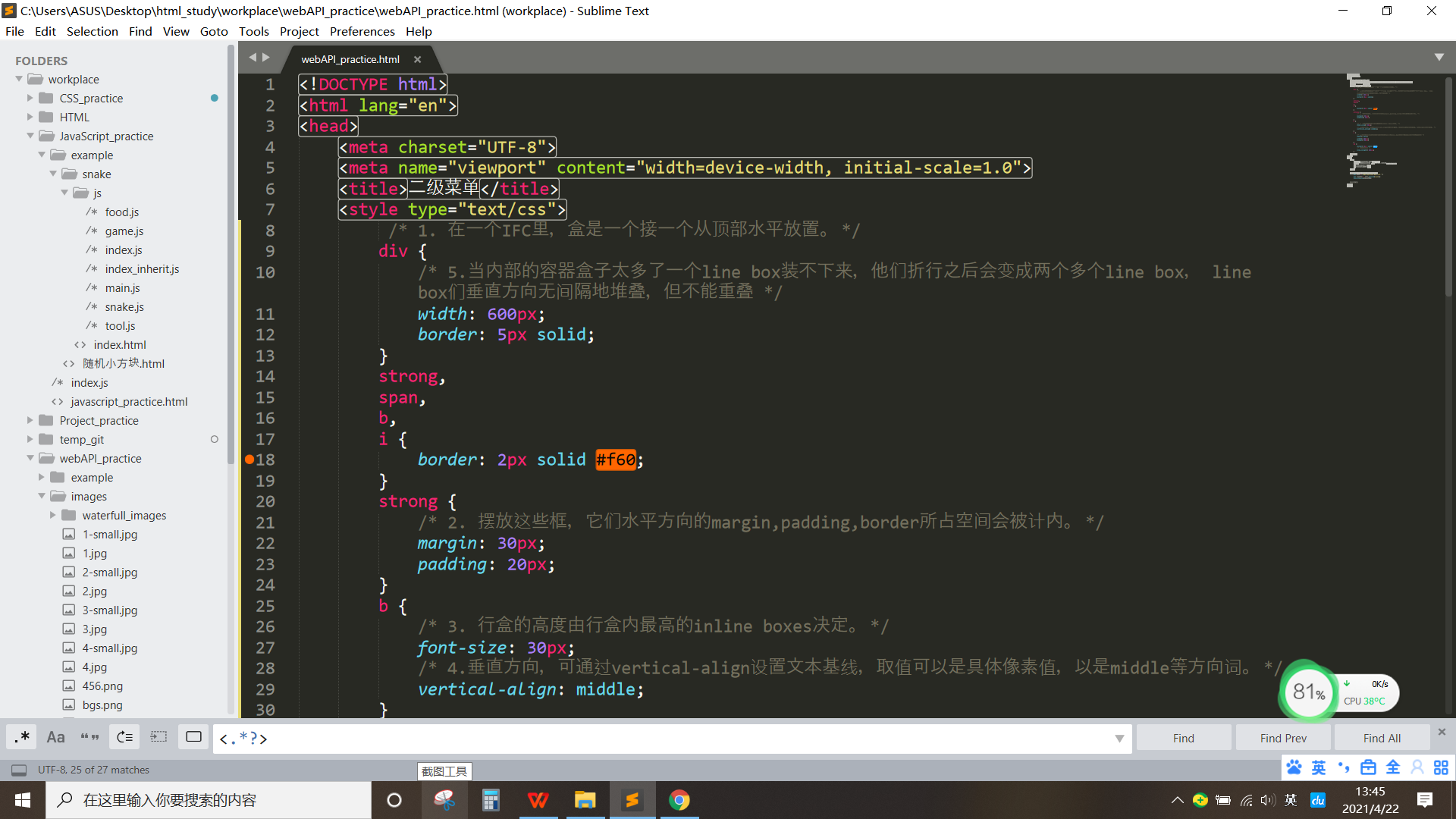
贪婪与非贪婪模式影响的是被量词修饰的子表达式的匹配行为。

贪婪模式在整个表达式匹配成功的前提下，尽可能多的匹配。

非贪婪模式在整个表达式匹配成功的前提下，尽可能少的匹配。非贪婪模式只被部分NFA引擎所支持。



这个属于贪婪模式，<title>二级菜单</title>中的二级菜单本来是不想要的



非贪婪模式

在一些使用NFA引擎的语言中，在匹配优先量词（量词就是数量意思，限定符）后加上“?”，即变成属于非贪婪模式的量词，也叫做忽略优先量词，包括：

“{m,n}?”、“{m,}?”、“??”、“\*?”和“+?”。

从正则语法的角度来讲， 被匹配优先量词修饰的子表达式使用的就是贪婪模式，如“(Expression)+”；

被忽略优先量词修饰的子表达式使用的就是非贪婪模式，如“(Expression)+?”。

10、正则表a达式中?=和?:和?!的理解

（1）前瞻，后顾，负前瞻，负后顾四个概念：理解?=和?!

// 前瞻：

exp1(**?=**exp2) 查找exp2前面的exp1

// 后顾：

(**?<=**exp2)exp1 查找exp2后面的exp1

// 负前瞻：

exp1(**?!**exp2) 查找后面不是exp2的exp1

// 负后顾：

(**?<!**exp2)exp1 查找前面不是exp2的exp1

|  |
| --- |
| var str = "中国人";  var newStr = str.replace(/(?<=中国)人/, "rr");  console.log(newStr);  输出 中国rr |
| var str = "fa国人";  var newStr = str.replace(/(?<=中国)人/, "rr");  console.log(newStr);  *结果为 fa国人，因为人前面不是中国，所以无法匹配到* |

（2）理解?:则需要理解捕获分组和非捕获分组的概念：

()表示捕获分组，()会把每个分组里的匹配的值保存起来，使用$n(n是一个数字，表示第n个捕获组的内容)

(?:)表示非捕获分组，和捕获分组唯一的区别在于，非捕获分组匹配的值不会保存起来

|  |
| --- |
| // 数字格式化 1,123,000  "1234567890".replace(/\B(?=(?:\d{3})+(?!\d))/g,",") // 结果：1,234,567,890，匹配的是后面是3\*n个数字的非单词边界(\B) |

1. 表格排序案例
2. 遇到排序的问题，如果是对象的排序，可以先采取sort来尝试。等于0的情况，它们的相对位置不会变化，详细见MDN
3. 动态添加元素的时候，可以采用数组来拼接字符串，使用innerHTML；或者createElement来操作。

写完之后，要执行在html中确认元素是否出现

是哦也能够innerHTML清空了table下边的元素，是否会引起其他的问题