1 变量

1.1 变量类型

基本类型：undefined、string、number、boolean、null
引用类型：object

1.2 变量赋值

从一个基本类型变量向另一个变量赋值时，会在内存中新建一个地址，存放新的变量和复制过来的值；
从一个引用类型变量向另一个变量赋值时，同上，但引用类型的值，实际上是一个指针，与初始变量指向同一个堆内存的对象。因此，这两个变量会互相影响。

1.3 typeof 判断变量类型：

typeof 判断变量类型可能返回以下几种：
‘string’ ‘number’ ‘boolean’ ‘undefined’ ‘null’ ‘object’ ‘function’ ‘symbol’

1.4 强制类型转换情形：

- 字符串拼接（将其他类型转换为字符串）
- == 运算符（两边类型不同时，先转换类型再进行值比较）
- if 语句（运算符两边条件语句转换为布尔类型）
- 逻辑运算（运算符两边转换为布尔类型）

何时使用 == ：
obj.a == null 相当于 obj.a === null || obj.a === undefined

1.5 JSON

JSON 是 JS 中的内置对象，也是一种数据格式。
JSON.**stringify**({a:10,b:20})：将一个 JS 值（对象/数组）转换为 JSON 字符串；
JSON.**parse**(‘{“a”:10,”b”:20}’)：解析 JSON 字符串，构造由字符串描述的 JS 值或对象

2 原型和原型链

2.1 构造函数

构造函数 ，是一种特殊的方法。主要用来在创建对象时初始化对象， 即为对象成员变量赋初始值，总与 new 运算符一起使用在创建对象的语句中。

实例化一个对象的过程：

new 一个新对象 => this 指向这个对象 => 执行代码（对 this 赋值） => 返回 this

2.2 原型

隐式原型：所有引用类型（数组、对象、函数），都有一个 _ proto _ 属性，属性值是一个普通对象。

显式原型：所有的函数都有一个 prototype 属性，属性值是一个普通对象。

所有引用类型其隐式原型都指向它的构造函数的显式原型

obj._ proto _ === Object.prototype

- 当试图得到一个对象的某个属性时，如果这个对象本身没有这个属性，会去它的 _ proto _ （即其构造函数的 prototype ）中寻找。

hasOwnProperty： 只判断对象本身是否包含某属性，不去其原型链中寻找。

2.3 原型链

instanceof

变量 instanceof 函数：判断一个函数是否是变量的构造函数

判断逻辑： 变量的 _ proto _ 一层一层往上，看能否对应到 函数.prototype

2.3.1 判断 JavaScript 数据类型的方法

typeof 操作符：返回给定变量的数据类型，返回值为字符串

```
返回字符串 —— 数据类型

'undefined'——Undefined

'boolean'——Boolean

'string'——String

'number'——Number

'symbol'——Symbol
```

```
'object'——Object / Null （Null 为空对象的引用）

'function'——Function
```

语法： object instanceof constructor

检测 constructor.prototype 是否存在于参数 object 的原型链上，是则返回 true，不是则返回 false。

注：instanceof 只能用来判断对象类型，则后面一定要是对象类型，且大小写不能错。

```
    alert([1,2,3] instanceof Array) ---------------> true

    alert(new Date() instanceof Date)

    alert(function(){this.name="22";} instanceof Function) ------------> true

    alert(function(){this.name="22";} instanceof function) ------------> false
```

constructor：

返回对象对应的构造函数

```
alert({}.constructor === Object);  =>  true

alert([].constructor === Array);  =>  true

alert('abcde'.constructor === String);  =>  true

alert((1).constructor === Number);  =>  true

alert(true.constructor === Boolean);  =>  true

alert(false.constructor === Boolean);  =>  true

alert(function s(){}.constructor === Function);  =>  true

alert(new Date().constructor === Date);  =>  true

alert(new Array().constructor === Array);  =>  true

alert(new Error().constructor === Error);  =>  true

alert(document.constructor === HTMLDocument);  =>  true

alert(window.constructor === Window);  =>  true
```

```
alert(Symbol().constructor);    =>    undefined
```

Symbol 值通过 Symbol 函数生成，是一个原始类型的值，不是对象，不能通过 constructor 判断；

null 和 undefined 是无效的对象，没有 constructor，因此无法通过这种方式来判断。

函数的 constructor 不稳定。
当一个函数被定义时，JS 引擎会为其添加 prototype 原型，然后在 prototype 上添加一个 constructor 属性，并让其指向函数的引用。
但函数的 prototype 被重写后，原有的 constructor 引用会丢失。再次新建一个次函数的实例后，其 constructor 指向的内容已发生改变。
因此为了规范开发，在重写对象原型时，一般都需要重新给 constructor 赋值，以保证对象实例的类型不被更改。

Object.prototype.toString()：

toString() 是 Object 的原型方法，调用该方法，默认返回当前对象的 [[Class]] 。这是一个内部属性，其格式为 [object Xxx] ，是一个字符串，其中 Xxx 就是对象的类型。
对于 Object 对象，直接调用 toString() 就能返回 [object Object] 。而对于其他对象，则需要通过 call / apply 来调用才能返回正确的类型信息。

```
Object.prototype.toString.call(new Date);//[object Date]

Object.prototype.toString.call(new String);//[object String]

Object.prototype.toString.call(Math);//[object Math]

Object.prototype.toString.call(undefined);//[object Undefined]

Object.prototype.toString.call(null);//[object Null]

Object.prototype.toString.call('') ;   // [object String]

Object.prototype.toString.call(123) ;    // [object Number]

Object.prototype.toString.call(true) ; // [object Boolean]

Object.prototype.toString.call(Symbol()); //[object Symbol]

Object.prototype.toString.call(new Function()) ; // [object Function]

Object.prototype.toString.call(new Date()) ; // [object Date]

Object.prototype.toString.call([]) ; // [object Array]

Object.prototype.toString.call(new RegExp()) ; // [object RegExp]

Object.prototype.toString.call(new Error()) ; // [object Error]
```

```
Object.prototype.toString.call(document) ; // [object HTMLDocument]

Object.prototype.toString.call(window) ; //[object global] window 是全局对象 global 的引用
```

类型判断小结：
1）typeof 更适合判断基本类型数据，因为对于引用类型数据，typeof 只会返回 ‘function’ 或 ‘object’，不会返回其他的数组等类型；
2）instanceof 只能用来判断实例类型，包括 Array、Date 等，判断基本类型会永远返回 true，无意义；
3）constructor 不能用来判断 Symbol、Null 及 Undefined 类型
4）注：new String()、new Number() 生成的实际上为对象，但只能通过 typeof 能判断出来，后面三种只会返回 String 或 Number，无法判断是基本类型或是引用类型。

2.4 原型继承

写一个原型继承的例子

```
function Elem(id){

    this.Elem = id ? document.getElementById(id) : null;

}

Elem.prototype.on = function(type,fn){

    if(this.Elem){

        this.Elem.addEventListener(type, fn)

        return this //链式操作

    }

}

let html = new Elem('a');

html.on('click', function(){

    console.log('click')

})
```

3 作用域和闭包

3.1 this

this 在执行时才能确认值，定义时无法确认

3.1.1 this 的几种使用场景

- **构造函数：** this 指向新对象
- **对象方：** this 指向这个对象本身
- **函数：** this 指向 window
- **call apply bind：** this 指向传入的 对象

call apply bind

```
//call & apply 立即调用

//call()

function.call(thisObj[, arg1[, arg2[, [,...argN]]]]);

//apply()

function.apply(thisObj[, argArray])

//bind()    暂不调用，把函数中 this 的指向定为 thisObj，便于后面调用

function.bind(thisObj
```

3.2 作用域

ES6 之前没有块级作用域，只有函数和全局作用域。
ES6 开始有了块级作用域：let、const。

简述 var 与 let 的区别：
1）let 是块级作用域，var 没有块级作用域，let 的作用域更小；
2）let 无变量提升。下面定义的变量，在上面使用会报错；var 有变量提升，下面定义的变量，在上面值为 undefined。
3）let 同一个变量只能声明一次，而 var 可声明多次。
let 和 const 均有块级作用域，let 声明的是变量，const 声明的是常量，不可改变。

3.3 作用域链

函数内未定义的变量，称为**自由变量**，函数执行时，会去函数定义时的父级作用域寻找自由变量，若没有找到，则逐级向上寻找。
函数的父级作用域由其定义时所在位置决定

3.4 闭包

闭包是由函数以及创建该函数的词法环境组合而成。
这个环境包含了这个闭包创建时所能访问的所有局部变量。

```
//函数作为返回值，参数作为局部变量传入

function makeAdder(x) {

  return function(y) {

    return x + y;

  };}



var add5 = makeAdder(5);

var add10 = makeAdder(10);



console.log(add5(2));  // 7

console.log(add10(2)); // 12
```

add5 和 add10 都是闭包。它们共享相同的函数定义，但是保存了不同的词法环境。

闭包小总结：
局部变量相同时，可实现单例模式；
局部变量作为参数传递时，具有不同的词法环境；

闭包缺点：
1）性能：闭包在**处理速度和内存消耗**方面对脚本性能具有负面影响。在创建新的对象或者类时，方法通常应该关联于对象的原型，而不是定义到对象的构造器中。每次构造器被调用时，闭包内的方法都会被重新赋值一次（也就是每个对象的创建）。闭包会使得函数中的变量都被保存在内存中，内存消耗很大，所以不能滥用；

2）闭包会在父函数外部改变父函数内部变量的值，不便于维护。

4 异步和单线程

4.1 概述

JavaScript 是单线程： 同一时间支持能做一件事儿。
异步： JavaScript 有些部分是异步的，比如定时器、网络请求、事件绑定，这些会放在任务队列中，等其他任务结束后再来看是否执行，实现了异步操作。不会发生事件阻塞。（alert 同步，setTimeout 异步）

4.2 前端中使用异步的场景

- 定时任务： setTimeout、 setInterval
- 网络请求： ajax 请求，动态 `<img>` 加载
- 事件绑定

5 JS 操作节点

5.1 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

5.2 添加、移除、替换、插入

append ()
removeChild()
replaceChild()
insertBefore()

5.3 查找

getElementsByTagName_r() //通过标签名称
getElementsByName() //通过元素的 Name 属性的值
getElementByClassName() //通过元素 class
getElementById() //通过元素 Id，唯一性

6. JavaScript 事件

6.1 事件冒泡：从内到外

事件开始时由最具体的元素接收，然后逐级向上传播到较为不具体的节点。
事件捕获：由外到内
不太具体的节点更早接收到事件，具体的则较晚，与事件冒泡顺序相反。
DOM 事件流：
‘ DOM2 级事件 ’ 规定的事件流包括三个阶段：事件捕获 =>处于目标 => 事件冒泡

6.2 HTML 事件处理程序：

```
<input type="button" value="Click Me" onclick="alert(event.value)" /> //输出 "Click Me"
```

缺点：
1）HTML 与 JavaScript 代码紧密耦合，不利于代码维护
2）存在时差问题，事件触发时函数不一定解析完
DOM0 级事件处理程序：
将一个函数赋值给一个事件处理程序属性。this 指向当前元素。
以这种方式添加的事件处理程序会在事件流的冒泡阶段被处理。
DOM2 级事件处理程序：
“DOM2 级事件”定义了两个方法，用于处理指定和删除事件处理程序的操作：addEventListener()和 removeEventListener()。所有 DOM 节点中都包含这两个方法，并且它们都接受 3 个参数：要处理的事件名、作为事件处理程序的函数和一个布尔值。最后这个布尔值参数如果是 true，表示在捕获阶段调用事件处理程序；如果是 false，表示在冒泡阶段调用事件处理程序。this 指向当前元素。
优点：可以添加多个事件处理程序，事件处理程序会按照添加他们的顺序触发。
注：移除事件处理程序 通过 addEventListener()添加的事件处理程序只能使用 removeEventListener()来移除；移除时传入的参数与添加处理程序时使用的参数相同。这也意味着通过 addEventListener()添加的匿名函数将无法移除

大多数情况下，都是将事件处理程序添加到事件流的冒泡阶段，这样可以最大限度地兼容各种浏览器。

6.3 DOM 中的事件对象：

1）type 属性：用于获取事件类型 (event.type)
2）target 属性：用于获取事件目标 (event.target / event.target.nodeName)
3）stopPropagation():用于阻止事件冒泡
4）preventDefault():阻止事件的默认行为
IE 中的事件对象：
1）type 属性：用于获取事件类型 (event.type)
2）srcElement 属性：用于获取事件的目标

3）cancelBubble 属性：用于阻止事件冒泡(设置为 true 表示阻止冒泡 设置 false 表示不阻止冒泡)
4）returnValue 属性：用于阻止事件的默认行为（设置为 false 表示阻止事件默认行为）

6.4 事件类型

DOM3 级事件规定了以下几类事件：
UI 事件：
load、unload、resize、scroll
焦点事件：
blur、focus、focusin（与 focus 等价，会冒泡）、focusout（与 blur 等价，会冒泡）
鼠标和滚轮事件：
click、dblclick、mousedown、mouseenter、mouseleave、mousemove（包括被选元素即其子元素）、mouseover、mouseout、mouseup、mousewheel （鼠标滚轮）
键盘和文本事件：
keydown、keypress、keyup

6.5 内存和性能

每个函数都是对象，都会占用内存；内存中对象越多，性能就越差。
事件委托：
利用事件冒泡，只指定一个事件处理程序，就可以管理某一类型的所有事件。
eg：`<ul id="myLinks">`
`<li id="goSomewhere">Go somewhere</li>`
`<li id="doSomething">Do something</li>`
`<li id="sayHi">Say hi</li>`
`</ul>`
按传统做法，要给三个 li 都添加点击事件，需要逐个添加事件处理程序，利用事件委托，只需要再 DOM 树中尽量最高的层次上添加一个事件处理程序，即给 ul 添加事件处理程序，再通过判断 id 进行处理。
移除事件处理程序：
每当将事件处理程序指定给元素时，运行中的浏览器代码与支持页面交互的 JavaScript 代码之间就会建立一个连接。这种连接越多，页面执行起来就越慢。
除采用事件委托技术，限制建立的连接数量外，在不需要的时候移除事件处理程序，也是解决这个问题的一种方案。

6.6 DOM 中模拟事件：createEvent()、initMouseEvent、dispatchEvent

```
//模拟鼠标事件

var btn = document.getElementById("myBtn");
```

```
//创建事件对象

var event = document.createEvent("MouseEvents");

//初始化事件对象

event.initMouseEvent("click", true, true, document.defaultView, 0, 0, 0, 0, 0,

false, false, false, false, 0, null);

//触发事件

btn.dispatchEvent(event);

//模拟键盘事件:按住 Shift 的同时又按下 A 键

var textbox = document.getElementById("myTextbox"),

    event;

//以 DOM3 级方式创建事件对象

if (document.implementation.hasFeature("KeyboardEvents", "3.0")){

    event = document.createEvent("KeyboardEvent");

//初始化事件对象

    event.initKeyboardEvent("keydown", true, true, document.defaultView, "a",0, "Shift",
0);

}

//触发事件

textbox.dispatchEvent(event);
```

6.7 IE 中事件模拟：createEventObject()、fireEvent

```
var btn = document.getElementById("myBtn");

//创建事件对象

var event = document.createEventObject();
```

```
//初始化事件对象

event.screenX = 100;

event.screenY = 0;

event.clientX = 0;

event.clientY = 0;

event.ctrlKey = false;

event.altKey = false;

event.shiftKey = false;

event.button = 0;

//触发事件

btn.fireEvent("onclick", event);
```

查看 JavaScrpit 事件篇详情

7 AJAX

7.1 XMLHTTPRequest 对象

Ajax 其核心由 JavaScript、XMLHTTPRequest、DOM 对象组成，通过 XmlHttpRequest 对象来向服务器发异步请求，从服务器获得数据，然后用 JavaScript 来操作 DOM 而更新页面。
这其中最关键的一步就是从服务器获得请求数据。

7.2 get 和 post 的区别

GET 请求会将参数跟在 URL 后进行传递，而 POST 请求则是作为 HTTP 消息的实体内容发送给 WEB 服务器

区别：
- post 更安全（不会作为 url 的一部分，不会被缓存、保存在服务器日志、以及浏览器浏览记录中）
- post 发送的数据更大（get 有 url 长度限制）
- post 能发送更多的数据类型（get 只能发送 ASCII 字符）
- post 比 get 慢

- post 用于修改和写入数据，get 一般用于搜索排序和筛选之类的操作（淘宝，支付宝的搜索查询都是 get 提交），目的是资源的获取，读取数据

若符合下列任一情况，则用 POST 方法：
- 请求的结果有持续性的副作用，例如，数据库内添加新的数据行
- 若使用 GET 方法，则表单上收集的数据可能让 URL 过长
- 要传送的数据不是采用 7 位的 ASCII 编码

若符合下列任一情况，则用 GET 方法：
- 请求是为了查找资源，HTML 表单数据仅用来帮助搜索
- 请求结果无持续性的副作用
- 收集的数据及 HTML 表单内的输入字段名称的总长不超过 1024 个字符

8 跨域

8.1 同源策略

同源： 协议+域名+端口 三者相同。

同源策略：同源策略限制以下几种行为：

- Cookie、LocalStorage 和 IndexDB 无法读取
- DOM 和 Js 对象无法获得
- AJAX 请求不能发送

同源策略是浏览器最核心也最基本的安全功能，如果缺少了同源策略，浏览器很容易受到 XSS、CSFR 等攻击。

8.2 跨域解决方案

1）JSONP
动态创建 script，再请求一个带参网址实现跨域通信。
缺点：只能实现 get 一种请求。

2）postMessage
postMessage 是 HTML5 XMLHttpRequest Level 2 中的 API，且是为数不多可以跨域操作的 window 属性之一，它可用于解决以下方面的问题：
a.） 页面和其打开的新窗口的数据传递
b.） 多窗口之间消息传递
c.） 页面与嵌套的 iframe 消息传递
d.） 上面三个场景的跨域数据传递

用法：postMessage(data,origin)方法接受两个参数

3）跨域资源共享（CORS）

* *只服务端请求头设置 Access-Control-Allow-Origin 即可，前端无须设置，若要带 cookie 请求：前后端都需要设置。