--

[object 1](#_Toc31177)

[属性名表达式 1](#_Toc8093)

[简洁表示法 1](#_Toc24908)

[.create 1](#_Toc19688)

[.create new 1](#_Toc22922)

[Object.create(Arrar.prototype) 2](#_Toc24826)

[.is() 3](#_Toc12181)

[.assign() 3](#_Toc3629)

[.defineProperty 3](#_Toc26519)

[.getOwnPropertyDescriptor 3](#_Toc15992)

[super 4](#_Toc2156)

[obj.hasOwnProperty("xx") 4](#_Toc5878)

[遍历对象的属性 4](#_Toc23955)

[原型链 5](#_Toc19646)

[关于prototype 5](#_Toc10541)

[相互关系 6](#_Toc32062)

[prototype关系 6](#_Toc20316)

[继承 6](#_Toc3776)

[双向绑定（vue） 7](#_Toc3402)

[es6 - class 7](#_Toc447)

[其他 9](#_Toc8593)

[学：单var形式 9](#_Toc8757)

[关于预解析、全局与局部的表现 9](#_Toc14573)

[关于var 9](#_Toc16395)

[全局变量 9](#_Toc18770)

[JS引擎 10](#_Toc4172)

[js中数据类型 10](#_Toc5855)

[闭包 11](#_Toc9431)

[闭包的条件 11](#_Toc14663)

[闭包的理论 11](#_Toc5786)

[作用 11](#_Toc28072)

[DOM循环添加事件 12](#_Toc7834)

[匿名函数与闭包 12](#_Toc8164)

[js建立对象：工厂 构造 原型 12](#_Toc31831)

[立即执行函数与模块 14](#_Toc12280)

[引用 15](#_Toc15447)

[关于this 15](#_Toc23275)

[关于settimeout 16](#_Toc17458)

[异步加载js方式 16](#_Toc26627)

[!function($){}(window.jQuery)是什么意思 17](#_Toc24502)

[JS匿名函数与自执行 18](#_Toc24410)

[面试题 19](#_Toc15270)

[立即执行函数: (function(){...})() 与 (function(){...}()) 有什么区别? 19](#_Toc9928)

object

|  |  |
| --- | --- |
|  | <https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Reference/Global_Objects/Object> |
|  | Object.is()方法判断两个值是否是相同的值。这与===运算符也不一样。===运算符（和==运算符）将数字值-0和+0视为相等，并认为Number.NaN不等于NaN。  Object.assign()  Object.create() 方法会使用指定的原型对象及其属性去创建一个新的对象。  Object.entries()  Object.values()  Object.keys()  Object.defineProperties()  Object.defineProperty()  Object.getOwnPropertyDescriptor() 方法返回指定对象上一个自有属性对应的**属性描述符**。（自有属性指的是直接赋予该对象的属性，不需要从原型链上进行查找的属性）  Object.getOwnPropertyDescriptors() 方法用来获取一个对象的所有自身属性的描述符。  Object.getOwnPropertyNames()方法返回一个由指定对象的所有自身属性的属性名（包括不可枚举属性但不包括Symbol值作为名称的属性）组成的数组。  Object.getOwnPropertySymbols() 方法会返回一个数组，该数组包含了指定对象自身的（非继承的）所有 symbol 属性键。  Object.setPrototypeOf() 方法设置一个指定的对象的原型 ( 即, 内部[[Prototype]]属性）到另一个对象或 null。  Object.getPrototypeOf() 方法返回指定对象的原型（内部[[Prototype]]属性的值）。  Object.seal() 方法可以让一个对象密封，并返回被密封后的对象。密封对象将会阻止向对象添加新的属性，并且会将所有已有属性的可配置性（configurable）置为不可配置（false），即不可修改属性的描述或删除属性。但是可写性描述（writable）为可写（true）的属性的值仍然被修改。  Object.isSealed() 方法判断一个对象是否被密封。  Object.freeze() 方法可以冻结一个对象，冻结指的是不能向这个对象添加新的属性，不能修改其已有属性的值，不能删除已有属性，以及不能修改该对象已有属性的可枚举性、可配置性、可写性。也就是说，这个对象永远是不可变的。该方法返回被冻结的对象。  Object.isFrozen() 方法判断一个对象是否被冻结 frozen。  Object.preventExtensions() 方法让一个对象变的不可扩展，也就是永远不能再添加新的属性。(阻碍扩展)  Object.isExtensible() 方法判断一个对象是否是可扩展的（是否可以在它上面添加新的属性）。  -Object.observe()  -Object.unobserve()  Object.prototype.\_\_defineGetter\_\_()  Object.prototype.\_\_defineSetter\_\_()  Object.prototype.\_\_lookupGetter\_\_()  Object.prototype.\_\_lookupSetter\_\_()  Object.prototype.eval()  Object.prototype.propertyIsEnumerable()  Object.prototype.toLocaleString()  Object.prototype.toSource()  Object.prototype.toString()  Object.prototype.unwatch()  Object.prototype.valueOf()  Object.prototype.watch()  Object.prototype.hasOwnProperty() 方法会返回一个布尔值，指示对象是否具有指定的属性作为自身（不继承）属性。  Object.prototype.isPrototypeOf()  Reflect.ownKeys() 方法用于返回对象的所有属性，基本等同于Object.getOwnPropertyNames与Object.getOwnPropertySymbols之和。 |
| 属性名表达式  简洁表示法 | **let birth** = '2000/01/01'; **let keyWord** = 'testKey'; **const Point** = {  name: '张三',  birth, //等同于birth: birth  hello() {  **console**.log('我的名字是', **this**.name);  }, // 等同于hello: function () {}  **class**() {  },// 'class': function() {} class是字符串，所以不会因为它属于关键字，而导致语法解析报错  ['a' + 'bc']: 123,  [**keyWord**]: "test",  ['h' + 'ello']() {  **return** 'hi';  },  // [foo] 属性名表达式与简洁表示法，不能同时使用，会报错。正确使用方式{ [foo]: 'abc'}  // [{a:"a"}]: 'valueB' // Object {[object Object]: "valueB"} 属性名表达式如果是一个对象，默认情况下会自动将对象转为字符串[object Object] }; |
| .create | Object.create 使用指定的原型对象和其属性创建了一个新的对象。  语法  Object.create(proto[, propertiesObject])  参数  proto 新创建对象的原型对象。  propertiesObject 可选。如果没有指定为 undefined，则是要添加到新创建对象的可枚举属性（即其自身定义的属性，而不是其原型链上的枚举属性）对象的属性描述符以及相应的属性名称。这些属性对应Object.defineProperties()的第二个参数。  返回值  在指定原型对象上添加新属性后的对象。  例外  如果propertiesObject参数不是 null 或一个对象，则抛出一个 TypeError 异常。  Object.create(proto [, propertiesObject ]) 是E5中提出的一种新的对象创建方式，第一个参数是要继承的原型，如果不是一个子函数，可以传一个null，第二个参数是对象的属性描述符，这个参数是可选的。  对象的\_\_proto\_\_链接是不能直接修改的，而函数对象的prototype属性是可以修改的，因此，Object.create实际上是利用了这个特点，结合new来完成了一个拷贝创建的过程 |
| .create new | 你不知道的javascript之Object.create 和new区别  <http://blog.csdn.net/blueblueskyhua/article/details/73135938>   1. ### 两种创建对象的方式   var Base = function () {}  var o1 = new Base();  var o2 = Object.create(Base);   1. new的实现方式：   var o1 = new Object();  o1.\_\_proto\_\_= Foo.prototype;  Foo.call(o1);  new做法是新建一个obj对象o1，并且让o1的\_\_proto\_\_指向了Foo.prototype对象。并且使用call 进行强转作用环境。从而实现了实例的创建。   1. Object.create的实现方式：   Object.create = function (o) {  var F = function () {};  F.prototype = o;  return new F();  };  Object.create是内部定义一个对象，并且让F.prototype对象 赋值为引进的对象/函数 o，并return出一个新的对象。  我们来看看两个对象打印情况。  console.log(o1);//Base {}  console.log(o2);//Function {} |
|  | 1. instanceof 和 isPrototypeOf   写了创建一个对象实例，并且说了通过原型链来完成这一个个对象之间的联系，但是你怎么知道就一定含有呢？所以我们需要一个判断机制。  var Base = function () {  this.a = 2  }  Base.prototype.b = 3;  var o1 = new Base();  var o2 = Object.create(Base.prototype);  console.log(o1 instanceof Base); // true  console.log(o2 instanceof Base); // true  console.log(Base.prototype.isPrototypeOf(o1));// true  console.log(Base.prototype.isPrototypeOf(o2));// true  instanceof 说的是在a的整条[[Prototype]] 是否含有Foo.prototype对象。 但是这个方法只能实现对象(a)和函数(带.prototype引用的Foo),如果你想判断两个对象（a 和 b）是否通过[[Prototype]]链关联。只用instanceof就无法实现。  所以这里用到了isPrototypeOf。  var a = {};  var b = Object.create(a);  console.log(a.isPrototypeOf(b)); // true 在a的[[Prototype]]是否出现过b来判断。   1. isPrototypeOf实现方式:   function isRelatedTo(o1,o2){  function F(){}  F.prototype = o2;  return o1 instanceof F;  }  console.log(a.isPrototypeOf(b) === isRelatedTo(b,a));// true  上述函数通过了构建一个辅助函数F，构建了一个prototype对象。从而达到instanceof比较的条件。 |
|  | 1. constructor   我们先来看看下面的代码。  function Foo(){  }  console.log(Foo.prototype.constructor === Foo);//true  var a = new Foo();  console.log(a.constructor === Foo);//true  看起来a.constructor === Foo 为真意味着a的确有一个.constructor指向Foo的.constructor属性，但事实并不是这样的。  function Foo(){  }  Foo.prototype = {}  var a1 = new Foo();  console.log(a1.constructor === Foo);//false  console.log(a1.constructor === Object);//true  可以看到a1并没有.constructor属性。那是为什么呢。？因为a1没有.constructor属性，他会委托[[prototype]]链上的Foo.prototype。但是新建的Foo.prototype也没有.constructor,所以继续往上找，一直到了顶端的Object.prototype。  你可以手动地进行修正.constructor的指向。  所以可以看出.constructor是一个非常不可靠，并且不安全的引用。在开发中尽量避免使用这些引用，。 |
|  | 改造代码:  var Base = function () {  this.a = 2  }  Base.prototype.b = 3;  var o1 = new Base();  var o2 = Object.create(Base);  console.log(o1.a); //2  console.log(o2.a); //undefined  console.log(o1.b); //3  console.log(o2.b); //undefined  改造代码:  var Base = function () {  this.a = 2  }  Base.prototype.b = 3;  var o1 = new Base();  var o2 = Object.create(Base.prototype);  console.log(o1.a); //2  console.log(o2.a); //undefined  console.log(o1.b); //3  console.log(o2.b); //3  可以看到Object.create 失去了原来对象的(属性,[prototype])的访问  （一开始没理解prototype和\_\_proto\_\_ 的关系。造成对这两种方式的创建理解非常费解）。  那我们就以图说话。  IMG_260 IMG_261  （F在创建后被销毁）  看完上图，我们就知道了，为什么通过Object.create构造的连Base原型上的属性都访问不到，因为他压根就没有指向他的prototype。这也就说明了\_\_proto\_\_ 和 prototype 的区别。所以上面在prototype定义的a，只是Base的prototype对象上的一个属性。  再来看看就是：  new关键字必须是以function定义的。  Object.create 则 function和object都可以进行构建。 |
| Object.create(Arrar.prototype) | 1. 创建   const arrayProto = Array.prototype  const arrayMethods = Object.create(arrayProto)  这一步是通过 Object.create 创建了一个原型对象为Array.prototype的空对象。然后通过Object.defineProperty方法对这个对象定义几个变异的数组方法。有些新手可能会直接修改 Array.prototype 上的方法，这是很危险的行为，这样在引入的时候会全局影响Array 对象的方法，而使用Object.create实质上是完全了一份拷贝，新生成的arrayMethods对象的原型指针\_\_proto\_\_指向了Array.prototype，修改arrayMethods 对象不会影响Array.prototype。  基于这种原理，我们通常会使用Object.create 实现类式继承。  // 实现继承  var extend = function(Child, Parent) {  // 拷贝Parent原型对象  Child.prototype = Object.create(Parent.prototype);  // 将Child构造函数赋值给Child的原型对象  Child.prototype.constructor = Child;  }  // 实例  var Parent = function () {  this.name = 'Parent';  }  Parent.prototype.getName = function () {  return this.name;  }  var Child = function () {  this.name = 'Child';  }  extend(Child, Parent);  var child = new Child();  console.log(child.getName()) |
|  |  |
| .is() | ES6 提出“Same-value equality”（同值相等）算法，用来解决这个问题。Object.is就是部署这个算法的新方法。它用来比较两个值是否严格相等，与严格比较运算符（===）的行为基本一致。  Object.is('foo', 'foo') // true  Object.is({}, {}) // false  不同之处只有两个：一是+0不等于-0，二是NaN等于自身。  +0 === -0 //true  NaN === NaN // false  Object.is(+0, -0) // false  Object.is(NaN, NaN) // true |
| Object.defineProperty(Object, 'is', {  value: function (x, y) {  if (x === y) {  // 针对+0 不等于 -0的情况  return x !== 0 || 1 / x === 1 / y;  }  // 针对NaN的情况  return x !== x && y !== y;  },  configurable: true,  enumerable: false,  writable: true  }); |
| .assign() | 1. 复制一个对象   var obj = { a: 1 };  var copy = Object.assign({}, obj);  console.log(copy); // { a: 1 }   1. 深拷贝问题   针对深拷贝，需要使用其他方法，因为 Object.assign()拷贝的是属性值。假如源对象的属性值是一个指向对象的引用，它也只拷贝那个引用值。  function test() {  'use strict';  let obj1 = { a: 0 , b: { c: 0}};  let obj2 = Object.assign({}, obj1);  console.log(JSON.stringify(obj2)); // { a: 0, b: { c: 0}}    obj1.a = 1;  console.log(JSON.stringify(obj1)); // { a: 1, b: { c: 0}}  console.log(JSON.stringify(obj2)); // { a: 0, b: { c: 0}}    obj2.a = 2;  console.log(JSON.stringify(obj1)); // { a: 1, b: { c: 0}}  console.log(JSON.stringify(obj2)); // { a: 2, b: { c: 0}}    obj2.b.c = 3;  console.log(JSON.stringify(obj1)); // { a: 1, b: { c: 3}}  console.log(JSON.stringify(obj2)); // { a: 2, b: { c: 3}}    // Deep Clone  obj1 = { a: 0 , b: { c: 0}};  let obj3 = JSON.parse(JSON.stringify(obj1));  obj1.a = 4;  obj1.b.c = 4;  console.log(JSON.stringify(obj3)); // { a: 0, b: { c: 0}}}  test();   1. 合并对象   var o1 = { a: 1 };var o2 = { b: 2 };var o3 = { c: 3 };  var obj = Object.assign(o1, o2, o3);  console.log(obj); // { a: 1, b: 2, c: 3 }  console.log(o1); // { a: 1, b: 2, c: 3 }, 注意目标对象自身也会改变。   1. 合并具有相同属性的对象   var o1 = { a: 1, b: 1, c: 1 };var o2 = { b: 2, c: 2 };var o3 = { c: 3 };  var obj = Object.assign({}, o1, o2, o3);  console.log(obj); // { a: 1, b: 2, c: 3 } |
| .defineProperty | Object.defineProperty()方法可以直接在一个对象上定义一个新属性，或者修改一个已经存在的属性， 并返回这个对象。   1. 语法：   Object.defineProperty(obj, prop, descriptor)   1. 参数：   obj：需要定义属性的对象。  prop：需被定义或修改的属性名。  descriptor：需被定义或修改的属性的描述符。   1. 返回值：   返回传入函数的对象，即第一个参数obj.  该方法重点是描述，对象里目前存在的属性描述符有两种主要形式：数据描述符和存取描述符。数据描述符是一个拥有可写或不可写值的属性。存取描述符是由一对 getter-setter 函数功能来描述的属性。描述符必须是两种形式之一；不能同时是两者。     1. 数据描述符和存取描述符均具有以下可选键值：   configurable：当且仅当该属性的 configurable 为 true 时，该属性才能够被改变，也能够被删除。默认为 false。  enumerable：当且仅当该属性的 enumerable 为 true 时，该属性才能够出现在对象的枚举属性中。默认为 false。   1. 数据描述符同时具有以下可选键值：   value：该属性对应的值。可以是任何有效的 JavaScript 值（数值，对象，函数等）。默认为 undefined。  writable：当且仅当仅当该属性的writable为 true 时，该属性才能被赋值运算符改变。默认为 false。   1. 存取描述符同时具有以下可选键值：   get：一个给属性提供 getter 的方法，如果没有 getter 则为 undefined。该方法返回值被用作属性值。默认为undefined。  set：一个给属性提供 setter 的方法，如果没有 setter 则为 undefined。该方法将接受唯一参数，并将该参数的新值分配给该属性。默认为undefined。  我们可以通过Object.defineProperty()方法精确添加或修改对象的属性。比如，直接赋值创建的属性默认情况是可以枚举的，但是我们可以通过Object.defineProperty()方法设置enumerable属性为false为不可枚举。  var obj = {  a: 0,  b: 1  };  for (var prop in obj) {  console.log(`obj.${prop} = ${obj[prop]}`);  }  结果："obj.a = 0""obj.b = 1"  我们通过Object.defineProperty()修改如下：  var obj = {  a: 0,  b: 1  }  Object.defineProperty(obj, 'b', {  enumerable: false  })for (var prop in obj) {  console.log(`obj.${prop} = ${obj[prop]}`);  }  结果："obj.a = 0"  这里需要说明的是我们使用Object.defineProperty()默认情况下是enumerable属性为false，例如：  var obj = {  a: 0  }  Object.defineProperty(obj, 'b', {  value: 1  })  for (var prop in obj) {  console.log(`obj.${prop} = ${obj[prop]}`);  }  结果："obj.a = 0"  其他描述属性使用方法类似，不做赘述。Vue源码core/util/lang.jsS中定义了这样一个方法：  /\*\*  \* Define a property.  \*/  export function def(obj: Object, key: string, val: any, enumerable ? : boolean) {  Object.defineProperty(obj, key, {  value: val,  enumerable: !!enumerable,  writable: true,  configurable: true  })  } |
| .getOwnPropertyDescriptor | Object.getOwnPropertyDescriptor() 返回指定对象上一个自有属性对应的属性描述符。（自有属性指的是直接赋予该对象的属性，不需要从原型链上进行查找的属性） 语法：Object.getOwnPropertyDescriptor(obj, prop)  参数：  obj：在该对象上查看属性。  prop：一个属性名称，该属性的属性描述符将被返回。  返回值：如果指定的属性存在于对象上，则返回其属性描述符（property descriptor），否则返回 undefined。可以访问“属性描述符”内容，例如前面的例子：  var o = {  a: 0  };  Object.defineProperty(o, "b", {  get: function () {  return this.a + 1;  },  set: function (value) {  this.a = value / 2;  }  });  var des = Object.getOwnPropertyDescriptor(o, 'b');  console.log(des); // { get: [Function: get], set: [Function: set], enumerable: false, configurable: false }  console.log(des.get); // [Function: get] |
| super | //this关键字总是指向函数所在的当前对象，ES6 又新增了另一个类似的关键字super，指向当前对象的原型对象。 **const proto** = {  foo: 'hello' };  **const obj** = {  find() {  **return super**.**foo**;  } }; **Object**.setPrototypeOf(**obj**, **proto**); **obj**.find() // "hello"  super关键字表示原型对象时，只能用在对象的方法之中，用在其他地方都会报错。  目前，只有对象方法的简写法可以让 JavaScript 引擎确认，定义的是对象的方法。 |
| obj.hasOwnProperty("xx") | hasOwnProperty() 方法会返回一个布尔值，指示对象是否具有指定的属性作为自身（不继承）属性。  obj.hasOwnProperty(prop)  prop:要检测的属性  [字符串](https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Reference/String" \o "此页面仍未被本地化, 期待您的翻译!) 名称或者 [Symbol](https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Reference/Global_Objects/Symbol" \o "Symbol()函数会返回symbol类型的值，该类型具有静态属性和静态方法。它的静态属性会暴露几个内建的成员对象；它的静态方法会暴露全局的symbol注册，且类似于内建对象类，但作为构造函数来说它并不完整，因为它不支持语法：\"new Symbol()\"。)。 |
| 1. 使用 hasOwnProperty 方法判断属性是否存在   o = new Object();  o.prop = 'exists';  o.hasOwnProperty('prop'); // 返回 true  delete o.prop;  o.hasOwnProperty('prop'); // 返回 false   1. 自身属性与继承属性   下面的例子演示了 hasOwnProperty 方法对待自身属性和继承属性的区别：  o = new Object();  o.prop = 'exists';  o.hasOwnProperty('prop'); // 返回 true  o.hasOwnProperty('toString'); // 返回 false  o.hasOwnProperty('hasOwnProperty'); // 返回 false   1. 使用 hasOwnProperty 作为属性名   JavaScript 并没有保护 hasOwnProperty 属性名，因此某个对象是有可能存在使用这个属性名的属性，使用外部的 hasOwnProperty 获得正确的结果是需要的：  // 如果担心这种情况，可以直接使用原型链上真正的 hasOwnProperty 方法  ({}).hasOwnProperty.call(foo, 'bar'); // true  // 也可以使用 Object 原型上的 hasOwnProperty 属性  Object.prototype.hasOwnProperty.call(foo, 'bar'); // true |
|  | function siteAdmin(nickName, siteName) {   this.nickName = nickName;   this.siteName = siteName;  }  siteAdmin.prototype.showAdmin = function () {   console.log(this.nickName + "是" + this.siteName + "的站长!")  };  siteAdmin.prototype.showSite = function (siteUrl) {   this.siteUrl = siteUrl;   return this.siteName + "的地址是" + this.siteUrl;  };  var matou = new siteAdmin("脚本之家", "WEB前端开发");  var matou2 = new siteAdmin("脚本之家", "WEB前端开发");  matou.age = "30";  // matou.showAdmin();  // matou.showSite("http://www.jb51.net/");  console.log(matou.hasOwnProperty("nickName")); //true  console.log(matou.hasOwnProperty("age")); //true  console.log(matou.hasOwnProperty("showAdmin")); //false  console.log(matou.hasOwnProperty("siteUrl")); //false //true  console.log(siteAdmin.prototype.hasOwnProperty("showAdmin")); //true  console.log(siteAdmin.prototype.hasOwnProperty("siteUrl")); //false |
|  | 1. #如何判断某个属性是不是某个对象自身的属性呢？   用 in 操作符（不严谨，它其实判定的是这个属性在不在该对象的原型链上）  'age' in obj; // true  'userProp' in obj; // true （userProp 是 obj 原型链上的属性）  'name' in Object; // true  // 上面这个也是 true 的原因是，Object 是一个构造函数，而函数恰巧也有一个 name 属性  Object.name; // 'Object'  Array.name; // 'Array'   1. 用 hasOwnProperty()，这个方法只会检测某个对象上的属性，而不是原型链上的属性。   obj.hasOwnProperty('age'); // true  obj.hasOwnProperty('skills'); // true  obj.hasOwnProperty('userProp'); // false   1. 但是它还是有不足之处的。举例~   // 利用 Object.create() 新建一个对象，并且这个对象没有任何原型链  var obj2 = Object.create(null, {  name: { value: 'percy' },  age: { value: 21 },  skills: { value: ['html', 'css', 'js'] }  });  obj2.hasOwnProperty('name'); // 报错  obj2.hasOwnProperty('skills'); // 报错   1. #针对上面的情况，我们用一个更完善的解决方案来解决。   使用 Object.prototype.hasOwnProperty.call(obj,'prop'…)  Object.prototype.hasOwnProperty.call(obj2,'name'); // true  Object.prototype.hasOwnProperty.call(obj2,'skills'); // true  Object.prototype.hasOwnProperty.call(obj2,'userProp'); // false |
| 遍历对象的属性 | |  | | --- | | // 为 Object 设置三个自定义属性（可枚举）  Object.prototype.userProp = 'userProp';  Object.prototype.getUserProp = function() {  return Object.prototype.userProp;  };  // 定义一个对象，隐式地继承自 Object.prototype  var obj = {  name: 'percy',  age: 21,  [Symbol('symbol 属性')]: 'symbolProp',  unEnumerable: '我是一个不可枚举属性',  skills: ['html', 'css', 'js'],  getSkills: function() {  return this.skills;  }  };  // 设置 unEnumerable 属性为不可枚举属性  Object.defineProperty(obj, 'unEnumerable', {  enumerable: false  }); |   ES6 之后，共有以下 5 种方法可以遍历对象的属性。  // for…in： 遍历对象自身的和继承的可枚举属性（不含 Symbol 类型的属性）  // 不要使用 for…in 来遍历数组，虽然可以遍历，但是如果为 Object.prototype 设置了可枚举属性后，也会把这些属性遍历到，因为数组也是一种对象。  for (let key in obj) {  console.log(key); // name age skills getSkills userProp getUserProp  console.log(obj[key]); // 通过obj.key获取不到 console.log(obj.key);  }  // Object.keys(obj)：返回一个数组，包括对象自身的所有可枚举属性（不含 Symbol 类型的属性）（不含继承的）  Object.keys(obj); // ["name", "age", "skills", "getSkills"]  // Object.getOwnPropertyNames(obj)：返回一个数组，包含对象自身的所有属性（不含 Symbol 类型的属性，不包含继承属性，但是包括不可枚举属性）  Object.getOwnPropertyNames(obj);// ["name", "age", "unEnumerable", "skills", "getSkills"]  // Object.getOwnPropertySymbols(obj)：返回一个数组，包含对象自身的所有 Symbol 类型的属性（不包括继承的属性）  Object.getOwnPropertySymbols(obj);// [Symbol(symbol 属性)]  // Reflect.ownKeys(obj)：返回一个数组，包含对象自身的所有属性（包含 Symbol 类型的属性，还有不可枚举的属性，但是不包括继承的属性）  Reflect.ownKeys(obj);// ["name", "age", "unEnumerable", "skills", "getSkills", Symbol(symbol 属性)]  Reflect.ownKeys({ [Symbol()]:0, b:0, 10:0, 2:0, a:0 })// ['2', '10', 'b', 'a', Symbol()]  上面代码中，Reflect.ownKeys方法返回一个数组，包含了参数对象的所有属性。这个数组的属性次序是这样的，首先是数值属性2和10，其次是字符串属性b和a，最后是 Symbol 属性。   |  |  |  |  |  | | --- | --- | --- | --- | --- | |  | 枚举 | 不可枚举 | symbol | 继承(可枚举) | | for in | + |  |  | + | | keys | + |  |  |  | | getOwnPropertyNames | + | + |  |  | | getOwnPropertySymbols |  |  | + |  | | ownKeys | + | + | + |  |  1. 以上的5种方法遍历对象的属性，都遵守同样的属性遍历的次序规则   1.首先遍历所有属性名为数值的属性，按照数字排序  2.其次遍历所有属性名为字符串的属性，按照生成时间排序  3.最后遍历所有属性名为Symbol值的属性，按照生成时间排序 |
|  |  |
|  |  |
|  |  |

原型链

|  |  |
| --- | --- |
|  | <http://www.jianshu.com/p/15ac7393bc1f>  前端基础进阶（九）：详解面向对象、构造函数、原型与原型链 |
|  | 构造函数  function Test(){}  实例化对象  let test = new Test();  原型对象  Test.prototype  test.\_\_proto\_\_.constructor  test instanceof Test  检测constructor.prototype是否存在于对象的原型链  Test.prototype.constructor  test.\_\_protp\_\_  判断对象是否存在于另一个对象的原型链中  Test.prototype.isPrototypeOf(test)  Test.prototype test.\_\_prototype\_\_ Object.getPrototypeOf(test) |
|  | function A() {}  var a = new A();  console.log(A.prototype == a.\_\_proto\_\_); //true  console.log(A.prototype == Object.getPrototypeOf(a)); //true  console.log(a.constructor == A); //true (A.prototype == a.constructor.prototype); //true  console.log(a.constructor == A.prototype.constructor); //true |
|  | IMG_256  原型对象是构造函数的prototype属性，是所有实例化对象共享属性和方法的原型对象。  实例化对象通过new构造函数得到，都继承了原型对象的属性和方法。  原型对象中有个隐式的constructor，指向了构造函数本身。 |

关于prototype

|  |  |
| --- | --- |
| 三种js方法 | // 对象方法  function People(name) {      this.name = name;      //对象方法      this.Introduce = function () {          console.log("My name is " + this.name);      }  }  //类方法  People.Run = function () {      console.log("I can run");  }  //原型方法  People.prototype.IntroduceChinese = function () {      console.log("我的名字是" + this.name);  }  var p1 = new People("Windking");  p1.Introduce();  People.Run();  // p1.run() p1.run() is not a function  p1.IntroduceChinese(); |
| A.prototype = new B(); | **function** baseClass() {  **this**.showMsg = **function** () {  console.log(**"baseClass::showMsg"**);  } }  **function** extendClass() { }  extendClass.**prototype** = **new** baseClass(); **var instance** = **new** extendClass(); **instance**.showMsg(); // 显示baseClass::showMsg |
| 理解prototype不应把它和继承混淆。A的prototype为B的一个实例，可以理解A将B中的方法和属性全部克隆了一遍。A能使用B的方法和属性。这里强调的是克隆而不是继承。可以出现这种情况：A的prototype是B的实例，同时B的prototype也是A的实例。  我们首先定义了baseClass类，然后我们要定义extentClass，但是我们打算以baseClass的一个实例为原型，来克隆的extendClass也同时包含showMsg这个对象方法。  extendClass.prototype = new baseClass()就可以阅读为：extendClass是以baseClass的一个实例为原型克隆创建的。  那么就会有一个问题，如果extendClass中本身包含有一个与baseClass的方法同名的方法会怎么样？ |
| **function** baseClass() {  **this**.showMsg = **function** () {  console.log(**"baseClass::showMsg"**);  } }  **function** extendClass() {  **this**.showMsg = **function** () {  console.log(**"extendClass::showMsg"**);  } }  extendClass.**prototype** = **new** baseClass(); **var instance** = **new** extendClass();  **instance**.showMsg();//显示extendClass::showMsg  实验证明：函数运行时会先去本体的函数中去找，如果找到则运行，找不到则去prototype中寻找函数。或者可以理解为prototype不会克隆同名函数。 |
| func.call(abj) | 如果我想使用extendClass的一个实例instance调用baseClass的对象方法showMsg怎么办？  答案是可以使用call:  extendClass.prototype = new baseClass(); var instance = new extendClass();  var baseinstance = new baseClass(); baseinstance.showMsg.call(instance);//显示baseClass::showMsg  这里的baseinstance.showMsg.call(instance);阅读为“将instance当做baseinstance来调用，调用它的对象方法showMsg”  好了，这里可能有人会问，为什么不用baseClass.showMsg.call(instance);  这就是对象方法和类方法的区别，我们想调用的是baseClass的对象方法 |

相互关系

|  |  |  |
| --- | --- | --- |
| prototype关系 | <http://www.blogjava.net/heavensay/archive/2013/10/20/405440.html>  Js中Prototype、\_\_proto\_\_、Constructor、Object、Function关系介绍(转) | |
| javascript中的每个对象都有prototype属性，Javascript中对象的prototype属性的解释是：返回对象类型原型的引用 | |
| Function、Object：Js自带的函数对象。  prototype:每一个函数对象都有一个显示的prototype属性,它代表了对象的原型(Function.prototype函数对象是个例外，没有prototype属性)。  \_\_proto\_\_:每个对象都有一个名为\_\_proto\_\_的内部隐藏属性，指向于它所对应的原型对象(chrome、firefox中名称为\_\_proto\_\_，并且可以被访问到)。原型链正是基于\_\_proto\_\_才得以形成(note：不是基于函数对象的属性prototype)。 | |
| function Animal(){  } var anim = new Animal();  console.log('\*\*\*\*\*\*\*\*\*\*\*Animal anim proto\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*');  console.log('typeof Animal:' +typeof Animal); //function console.log('typeof Animal.prototype:' +typeof Animal.prototype); //object console.log('anim.\_\_proto\_\_===Animal.prototype:'+(anim.\_\_proto\_\_===Animal.prototype)); //true console.log('Animal.\_\_proto\_\_===Function.prototype:'+(Animal.\_\_proto\_\_===Function.prototype)); //true console.log('Animal.prototype.\_\_proto\_\_===Object.prototype:'+(Animal.prototype.\_\_proto\_\_===Object.prototype)); //true  console.log('\*\*\*\*\*\*\*\*\*\*\*Function proto\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*'); console.log('typeof Function.prototype:'+typeof Function.prototype); //function console.log('typeof Function.\_\_proto\_\_:'+typeof Function.\_\_proto\_\_); //function console.log('typeof Function.prototype.prototype:'+typeof Function.prototype.prototype); //undefined console.log('typeof Function.prototype.\_\_proto\_\_:'+typeof Function.prototype.\_\_proto\_\_); //object console.log('Function.prototype===Function.\_\_proto\_\_:'+(Function.prototype===Function.\_\_proto\_\_)); //true  console.log('\*\*\*\*\*\*\*\*\*\*\*Object proto\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*'); console.log('typeof Object.prototype:'+typeof Object.prototype); //object console.log('typeof Object.\_\_proto\_\_:'+typeof Object.\_\_proto\_\_); //function console.log('Object.prototype.prototype:'+Object.prototype.prototype); //undefied console.log('Object.prototype.\_\_proto\_\_===null:'+(Object.prototype.\_\_proto\_\_===null)); //null  console.log('\*\*\*\*\*\*\*\*\*\*\*Function Object proto关系\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*'); console.log('Function.prototype===Object.\_\_proto\_\_:'+(Function.prototype===Object.\_\_proto\_\_)); //true console.log('Function.\_\_proto\_\_===Object.\_\_proto\_\_:'+(Function.\_\_proto\_\_===Object.\_\_proto\_\_)); //true console.log('Function.prototype.\_\_proto\_\_===Object.prototype:'+(Function.prototype.\_\_proto\_\_===Object.prototype)); //true  /\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* 系统定义的对象Array、Date \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/ console.log('\*\*\*\*\*\*\*\*\*\*\*\*\*\*test Array、Date\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*'); var array = new Array(); var date = new Date(); console.log('array.\_\_proto\_\_===Array.prototype:'+(array.\_\_proto\_\_===Array.prototype)); //true console.log('Array.\_\_proto\_\_===Function.prototype:'+(Array.\_\_proto\_\_===Function.prototype)); //true console.log('date.\_\_proto\_\_===Date.prototype:'+(date.\_\_proto\_\_===Date.prototype)); //true console.log('Date.\_\_proto\_\_===Function.prototype:'+(Date.\_\_proto\_\_===Function.prototype)); //true console.log('\*\*\*\*\*\*\*\*\*\*\*\*\*\*constructor\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*');  console.log('anim.constructor===Animal:'+(anim.constructor===Animal)) ; //true console.log('Animal===Animal.prototype.constructor:'+(Animal===Animal.prototype.constructor)) ; //true console.log('Animal.constructor===Function.prototype.constructor:'+(Animal.constructor===Function.prototype.constructor)); //true console.log('Function.prototype.constructor===Function:'+(Function.prototype.constructor===Function)); //true console.log('Function.constructor===Function.prototype.constructor:'+(Function.constructor===Function.prototype.constructor)); //true  console.log('Object.prototype.constructor===Object:'+(Object.prototype.constructor===Object)); //true console.log('Object.constructor====Function:'+(Object.constructor===Function)); //true | |
|  | |  | |
| 继承 | | [前端开发必须知道的JS（一） 原型和继承](http://www.cnblogs.com/ljchow/archive/2010/06/08/1753526.html)  <http://www.cnblogs.com/ljchow/archive/2010/06/08/1753526.html>  Angular系列学习笔记（一）—— 聊聊angular的模块化  <http://www.jianshu.com/p/0fc26c75f089>  一. 原型与构造函数  ### 原型  Js所有的函数都有一个prototype属性，这个属性引用了一个对象，即原型对象，也简称原型。这个函数包括构造函数和普通函数，我们讲的更多是构造函数的原型，但是也不能否定普通函数也有原型。譬如普通函数：  function F(){}  console.log(F.prototype instanceof Object) //true  ### 构造函数  构造函数，也即构造对象。首先了解下通过构造函数实例化对象的过程。  function A(x){  　　this.x=x;  }  var obj=new A(1);   1. 实例化obj对象有三步：   1. 创建obj对象：obj=new Object();  2. 将obj的内部\_\_proto\_\_指向构造他的函数A的prototype，同时，obj.constructor===A.prototype.constructor，  (这个是永远成立的；即使A.prototype不再指向原来的A原型，也就是说：类的实例对象的constructor属性永远指向"构造函数"的prototype.constructor)  从而使得obj.constructor.prototype指向A.prototype（obj.constructor.prototype===A.prototype，当A.prototype改变时则不成立，下文有遇到）。  obj.constructor.prototype与的内部\_proto\_是两码事，实例化对象时用的是\_\_proto\_\_，obj是没有prototype属性的，但是有内部的\_\_proto\_\_，通过\_\_proto\_\_来取得原型链上的原型属性和原型方法，FireFox公开了\_\_proto\_\_，可以在FireFox中alert（obj.\_\_proto\_\_）；  3. 将obj作为this去调用构造函数A，从而设置成员（即对象属性和对象方法）并初始化。  当这3步完成，这个obj对象就与构造函数A再无联系，这个时候即使构造函数A再加任何成员，都不再影响已经实例化的obj对象了。  此时，obj对象具有了x属性，同时具有了构造函数A的原型对象的所有成员，当然，此时该原型对象是没有成员的。原型对象初始是空的，也就是没有一个成员（即原型属性和原型方法）。  但是，一旦定义了原型属性或原型方法，则所有通过该构造函数实例化出来的所有对象，都继承了这些原型属性和原型方法，这是通过内部的\_proto\_链来实现的。  譬如  A.prototype.say=function(){console.log("Hi")};  那所有的A的对象都具有了say方法，这个原型对象的say方法**是唯一的副本给大家共享**的，而不是每一个对象都有关于say方法的一个副本。 | |
|  | |  | |
|  | | 原型参与实现的完美继承。（本文核心在此^\_^）  1 function A(x){  2 　　this.x = x;  3 }  4 A.prototype.a = "a";  5 function B(x,y){  6 　　this.y = y;  7 　　A.call(this,x);  8 }  9 B.prototype.b1 = function(){ 10 　　alert("b1"); 11 } 12 B.prototype = new A(); 13 B.prototype.b2 = function(){ 14 　　alert("b2"); 15 } 16 B.prototype.constructor = B; 17 var obj = new B(1,3); | |
|  | | 这个例子讲的就是B继承A。第7行类继承：A.call(this,x);上面已讲过。实现原型继承的是第12行：B.prototype = new A();  就是说把B的原型指向了A的1个实例对象，这个实例对象具有x属性，为undefined，还具有a属性，值为"a"。所以B原型也具有了这2个属性（或者说，B和A建立了原型链，B是A的下级）。  而因为方才的类继承，B的实例对象也具有了x属性，也就是说obj对象有2个同名的x属性，此时原型属性x要让位于实例对象属性x，所以obj.x是1，而非undefined。第13行又定义了原型方法b2，所以B原型也具有了b2。虽然第9~11行设置了原型方法b1，但是你会发现第12行执行后，B原型不再具有b1方法，也就是obj.b1是undefined。因为第12行使得B原型指向改变，原来具有b1的原型对象被抛弃，自然就没有b1了。  第12行执行完后，B原型（B.prototype）指向了A的实例对象，而A的实例对象的构造器是构造函数A，所以B.prototype.constructor就是构造对象A了（换句话说，A构造了B的原型）。  alert(B.prototype.constructor)出来后就是"function A(x){...}" 。同样地，obj.constructor也是A构造对象，alert(obj.constructor)出来后就是"function A(x){...}" ，也就是说B.prototype.constructor===obj.constructor（true），但是B.prototype===obj.constructor.prototype（false），因为前者是B的原型，具有成员：x,a,b2，后者是A的原型，具有成员：a。如何修正这个问题呢，就在第16行，将B原型的构造器重新指向了B构造函数，那么B.prototype===obj.constructor.prototype（true），都具有成员：x,a,b2。  如果没有第16行，那是不是obj = new B(1,3)会去调用A构造函数实例化呢？答案是否定的，你会发现obj.y=3，所以仍然是调用的B构造函数实例化的。虽然obj.constructor===A(true)，但是对于new B()的行为来说，执行了上面所说的通过构造函数创建实例对象的3个步骤，第一步，创建空对象；第二步，obj.\_\_proto\_\_ === B.prototype，B.prototype是具有x,a,b2成员的，obj.constructor指向了B.prototype.constructor，即构造函数A；第三步，调用的构造函数B去设置和初始化成员，具有了属性x,y。虽然不加16行不影响obj的属性，但如上一段说，却影响obj.constructor和obj.constructor.prototype。所以在使用了原型继承后，要进行修正的操作。  关于第12、16行，总言之，第12行使得B原型继承了A的原型对象的所有成员，但是也使得B的实例对象的构造器的原型指向了A原型，所以要通过第16行修正这个缺陷。 | |
|  | | function A(x) {      this.x = x;  }  A.prototype.a = "a";  function B(x, y) {      this.y = y;      A.call(this, x);//类继承  }  B.prototype = new A();//原型继承  B.prototype.b2 = function () {      console.log("b2");  }  B.prototype.constructor = B;  var obj = new B(1, 3);  console.log(obj.x);  console.log(obj.a);  console.log(obj.y);  console.log(obj.b2); | |

双向绑定（vue）

|  |  |
| --- | --- |
|  | JavaScript 进阶之深入理解数据双向绑定(主要讲了vue)  <http://gitbook.cn/books/593faaf7d3845323661a4cec/index.html>  Object.defineProperty()方法可以更方便的监听一个对象的变化。当我们的视图和数据任何一方发生变化的时候，我们希望能够通知对方也更新，这就是所谓的数据双向绑定。  Object.getOwnPropertyDescriptor() 返回指定对象上一个自有属性对应的属性描述符。 |
|  | Vue源码分析  本次我们主要分析一下Vue 数据绑定的源码，这里我直接将 [Vue.js 1.0.28](https://github.com/vuejs/vue/blob/v1.0.28/src/observer/index.js) 版本的代码稍作删减拿过来进行，2.x 的代码基于 flow 静态类型检查器书写的，代码除了编码风格在整体结构上基本没有太大改动，所以依然基于 1.x 进行分析，对于存在差异的部分加以说明。  IMG_256  监听对象变动  // 观察者构造函数function Observer (value) {  this.value = value  this.walk(value)  }  // 递归调用，为对象绑定getter/setter  Observer.prototype.walk = function (obj) {  var keys = Object.keys(obj)  for (var i = 0, l = keys.length; i < l; i++) {  this.convert(keys[i], obj[keys[i]])  }  }  // 将属性转换为getter/setter  Observer.prototype.convert = function (key, val) {  defineReactive(this.value, key, val)  }  // 创建数据观察者实例function observe (value) {  // 当值不存在或者不是对象类型时，不需要继续深入监听  if (!value || typeof value !== 'object') {  return  }  return new Observer(value)  }  // 定义对象属性的getter/setterfunction defineReactive (obj, key, val) {  var property = Object.getOwnPropertyDescriptor(obj, key)  if (property && property.configurable === false) {  return  }  // 保存对象属性预先定义的getter/setter  var getter = property && property.get  var setter = property && property.set  var childOb = observe(val)  Object.defineProperty(obj, key, {  enumerable: true,  configurable: true,  get: function reactiveGetter () {  var value = getter ? getter.call(obj) : val  console.log("访问："+key)  return value  },  set: function reactiveSetter (newVal) {  var value = getter ? getter.call(obj) : val  if (newVal === value) {  return  }  if (setter) {  setter.call(obj, newVal)  } else {  val = newVal  }  // 对新值进行监听  childOb = observe(newVal)  console.log('更新：' + key + ' = ' + newVal)  }  })  }  定义一个对象作为数据模型，并监听这个对象。  let data = {  user: {  name: 'zhaomenghuan',  age: '24'  },  address: {  city: 'beijing'  }  }  observe(data)  console.log(data.user.name) // 访问：user // 访问：name  data.user.name = 'ZHAO MENGHUAN'// 访问：user// 更新：name = ZHAO MENGHUAN  效果如下： |

es6 - class

|  |  |
| --- | --- |
|  | class Point {  constructor(x, y) {  this.x = y;  this.y = y;  }  getMsg() {  // 公共方法  var msg = `name: ${this.x} , age: ${this.y}`;  return msg;  }  static staticFunc() {  // 静态方法 Point.staticFunc  }  \_privateFunc() {  // 私有方法  }  get a() {  return "a";  }  set a(value) {  console.log("change")  }  }  var p = new Point("100", "200");  // Class 可以通过extends关键字实现继承，这比 ES5 的通过修改原型链实现继承，要清晰和方便很多。  class ColorPoint extends Point {  constructor(x, y, color) {  super(x, y); // 调用父类的constructor(x, y)  this.color = color;  }  getMsgWithColor() {  var msg = this.color + ' ' + super.getMsg();  return msg; // 调用父类的toString()  }  }  var cp = new ColorPoint(100, 200, "yellow");  console.log(cp.getMsgWithColor()); |
|  | 1. 类   1.类的数据类型就是函数，类本身就指向构造函数。  p是Point类的实例，它的constructor方法就是Point类原型的constructor方法。  prototype对象的constructor属性，直接指向“类”的本身，这与 ES5 的行为是一致的。  typeof Point // "function"  Point === Point.prototype.constructor // true  // p.constructor <=> Point.prototype.constructor <=> Point  2.类必须使用new调用，否则会报错。这是它跟普通构造函数的一个主要区别，后者不用new也可以执行。   1. 类的方法   1.在类的实例上面调用方法，其实就是调用原型上的方法。  Point.pertotype={say:function(){}}  2.由于类的方法都定义在prototype对象上面，所以类的新方法可以添加在prototype对象上面。Object.assign方法可以很方便地一次向类添加多个方法。  Object.assign(Point.prototype, {  toString(){},  toValue(){}  });  3.类的内部所有定义的方法[say]，都是不可枚举的（non-enumerable）。这一点与 ES5 的行为不一致。  console.log(Object.keys(p));//[ 'name', 'age' ]  console.log(Object.getOwnPropertyNames(p));//[ 'name', 'age' ]  console.log(Object.keys(Point.prototype))//[]  console.log(Object.getOwnPropertyNames(Point.prototype))//[ 'constructor', 'say' ] |
|  | 实例对象的属性除非显示定义在其本身（this对象），否则都是定义在原型上（prototype）  class不存在变量提升，“使用在前，定义在后”的情况会报错  class不存在私有方法；  通过特殊标识实现（在方法名前加下划线）；  将私有方法移出模块，在内部使用funout.call(this,a)；  利用Symbol的唯一性；   1. this默认指向类的实例，其他情况：   const logger = new Logger();  const { printName } = logger;  printName(); // TypeError: Cannot read property 'print' of undefined  1.在构造方法中绑定this，这样就不会找不到print方法  class Logger {  constructor() {  this.printName = this.printName.bind(this);  }  // ...  }  2.使用箭头函数  class Logger {  constructor() {  this.printName = (name = 'there') => {  this.print(`Hello ${name}`);  };  }  // ...  }     1. get set   与 ES5 一样，在“类”的内部可以使用get和set关键字，对某个属性设置存值函数和取值函数，拦截该属性的存取行为。  class CustomHTMLElement {  constructor(element) {  this.element = element;  }  get html() {  return this.element.innerHTML;  }  set html(value) {  this.element.innerHTML = value;  }  }  var descriptor = Object.getOwnPropertyDescriptor(  CustomHTMLElement.prototype, "html"  );  "get" in descriptor // true  "set" in descriptor // true   1. 静态方法   类相当于实例的原型，所有在类中定义的方法，都会被实例继承。如果在一个方法前，加上static关键字，就表示该方法不会被实例继承，而是直接通过类来调用，这就称为“静态方法”。  静态方法，可以直接在Foo类上调用（Foo.classMethod()），而不是在Foo类的实例上调用。如果在实例上调用静态方法，会抛出一个错误，表示不存在该方法。  注意，  1.如果静态方法包含this关键字，这个this指的是类，而不是实例。  2.静态方法可以与非静态方法重名。  3.父类的静态方法，可以被子类继承。   1. 静态属性   静态属性指的是 Class 本身的属性，即Class.propName，而不是定义在实例对象（this）上的属性。  因为 ES6 明确规定，Class 内部只有静态方法，没有静态属性。  唯一有效写法：  class Foo {  }  Foo.prop = 1;  Foo.prop // 1   1. new.target   ES6 为new命令引入了一个new.target属性，该属性一般用在构造函数之中，返回new命令作用于的那个构造函数。如果构造函数不是通过new命令调用的，new.target会返回undefined，因此这个属性可以用来确定构造函数是怎么调用的。   1. Point.name // "Point" |
| extend | 1. super关键字，可以作为函数、对象使用   \* constructor方法和toString方法之中，都出现了super关键字，它在这里表示父类的构造函数，用来新建父类的this对象。  \* 作为函数使用时，super()代表父类的构造函数，子类构造函数必须执行一次super函数；  ---super虽然代表父类的构造函数，但是返回的是子类的实例，因此super内部的this指的的子类，因此super()相当于A.prototype.constructor.call(this);  ---子类调用super()，父类中new.target.name为子类name  ---作为函数使用，只能在子类的构造函数中使用  \* 作为对象使用时，  ---在普通方法中指向父类的原型对象，可以通过super.func()执行父类的普通方法  ---super指向父类的原型对象，所以定义在父类实例上的方法或属性，无法通过super获取父类constructor中定义在this上的属性（this.name）;只能获取A.prototype上的方法，即父类中定义的简单方法；  ---通过super.x=3;进行赋值，可通过this.xx获取；赋值的属性会变成子类实例的属性；super.x赋值为3，这时等同于对this.x赋值为3。而当读取super.x的时候，读的是A.prototype.x，所以返回undefined。  ---在静态方法中指向父类；  \* 在使用的过程中必须显示指定是作为函数、还是作为对象使用，否则会报错。 console.log(super); // 报错   1. constructor   constructor方法是类的默认方法，通过new命令生成对象实例时，自动调用该方法。一个类必须有constructor方法，如果没有显式定义，一个空的constructor方法会被默认添加。若没有书写，javascript引擎会自动添加一个空的constructor方法；  子类默认拥有constructor方法，且必须在子类中调用super方法，否则在新建实例的时候会报错；  因为子类没有自己的this对象，而是继承父类的this对象，然后进行加工。如果不调用super方法，子类就得不到this对象。  注意：this关键字要在super调用后使用；这是因为子类实例的构建，是基于对父类实例加工，只有super方法才能返回父类实例。   1. 类的 prototype 属性和\_\_proto\_\_属性   \* 表现  B.\_\_proto\_\_ === A // true  B.prototype.\_\_proto\_\_ === A.prototype // true  大多数浏览器的 ES5 实现之中，每一个对象都有\_\_proto\_\_属性，指向对应的构造函数的prototype属性。Class 作为构造函数的语法糖，同时有prototype属性和\_\_proto\_\_属性，因此同时存在两条继承链。  （1）子类的\_\_proto\_\_属性，表示构造函数的继承，总是指向父类。  （2）子类prototype属性的\_\_proto\_\_属性，表示方法的继承，总是指向父类的prototype属性。  这两条继承链，可以这样理解：作为一个对象，子类（B）的原型（\_\_proto\_\_属性）是父类（A）；作为一个构造函数，子类（B）的原型对象（prototype属性）是父类的原型对象（prototype属性）的实例。  \* 原因  Object.setPrototypeOf(B.prototype, A.prototype);// B 的实例继承 A 的实例  Object.setPrototypeOf(B, A);// B 的实例继承 A 的静态属性  const b = new B();  Object.setPrototypeOf = function (obj, proto) {  obj.\_\_proto\_\_ = proto;  return obj;  }  实例对象cp同时是ColorPoint和Point两个类的实例，这与 ES5 的行为完全一致。  子类实例的\_\_proto\_\_属性的\_\_proto\_\_属性，指向父类实例的\_\_proto\_\_属性。也就是说，子类的原型的原型，是父类的原型。  var p1 = new Point(2, 3);  var p2 = new ColorPoint(2, 3, 'red');  p2.\_\_proto\_\_ === p1.\_\_proto\_\_ // false  p2.\_\_proto\_\_.\_\_proto\_\_ === p1.\_\_proto\_\_ // true |
|  |  |
|  |  |
|  |  |

****其他****

var

|  |  |
| --- | --- |
|  | 【var声明的全局变量与隐式全局变量】  通过var创建的全局变量（任何函数之外的程序中创建）是不能被删除的。  无var创建的隐式全局变量（无视是否在函数中创建）是能被删除的。  在技术上，属性是可以通过delete操作符删除的，而变量是不能的。这表明，隐式全局变量并不是真正的全局变量，而是全局对象的属性。  在ES5严格模式下，未声明的变量（如在前面的代码片段中的两个反面教材）工作时会抛出一个错误。 |
|  | // 定义三个全局变量  **var global\_var** = 1;  **global\_novar** = 2; // 反面教材  (**function** () {  **global\_fromfunc** = 3; // 反面教材  }());   // 试图删除  **delete global\_var**; // false (删除不了（对象与值）)  **delete global\_novar**; // true （将其直接删除（对象与值））  **delete global\_fromfunc**; // true （将其直接删除（对象与值））   //测试其值  **console**.log(**global\_var**) //1  // console.log(global\_novar) //报错，not defined  // console.log(global\_fromfunc) //报错，not defined  // 测试该删除  **console**.log(**typeof global\_var**); // "number"  **console**.log(**typeof global\_novar**); // "undefined"  **console**.log(**typeof global\_fromfunc**); // "undefined" |
|  | console.log(aaa);aaa=1;    console.log(aaa);var aaa=1; |
|  | 在函数的顶部使用单var声明的变量，好处如下：   1. 所有的变量好找 2. 少代码，单线   用法：   1. 使用一个var来声明多个变量，并以逗号分隔 2. 可以在声明的过程中作一些有意义的事情（所有声明未初始化的值都是”undefined”）   **function** func() {  **var a** = 1,  **b** = 2,  **sum** = **a** + **b**,  **myobject** = {},  **i**,  **J**,  **el** = **document**.getElementById(**"result"**),  **style** = **el**.**style**;  // 使用el和style干点其他什么事...  // function body...  } |

关于预解析、全局与局部的表现

|  |  |
| --- | --- |
|  | 执行层上代码处理分两个阶段，第一阶段是变量，函数声明，以及正常格式的参数创建，这是一个解析和进入上下文 的阶段。第二个阶段是代码执行，函数表达式和不合格的标识符（为声明的变量）被创建。 |
|  | // 反例 **var myname** = **"global"**; // 全局变量 **function** func() {  console.log(**myname**); // "undefined"  **var myname** = **"local"**;  console.log(**myname**); // "local" } func(); |
|  | // 反例 **var myname** = **"global"**; // 全局变量 **function** func() {  console.log(**myname**); // "global" } func(); |
|  | // 反例  myname = **"global"**; // global variable **function** func() {  **var myname**; // 等同于 -> var myname = undefined;  console.log(myname); // "undefined"  myname = **"local"**;  console.log(myname); // "local"  } func(); |

全局变量

|  |  |  |
| --- | --- | --- |
|  | 1. 什么是全局变量   在浏览器中，全局对象可以通过window属性在代码的任何位置访问（除非你做了些比较出格的事情，像是声明了一个名为window的局部变量）。但是在其他环境下，这个方便的属性可能被叫做其他什么东西（甚至在程序中不可用）。如果你需要在没有硬编码的window标识符下访问全局对象，你可以在任何层级的函数作用域中做如下操作：  var global = (function () {  return this;  }());  这种方法可以随时获得全局对象，因为其在函数中被当做函数调用了（不是通过new构造），this总是指向全局对象。实际上这个病不适用于ECMAScript 5严格模式，所以，在严格模式下时，你必须采取不同的形式。例如，你正在开发一个JavaScript库，你可以将你的代码包裹在一个即时函数中，然后从全局作用域中，传递一个引用指向this作为你即时函数的参数。   1. 谨慎使用全局变量   全局变量的问题在于，你的JavaScript应用程序和web页面上的所有代码都共享了这些全局变量，他们住在同一个全局命名空间，所以当程序的两个不同部分定义同名但不同作用的全局变量的时候，命名冲突在所难免。  web页面包含不是该页面开发者所写的代码也是比较常见的，例如：  第三方的JavaScript库  广告方的脚本代码  第三方用户跟踪和分析脚本代码  不同类型的小组件，标志和按钮  比方说，该第三方脚本定义了一个全局变量，叫做result；接着，在你的函数中也定义一个名为result的全局变量。其结果就是后面的变量覆盖前面的，第三方脚本就一下子嗝屁啦！  因此，要想和其他脚本成为好邻居的话，尽可能少的使用全局变量是很重要的。一些减少全局变量的策略，例如命名空间模式或是函数立即自动执行，但是要想让全局变量少最重要的还是始终使用var来声明变量。  由于JavaScript的两个特征，不自觉地创建出全局变量是出乎意料的容易。第一，你可以甚至不需要声明就可以使用变量；第二，JavaScript有隐含的全局概念，意味着你不声明的任何变量都会成为一个全局对象属性。  另外一个避免全局变量的原因是可移植性 | |
|  | // 建立和显示全局变量的方法 **myglobal** = **"nowamagic"**; // 不推荐写法 **console**.log(**myglobal**); // "hello" **console**.log(**window**.**myglobal**); // "hello" **console**.log(**window**[**"myglobal"**]); // "hello" **console**.log(**this**.**myglobal**); // "hello" | |
|  | function sum(x, y) {  // 不推荐写法: 隐式全局变量  result = x + y;  return result;  } | function sum(x, y) {  var result = x + y;  return result;  }  （推荐的写法） |
|  | // 反例，勿使用  function foo() {  var a = b = 0;  // ...  }  （a是本地变量）  （b是全局变量）  （相当于：**var** a = (b = 0)） | function foo() {  var a, b;  // ... a = b = 0; // 两个均局部变量  } |
|  | 函数声明和函数表达式的区别  <http://web.jobbole.com/87534/?utm_source=hao.caibaojian.com> | |

JS引擎

|  |  |
| --- | --- |
|  | 1. 什么是JS解析引擎   JavaScript解析引擎就是能够“读懂”JavaScript代码，并准确地给出代码运行结果的一段程序。比方说，当你写了 var a = 1 + 1; 这样一段代码，JavaScript引擎做的事情就是看懂（解析）你这段代码，并且将a的值变为2。  学过编译原理的人都知道，对于静态语言来说（如Java、C++、C），处理上述这些事情的叫编译器（Compiler），相应地对于JavaScript这样的动态语言则叫解释器（Interpreter）。这两者的区别用一句话来概括就是：编译器是将源代码编译为另外一种代码（比如机器码，或者字节码），而解释器是直接解析并将代码运行结果输出。 比方说，firebug的console就是一个JavaScript的解释器。  但是，现在很难去界定说，JavaScript引擎它到底算是个解释器还是个编译器，因为，比如像V8（Chrome的JS引擎），它其实为了提高JS的运行性能，在运行之前会先将JS编译为本地的机器码（native machine code），然后再去执行机器码（这样速度就快很多），相信大家对JIT（Just In Time Compilation）一定不陌生吧  要强调的就是，JavaScript引擎本身也是程序，代码编写而成。比如V8就是用C/C++写的。 |
|  | 1. JavaScript解析引擎与ECMAScript是什么关系？   JavaScript引擎需要根据一些标准去解析JavaScript代码，而ECMAScript就是定义了这些标准、规则。其中ECMAScript 262这份文档，就是对JavaScript这门语言定义了一整套完整的标准。其中包括：  var，if，else，break，continue等是JavaScript的关键词  abstract，int，long等是JavaScript保留词  怎么样算是数字、怎么样算是字符串等等  定义了操作符（+，-，>，<等）  定义了JavaScript的语法  定义了对表达式，语句等标准的处理算法，比如遇到==该如何处理  ⋯⋯  标准的JavaScript引擎就会根据这套文档去实现，注意这里强调了标准，因为也有不按照标准来实现的，比如IE的JS引擎。这也是为什么JavaScript会有兼容性的问题。至于为什么IE的JS引擎不按照标准来实现，就要说到浏览器大战了，这里就不赘述了，自行Google之。  所以，简单的说，ECMAScript定义了语言的标准，JavaScript引擎根据它来实现，这就是两者的关系。 |
|  | 1. JavaScript解析引擎与浏览器又是什么关系？   简单地说，JavaScript引擎是浏览器的组成部分之一。因为浏览器还要做很多别的事情，比如解析页面、渲染页面、Cookie管理、历史记录等等。那么，既然是组成部分，因此一般情况下JavaScript引擎都是浏览器开发商自行开发的。比如：IE9的Chakra、Firefox的TraceMonkey、Chrome的V8等等。  从而也看出，不同浏览器都采用了不同的JavaScript引擎。因此，我们只能说要深入了解哪个JavaScript引擎。 |
|  | 1. 深入了解其内部原理的途径有哪些？   搞清楚了前面三个问题，那这个问题就好回答了。个人认为，主要途径有如下几种（依次由浅入深）：  看讲JavaScript引擎工作原理的书  这种方式最方便，不过我个人了解到的这样的书几乎没有，但是[Dmitry A.Soshnikov](http://dmitrysoshnikov.com/" \t "http://www.nowamagic.net/librarys/veda/detail/_blank)博客上的文章真的是非常的赞，建议直接看英文，实在英文看起来吃力的，可以看[译本](http://blog.goddyzhao.me/JavaScript-Internal" \t "http://www.nowamagic.net/librarys/veda/detail/_blank)  看ECMAScript的标准文档 这种方式相对直接，原汁原味，因为引擎就是根据标准来实现的。目前来说，可以看[第五版](http://www.ecma-international.org/publications/files/ECMA-ST/Ecma-262.pdf" \t "http://www.nowamagic.net/librarys/veda/detail/_blank)和[第三版](http://www.ecma-international.org/publications/files/ECMA-ST-ARCH/ECMA-262, 3rd edition, December 1999.pdf" \t "http://www.nowamagic.net/librarys/veda/detail/_blank)，不过要看懂也是不容易的。  看JS引擎源代码  这种方式最直接，当然也最难了。因为还牵涉到了如何实现词法分析器，语法分析器等等更加底层的东西了，而且并非所有的引擎代码都是开源的。 |
|  | 1. 以上几种方式中第一种都很难看明白怎么办？   其实第一种方式中的文章，作者已经将文档中内容提炼出来，用通俗易懂的方式阐述出来了。如果，看起来还觉得吃力，那说明还缺少两块的东西：   1. 对JavaScript本身还理解的不够深入   如果你刚刚接触JavaScript，或者说以前甚至都没有接触过。那一下子就想要去理解内部工作原理，的确是很吃力的。首先应该多看看书，多实践实践，从知识和实践的方式来了解JavaScript预言特性。这种情况下，你只需要了解现象。比方说，(function(){})() 这样可以直接调用该匿名函数、用闭包可以解决循环中的延迟操作的变量值获取问题等等。要了解这些，都是需要多汲取和实践的。实践这里就不多说了，而知识汲取方面可以多看看书和博客。这个层面的书就相对比较多了，[《Professional JavaScript for Web Developers》](http://www.amazon.com/Professional-JavaScript-Developers-Nicholas-Zakas/dp/1118026691/" \t "http://www.nowamagic.net/librarys/veda/detail/_blank)就是本很好的书（中文版请自行寻找）。   1. 缺乏相应的领域知识   当JavaScript也达到一定深度了，但是，还是看不大明白，或者没法很深入到内部去一探究竟。那就意味着缺少对应的领域知识。这里明显的就是编译原理相关的知识。不过，其实对这块了解个大概基本看起来就没问题了。要再继续深入，那需要对编译原理了解的很深入，比如说词法分析采用什么算法，一般怎么处理。会有什么问题，如何解决，AST生成算法一般有哪几种等等。那要看编译原理方面的书，也有基本经典的书，比如[《Compilers: Principles, Techniques, and Tools》](http://www.amazon.com/Compilers-Principles-Techniques-Tools-2nd/dp/0321486811/" \t "http://www.nowamagic.net/librarys/veda/detail/_blank)这本也是传说中的龙书，还有非常著名的[《SICP》](http://mitpress.mit.edu/sicp/full-text/book/book.html" \t "http://www.nowamagic.net/librarys/veda/detail/_blank)和[《PLAI》](http://www.cs.brown.edu/~sk/Publications/Books/ProgLangs/" \t "http://www.nowamagic.net/librarys/veda/detail/_blank)。不过其实根据个人经验，对于Dmitry的文章，要看懂它，只要你对JavaScript有一定深度的了解，同时你大学计算机的课程都能大致掌握了（尤其是操作系统），也就是说基础不错，理解起来应该没问题。因为这些文章基本没有涉及底层编译相关的，只是在解释文档的内容，并且其中很多东西都是相通的，比如：context的切换与CPU的进程切换、函数相关的的局部变量的栈存储、函数退出的操作等等都是一致的。  以上就是个人对这个问题的看法，除此之外，我觉得，学习任何技术都不能操之过急，要把基础打扎实了，这样学什么都会很快。 |

闭包

|  |  |  |
| --- | --- | --- |
|  | [http://www.csstop.com//jsclosure/2016-0919-4952.html](http://www.csstop.com/jsclosure/2016-0919-4952.html)  <https://segmentfault.com/a/1190000000652891> | |
| 闭包的条件 | 1.调用的函数是父级作用域内部声明的 2.调用的函数是在父级作用域之外进行调用 3.调用的函数内部使用了父级作用域的内部变量  ++内部申明的函数再外部进行调用，并使用了内部的变量  1.函数嵌套函数  2.函数内部可以引用外部的参数和变量  3.参数和变量不会被垃圾回收机制回收 | |
|  | 在动态执行环境中，数据实时地发生变化，为了保持这些非持久型变量的值，我们用闭包这种载体来存储这些动态数据。这就是闭包的作用。也就说遇到需要存储动态变化的数据或将被回收的数据时，我们可以通过外面再包裹一层函数形成闭包来解决。  当然，闭包会导致很多外部函数的调用对象不能释放，滥用闭包会使得内存泄露，所以在频繁生成闭包的情景下我们要估计下他带来的副作用。 | |
|  | 闭包的定义及其优缺点  闭包 是指有权访问另一个函数作用域中的变量的函数，创建闭包的最常见的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  闭包的缺点就是常驻内存，会增大内存使用量，使用不当很容易造成内存泄露。  闭包是javascript语言的一大特点，主要应用闭包场合主要是为了：**设计私有的方法和变量**。  一般函数执行完毕后，局部活动对象就被销毁，内存中仅仅保存全局作用域。但闭包的情况不同！ | |
| 闭包的理论 | 执行环境  每调用一个函数时（执行函数时），系统会为该函数创建一个封闭的局部的运行环境，即该函数的执行环境。函数总是在自己的执行环境中执行，如读写局部变量、函数参数、运行内部逻辑。创建执行环境的过程包含了创建函数的作用域，函数也是在自己的作用域下执行的。从另一个角度说，每个函数执行环境都有一个作用域链，子函数的作用域链包括它的父函数的作用域链。关于作用域、作用域链请看下面。    作用域、作用域链、调用对象  函数作用域分为词法作用域和动态作用域。  词法作用域是函数定义时的作用域，即静态作用域。当一个函数定义时，他的词法作用域就确定了，词法作用域说明的是在函数结构的嵌套关系下，函数作用的范围。这个时候也就形成了该函数的作用域链。作用域链就是把这些具有嵌套层级关系的作用域串联起来。函数的内部[[scope]]属性指向了该作用域链。  动态作用域是函数调用执行时的作用域。当一个函数被调用时，首先将函数内部[[scope]]属性指向了函数的作用域链，然后会创建一个调用对象，并用该调用对象记录函数参数和函数的局部变量，将其置于作用域链顶部。动态作用域就是通过把该调用对象加到作用域链的顶部来创建的，此时的[[scope]]除了具有定义时的作用域链，还具有了调用时创建的调用对象。换句话说，执行环境下的作用域等于该函数定义时就确定的作用域链加上该函数刚刚创建的调用对象，从而也形成了新的作用域链。所以说是动态的作用域，并且作用域链也随之发生了变化。再看这里的作用域，其实是一个对象链，这些对象就是函数调用时创建的调用对象，以及他上面一层层的调用对象直到最上层的全局对象。  譬如全局环境下的函数A内嵌套了一个函数B，则该函数B的作用域链就是：函数B的作用域—>函数A的作用域—>全局window的作用域。当函数B调用时，寻找某标识符，会按函数B的作用域—>函数A的作用域—>全局window的作用域去寻找，实际上是按函数B的调用对象—>函数A的调用对象—>全局对象这个顺序去寻找的。也就是说当函数调用时，函数的作用域链实际上是调用对象链。    闭包  在动态执行环境中，数据实时地发生变化，为了保持这些非持久型变量的值，我们用闭包这种载体来存储这些动态数据（看完下面的应用就会很好的体会这句话）。闭包的定义：所谓“闭包”，指的是一个拥有许多变量和绑定了这些变量的环境的表达式（通常是一个函数），因而这些变量也是该表达式的一部分。  闭包就是嵌套在函数里面的内部函数，并且该内部函数可以访问外部函数中声明的所有局部变量、参数和其他内部函数。当该内部函数在外部函数外被调用，就生成了闭包。（实际上任何函数都是全局作用域的内部函数，都能访问全局变量，所以都是window的闭包）  垃圾回收机制：如果某个对象不再被引用，该对象将被回收。  再结合前面所讲的一些概念，在执行var test=f(1)时创建了f的调用对象，这里暂且记作obj，执行完后虽然退出了外部执行环境，但内部函数inner被外部函数f外面的一个变量test引用。由于外部函数创建的调用对象obj有一个属性指向此内部函数，而现在这个内部函数又被引用，所以调用对象obj会继续存在，不会被垃圾回收器回收，其函数参数x和局部变量a都会在这个调用对象中得以维持。虽然调用对象不能被直接访问，但是该调用对象已成为内部函数作用域链中的一部分，可以被内部函数访问并修改，所以执行test()时，可以正确访问x和a。所以说， 当执行了外部函数时，生成了闭包，被引用的外部函数的变量将继续存在。 | |
|  | 嵌套函数的闭包  function aaa() {  var a = 1;  return function () {  alert(a++)  };  }  var fun = aaa();  fun(); // 1 执行后 a++，，然后a还在~  fun(); // 2  fun = null; //a被回收！！  闭包会使变量始终保存在内存中，如果不当使用会增大内存消耗。  javascript的垃圾回收原理  （1）、在javascript中，如果一个对象不再被引用，那么这个对象就会被GC回收；  （2）、如果两个对象互相引用，而不再被第3者所引用，那么这两个互相引用的对象也会被回收。 | |
| 作用 | 私有成员的存在  模块化代码，减少全局变量的污染  使用匿名函数实现累加  在循环中直接找到对应元素的索引 | |
| demo1-局部变量的累加 | **var outerFun** = **null**; (**function** () {  **var one** = 0;  **outerFun** = **function** () {  **one** += 1;  log(**one**); // 1 2 3   **return one**;  }; })();  log(**outerFun**()); //1 log(**outerFun**()); //2 log(**outerFun**()); //3  outerFun是在外部进行调用的  /\*这段代码中的变量one是一个局部变量（因为它被定义在一个函数之内），因此外部是不可以访问的。但是这里我们创建了inner函数，inner函数是可以访问变量one的；又将全局变量outer引用了inner，所以三次调用outer会弹出递增的结果。\*/ | |
|  | 在js中我们可以通过关键字function来声明一个函数，并进行调用  function abc(){  alert(123);  }  abc();  我们也可以通过一个"()"来将这个声明变成一个表达式：  (function (){  alert(123);  })();  //然后通过()直接调用前面的表达式即可，因此函数可以不必写名字 | |
| demo2 | var fun1; var fun2 = function() {         var x = 0;         fun1 = function() {                 console.log(x);         } } fun2(); fun1();  其中fun1是个闭包，条件都满足。有疑问的地方可能是，是否内部声明的呢？是的，因为fun1是个函数，是fun2的内部函数 | var fun2 = function() {  var btn = dom("myBtn");         var x;         btn.onclick = function() {                 console.log(x);         } } fun2();  主要看看是否是外部调用。是的。  因为用户点击时触发事件，不是在fun2中内部调用的。 |
| demo3-cache | 下面的code是缓存的应用，catchNameArr。在匿名函数的调用对象中保存catch的值，返回的对象由于被CachedBox变量引用导致匿名函数的调用对象不会被回收，从而保持了catch的值。可以通过CachedBox.getCatch("regionId");来操作，若找不到regionId则从后台取，catchNameArr 主要是为了防止缓存过大。  var CachedBox = (function() {      var cache = {}, catchNameArr = [], catchMax = 10000;      return {          getCatch: function(name) {              if (name in cache) {                  return cache[name];              }              var value = GetDataFromBackend();              cache[name] = value;              catchNameArr.push(name);              this.clearOldCatch();              return value;          },          clearOldCatch: function() {              if (catchNameArr.length > catchMax) {                  delete cache[catchNameArr.shift()];              }          }      };  })();  同理，也可以用这种思想实现自增长的ID。  var GetId = (function() {      var id = 0;      return function() {          return id++;      }  })();  var newId1 = GetId();  var newId2 = GetId(); | |
| DOM循环添加事件  onclick | [在循环中正确找到对应DOM元素的索引](http://www.cnblogs.com/snandy/archive/2011/03/01/1967628.html)  <http://www.cnblogs.com/snandy/archive/2011/03/01/1967628.html> | |
| /\*\*  \* <body>  \* <ul>  \* <li>one</li>  \* <li>two</li>  \* <li>three</li>  \* <li>one</li>  \* </ul>  \*/  **var lists** = **document**.getElementsByTagName(**'li'**); **for**(**var i** = 0 , **len** = **lists**.**length** ; **i** < **len** ; **i**++){  **lists**[**i**].onmouseover = **function**(){  console.log(**i**);  }; }  //你会发现当鼠标移过每一个<li>元素时，总是弹出4，而不是我们期待的元素下标。这是为什么呢？注意事项里已经讲了（最终值）。显然这种解释过于简单，当mouseover事件调用监听函数时，首先在匿名函数（ function(){ console.log(i); }）内部查找是否定义了 i，结果是没有定义；因此它会向上查找，查找结果是已经定义了，并且i的值是4（循环后的i值）；所以，最终每次弹出的都是4。 | **var lists** = **document**.getElementsByTagName(**'li'**); **for**(**var i** = 0 , **len** = **lists**.**length** ; **i** < **len** ; **i**++){  (**function**(index){  **lists**[ index ].onmouseover = **function**(){  console.log(index);  };  })(**i**); }  **var lists** = **document**.getElementsByTagName(**'li'**); **for**(**var i** = 0, **len** = **lists**.**length**; **i** < **len**; **i**++){  **lists**[ **i** ].**$$index** = **i**; //通过在Dom元素上绑定$$index属性记录下标  **lists**[ **i** ].onmouseover = **function**(){  console.log(**this**.**$$index**);  }; }  **function** eventListener(list, index){  list.onmouseover = **function**(){  console.log(index);  }; } **var lists** = **document**.getElementsByTagName(**'li'**); **for**(**var i** = 0 , **len** = **lists**.**length** ; **i** < **len** ; **i**++){  eventListener(**lists**[ **i** ] , **i**); }  **var list\_obj** = **document**.getElementsByTagName(**'li'**); **for** (**var i** = 0; **i** < **list\_obj**.*length*; **i**++) {  **list\_obj**[**i**].**onclick** = **function**() {  console.log(**this**.**innerHTML**);  console.log(**i**);  console.log(**list\_obj**[**i**-1].**innerHTML**);  } } |
| setTimeout | 异步操作，setTimeout或setInterval  var fun = function(x) {         setInterval(function() {                 console.log(x++);         }, 3000) } fun(1);  setInterval第一个参数，是一个函数，此函数的执行必须是在全局调用的。因此是闭包。 | |
| 异步操作  ajax请求回调 |  | |
| 特权函数 | var Person = function() {         this.sayName = function() {                 console.log(name);         }         var name = "laoyao"; } var p = new Person(); p.sayName(); | |
| 匿名函数与闭包 | 1、闭包的含义：闭包说白了就是函数的嵌套，内层的函数可以使用外层函数的所有变量，即使外层函数已经执行完毕（这点涉及JavaScript作用域链）。  2、匿名函数最大的用途是创建闭包（这是JavaScript语言的特性之一），并且还可以构建命名空间，以减少全局变量的使用。  1、闭包允许内层函数引用父函数中的变量，但是该变量是最终值  2、内存泄露：使用闭包十分容易造成浏览器的内存泄露，严重情况下会是浏览器挂死  1.函数嵌套函数  2.函数内部可以引用外部的参数和变量  3.参数和变量不会被垃圾回收机制回收 | |
| **function** dodo(x, y) {  console.log(x + y); } **function** delay1(x, y, time) {  setTimeout(**'dodo('** + x + **','** + y + **')'**, time); } //上面的delay1函数十分难以阅读，也不容易编写，但如果使用**闭包**就可以让代码更加清晰 **function** delay2(x, y, time) {  setTimeout(**function** () {console.log(**"1"**); dodo(x, y) }, time); } | |
| **var oEvent** = {}; (**function**(){  **var** addEvent = **function**(){};//函数方式1  **function** removeEvent(){};//函数方式2  console.log(**"111"**)//这段话会立即执行  **oEvent**.addEvent = addEvent;  **oEvent**.removeEvent = removeEvent; })(); /\*  在这段代码中函数addEvent和removeEvent都是局部变量，但我们可以通过全局变量oEvent使用它，这就大大减少了全局变量的使用，增强了网页的安全性。  我们要想使用此段代码：oEvent.addEvent(document.getElementById('box') , 'click' , function(){});  \*/ | |
| **var abc** = (**function**(x , y){  **return** x + y; })(2 , 3);  //初始化一个变量 | |
| 内存泄露问题 | //由于IE的js对象和DOM对象使用不同的垃圾收集方法，因此闭包在IE中会导致内存泄露问题，也就是无法销毁驻留在内存中的元素  function closure(){  var oDiv = document.getElementById('oDiv');//oDiv用完之后一直驻留在内存中  oDiv.onclick = function () {  alert('oDiv.innerHTML');//这里用oDiv导致内存泄露  };  }  closure();  //最后应将oDiv解除引用来避免内存泄露  function closure(){  var oDiv = document.getElementById('oDiv');  var test = oDiv.innerHTML;  oDiv.onclick = function () {  alert(test);  };  oDiv = null;  } | |

js建立对象：工厂 构造 原型

|  |  |
| --- | --- |
|  | //use factory 工厂模式 **function** O2(user,pwd){  **var obj**=**new** Object();  **obj**.**user**=user;  **obj**.**pwd**=pwd;  **obj**.**get**=**get**;  **return obj**; } //use constructor 构造函数 **function** O(user,pwd){  **this**.**user**=user;  **this**.**pwd**=pwd;  **this**.**get**=**get**;  **return this**; } //use prototype 原型模式 **function** O3(){ } O3.**prototype**.**user**=**'abc'**; O3.**prototype**.**pwd**=**'abc'**; O3.**prototype**.**get**=**get**;  // 混合：构造+原型 **function** O4(user,pwd){  **this**.**user**=user;  **this**.**pwd**=pwd;  **return this**; } O4.**prototype**.get=**function**(){console.log(**'123'**);};  // 共有函数get **function get**() {  console.log(**"This User:"** + **this**.**user**); } |
| 工厂模式 | **function** createPerson(name,age,job){  **var o** = {};  **o**.**name** = name;  **o**.**age** = age;  **o**.**job** = job;  **o**.sayName = **function**(){  console.log(**this**.**name**);  };  **return o**; } **var tanya** = createPerson(**"tanya"**,**"30"**,**"female"**); **var ansel** = createPerson(**"ansel"**,**"30"**,**"male"**); **tanya**.sayName(); **ansel**.sayName();  这里先定义o为一个空的对象，然后为o设置了一堆属性；  其实也可以直接给o属性的嘛，所以如果这样写也是ok的：  **function** createPerson(name,age,job){  **var o** = {  **name** : name,  **age** : age,  **job** : job,  sayName : **function**(){  console.log(**this**.**name**);  }  };  **return** o; } **var tanya** = createPerson(**"tanya"**,**"30"**,**"female"**); **var ansel** = createPerson(**"ansel"**,**"30"**,**"male"**); **tanya**.sayName(); **ansel**.sayName();  说明：  1.在函数中定义对象,并定义对象的各种属性，虽然属性可以为方法，但是建议将属性为方法的属性定义到函数之外，这样可以避免重复创建该方法  2.引用该对象的时候，这里使用的是 var x = createPerson()而不是 var x = new Parent();因为后者会可能出现很多问题（前者也成为工厂经典方式,后者称之为混合工厂方式），不推荐使用new的方式使用该对象  3.在函数的最后返回该对象  4.不推荐使用这种方式创建对象，但应该了解 |
| 构造函数 | 还有一种办法是利用无敌的this，因为this就表示当前运行时的对象，将构造函数this的作用域指向新对象，将当前运行对象的属性和方法都赋给新对象，这样对象模式称为构造函数模式：  **function** Person(name,age,job){  **this**.**name** = name;  **this**.**age** = age;  **this**.**job** = job;  **this**.sayName = **function**(){  console.log(**this**.**name**);  }; } **var tanya** = **new** Person(**"tanya"**,**"30"**,**"female"**); **var ansel** = **new** Person(**"ansel"**,**"30"**,**"male"**); **tanya**.sayName(); **ansel**.sayName();  说明：  1.与工厂方式相比，使用构造函数方式创建对象，无需再函数内部重建创建对象，而使用this指代，并而函数无需明确return  2.同工厂模式一样，虽然属性的值可以为方法，扔建议将该方法定义在函数之外  3.同样的，不推荐使用这种方式创建对象，但仍需要了解  在这个例子中，tanya和ansel都有一个constructor属性，该属性指向person。 考虑一下如下的情况： 代码如下:  **function** Person(name,age,job){  **this**.**name** = name;  **this**.**age** = age;  **this**.**job** = job;  **this**.sayName = **function**(){  console.log(**this**.**name**);  }; } Person(**"tanya"**,**"30"**,**"female"**); Person(**"ansel"**,**"30"**,**"male"**); **window**.sayName(); **window**.sayName();  发现两次弹出的都是ansel，这是因为不用new的话，就不是一个person的实例，而仅仅在执行函数。而在全局作用域调用一个函数时this总是指向Global对象。而Global对象在浏览器中就是window对象。 我们还可以用构造模式在另外一个对象中调用sayName方法，还记得Apply和call么，来吧再考虑另外一种情况， 代码如下: **function** Person(name,age,job){  **this**.**name** = name;  **this**.**age** = age;  **this**.**job** = job;  **this**.sayName = **function**(){  console.log(**this**.**name**);  }; } **var olivia** = {}; Person.call(**olivia**,**"tanya"**,**"30"**,**"female"**); **olivia**.sayName(); **var philip** = {} Person.apply(**philip**,[**"ansel"**,**"30"**,**"male"**]); **philip**.sayName(); |
| 原型 | **var** lev=**function**(){  **return** **"啊打"**; }; **function** Parent(){  }; Parent.**prototype**.**name**=**"李小龙"**; Parent.**prototype**.**age**=**"30"**; Parent.**prototype**.lev=lev;  **var**  **x** =**new**  Parent(); console.log(**x**.**name**); console.log(**x**.lev());    说明：  1.函数中不对属性进行定义  2.利用prototype属性对属性进行定义  3.同样的，不推荐使用这样方式创建对象 |
| 混合：  构造函数  +  原型方式 | 原型模式就要考虑原型链了，分析一下，sayName方法在实例中被重复定义了两次，但其实没有必要创造两个一样的副本。使用原型方法，可以使是tanya和ansel的共享一个sayName方法。 于是原型模式的写法如下： 代码如下:  **function** Person(name,age,job){  **this**.**name** = name;  **this**.**age** = age;  **this**.**job** = job; } Person.**prototype**.sayName= **function**(){  console.log(**this**.**name**); }; **var tanya** = **new** Person(**"tanya"**,**"30"**,**"female"**); **var ansel** = **new** Person(**"ansel"**,**"30"**,**"male"**); **tanya**.sayName(); **ansel**.sayName();  说明：  1、该模式是指混合搭配使用构造函数方式和原型方式  2、将所有属性不是方法的属性定义在函数中（构造函数方式）  将所有属性值为方法的属性利用prototype在函数之外定义（原型方式）  3.推荐使用这样方式创建对象,这样做有好处和为什么不单独使用构造函数方式和原型方式，由于篇幅问题这里不予讨论 |
| 动态原型 | 实际应用时，不是一成不变的套用某种模式，活学活用。需要共享方法的时候就用原型模式，需要使用副本的时候就用构造模式，还可以结合起来，把所有信息都封装在构造函数中，而通过在构造函数中初始化原型，使得对象保持了同时使用构造函数和原型的优点。 **function** Person(name,age,job){  **this**.**name** = name;  **this**.**age** = age;  **this**.**job** = job;  **if** (**typeof** sayName != **"function"** ){  Person.**prototype**.sayName= **function**(){  console.log(**this**.**name**);  };  } } **var tanya** = **new** Person(**"tanya"**,**"30"**,**"female"**); **var ansel** = **new** Person(**"ansel"**,**"30"**,**"male"**); **ansel**.sayName = **function** () {  console.log(**"Hi ansel, how hansome you are!"**); } **tanya**.sayName(); **ansel**.sayName();  说明：  1.动态原型方式可以理解为混合构造函数，原型方式的一个特例  2.该模式中,属性为方法的属性直接在函数中进行了定义，但是因为  if(typeof Parent.\_lev=="undefined"){ Parent.\_lev=true; }  从而保证创建该对象的实例时，属性的方法不会被重复创建   1. 推荐使用这种模式   **function** Parent() {  **this**.**name** = **"李小龙"**;  **this**.**age** = 32;  **if** (**typeof** Parent.\_lev == **"undefined"**) {  Parent.**prototype**.lev = **function** () {  **return this**.name;  }  Parent.**\_lev** = **true**;  } } **var x** = **new** Parent(); console.log(**x**.lev()); |

立即执行函数与模块

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | ****模块****  有多少人在一个文件中写的JS像下面的代码块一样？（注意：我可没有说内嵌在HTML文件中哦）:   |  |  | | --- | --- | | 1  2  3 | var someSharedValue = 10;  var myFunction = function(){ //do something }  var anotherImportantFunction = function() { //do more stuff } |   如果你做到了这一点，那么很有可能你正在写这样的代码。我不是在给你下定义，因为在相当长的一段时间里我也曾这么写程序。事实上这段代码有很多毛病，不过我们会专注在讨论全局命名空间的污染问题上。这样的代码代码会把方法和变量都暴露在了全局中，我们需要将让这些数据与全局命名空间独立开来，我们将会采用模块模式（Module Pattern）来实现这个目的。模块中可以有很多不同的形式达到我们的目标，我会从最简单的方法开始说：立即调用的函数表达式（Immediately Invoked Function Expression，简写为：IIFE）。  名字听起来很高大上，不过它的实现其实很简单：   |  |  | | --- | --- | | 1  2  3 | (function(){      //do some work  })(); |   如果在此之前你从未接触过这种函数，可能现在你会觉得它很怪 — 怎么会有这么多括号！它是会立即执行的函数，你可以这么理解：一个函数被创建了后又立刻被调用。它应该是一个表达而不是一个语句：一个函数语句是一定要有一个名字的，但是大家也看到了，立即执行函数是没有名字的。在函数定义的外部还有一组括号，这一点也能很好地帮助我们在代码中轻易找到匿名函数的身影。  现在我们知道要怎么写一个立即执行函数了，那就来聊聊为什么要使用它吧。在JS中我们都是在和各种作用域之中的函数打交道，所以如果我们想要创建一个作用域，就可以使用函数。匿名函数中的变量和方法的作用域仅仅在匿名函数中，就不会污染全局的命名空间，那么现在还需要考虑的一个问题是，我们要如何从外部取得那些在匿名函数作用域中的变量和方法呢？答案就是全局命名空间：将变量放入全局命名空间中，或者至少将作用变量与全局命名空间关联起来  想要在立即执行函数外部调用方法，我们可以将window对象传入立即执行函数，再将函数或变量值赋值到这个对象上。为了保证这个window对象的引入不会造成什么混乱，我们可以将widow对象作为一个变量传入我们的立即执行函数中。当做函数传入参数的方法同样适用于第三方库，甚至undefined这样的值。  现在我们的立即执行函数看起来是这样的：   |  |  | | --- | --- | | 1  2  3 | (function(window, $, undefined){      //do some work  })(window, jQuery); |   正如你所看到的，我们将window和jQuery传入函数中（’$'符号表示的就是’jQuery’，把它用在这的原因是防止其他库也定义了’$'），但是这个函数其实是接收了3个参数。如果我们没有传入第三个参数，形参undefined 的值会保持这个状态，而不会由于其他代码把全局的undefined改变了它也跟着改变。其实在函数内我们也是可以直接使用这些值，能这么做的原理是，JS的闭包会覆盖他们所处的上下文。对于这个话题，我曾写过[一篇关于C#的文章](http://www.codethinked.com/c-closures-explained" \t "http://blog.jobbole.com/66135/_blank)以解释这个概念，这两者是互通的。  现在我们有了一个会立即执行的方法，还有一个相对安全的执行上下文，其中还包含有window、$和undefined变量（这几个变量还是有可能在这个脚本被执行前就被重新赋值，不过现在的可能性要小的多了）。现在我们已经做得很好了：把我们的代码从全局环境下的一团混乱的局面中拯救了出来；降低了与其他在同一应用中使用的脚本的冲突可能性。  任何我们想要从模块中获取的东西都可以通过window对象拿到。但是通常我不会直接将模块中的内容直接复制到window对象上，而是会用更有组织性地将模块中的内容。在大部分语言中，我们将这些容器称为“命名空间”，在JS中我们可以用“对象”的方式来模拟。  ****命名空间****  如果我们想要声明一个命名空间，将一个函数放进这个空间中，代码可以写成这样：   |  |  | | --- | --- | | 1  2  3  4 | window.myApp = window.myApp || {};  window.myApp.someFunction = function(){      //so some work  }; |   我们是在全局环境中创建了一个对象，它是用于查看另外的某个对象是否已经存在，如果已经存在了，那么我们就可以直接使用；不然就需要用’{}’来创建一个新的对象。接着，我们可以开始添加这个命名空间的内容，将各种函数放入这个空间中，就像上面的代码片段所做的那样，但是我们又不希望这些函数就随便的放在那里，而是希望将模块和命名空间联系在一起，就像下面这样：   |  |  | | --- | --- | | 1  2  3 | (function(myApp, $, undefined){      //do some work  }(window.myApp = window.myApp || {}, jQuery)); |   还可以这么写：   |  |  | | --- | --- | | 1  2  3  4 | window.myApp = (function(myApp, $, undefined){      //do some work      return myApp;  })(window.myApp || {}, jQuery); |   现在，我们不再是将window传入我们的模块中，我们将一个和window对象联系在一起的命名空间传入模块中。之所以**使用’||’的原因是我们可以重复使用同一个命名空间**，而不是每次需要使用命名空间的时候我们又要重新创建一个。许多包含有命名空间方法的库会帮你创建好空间的，或者你可以使用一些想namespace.js这样的工具来构建嵌套的命名空间。由于在JS中，每一个在命名空间中的项你都不得不指定它的命名空间，所以通常我都尽量不会去创建深度嵌套的命名空间。  如果你在MyApp.MyModule.MySubModule中创建了一个doSomething方法，你需要这么引用它：  MyApp.MyModule.MySubModule.doSomething();  每次你要调用它，或者你可以在你的模块中给这个命名空间一个别名：  var MySubModule = MyApp.MyModule.MySubModule;  这样定义以后，如果你想用doSomething这个方法可以用MySubModule.doSomething()来调用。不过这个方式其实是不必要的，除非你有非常非常多的代码，不然这么做只会将问题复杂化。  ****揭秘模块模式****  在创建模块时你也常会看到另一种设计模式：揭秘模块模式（Revealing Module Pattern）。它和模块模式有一些不同：所有定义在模块中的内容都是私有的，然后你可以把所有要暴露到模块外部的内容放在一个对象中，再返回这个对象。你可以这么做：  var myModule = (function($, undefined){       var myVar1 = '',       myVar2 = '';       var someFunction = function(){           return myVar1 + " " + myVar2;       };       return {           getMyVar1: function() { return myVar1; }, //myVar1 public getter           setMyVar1: function(val) { myVar1 = val; }, //myVar1 public setter           someFunction: someFunction //some function made public       }  })(jQuery);  一次就建立一个模块，然后返回一个包含有需要公有化的模块片段的对象，同时模块中需要保持私有的变量也不会被暴露。myModule变量会包含有两个共有的项，不过其中Somefunction中的myVar2是从外部获取不到的。  ****创建构造器（类）****  在JS中没有“类”这个概念，但是我们可以通过创建构造器来创建“对象”。假设现在我们要创建一系列Person对象，还需要传入姓、名和年龄，我们可以将构造器定义成下面这样（这部分代码应该放在模块之中）：  var Person = function(firstName, lastName, age){      this.firstName = firstName;      this.lastName = lastName;      this.age = age;  }  Person.prototype.fullName = function(){      return this.firstName + " " + this.lastName;  };  现在先看第一个函数，你会看到我们创建了一个Person构造器。我们用它来构造新的person对象。这个构造器需要3个传入参数，然后将这3个参数赋值到执行上下文中。我们也是通过这种方式获取到公有实例变量。这里也可以创建私有变量：将传入参数赋值到这个构造器中的局部变量。但是这么做以后，公有的方法就没法获取这些私有的变量了，所以你最好还是把它们都变成公有的。也可以把方法放在构造器中同时还能从外部获取到它，这样方法就能拿到构造器里的私有变量了，不过这么做的话又会出现一系列新的问题。  第二个方法中我们使用了Person构造器的”原型”（prototype）。一个函数的原型就是一个对象，当你需要在某个实例上解析它所调用到的字段或者函数时你需要遍历这个函数上所有的实例。所以这几行代码所做的就是创建一个fullName方法的实例，然后所有的Person的实例都能直接调用到这方法，而不是对每个Person实例都添加一个fullName方法，造成方法的泛滥。我们也可以在构造器中用this.fullName = function() { … 的方式定义fullName，但这样每一个Person实例都会有fullName方法的副本，这不是我们希望的。  如果我们想要创建一个Person实例，我们可以这么做：  var person = new Person("Justin", "Etheredge");  console.log(person.fullName());  我们也可以创建一个继承自Person的构造器：Spy构造器，我们会创建Spy的一个实例，不过只会声明一个方法：  var Spy = function(firstName, lastName, age){      this.firstName = firstName;      this.lastName = lastName;      this.age = age;  };  Spy.prototype = new Person();  Spy.prototype.spy = function(){      console.log(this.fullName() + " is spying.");  }  var mySpy = new Spy("Mr.", "Spy", 50);  mySpy.spy();  正如你所看到的，我们创建了一个和Person很相似的构造器，但是它的原型是Person的一个实例。现在我们又添加上一些方法，使得Spy的实例又可以调用到Person的方法，同时还能直接取得Spy中的变量。这个方法比较复杂，不过一旦你明白怎么使用了，你的代码就会变得很优雅。  ****结语****  看到这里，希望你已经学到了一些东西。不过这篇文章里并没有介绍多少关于“现代”JS的开发。这篇文章中涉及的还是旧知识，在过去几年里它们的使用面相当广。希望你看完这篇文章以后，找到了学习JS的正确的方向。现在可能你把代码放到了不同的模块不同的文件中（你应该做到这一点！），那么下一步你要开始着手研究如何将JS结合和压缩。如果你是使用Rails 3的开发者，可以在asset pipeline上免费获取这些信息或者工具。如果你是.NET开发者，你可以看看[SquishIt](https://github.com/jetheredge/SquishIt" \t "http://blog.jobbole.com/66135/_blank)框架，我就是从这里开始的。如果你是ASP.NET MVC 4的开发者，也有相关的资源。 |
| A | 最简单的函数： var some=funciton(){}  立即函数：(function(){ === })()  传参的立即函数：(function(window,$,undifined){ === })(window,jQuery)  [立即执行的方法，执行上下文，几个变量（在此段脚本被执行之前就被重新赋值）] | |
| B | 简单的函数：  window.app=window.app||{};  window.app.somefuc=function(){ === }  模块化编写：  (function(app,$){ === })( window.app=window.app||{},jquery)  window.app=(function(app,$){ === return app; })(window.app,jquery)  [使用||的原因是我们可以重复使用同一个命名空间] | |

引用

|  |  |
| --- | --- |
|  | 在 Node.js 中看 Javascript 的引用  <https://zhuanlan.zhihu.com/p/26128007>  [javascript 连等赋值问题](https://segmentfault.com/q/1010000002637728)  <https://segmentfault.com/q/1010000002637728>  var a = {n:1}; var b = a; // 持有a，以回查  a.x = a = {n:2};  console.log(a.x);// --> undefined  console.log(b.x);// --> {n:2}  [从连续赋值到：词法分析、函数执行原理](https://segmentfault.com/a/1190000007063202)  https://segmentfault.com/a/1190000007063202 |
| 关于& | for (i = 1; i < 10; i++) {  // i && 1?console.log("第1段代码"):console.log("第2段代码");  i & 1 ? console.log("第1段代码") : console.log("第2段代码");  }  //轮流执行（第一段）（第二段） |
| apply | apply函数中的第一个参数为函数所属对象，通常为当前页面(this)。第二个参数为传入到函数中的参数数组（必须为数组）。 例子：  var add = function(a,b){ return a+b; } var args = [2,3]; console.log(add.apply(this,args));  Math.max.apply(Math,values); 这句代码的意思执行Math对象中的max函数，参数为values，并将返回结果赋值于max 其中apply 函数为JS的内置函数，作用为执行该函数。 |
| 关于XMLHttpRequest | XMLHttpRequest 是 AJAX 的基础。  所有现代浏览器均支持 XMLHttpRequest 对象（IE5 和 IE6 使用 ActiveXObject）。  XMLHttpRequest 用于在后台与服务器交换数据。这意味着可以在不重新加载整个网页的情况下，对网页的某部分进行更新。  所有现代浏览器（IE7+、Firefox、Chrome、Safari 以及 Opera）均内建 XMLHttpRequest 对象。  variable=new XMLHttpRequest();  老版本的 Internet Explorer （IE5 和 IE6）使用 ActiveX 对象：  variable=new ActiveXObject("Microsoft.XMLHTTP");  为了应对所有的现代浏览器，包括 IE5 和 IE6，请检查浏览器是否支持 XMLHttpRequest 对象。如果支持，则创建 XMLHttpRequest 对象。如果不支持，则创建 ActiveXObject ：  var xmlhttp;  if (window.XMLHttpRequest) { // code for IE7+, Firefox, Chrome, Opera, Safari  xmlhttp = new XMLHttpRequest();  } else { // code for IE6, IE5  xmlhttp = new ActiveXObject("Microsoft.XMLHTTP");  }  request.open("GET"，"/get,true);  request.send();  request.open("POST","/post",true);  request.send();  request.open("Post","/create",true);  request.setRequestHeader("Content-type","application/x-www-form-urlencoded");  request.send("name=hehe&age=11");        var rq = new XMLHttpRequest();  rq.open("GET", "/get", true);  rq.send();  request.onreadystatechange = function () {  if (rq.readyState === 4 && rq.status === 200) {}  } |
| 跨域 | jsonp 不支持post请求 |
| 关于settimeout | setTimeout("console.log('3秒后执行我')",3000);  //直接把执行语句放在双引号里当参数  setTimeout(function(){console.log("3秒后执行我");},3000);  //直接把一个包含执行语句的匿名函数当参数  setTimeout(count,3000);  //把指针当参数  setTimeout("count()",3000);  //把count()函数放在双引号里当参数，相当于用eval()函数来解析他  setTimeout(function(){count();},3000);  //把要执行的函数放在一个匿名函数里当参数  setTimeout(arguments.callee,3000);  //如果是调用自身的话可以直接用arguments.callee方法，优于把自身函数的指针，因为arguments.callee始终指向自身，不用管函数名变动的问题。 |
|  | setTimeout("count('"+arg1+"',"+"'"+arg2+"',"+....+"')",3000);  //不推荐这种方法，麻烦，还容易弄错  setTimeout(count,3000,arg1,arg2....);  //第一个参数是指针，第二个是延迟执行的时间，后面紧跟着的是调用函数所执行的参数  setTimeout(function(){count(arg1,arg2...);},3000);  //直接把要执行的函数已经参数放在匿名函数里当setTimeout的第一个参数  setTimeout(arguments.callee,3000,arg1,arg2....);  //如果是调用自身的话可以用arguments.callee当自身指针 |
| 闭包相关 | for (var i = 0; i < 10; i++) {  setTimeout(function () {  console.log(i);  }, 0); }  输出结果使10个10；  相当于for(var i=0;i<10;i++){} setTimeout(function(){console.log(i)},0);  相当于for(var i=0;i<10;i<10){} console.log(i) |
| for(var i = 0 ; i < 10; i++){  setTimeout((function(i){  return function(){console.log(i);}  })(i), 0); }  输出结果为1234567890 |

异步加载js方式

|  |  |
| --- | --- |
| js加载方式 | 同步加载  在介绍js异步加载之前，我们先来看看什么是js同步加载。我们平时最常使用的就是这种同步加载形式：  <script src="http://XXX.com/script.js"></script>  同步模式，又称阻塞模式，会阻止浏览器的后续处理，停止了后续的解析，因此停止了后续的文件加载（如图像）、渲染、代码执行。一般的script标签（不带async等属性）加载时会阻塞浏览器，也就是说，浏览器在下载或执行该js代码块时，后面的标签不会被解析，例如在head中添加一个script，但这个script下载时网络不稳定，很长时间没有下载完成对应的js文件，那么浏览器此时一直等待这个js文件下载，此时页面不会被渲染，用户看到的就是白屏。以前的一般建议是把<script>放在页面末尾</body>之前，这样尽可能减少这种阻塞行为，而先让页面展示出来。  异步加载  它允许无阻塞资源加载，并且使 onload 启动更快，允许页面内容加载，而不需要刷新页面，也可以根据页面内容延迟加载依赖。  异步加载，需要将所有 js 内容按模块化的方式来切分组织，其中就存在依赖关系，而异步加载不保证执行顺序。​js模块化解决了这个问题。  延迟加载  有些 js 代码并不是页面初始化的时候就立刻需要的，而稍后的某些情况才需要的。延迟加载就是一开始并不加载这些暂时不用的js，而是在需要的时候或稍后再通过js 的控制来异步加载。  也就是将 js 切分成许多模块，页面初始化时只加载需要立即执行的 js ，然后其它 js 的加载延迟到第一次需要用到的时候再加载。  特别是页面有大量不同的模块组成，很多可能暂时不用或根本就没用到。  就像图片的延迟加载，在图片出现在可视区域内时（在滚动条下拉）才加载显示图片。 |
|  | 方案一：async="true"属性  点评：HTML5中新增的属性，Chrome、FF、IE9&IE9+均支持（IE6~8不支持）。此外，这种方法不能保证脚本按顺序执行。  <script type="text/javascript" src="load.js" async="true"></script>  方案二：defer="defer"属性  <script type="text/javascript"  defer="defer">意思是延迟脚本的执行，直到整个页面都加载完才执行，防止还没加载报对象不存在。  浏览器支持：注意：只有 Internet Explorer 支持 defer 属性。  如果您的脚本不会改变文档的内容，可将 defer 属性加入到 <script> 标签中，以便加快处理文档的速度。因为浏览器知道它将能够安全地读取文档的剩余部分而不用执行脚本，它将推迟对脚本的解释，直到文档已经显示给用户为止。  方案三：动态创建<script>标签  示例：  (function () {  var s  =  document.createElement\_x('script');  s.type  =   'text/javascript';  s.src  =   "http://code.jquery.com/jquery-1.7.2.min.js";  var tmp  =  document.getElementsByTagName\_r('script')[0];  tmp.parentNode.insertBefore(s,  tmp);  })();  这种方法是在页面中<script>标签内，用 js 创建一个 script 元素并插入到 document 中。这样就做到了非阻塞的下载 js 代码。  点评：兼容所有浏览器。  方案四：AJAX eval（使用AJAX得到脚本内容，然后通过eval\_r(xmlhttp.responseText)来运行脚本）  点评：兼容所有浏览器。  方案五：iframe方式（这里可以参照：[iframe异步加载技术及性能](http://hi.baidu.com/flondon/blog/item/3759c4b5ad433d538bd4b297.html" \t "http://blog.sina.com.cn/s/_blank) 中关于Meboo的部分）  点评：兼容所有浏览器。 |
|  | 用于确保页面中JS加载完全。    var n = document.createElement\_x("script");  n.type = "text/javascript";  //以上省略部分代码  //ie支持script的readystatechange属性(IE support the readystatechange event for script and css nodes)  if(ua.ie){     n.onreadystatechange = function(){         var rs = this.readyState;         if('loaded' === rs || 'complete'===rs){             n.onreadystatechange = null;             f(id,url); //回调函数         }  };  //省略部分代码  //safari 3.x supports the load event for script nodes(DOM2)     n.addEventListener('load',function(){         f(id,url);     });  //firefox and opera support onload(but not dom2 in ff) handlers for  //script nodes. opera, but no ff, support the onload event for link  //nodes.  }else{     n.onload = function(){         f(id,url);     };  } |
|  | 1. 使用XMLHttpRequest对象加载   var xhr = new XMLHttpRequest();  xhr.open("get","load.js",true);  xhr.onreadystatechange=function(){      if(readyState==4){          if(status>=200&&status<300||status==304){             var script = document.createElement("script");              script.type = "text/javascript";              script.src = xhr.responseText;              document.body.appendChild(script);          }      }  }  xhr.send(null); |
|  | 1. 使用推荐的无阻塞模式：   function loadScript（url ,  callback） {  var script  =  document.createElement("script");  script.type = "text/javascript";  if (script.readyState) {  script.onreadystatechange  =  function() {  if (script.readyState == "loaded" || script.readyState == "complete") {  script.onreadystatechange = null;  callback();  }  }  } else {  script.onload  =  function() {  callback();  }  }  script.src  =  url;  document.getElementsByName("head")[0].appendChild(script);  } |
|  | 1. [关于JS异步加载的测试](http://www.cnblogs.com/mxw09/archive/2010/11/22/1884336.html)   对IE8和Firefox3.6测试结果： 1.常规的在head里面使用Script标签引用js,不再是堵塞模式加载（也就是说脚本文件不再是一个一个被加载，加载完成一个才开始加载另一个），而是最大6个JS同时开始加载（并发限制为6个），但是script的执行顺序还是按照Script标签顺序，而且会阻碍页面的呈现。 2.由于1的原因，所以使用document.write("<script>...</script>")(此方式对Firefox无效)的方式和head.append(script)（此方式兼容IE和Firefox）的方式使用异步加载意义不大 3.但是使用head.append(script)的方式，不会阻碍界面的呈现，但是要注意： （1）如果对于window.onload事件的触发有所影响：在 IE中，window.onload事件会在页面其它元素被加载完毕之后立即触发（即不必等所有脚本加载完，而是html页面加载完），而FireFox里的window.onload事件会等待动态添加的那些脚本文件也被加载完毕后才触发。 （2）虽然我们动态加载的script元素是有严格顺序的，但是浏览器可不一定这样认为。在FireFox中，脚本文件会按照它动态加载的script元素的顺序执行，而IE会根据脚本文件下载完毕的顺序执行。  注意： 1.在IE6和IE7中，常规的在head里面使用Script标签引用js，是堵塞模式加载，也就是说必须加载完一个JS，才能去加载下一个JS 2.在IE6和IE7中，并发限制是2个。也就是即使使用head.append(script)的方式异步加载3个JS，也会等前两个JS加载完成以后才会加载第三个JS。 |

!function($){}(window.jQuery)是什么意思

|  |  |
| --- | --- |
|  | !function($){          //一些代码      }(window.jQuery)  应该等同于：      function Method1($){          //一些代码      };  !Method1(window.jQuery);  匿名函数：  (function($){      console.log($)  })(this);  !function($){      console.log($)  }(this);  ~function($){      console.log($)  }(this);  (function(){})()匿名函数直接执行，用取反的标志可以活力匿名函数左右的括号！以省略匿名函数左右的括号！  我们都知道，函数的声明方式有这两种  function fnA(){console.log('msg');}//声明式定义函数  var fnB = function(){console.log('msg');}//函数赋值表达式定义函数  楼主问题中出现的两个函数，都是匿名函数。通常，我们调用一个方法的方式就是 FunctionName()  但是，如果我们尝试为一个“定义函数”末尾加上()，解析器是无法理解的。  function msg(){  console.log('message');  }();  function() {}();  //解析器是无法理解的,会报错,因为这是一个函数定义  定义函数的调用方式应该是 msg() ; 那为什么将函数体部分用()包裹起来就可以了呢？  原来，使用括号包裹定义函数体，解析器将会以函数表达式的方式去调用定义函数。也就是说，任何能将函数变成一个函数表达式的作法，都可以使解析器正确的调用定义函数。而 ! 就是其中一个，而 + - || 都有这样的功能。  另外，用 ! 可能更多的是一个习惯问题，不同的运算符，性能是不同的。  [What does !function ($) { $(function(){ }) }(window.jQuery) do?](http://stackoverflow.com/questions/10896749/what-does-function-function-window-jquery-do)  <http://stackoverflow.com/questions/10896749/what-does-function-function-window-jquery-do?rq=1> |

JS匿名函数与自执行

|  |  |
| --- | --- |
|  | <http://www.jcodecraeer.com/a/jquery_js_ajaxjishu/2012/0628/290.html>  **函数**是JavaScript中最灵活的一种对象，这里只是讲解其匿名函数的用途。匿名函数：就是没有函数名的函数。  【1】函数的定义（3种方式）：   1. 常规的使用方式   **function** sth(){}   1. 使用Function的构造函数，把参数列表和函数体都作为字符串，很不方便，不建议使用。   **var sth**=**new Function**(**"x"**,**"console.log(x);return x\*2;"**);  console.log(**sth**(2));   1. 命名函数(将一个创建好的匿名函数，赋值给了变量)   **var** sth=**function**(x){console.log(x);**return** x\*2;};  console.log(sth(2));  【2】匿名函数的创立   1. 方式一，创建匿名函数较为常见的一种方式   **function**(x){console.log(x);**return** x\*2;};   1. 方式二，第一个刮号创建，第二个刮号进行调用，并传入参数。第一个刮号是表达式，是表达式就会有返回值，可以继续添加一对刮号让它们执行。   (**function**(x,y){  console.log(x+y);  })(2,3)  结果：5  console.log((**function**(x,y){  console.log(x+y);  **return** x\*y;  })(2,3))  结果：5 6  【3】关于匿名函数的执行：  1. 什么是自执行的匿名函数?     它是指形如这样的函数: (function (){// code})();     2. 疑问     为什么(function(){// code})();可以被执行, 而function(){// code}();却会报错?     3. 分析     (1). 首先, 要清楚两者的区别:     (function(){// code})是表达式, function(){// code}是函数声明.     (2). 其次, js"预编译"的特点:  js在"预编译"阶段, 会解释函数声明, 但却会忽略表式;  js在“执行”阶段，选择()，跳过申明的函数。()的返回值是一个函数，函数()，是函数名()形式，故会执行     (3). 当js执行到function(){//code}();时, 由于function(){//code}在"预编译"阶段已经被解释过, js会跳过function(){//code}, 试图去执行();故会报错;     当js执行到(function(){// code})();时, 由于(function(){// code})是表达式, js会去对它求解得到返回值, 由于返回值是一个函数, 故而遇到()时, 便会被执行.  另外， 函数转换为表达式的方法并不一定要靠分组操作符()，我们还可以用void操作符，~操作符，!操作符……  !**function**(){ console.log(**"匿名函数自执行！！！"**); }(); |
|  | ---------------------------------  (function(){  return typeof arguments;  })();   "object"   "array"   "arguments"   "undefined"  ---------------------------------  var f = function g(){  return 23;  };  typeof g();   "number"   "undefined"   "function"   Error  ---------------------------------  (function(x){  delete x;  return x;  })(1);   1   null   undefined   Error  ---------------------------------  var y = 1, x = y = typeof x;  x;   1   "number"   undefined   "undefined"  ---------------------------------  (function f(f){  return typeof f();  })(function(){ return 1; });   "number"   "undefined"   "function"   Error  ---------------------------------  var foo = {  bar: function() {  return this.baz;  },  baz: 1  };  (function(){  return typeof arguments[0]();  })(foo.bar);   "undefined"   "object"   "number"   "function"  ---------------------------------  var foo = {  bar: function(){  return this.baz;  },  baz: 1  }  typeof (f = foo.bar)();   "undefined"   "object"   "number"   "function"  ---------------------------------  var f = (  function f(){  return "1";  },  function g(){  return 2;  }  )();  typeof f;   "string"   "number"   "function"   "undefined"  ---------------------------------  var x = 1;  if (function f(){}) {  x += typeof f;  }  x;   1   "1function"   "1undefined"   NaN  ---------------------------------  var x = [typeof x, typeof y][1];  typeof typeof x;   "number"   "string"   "undefined"   "object"  ---------------------------------  (function(foo){  return typeof foo.bar;  })({ foo: { bar: 1 } });   "undefined"   "object"   "number"   Error  ---------------------------------  (function f(){  function f(){ return 1; }  return f();  function f(){ return 2; }  })();   1   2   Error (e.g. "Too much recursion")   undefined  ---------------------------------  function f(){ return f; }  new f() instanceof f;   true   false  ---------------------------------  with (function(x, undefined){}) length;   1   2   undefined   Error  思考题：  如何改变 undefined的typeof类型 ? （即 typeof undefined 或者 typeof(undefined) 都不为"undefined" ） |
|  |  |

面试题

|  |  |
| --- | --- |
| 立即执行函数: (function(){...})() 与 (function(){...}()) 有什么区别? | |
|  | 方式一，调用函数，得到返回值。强制函数直接量执行再返回一个引用，引用在去调用执行 方式二，调用函数，得到返回值。强制运算符使函数调用执行 (function(){})(); 是 把函数当作表达式解析，然后执行解析后的函数 相当于 var a = function(){}; a(); a得到的是函数 (function(){}()); 是把函数表达式和执行当作语句直接执行、 相当于 var a = function(){}(); a得到的是结果 最终结果是一样的、 ()只是起了 自执行的作用 和 () 一样的还有很多 比如 +function (){} 这个等于 (function (){}) 一般用(function (){}) 还有个作用,就是 避免全局变量 |
|  | <https://juejin.im/post/591afe90128fe1005ce339f7>  <https://juejin.im/post/57d4ef360e3dd90069d84be9> |
|  | React和Vue对比  相同点:  数据驱动视图，提供响应式的视图组件  都有Virtual DOM，组件化开发，通过props参数进行父子组件数据的传递，都实现webComponents规范  数据流动单向  都支持服务端渲染  都有支持native的方案，React的React native，Vue的weex  不同点：  社区：React社区还是要比vue大很多；  开发模式：React在view层侵入性还是要比Vue大很多的,React严格上只针对MVC的view层，Vue则是MVVM模式的一种实现；  数据绑定：Vue有实现了双向数据绑定，React数据流动是单向的  数据渲染：对于大规模数据渲染，React要比Vue更快，渲染机制启动时候要做的工作比较多；  数据更新方面：Vue 由于采用依赖追踪，默认就是优化状态：你动了多少数据，就触发多少更新，不多也不少。React在复杂的应用里有两个选择:  (1). 手动添加 shouldComponentUpdate 来避免不需要的 vdom re-render。 (2).Components 尽可能都用 pureRenderMixin，然后采用 redux 结构 + Immutable.js；  开发风格的偏好：React 推荐的做法是 JSX + inline style，也就是把 HTML 和 CSS 全都写进 JavaScript 了，即"all in js"；Vue进阶之后推荐的是使用 webpack + vue-loader 的单文件组件格式，即html,css,js写在同一个文件；  使用场景：React配合Redux架构适合超大规模多人协作的复杂项目;Vue则适合小快灵的项目。对于需要对 DOM 进行很多自定义操作的项目，Vue 的灵活性优于 React；  Vue要比React更好上手，具体可能体现在很多人不熟悉React的JSX语法和函数式编程的思想，以及想要发挥出React的最大威力需要学习它一系列生态的缘故；  Vue着重提高开发效率,让前端程序员更快速方便的开发应用。React着重于变革开发思想，提升前端程序员编程的深度与创造力,让前端工程师成为真正的程序员而不是UI的构建者； |
|  | 1. get与post   常见解释  GET使用URL或Cookie传参。而POST将数据放在BODY中  GET的URL会有长度上的限制，则POST的数据则可以非常大  POST比GET安全，因为数据在地址栏上不可见  www  http://www.nowamagic.net/librarys/veda/detail/1919 |
|  | 1. gulp和webpack区别   gulp是一种工具，我们可以用它来优化前端的工作流程，比如自动刷新页面、combo、压缩css、js、编译less等等。具体体现为：在gulp的配置文件中书写一个个的task，webpack则是一种打包工具，或者说是一种模块化解决方案，实际上很大一部分人刚开始使用webpack的方式就是通过gulp-webpack这个插件，写好task来使用webpack对前端的一些文件进行打包;  gulp的处理任务需要自己去写，webpack则有现成的解决方案，只需要在webpack.config.js配置好即可;   1. 防止重复发送Ajax请求   用户点击之后按钮disabled;  函数节流  abort掉上一个请求。   1. 事件模型   事件捕获阶段(capturing phase)。事件从document一直向下传播到目标元素, 依次检查经过的节点是否绑定了事件监听函数，如果有则执行。  事件处理阶段(target phase)。事件到达目标元素, 触发目标元素的监听函数。  事件冒泡阶段(bubbling phase)。事件从目标元素冒泡到document, 依次检查经过的节点是否绑定了事件监听函数，如果有则执行。 |
|  | 1. 什么是原型链   每一个对象都会在内部链接到另一个对象(该对象的原型对象)，该对象有一个原型prototype，当访问对象的属性或是方法的时候，不仅仅会在原对象上查找，还会顺着原型链在原型对象的原型链上查找，直到查到null(所有原型链的顶层)为止。原型是JavaScript实现继承的基础，new关键字做的主要的事情就是将实例对象的\_\_proto\_\_属性指向原型对象的prototype。   1. 什么是闭包   闭包是javascript支持[头等函数](https://link.juejin.im/?target=https://zh.wikipedia.org/wiki/%E5%A4%B4%E7%AD%89%E5%87%BD%E6%95%B0" \t "https://juejin.im/post/_blank)的一种方式，它是一个能够引用其内部作用域变量(在本作用域第一次声明的变量)的表达式，这个表达式可以赋值给某个变量，可以作为参数传递给函数，也可以作为一个函数返回值返回。  闭包是函数开始执行的时候被分配的一个[栈帧](https://link.juejin.im/?target=http://baike.baidu.com/link?url=x9za8fl-K8Gsdc0IFBbC5fTininX3H8qVBuSPsChIJd8bmzTRXvd8scDL1uCYKLS26m6GMbXgHFC5K8yXz7nZ3eImibufpfwiBWzlBDAyT_" \t "https://juejin.im/post/_blank)，在函数执行结束返回后仍不会被释放(就好像一个栈帧被分配在堆里而不是栈里！)   1. 闭包的应用：   比如写柯里化函数的时候利用闭包，保存参数在内存中；  var currying = function(fun) {  //格式化arguments  var args = Array.prototype.slice.call(arguments, 1);  return function() {  //收集所有的参数在同一个数组中，进行计算  var \_args = args.concat(Array.prototype.slice.call(arguments));  return fun.apply(null, \_args);  };  }  ​  模拟私有变量或是私有方法；  const people = (num) => {  var num = num;  return {  increase: () => {  num++;  },  get: () => {  return num;  }  }  }  const man = people(4);  man.increase();  man.get();  避免引用错误  for (var i = 0; i < 4; i++) {  (function(\_i) {  setTimeout(function() {  console.log(\_i)  }, 1000)  })(i)  } |
|  | 1. 说说你对闭包的理解   使用闭包主要是为了设计私有的方法和变量。闭包的优点是可以避免全局变量的污染，缺点是闭包会常驻内存，会增大内存使用量，使用不当很容易造成内存泄露。在js中，函数即闭包，只有函数才会产生作用域的概念  闭包有三个特性：  1.函数嵌套函数  2.函数内部可以引用外部的参数和变量  3.参数和变量不会被垃圾回收机制回收 |
|  | 1. 图片懒加载与预加载   图片懒加载的原理就是暂时不设置图片的src属性，而是将图片的url隐藏起来，比如先写在data-src里面，等某些事件触发的时候(比如滚动到底部，点击加载图片)再将图片真实的url放进src属性里面，从而实现图片的延迟加载  图片预加载，是指在一些需要展示大量图片的网站，实现图片的提前加载。从而提升用户体验。常用的方式有两种，一种是隐藏在css的background的url属性里面，一种是通过javascript的Image对象设置实例对象的src属性实现图片的预加载。相关代码如下：  CSS预加载图片方式：  #preload-01 { background: url(http://domain.tld/image-01.png) no-repeat -9999px -9999px; }  #preload-02 { background: url(http://domain.tld/image-02.png) no-repeat -9999px -9999px; }  #preload-03 { background: url(http://domain.tld/image-03.png) no-repeat -9999px -9999px; }  Javascript预加载图片的方式：  function preloadImg(url) {  var img = new Image();  img.src = url;  if(img.complete) {  //接下来可以使用图片了  //do something here  } else {  img.onload = function() {  //接下来可以使用图片了  //do something here  };  }  } |
|  | 1. 函数节流和函数防抖   函数节流让指函数有规律的进行调用，应用场景：window.resize，游戏中子弹发射(1s只能发射一颗子弹)等；  函数防抖让函数在"调用''之后的一段时间后生效，应用场景:输入框(例：在用户停止输入的500ms后再处理用户数据)。  //函数节流/\*  \* @params {Function} fun 调用函数  \* @params {delay} number 延迟时间  \*/const throttle = (fun, delay, ...rest) => {  let last = null;  return () => {  const now = + new Date();  if (now - last > delay) {  fun(rest);  last = now;  }  }  }//实例const throttleExample = throttle(() => console.log(1), 1000);//调用  throttleExample();  throttleExample();  throttleExample();//函数防抖const debouce = (fun, delay, ...rest) => {  let timer = null;  return () => {  clearTimeout(timer);  timer = setTimeout(() => {  fun(rest);  }, delay);  }  }//实例const debouceExample = debouce(() => console.log(1), 1000);//调用  debouceExample();  debouceExample();  debouceExample();   1. 快速排序   从数列中挑出一个元素，称为"基准"（pivot），  重新排序数列，所有比基准值小的元素摆放在基准前面，所有比基准值大的元素摆在基准后面（相同的数可以到任一边）。在这个分区结束之后，该基准就处于数列的中间位置。这个称为分区（partition）操作。  [递归](https://link.juejin.im/?target=https://zh.wikipedia.org/wiki/%E9%80%92%E5%BD%92" \t "https://juejin.im/post/_blank)地（recursively）把小于基准值元素的子数列和大于基准值元素的子数列排序。  时间复杂度平均情况：O(n\log n) 最快：O(n^{2}) 空间复杂度: O(\log n)  var quickSort = function(arr) {  console.time('2.快速排序耗时');  　　if (arr.length <= 1) { return arr; }  　　var pivot = arr.splice(0, 1)[0];  　　var left = [];  　　var right = [];  　　for (var i = 0; i < arr.length; i++){  　　　　if (arr[i] < pivot) {  　　　　　　left.push(arr[i]);  　　　　} else {  　　　　　　right.push(arr[i]);  　　　　}  　　}console.timeEnd('2.快速排序耗时');  　　return quickSort(left).concat([pivot], quickSort(right));  };var arr=[3,44,38,5,47,15,36,26,27,2,46,4,19,50,48];console.log(quickSort(arr));//[2, 3, 4, 5, 15, 19, 26, 27, 36, 38, 44, 46, 47, 48, 50]   1. AMD和CMD的区别   AMD 是 RequireJS 在推广过程中对模块定义的规范化产出。 CMD 是 SeaJS 在推广过程中对模块定义的规范化产出。  对于依赖的模块，AMD 是提前执行，CMD 是延迟执行。不过 RequireJS 从 2.0 开始，也改成可以延迟执行（根据写法不同，处理方式不同）。CMD 推崇 as lazy as possible.  CMD 推崇依赖就近，AMD 推崇依赖前置。  AMD 的 API 默认是一个当多个用，CMD 的 API 严格区分，推崇职责单一。比如 AMD 里，require 分全局 require 和局部 require，都叫 require。CMD 里，没有全局 require，而是根据模块系统的完备性，提供 seajs.use 来实现模块系统的加载启动。CMD 里，每个 API 都简单纯粹。 |
|  | 1. JavaScript内存泄露的原因以及如何去手动释放内存   易出现泄露的场景  XMLHttpRequest 泄漏发生在IE7-8,释放方法，将XMLHttpRequest实例对象设置为Null；  DOM&BOM等COM对象循环绑定 泄漏发生在IE6-8，释放方法，切断循环引用，将对对象的应用设置为Null；  定时器(严格上说不能算是泄露，是被闭包持有了，是正常的表现)，对于闭包中无用的变量可以使用delete操作符进行释放；  JavaScript垃圾回收机制  引用计数  此算法把“对象是否不再需要”简化定义为“对象有没有其他对象引用到它”。如果没有引用指向该对象（零引用），对象将被垃圾回收机制回收。  限制：无法处理循环引用。在下面的例子中，两个对象被创建，并互相引用，形成了一个循环。它们被调用之后不会离开函数作用域，所以它们已经没有用了，可以被回收了。然而，引用计数算法考虑到它们互相都有至少一次引用，所以它们不会被回收。  标记清除  当变量进入环境时，例如，在函数中声明一个变量，就将这个变量标记为“进入环境”。从逻辑上讲，永远不能释放进入环境的变量所占用的内存，因为只要执行流进入相应的环境，就可能会用到它们。而当变量离开环境时，则将其标记为“离开环境”。  垃圾回收器在运行的时候会给存储在内存中的所有变量都加上标记（当然，可以使用任何标记方式）。然后，它会去掉环境中的变量以及被环境中的变量引用的变量的标记（闭包）。而在此之后再被加上标记的变量将被视为准备删除的变量，原因是环境中的变量已经无法访问到这些变量了。最后，垃圾回收器完成内存清除工作，销毁那些带标记的值并回收它们所占用的内存空间。 |
|  | 1. 柯里化函数   所谓的柯里化函数简单的说就是将本来接受多个参数的函数变为只接受一个参数的函数。柯里化函数的模板和实例如下：  var currying = function(fun) {  //格式化arguments  var args = Array.prototype.slice.call(arguments, 1);  return function() {  //收集所有的参数在同一个数组中，进行计算  var \_args = args.concat(Array.prototype.slice.call(arguments));  return fun.apply(null, \_args);  };  }  var add = currying(function() {  var args = Array.prototype.slice.call(arguments);  return args.reduce(function(a, b) {  return a + b;  });  })  add(1, 2, 4)  /\*  \* 经典面试题  \* 函数参数不定回调函数数目不定  \* 编写函数实现:  \* add(1,2,3,4,5)==15  \* add(1,2)(3,4)(5)==15  \*/  function add() {  // 第一次执行时，定义一个数组专门用来存储所有的参数  var \_args = [].slice.call(arguments);  // 在内部声明一个函数，利用闭包的特性保存\_args并收集所有的参数值  var adder = function () {  var \_adder = function() {  [].push.apply(\_args, [].slice.call(arguments));  return \_adder;  };  // 利用隐式转换的特性，当最后执行时隐式转换，并计算最终的值返回  \_adder.toString = function () {  return \_args.reduce(function (a, b) {  return a + b;  });  }  return \_adder;  }  return adder.apply(null, \_args);  }  // 输出结果，可自由组合的参数  console.log(add(1, 2, 3, 4, 5)); // 15  console.log(add(1, 2, 3, 4)(5)); // 15  console.log(add(1)(2)(3)(4)(5)); // 15 |
|  | 1. Javascript垃圾回收方法   标记清除（mark and sweep）  这是JavaScript最常见的垃圾回收方式，当变量进入执行环境的时候，比如函数中声明一个变量，垃圾回收器将其标记为“进入环境”，当变量离开环境的时候（函数执行结束）将其标记为“离开环境”。  垃圾回收器会在运行的时候给存储在内存中的所有变量加上标记，然后去掉环境中的变量以及被环境中变量所引用的变量（闭包），在这些完成之后仍存在标记的就是要删除的变量了  引用计数(reference counting)  在低版本IE中经常会出现内存泄露，很多时候就是因为其采用引用计数方式进行垃圾回收。引用计数的策略是跟踪记录每个值被使用的次数，当声明了一个 变量并将一个引用类型赋值给该变量的时候这个值的引用次数就加1，如果该变量的值变成了另外一个，则这个值得引用次数减1，当这个值的引用次数变为0的时 候，说明没有变量在使用，这个值没法被访问了，因此可以将其占用的空间回收，这样垃圾回收器会在运行的时候清理掉引用次数为0的值占用的空间。  在IE中虽然JavaScript对象通过标记清除的方式进行垃圾回收，但BOM与DOM对象却是通过引用计数回收垃圾的， 也就是说只要涉及BOM及DOM就会出现循环引用问题。 |
|  |  |
|  | 1. XSS与CSRF介绍   XSS是一种跨站脚本攻击，是属于代码注入的一种，攻击者通过将代码注入网页中，其他用户看到会受到影响(代码内容有请求外部服务器);  CSRF是一种跨站请求伪造，冒充用户发起请求，完成一些违背用户请求的行为(删帖，改密码，发邮件，发帖等)  防御方法举例:  对一些关键字和特殊字符进行过滤(<>,?,script等)，或对用户输入内容进行URL编码(encodeURIComponent);  Cookie不要存放用户名和密码，对cookie信息进行MD5等算法散列存放，必要时可以将IP和cookie绑定; |
|  |  |
|  | 1. 跨域   跨域的方式有很多种，最常用的是jsonp主要利用了script的开放策略:通过script标签引入一个js或者是一个其他后缀形式（如php，jsp等）的文件，此文件返回一个js函数的调用。缺点在于只支持get请求而且存在安全问题。  CORS跨域，关键在于服务器，如果服务器实现了CORS跨域的接口，那么就可以使用ajax(请求路径为绝对路径)进行跨域请求。CORS请求分为两种，一种是简单请求，一种是非简单请求。简单请求是指请求方法在HEAD,GET,POST三者之间并且请求头信息局限在  Accept  Accept-Language  Content-Language  Content-Type：只限于三个值application/x-www-form-urlencoded、multipart/form-data、text/plain  非简单请求请求头：  （1）Access-Control-Request-Method  该字段是必须的，用来列出浏览器的CORS请求会用到哪些HTTP方法  （2）Access-Control-Request-Headers  该字段是一个逗号分隔的字符串，指定浏览器CORS请求会额外发送的头信息字段  执行简单请求的时候，浏览器会在请求头信息增加origin字段，服务器据此来判断请求域名是否在许可范围之内，来决定是否返回Access-Control-Allow-Origin字段。响应头有以下几种：  （1）Access-Control-Allow-Origin  该字段是必须的。它的值要么是请求时Origin字段的值，要么是一个\*，表示接受任意域名的请求。  （2）Access-Control-Allow-Credentials  该字段可选。它的值是一个布尔值，表示是否允许发送Cookie。默认情况下，Cookie不包括在CORS请求之中。设为true，即表示服务器明确许可，Cookie可以包含在请求中，一起发给服务器。这个值也只能设为true，如果服务器不要浏览器发送Cookie，删除该字段即可。  （3）Access-Control-Expose-Headers  该字段可选。CORS请求时，XMLHttpRequest对象的getResponseHeader()方法只能拿到6个基本字段：Cache-Control、Content-Language、Content-Type、Expires、Last-Modified、Pragma。如果想拿到其他字段，就必须在Access-Control-Expose-Headers里面指定。  (4)Access-Control-Max-Age  Access-Control-Max-Age 首部字段指明了预检请求的响应的有效时间。  (5)Access-Control-Allow-Methods  Access-Control-Allow-Methods 首部字段用于预检请求的响应。其指明了实际请求所允许使用的 HTTP 方法。  (6)Access-Control-Allow-Headers  Access-Control-Allow-Headers首部字段用于预检请求的响应。其指明了实际请求中允许携带的首部字段。  其他方法：document.domin,html5的postMessage,window.name等 |
|  | 1. 如何解决跨域问题   JSONP：  原理是：动态插入script标签，通过script标签引入一个js文件，这个js文件载入成功后会执行我们在url参数中指定的函数，并且会把我们需要的json数据作为参数传入。  由于同源策略的限制，XmlHttpRequest只允许请求当前源（域名、协议、端口）的资源，为了实现跨域请求，可以通过script标签实现跨域请求，然后在服务端输出JSON数据并执行回调函数，从而解决了跨域的数据请求。  优点是兼容性好，简单易用，支持浏览器与服务器双向通信。缺点是只支持GET请求。  JSONP：json+padding（内填充），顾名思义，就是把JSON填充到一个盒子里  <script>  function createJs(sUrl){  var oScript = document.createElement('script');  oScript.type = 'text/javascript';  oScript.src = sUrl;  document.getElementsByTagName('head')[0].appendChild(oScript);  }  createJs('jsonp.js');  box({  'name': 'test'  });  function box(json){  alert(json.name);  }  </script>  CORS  服务器端对于CORS的支持，主要就是通过设置Access-Control-Allow-Origin来进行的。如果浏览器检测到相应的设置，就可以允许Ajax进行跨域的访问。  通过修改document.domain来跨子域  将子域和主域的document.domain设为同一个主域.前提条件：这两个域名必须属于同一个基础域名!而且所用的协议，端口都要一致，否则无法利用document.domain进行跨域  主域相同的使用document.domain  使用window.name来进行跨域  window对象有个name属性，该属性有个特征：即在一个窗口(window)的生命周期内,窗口载入的所有的页面都是共享一个window.name的，每个页面对window.name都有读写的权限，window.name是持久存在一个窗口载入过的所有页面中的  使用HTML5中新引进的window.postMessage方法来跨域传送数据  还有flash、在服务器上设置代理页面等跨域方式。个人认为window.name的方法既不复杂，也能兼容到几乎所有浏览器，这真是极好的一种跨域方法。 |
|  | 1. 浏览器缓存机制   Expires策略  Expires是Web服务器响应消息头字段，在响应http请求时告诉浏览器在过期时间前浏览器可以直接从浏览器缓存取数据，而无需再次请求。Expires 是HTTP 1.0的东西，现在默认浏览器均默认使用HTTP 1.1，所以它的作用基本忽略。  Cache-Control策略  Cache-Control与Expires的作用一致，都是指明当前资源的有效期，控制浏览器是否直接从浏览器缓读取数据还是重新发请求到服务器取数据。只不过Cache-Control的选择更多，设置更细致，如果同时设置的话，其优先级高于Expires。  以上是设置缓存时间的两种方法。那么当缓存时间过了咋整呢？有人肯定说了，那就再次发起请求啊，这是对的。问题是如果服务器资源并没有更新呢？比如说我有一个jQuery.js文件已经缓存了，当它的缓存时间到了之后服务器的jQuery.js文件也没有更新，那实际上我们直接使用本地缓存的文件就可以啊！没必要浪费带宽和时间去重新请求一个新的文件啊！这时候我们就需要再进一步看一下HTTP协议里这几个参数的作用了。  Last-Modified/If-Modified-Since  首先Last-Modified/If-Modified-Since要配合Cache-Control使用。  Last-Modified：标示这个响应资源的最后修改时间。web服务器在响应请求时，告诉浏览器资源的最后修改时间(这个参数是和Cache-Control一起过来的)。  If-Modified-Since：当资源过期时（使用Cache-Control标识的max-age），发现资源具有Last-Modified声明，则再次向web服务器请求时带上头 If-Modified-Since，表示请求时间。web服务器收到请求后发现有头If-Modified-Since ，则与被请求资源的最后修改时间进行比对。若最后修改时间较新，说明资源又被改动过，则响应整片资源内容（写在响应消息包体内），HTTP 200；若最后修改时间较旧，说明资源无新修改，则响应HTTP 304 (无需包体，节省浏览)，告知浏览器继续使用所保存的cache。  ETag/If-None-Match  Etag/If-None-Match也要配合Cache-Control使用。  Etag：web服务器响应请求时，告诉浏览器当前资源在服务器的唯一标识（生成规则由服务器觉得）。Apache中，ETag的值，默认是对文件的索引节（INode），大小（Size）和最后修改时间（MTime）进行Hash后得到的。  If-None-Match：当资源过期时（使用Cache-Control标识的max-age），发现资源具有Etage声明，则再次向web服务器请求时带上头If-None-Match（Etag的值）。web服务器收到请求后发现有头If-None-Match 则与被请求资源的相应校验串进行比对，决定返回200或304。  ETag和Last-Modified  HTTP1.1中Etag的出现主要是为了解决几个Last-Modified比较难解决的问题：  Last-Modified标注的最后修改只能精确到秒级，如果某些文件在1秒钟以内，被修改多次的话，它将不能准确标注文件的修改时间  如果某些文件会被定期生成，当有时内容并没有任何变化，但Last-Modified却改变了，导致文件没法使用缓存  有可能存在服务器没有准确获取文件修改时间，或者与代理服务器时间不一致等情形  Etag是服务器自动生成或者由开发者生成的对应资源在服务器端的唯一标识符，能够更加准确的控制缓存。Last-Modified与ETag是可以一起使用的，服务器会优先验证ETag，一致的情况下，才会继续比对Last-Modified，最后才决定是否返回304。 |
|  | 1. 什么是Etag？   当发送一个服务器请求时，浏览器首先会进行缓存过期判断。浏览器根据缓存过期时间判断缓存文件是否过期。  情景一：若没有过期，则不向服务器发送请求，直接使用缓存中的结果，此时我们在浏览器控制台中可以看到 200 OK(from cache) ，此时的情况就是完全使用缓存，浏览器和服务器没有任何交互的。  情景二：若已过期，则向服务器发送请求，此时请求中会带上①中设置的文件修改时间，和Etag  然后，进行资源更新判断。服务器根据浏览器传过来的文件修改时间，判断自浏览器上一次请求之后，文件是不是没有被修改过；根据Etag，判断文件内容自上一次请求之后，有没有发生变化  情形一：若两种判断的结论都是文件没有被修改过，则服务器就不给浏览器发index.html的内容了，直接告诉它，文件没有被修改过，你用你那边的缓存吧—— 304 Not Modified，此时浏览器就会从本地缓存中获取index.html的内容。此时的情况叫协议缓存，浏览器和服务器之间有一次请求交互。  情形二：若修改时间和文件内容判断有任意一个没有通过，则服务器会受理此次请求，之后的操作同①  ① 只有get请求会被缓存，post请求不会   1. Expires和Cache-Control   Expires要求客户端和服务端的时钟严格同步。HTTP1.1引入Cache-Control来克服Expires头的限制。如果max-age和Expires同时出现，则max-age有更高的优先级。  Cache-Control: no-cache, private, max-age=0  ETag: abcde  Expires: Thu, 15 Apr 2014 20:00:00 GMT  Pragma: private  Last-Modified: $now // RFC1123 format   1. ETag应用:   Etag由服务器端生成，客户端通过If-Match或者说If-None-Match这个条件判断请求来验证资源是否修改。常见的是使用If-None-Match。请求一个文件的流程可能如下：  ====第一次请求===  1.客户端发起 HTTP GET 请求一个文件；  2.服务器处理请求，返回文件内容和一堆Header，当然包括Etag(例如"2e681a-6-5d044840")(假设服务器支持Etag生成和已经开启了Etag).状态码200  ====第二次请求===  客户端发起 HTTP GET 请求一个文件，注意这个时候客户端同时发送一个If-None-Match头，这个头的内容就是第一次请求时服务器返回的Etag：2e681a-6-5d0448402.服务器判断发送过来的Etag和计算出来的Etag匹配，因此If-None-Match为False，不返回200，返回304，客户端继续使用本地缓存；流程很简单，问题是，如果服务器又设置了Cache-Control:max-age和Expires呢，怎么办  答案是同时使用，也就是说在完全匹配If-Modified-Since和If-None-Match即检查完修改时间和Etag之后，  服务器才能返回304.(不要陷入到底使用谁的问题怪圈)  为什么使用Etag请求头?  Etag 主要为了解决 Last-Modified 无法解决的一些问题。 |
|  | React-router原理  1.History  老浏览器的history: 主要通过hash来实现，对应createHashHistory  高版本浏览器: 通过html5里面的history，对应createBrowserHistory  node环境下: 主要存储在memeory里面，对应createMemoryHistory  内部createHistory实现：  // 内部的抽象实现function createHistory(options={}) {  ...  return {  listenBefore, // 内部的hook机制，可以在location发生变化前执行某些行为，AOP的实现  listen, // location发生改变时触发回调  transitionTo, // 执行location的改变  push, // 改变location  replace,  go,  goBack,  goForward,  createKey, // 创建location的key，用于唯一标示该location，是随机生成的  createPath,  createHref,  createLocation, // 创建location  }  }  createLocation方法:  function createLocation() {  return {  pathname, // url的基本路径  search, // 查询字段  hash, // url中的hash值  state, // url对应的state字段  action, // 分为push、replace、pop三种  key // 生成方法为: Math.random().toString(36).substr(2, length)  }  }  三种方法各自执行URL前进的方式：  createBrowserHistory: pushState、replaceState  createHashHistory: location.hash=\*\*\* location.replace()  createMemoryHistory: 在内存中进行历史记录的存储  伪代码实现:  // createBrowserHistory(HTML5)中的前进实现function finishTransition(location) {  ...  const historyState = { key };  ...  if (location.action === 'PUSH') ) {  window.history.pushState(historyState, null, path);  } else {  window.history.replaceState(historyState, null, path)  }  }// createHashHistory的内部实现function finishTransition(location) {  ...  if (location.action === 'PUSH') ) {  window.location.hash = path;  } else {  window.location.replace(  window.location.pathname + window.location.search + '#' + path  );  }  }// createMemoryHistory的内部实现  entries = [];function finishTransition(location) {  ...  switch (location.action) {  case 'PUSH':  entries.push(location);  break;  case 'REPLACE':  entries[current] = location;  break;  }  }  React-router的基本原理  URL对应Location对象，而UI是由react的 components来决定的，这样就转变成location与components之间的同步问题。 |
|  | 1. 栈和队列的区别?   栈的插入和删除操作都是在一端进行的，而队列的操作却是在两端进行的。  队列先进先出，栈先进后出。  栈只允许在表尾一端进行插入和删除，而队列只允许在表尾一端进行插入，在表头一端进行删除   1. 栈和堆的区别？   栈区（stack）— 由编译器自动分配释放 ，存放函数的参数值，局部变量的值等。  堆区（heap） — 一般由程序员分配释放， 若程序员不释放，程序结束时可能由OS回收。  堆（数据结构）：堆可以被看成是一棵树，如：堆排序；  栈（数据结构）：一种先进后出的数据结构。   1. 你觉得jQuery或zepto源码有哪些写的好的地方   (答案仅供参考)  jquery源码封装在一个匿名函数的自执行环境中，有助于防止变量的全局污染，然后通过传入window对象参数，可以使window对象作为局部变量使用，好处是当jquery中访问window对象的时候，就不用将作用域链退回到顶层作用域了，从而可以更快的访问window对象。同样，传入undefined参数，可以缩短查找undefined时的作用域链。  (function( window, undefined ) {  //用一个函数域包起来，就是所谓的沙箱  //在这里边var定义的变量，属于这个函数域内的局部变量，避免污染全局  //把当前沙箱需要的外部变量通过函数参数引入进来  //只要保证参数对内提供的接口的一致性，你还可以随意替换传进来的这个参数  window.jQuery = window.$ = jQuery;  })( window );  jquery将一些原型属性和方法封装在了jquery.prototype中，为了缩短名称，又赋值给了jquery.fn，这是很形象的写法。  有一些数组或对象的方法经常能使用到，jQuery将其保存为局部变量以提高访问速度。  jquery实现的链式调用可以节约代码，所返回的都是同一个对象，可以提高代码效率。   1. ES6的了解   新增模板字符串（为JavaScript提供了简单的字符串插值功能）、  箭头函数（操作符左边为输入的参数，而右边则是进行的操作以及返回的值Inputs=>outputs。）、  for-of（用来遍历数据—例如数组中的值。）arguments对象可被不定参数和默认参数完美代替。  ES6将promise对象纳入规范，提供了原生的Promise对象。  增加了let和const命令，用来声明变量。增加了块级作用域。  let命令实际上就增加了块级作用域。  ES6规定，v  r命令和function命令声明的全局变量，属于全局对象的属性；  let命令、const命令、class命令声明的全局变量，不属于全局对象的属性。。  还有就是引入module模块的概念 |
|  |  |
|  |  |
|  | 介绍一下box-sizing属性？  box-sizing属性主要用来控制元素的盒模型的解析模式。默认值是content-box。  content-box：让元素维持W3C的标准盒模型。元素的宽度/高度由border + padding + content的宽度/高度决定，设置width/height属性指的是content部分的宽/高  border-box：让元素维持IE传统盒模型（IE6以下版本和IE6~7的怪异模式）。设置width/height属性指的是border + padding + content  标准浏览器下，按照W3C规范对盒模型解析，一旦修改了元素的边框或内距，就会影响元素的盒子尺寸，就不得不重新计算元素的盒子尺寸，从而影响整个页面的布局。  CSS 选择符有哪些？哪些属性可以继承？优先级算法如何计算？ CSS3新增伪类有那些？  1.id选择器（ # myid）  2.类选择器（.myclassname）  3.标签选择器（div, h1, p）  4.相邻选择器（h1 + p）  5.子选择器（ul > li）  6.后代选择器（li a）  7.通配符选择器（ \* ）  8.属性选择器（a[rel = "external"]）  9.伪类选择器（a: hover, li:nth-child）  优先级为:  !important > id > class > tag  important 比 内联优先级高,但内联比 id 要高  CSS3新增伪类举例：  p:first-of-type 选择属于其父元素的首个 <p> 元素的每个 <p> 元素。  p:last-of-type 选择属于其父元素的最后 <p> 元素的每个 <p> 元素。  p:only-of-type 选择属于其父元素唯一的 <p> 元素的每个 <p> 元素。  p:only-child 选择属于其父元素的唯一子元素的每个 <p> 元素。  p:nth-child(2) 选择属于其父元素的第二个子元素的每个 <p> 元素。  :enabled :disabled 控制表单控件的禁用状态。  :checked 单选框或复选框被选中。    CSS3有哪些新特性？  CSS3实现圆角（border-radius），  阴影（box-shadow），  对文字加特效（text-shadow、），  线性渐变（gradient），  旋转（transform）  transform:rotate(9deg) scale(0.85,0.90) translate(0px,-30px) skew(-9deg,0deg);//旋转,缩放,定位,倾斜  增加了更多的CSS选择器  多背景  rgba  在CSS3中唯一引入的伪元素是::selection.  媒体查询  多栏布局  border-image  新增了一种盒模型计算方式：box-sizing。盒模型默认的值是content-box, 新增的值是padding-box和border-box，几种盒模型计算元素宽高的区别如下：  对BFC规范的理解？  BFC，块级格式化上下文，一个创建了新的BFC的盒子是独立布局的，盒子里面的子元素的样式不会影响到外面的元素。在同一个BFC中的两个毗邻的块级盒在垂直方向（和布局方向有关系）的margin会发生折叠。  （W3C CSS 2.1 规范中的一个概念，它决定了元素如何对其内容进行布局，以及与其他元素的关系和相互作用。 |
|  | 1. 浮动元素引起的问题和解决办法？   浮动元素引起的问题：  （1）父元素的高度无法被撑开，影响与父元素同级的元素  （2）与浮动元素同级的非浮动元素（内联元素）会跟随其后  （3）若非第一个元素浮动，则该元素之前的元素也需要浮动，否则会影响页面显示的结构  解决方法：  1.额外标签法，<div style="clear:both;"></div>（缺点：不过这个办法会增加额外的标签使HTML结构看起来不够简洁。）  2.使用after伪类  #parent:after{  content:".";  height:0;  visibility:hidden;  display:block;  clear:both;  }  3.浮动外部元素  4.设置overflow为hidden或者auto   1. DOM操作——怎样添加、移除、移动、复制、创建和查找节点。   1）创建新节点  createDocumentFragment() //创建一个DOM片段  createElement() //创建一个具体的元素  createTextNode() //创建一个文本节点  2）添加、移除、替换、插入  appendChild()  removeChild()  replaceChild()  insertBefore() //并没有insertAfter()  3）查找  getElementsByTagName() //通过标签名称  getElementsByName() //通过元素的Name属性的值(IE容错能力较强， 会得到一个数组，其中包括id等于name值的)  getElementById() //通过元素Id，唯一性 |
|  | js延迟加载的方式有哪些？  defer和async、动态创建DOM方式（创建script，插入到DOM中，加载完毕后callBack）、按需异步载入js  call() 和 apply() 的区别和作用？  作用：动态改变某个类的某个方法的运行环境（执行上下文）。  区别参见：[JavaScript学习总结（四）function函数部分](https://link.juejin.im/?target=http://segmentfault.com/blog/trigkit4/1190000000660786%23articleHeader15" \t "https://juejin.im/post/_blank)  哪些操作会造成内存泄漏？  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。  垃圾回收器定期扫描对象，并计算引用了每个对象的其他对象的数量。如果一个对象的引用数量为 0（没有其他对象引用过该对象），或对该对象的惟一引用是循环的，那么该对象的内存即可回收。  setTimeout 的第一个参数使用字符串而非函数的话，会引发内存泄漏。  闭包、控制台日志、循环（在两个对象彼此引用且彼此保留时，就会产生一个循环）  详见：[详解js变量、作用域及内存](https://link.juejin.im/?target=http://segmentfault.com/blog/trigkit4/1190000000687844" \t "https://juejin.im/post/_blank) |
|  |  |
|  | 实现一个函数clone，可以对JavaScript中的5种主要的数据类型（包括Number、String、Object、Array、Boolean）进行值复制  Object.prototype.clone = function(){  var o = this.constructor === Array ? [] : {};  for(var e in this){  o[e] = typeof this[e] === "object" ? this[e].clone() : this[e];  }  return o;  } |
|  | 说说严格模式的限制  严格模式主要有以下限制：  变量必须声明后再使用  函数的参数不能有同名属性，否则报错  不能使用with语句  不能对只读属性赋值，否则报错  不能使用前缀0表示八进制数，否则报错  不能删除不可删除的属性，否则报错  不能删除变量delete prop，会报错，只能删除属性delete global[prop]  eval不会在它的外层作用域引入变量  eval和arguments不能被重新赋值  arguments不会自动反映函数参数的变化  不能使用arguments.callee  不能使用arguments.caller  禁止this指向全局对象  不能使用fn.caller和fn.arguments获取函数调用的堆栈  增加了保留字（比如protected、static和interface）  设立"严格模式"的目的，主要有以下几个：  消除Javascript语法的一些不合理、不严谨之处，减少一些怪异行为;  消除代码运行的一些不安全之处，保证代码运行的安全；  提高编译器效率，增加运行速度；  为未来新版本的Javascript做好铺垫。  注：经过测试IE6,7,8,9均不支持严格模式。  如何删除一个cookie  1.将时间设为当前时间往前一点。  var date = new Date();  date.setDate(date.getDate() - 1);//真正的删除  setDate()方法用于设置一个月的某一天。  2.expires的设置  document.cookie = 'user='+ encodeURIComponent('name') + ';expires = ' + new Date(0) |
|  | 说说你对MVC和MVVM的理解  MVC  View 传送指令到 Controller  Controller 完成业务逻辑后，要求 Model 改变状态  Model 将新的数据发送到 View，用户得到反馈  所有通信都是单向的。  Angular它采用双向绑定（data-binding）：View的变动，自动反映在 ViewModel，反之亦然。  组成部分Model、View、ViewModel  View：UI界面  ViewModel：它是View的抽象，负责View与Model之间信息转换，将View的Command传送到Model；  Model：数据访问层 |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |

