

JS用法

HTML载入外部JS

使用简洁的格式载入 JavaScript 文件（type 属性不是必须的）：

```
1 <script src="myscript.js"></script>
```

使用 JavaScript 访问 HTML 元素

```
1 var obj = getElementById("Demo")
```

HTML 中的 Javascript 脚本代码必须位于 与 标签之间。

Javascript 脚本代码可被放置在 HTML 页面的 和 部分中。

标签

如需在 HTML 页面中插入 JavaScript, 请使用 标签。

会告诉 JavaScript 在何处开始和结束。

IO

JavaScript 没有任何打印或者输出的函数。

JavaScript 可以通过不同的方式来输出数据：

- 使用 **window.alert()** 弹出警告框。
- 使用 **document.write()** 方法将内容写到 HTML 文档中。
- 使用 **innerHTML** 写入到 HTML 元素。
- 使用 **console.log()** 写入到浏览器的控制台。
- 如需从 JavaScript 访问某个 HTML 元素, 您可以使用 `document.getElementById("id")` 方法。请使用 "id" 属性来标识 HTML 元素, 并 `innerHTML` 来获取或插入元素内容：

```
1 <!DOCTYPE html>
2 <html>
3 <body>
4
5 <h1>我的第一个 Web 页面</h1>
6
7 <p id="demo">我的第一个段落</p>
8
9 <script>
10 document.getElementById("demo").innerHTML = "段落已修改。";
11 </script>
12
13 </body>
```

语法

字面量

```
1 //数字字面量, 字符串字面量, 表达式字面量
2
3 //数组字面量
4 var array = [40, 100, 1, 5, 25, 10];
5
6 //对象字面量
7 var object = {
8     firstName: "John",
9     lastName: "Doe",
10    age: 50,
11    eyeColor: "blue"
12 };
13
14 //函数字面量
15 function myFunction(a, b)
16 {
17     return a * b;
18 }
```

变量

```
1 var x;
```

数据类型

JavaScript 有多种数据类型：数字，字符串，数组，对象等等：

```
1 var length = 16; // Number 通过数字字面量赋值
2 var points = x * 10; // Number 通过表达式字面量赋值
3 var lastName = "Johnson"; // String 通过字符串字面量赋值
4 var cars = ["Saab", "Volvo", "BMW"]; // Array 通过数组字面量赋值
5 var person = {firstName: "John", lastName: "Doe"}; // Object 通过对象字面量赋值
```

换行符

您可以在文本字符串中使用**反斜杠 (\)** 对代码行进行换行。：

```
1 document.write("你好 \
2 世界!");
```

全等运算符

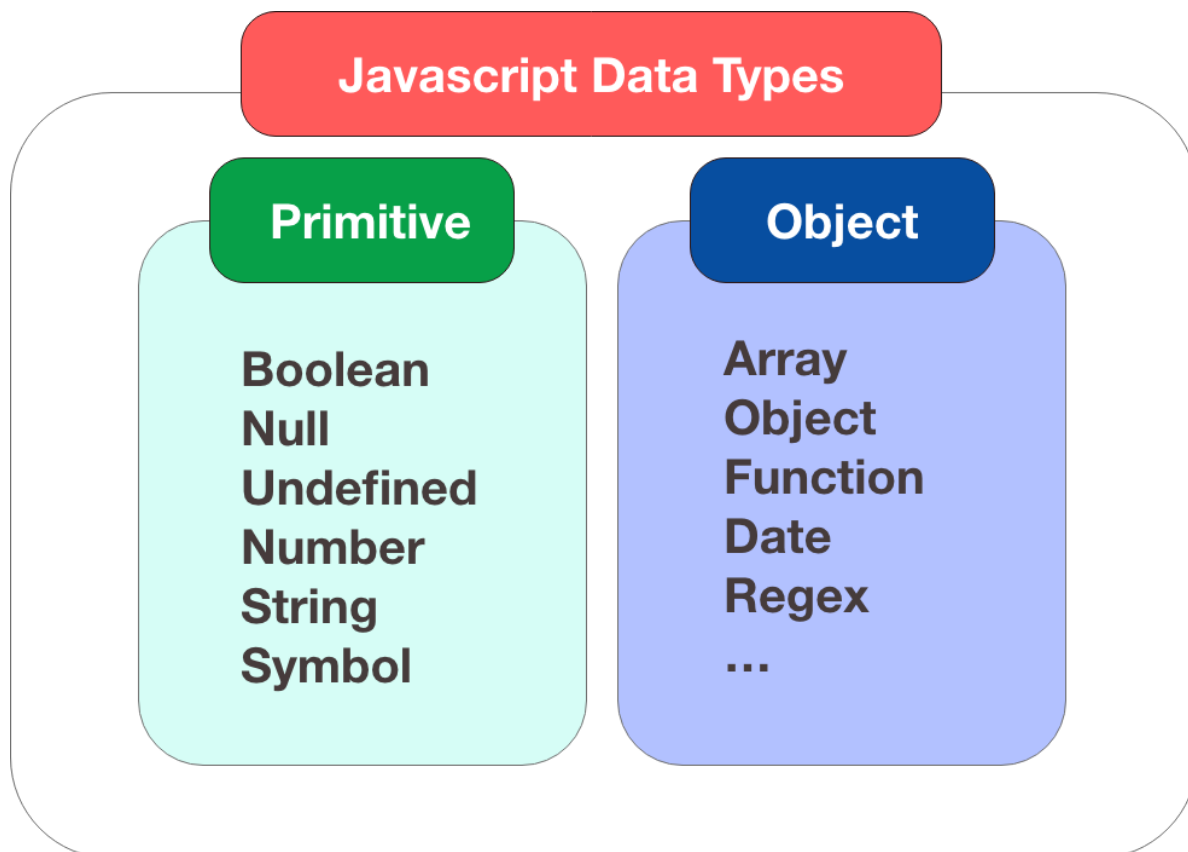
全等运算符（三个=）会检查它的两个操作数是否相等，并且返回一个布尔值结果。

JS数据类型

值类型(基本类型): 字符串 (String)、数字(Number)、布尔(Boolean)、空 (Null)、未定义 (Undefined)、Symbol。

Symbol 是 ES6 引入了一种新的原始数据类型，表示独一无二的值。

引用数据类型 (对象类型): 对象(Object)、数组(Array)、函数(Function)，还有两个特殊的对象：正则 (RegExp) 和日期 (Date)。



动态类型

JavaScript 拥有动态类型。这意味着相同的变量可用作不同的类型：

```
1 var x;           // x 为 undefined
2 var x = 5;       // 现在 x 为数字
3 var x = "John";  // 现在 x 为字符串
```

变量的数据类型可以使用 **typeof** 操作符来查看：

```
1 typeof "John"    // 返回 string
2 typeof 3.14      // 返回 number
3 typeof false     // 返回 boolean
4 typeof [1,2,3,4] // 返回 object
5 typeof {name:'John', age:34} // 返回 object
```

字符串

字符串是存储字符 (比如 "Bill Gates") 的变量。

```
1 //字符串可以是引号中的任意文本。您可以使用单引号或双引号
2 var carname="Volvo XC60";
3 var carname='Volvo XC60';
4
5 //可以在字符串中使用引号，只要不匹配包围字符串的引号即可：
6 var answer="He is called 'Johnny'";
7 var answer='He is called "Johnny"';
8 //也可以在字符串添加转义字符来使用引号
9 var x = 'It\'s alright';
10 var y = "He is called \"Johnny\"";
```

```
1 //可以使用索引位置来访问字符串中的每个字符：
2 var character = carname[7];
3
4 //可以使用内置属性 length 来计算字符串的长度：
5 var txt = "ABCDEFGHJKLMNOPQRSTUVWXYZ";
6 var sln = txt.length;
```

通常，JavaScript 字符串是原始值，可以使用字符创建：**var firstName = "John"**

但我们也可以使用 **new** 关键字将字符串定义为一个对象，但我们不要创建 **String** 对象。它会拖慢执行速度，并可能产生其他副作用！

```
1 var firstName = new String("John")
```

字符串属性和方法

原始值字符串，如 "John"，没有属性和方法（因为他们不是对象）。

原始值可以使用 JavaScript 的属性和方法，因为 JavaScript 在执行方法和属性时可以把原始值当作对象。

属性	描述
constructor	返回创建字符串属性的函数
length	返回字符串的长度
prototype	允许您向对象添加属性和方法

更多方法实例：[JavaScript String 对象](#)

方法	描述
charAt()	返回指定索引位置的字符
charCodeAt()	返回指定索引位置字符的 Unicode 值
concat()	连接两个或多个字符串，返回连接后的字符串
fromCharCode()	将 Unicode 转换为字符串
indexOf()	返回字符串中检索指定字符第一次出现的位置
lastIndexOf()	返回字符串中检索指定字符最后一次出现的位置
localeCompare()	用本地特定的顺序来比较两个字符串
match()	找到一个或多个正则表达式的匹配
replace()	替换与正则表达式匹配的子串
search()	检索与正则表达式相匹配的值
slice()	提取字符串的片断，并在新的字符串中返回被提取的部分
split()	把字符串分割为子字符串数组
substr()	从起始索引号提取字符串中指定数目的字符
substring()	提取字符串中两个指定的索引号之间的字符
toLocaleLowerCase()	根据主机的语言环境把字符串转换为小写，只有几种语言（如土耳其语）具有地方特有的大小写映射
toLocaleUpperCase()	根据主机的语言环境把字符串转换为大写，只有几种语言（如土耳其语）具有地方特有的大小写映射
toLowerCase()	把字符串转换为小写
toString()	返回字符串对象值
toUpperCase()	把字符串转换为大写
trim()	移除字符串首尾空白
valueOf()	返回某个字符串对象的原始值

数组

数组创建有三种方式：

```
1 var cars=new Array();
2 cars[0]="Saab";
3 cars[1]="Volvo";
4 cars[2]="BMW";
5
6 var cars=new Array("Saab","Volvo","BMW");
7
8 var cars=["Saab","Volvo","BMW"];
```

undefined 和 null

在 JavaScript 中

null是一个只有一个值的特殊类型。表示一个空对象引用。类型为object

undefined 是一个没有设置值的变量。类型为undefined

```
1 null 和 undefined 的值相等，但类型不等：
2
3 typeof undefined           // undefined
4 typeof null                 // object
5 null === undefined         // false
6 null = undefined           // true
```

let

let 声明的变量只在 **let** 命令所在的代码块内有效。

在 ES6 之前，JavaScript 只有两种作用域：**全局变量** 与 **函数内的局部变量**，现在引入了块级作用域。

使用 **var** 关键字声明的变量不具备块级作用域的特性，它在 **{ }** 外依然能被访问到

```
1 {
2     var x = 2;
3 }
4 // 这里可以使用 x 变量
```

```
1 var x = 10;
2 // 这里输出 x 为 10
3 {
4     var x = 2;
5     // 这里输出 x 为 2
6 }
7 // 这里输出 x 为 2
```

使用 **let** 关键字来实现块级作用域。

let 声明的变量只在 **let** 命令所在的代码块 **{ }** 内有效，在 **{ }** 之外不能访问。

```
1 {
2     let x = 2;
3 }
4 // 这里不能使用 x 变量
```

```
1 var x = 10;
2 // 这里输出 x 为 10
3 {
4     let x = 2;
5     // 这里输出 x 为 2
6 }
7 // 这里输出 x 为 10
```

局部变量

在函数体内使用 **var** 和 **let** 关键字声明的变量有点类似。

它们的作用域都是 **局部的**：

```
1 // 使用 var
2 function myFunction() {
3     var carName = "Volvo"; // 局部作用域
4 }
5
6 // 使用 let
7 function myFunction() {
8     let carName = "Volvo"; // 局部作用域
9 }
```

全局变量

在函数体外或代码块外使用 **var** 和 **let** 关键字声明的变量也有点类似。

它们的作用域都是 **全局的**：

```
1 // 使用 var
2 var x = 2; // 全局作用域
3
4 // 使用 let
5 let x = 2; // 全局作用域
```

作用域

在 JavaScript 中，全局作用域是针对 JavaScript 环境。

在 HTML 中，全局作用域是针对 window 对象。

使用 **var** 关键字声明的全局作用域变量属于 window 对象：

```
1 var carName = "Volvo";
2 // 可以使用 window.carName 访问变量
```

使用 **let** 关键字声明的全局作用域变量不属于 window 对象：

```
1 let carName = "Volvo";
2 // 不能使用 window.carName 访问变量
```

重置变量

使用 **var** 关键字声明的变量在任何地方都可以修改：

```
1 var x = 2;
2
3 // x 为 2
4
5 var x = 3;
6
7 // 现在 x 为 3
```

在相同的作用域或块级作用域中，不能使用 **let** 关键字来重置 **var** 关键字声明的变量：

```
1 var x = 2;      // 合法
2 let x = 3;      // 不合法
3
4 {
5     var x = 4;   // 合法
6     let x = 5;   // 不合法
7 }
```

在相同的作用域或块级作用域中，不能使用 **let** 关键字来重置 **let** 关键字声明的变量：

```
1 let x = 2;      // 合法
2 let x = 3;      // 不合法
3
4 {
5     let x = 4;   // 合法
6     let x = 5;   // 不合法
7 }
```

在相同的作用域或块级作用域中，不能使用 **var** 关键字来重置 **let** 关键字声明的变量：

```
1 let x = 2;      // 合法
2 var x = 3;      // 不合法
3
4 {
5     let x = 4;   // 合法
6     var x = 5;   // 不合法
7 }
```

let 关键字在不同作用域，或不同块级作用域中是可以重新声明赋值的：


```
1 let x = 2;      // 合法
2
3 {
4     let x = 3;  // 合法
5 }
6
7 {
8     let x = 4;  // 合法
9 }
```

变量提升

JavaScript 中, `var` 关键字定义的变量可以在使用后声明, 也就是变量可以先使用再声明 ([JavaScript 变量提升](#))。

```
1 // 在这里可以使用 carName 变量
2
3 var carName;
```

`let` 关键字定义的变量则不可以在使用后声明, 也就是变量需要先声明再使用。

```
1 // 在这里不可以使用 carName 变量
2
3 let carName;
```

const

`const` 用于声明一个或多个常量, 声明时必须进行初始化, 且初始化后值不可再修改:

```
1 const PI = 3.141592653589793;
2 PI = 3.14;      // 报错
3 PI = PI + 10;   // 报错
```

`const` 定义常量与使用 `let` 定义的变量相似:

- 二者都是块级作用域
- 都不能和它所在作用域内的其他变量或函数拥有相同的名称

两者还有以下两点区别:

- `const` 声明的常量必须初始化, 而 `let` 声明的变量不用
- `const` 定义常量的值不能通过再赋值修改, 也不能再次声明。而 `let` 定义的变量值可以修改。

`const` 的本质: `const` 定义的变量**并非常量, 并非不可变**, 它定义了一个常量引用一个值。使用 `const` 定义的对象或者数组, **其实是可变的**。下面的代码并不会报错:

```
1 // 创建常量对象
2 const car = {type:"Fiat", model:"500", color:"white"};
3
4 // 修改属性:
5 car.color = "red";
6
7 // 添加属性
8 car.owner = "Johnson";
```

但是我们不能对常量对象重新赋值:

```
1 const car = {type:"Fiat", model:"500", color:"white"};
2 car = {type:"Volvo", model:"EX60", color:"red"}; // 错误
```

以下实例修改常量数组:

```
1 // 创建常量数组
2 const cars = ["Saab", "Volvo", "BMW"];
3
4 // 修改元素
5 cars[0] = "Toyota";
6
7 // 添加元素
8 cars.push("Audi");
```

但是我们不能对常量数组重新赋值:

```
1 const cars = ["Saab", "Volvo", "BMW"];
2 cars = ["Toyota", "Volvo", "Audi"]; // 错误
```

声明变量类型

当您声明新变量时, 可以使用关键词 "new" 来声明其类型:

```
1 var carname=new String;
2 var x=      new Number;
3 var y=      new Boolean;
4 var cars=   new Array;
5 var person= new Object;
```

JS全局变量

变量在函数外定义, 即为全局变量。

全局变量有 **全局作用域**: 网页中所有脚本和函数均可使用。

```
1 var carName = " Volvo";
2
3 // 此处可调用 carName 变量
4 function myFunction() {
5     // 函数内可调用 carName 变量
6 }
```

如果变量在函数内没有声明（没有使用 `var` 关键字），该变量为全局变量。

以下实例中 `carName` 在函数内，但是为全局变量。

```
1 // 此处可调用 carName 变量
2
3 function myFunction() {
4     carName = "Volvo";
5     // 此处可调用 carName 变量
6 }
```

HTML全局变量

在 HTML 中，全局变量是 `window` 对象：所有数据变量都属于 `window` 对象。

```
1 //此处可使用 window.carName
2
3 function myFunction() {
4     carName = "Volvo";
5 }
```

类型转换

`Number()` 转换为数字，`String()` 转换为字符串，`Boolean()` 转换为布尔值。

[Number 方法](#)

方法	描述
toString()	将数字转换为字符串
toFixed()	把数字转换为字符串，结果的小数点后有指定位数的数字。
toPrecision()	把数字格式化为指定的长度。
toExponential()	把对象的值转换为指数计数法。

[Date 方法](#)

JS事件

事件可以用于处理表单验证，用户输入，用户行为及浏览器动作：

- 页面加载时触发事件
- 页面关闭时触发事件

- 用户点击按钮执行动作
- 验证用户输入内容的合法性
- 等等 ...

可以使用多种方法来执行 JavaScript 事件代码：

- HTML 事件属性可以直接执行 JavaScript 代码
- HTML 事件属性可以调用 JavaScript 函数
- 你可以为 HTML 元素指定自己的事件处理程序
- 你可以阻止事件的发生。
- 等等 ...

HTML 事件是发生在 HTML 元素上的事情。

当在 HTML 页面中使用 JavaScript 时，JavaScript 可以触发这些事件。

HTML事件

HTML 事件可以是浏览器行为，也可以是用户行为。

以下是 HTML 事件的实例：

- HTML 页面完成加载
- HTML input 字段改变时
- HTML 按钮被点击

通常，当事件发生时，你可以做些事情。在事件触发时 JavaScript 可以执行一些代码。

HTML 元素中可以添加事件属性，使用 JavaScript 代码来添加 HTML 元素。

```
1 //单引号
2 <some-HTML-element some-event='JavaScript 代码'>
3 //双引号
4 <some-HTML-element some-event="JavaScript 代码">
```

在以下实例中，按钮元素中添加了 onclick 属性（并加上代码）：

```
1 //修改id索引元素的内容
2 <button onclick="getElementById('demo').innerHTML=Date()">现在的时间是?
  </button>
3
4 //修改按钮自身内容
5 <button onclick="this.innerHTML=Date()">现在的时间是?</button>
```

常见的HTML事件

[HTML DOM 事件对象 | 菜鸟教程 \(runoob.com\)](https://www.runoob.com/html/html-dom-event-objects.html)

事件	描述
onchange	HTML 元素改变
onclick	用户点击 HTML 元素
onmouseover	鼠标指针移动到指定的元素上时发生
onmouseout	用户从一个 HTML 元素上移开鼠标时发生
onkeydown	用户按下键盘按键
onload	浏览器已完成页面的加载

JSON

[JSON 教程](#)

JSON 是用于存储和传输数据的格式。

JSON 通常用于服务端向网页传递数据。

- JSON 英文全称 **J**ava**S**cript **O**bject **N**otation
- JSON 是一种轻量级的数据交换格式。
- JSON是独立的语言
- JSON 易于理解。

JSON 使用 JavaScript 语法, 但是 JSON 格式仅仅是一个文本。
文本可以被任何编程语言读取及作为数据格式传递。

以下 JSON 语法定义了 sites 对象: 3 条网站信息 (对象) 的数组:

```
1  {"sites":[
2      {"name":"Runoob", "url":"www.runoob.com"},
3      {"name":"Google", "url":"www.google.com"},
4      {"name":"Taobao", "url":"www.taobao.com"}
5  ]}
```

语法规则

- 数据为 键/值 对。
- 数据由逗号分隔。
- 大括号保存对象
- 方括号保存数组

JSON 字符串转换为 JavaScript 对象

通常我们从服务器中读取 JSON 数据, 并在网页中显示数据。

简单起见, 我们网页中直接设置 JSON 字符串 (你还可以阅读我们的 [JSON 教程](#)):

首先, 创建 JavaScript 字符串, 字符串为 JSON 格式的数据:

```
1 var text = '{ "sites" : [' +
2   '{ "name":"Runoob" , "url":"www.runoob.com" },' +
3   '{ "name":"Google" , "url":"www.google.com" },' +
4   '{ "name":"Taobao" , "url":"www.taobao.com" } ]}';
```

然后，使用 JavaScript 内置函数 `JSON.parse()` 将字符串转换为 JavaScript 对象：

```
1 var obj = JSON.parse(text);
```

最后，在你的页面中使用新的 JavaScript 对象：

```
1 var text = '{ "sites" : [' +
2   '{ "name":"Runoob" , "url":"www.runoob.com" },' +
3   '{ "name":"Google" , "url":"www.google.com" },' +
4   '{ "name":"Taobao" , "url":"www.taobao.com" } ]}';
5
6 obj = JSON.parse(text);
7 document.getElementById("demo").innerHTML = obj.sites[1].name + " " +
  obj.sites[1].url;
```

函数	描述
JSON.parse()	用于将一个 JSON 字符串转换为 JavaScript 对象。
JSON.stringify()	用于将 JavaScript 值转换为 JSON 字符串。

函数

函数定义

JavaScript 使用关键字 **function** 定义函数。

函数可以通过声明定义，也可以是一个表达式。

函数表达式

JavaScript 函数可以通过一个表达式定义。

函数表达式可以存储在变量中：

```
1 var x = function (a, b) {return a * b};
```

在函数表达式存储在变量后，变量也可作为一个函数使用：

```
1 var x = function (a, b) {return a * b};
2 var z = x(4, 3);
```

以上函数实际上是一个 **匿名函数**（函数没有名称）。

函数存储在变量中，不需要函数名称，通常通过变量名来调用。

构造函数

在以上实例中，我们了解到函数通过关键字 **function** 定义。

函数同样可以通过内置的 JavaScript 函数构造器 (Function()) 定义。

```
1 var myFunction = new Function("a", "b", "return a * b");
2
3 var x = myFunction(4, 3);
```

实际上，你不必使用构造函数。上面实例可以写成：

```
1 var myFunction = function (a, b) {return a * b};
2
3 var x = myFunction(4, 3);
```

在 JavaScript 中，很多时候，你需要**避免使用 new 关键字**。

函数提升

在之前的教程中我们已经了解了 "hoisting(提升)"。

提升 (Hoisting) 是 JavaScript 默认将当前作用域提升到前面去的行为。

提升 (Hoisting) 应用在变量的声明与函数的声明。

因此，函数可以在声明之前调用：

```
1 myFunction(5);
2
3 function myFunction(y) {
4     return y * y;
5 }
```

使用表达式定义函数时无法提升。

自调用函数

函数表达式可以 "自调用"。

自调用表达式会自动调用。

如果表达式后面紧跟 ()，则会自动调用。

不能自调用声明的函数。

通过添加括号，来说明它是一个函数表达式：

```
1 (function () {
2     var x = "Hello!!";    // 我将调用自己
3 })();
```

以上函数实际上是一个 **匿名自我调用的函数**（没有函数名）。

函数是对象

在 JavaScript 中使用 **typeof** 操作符判断函数类型将返回 "function" 。

但是JavaScript 函数描述为一个对象更加准确。

JavaScript 函数有 **属性** 和 **方法**。

arguments.length 属性返回函数调用过程接收到的参数个数：

```
1 function myFunction(a, b) {  
2     return arguments.length;  
3 }
```

JS之箭头函数

箭头函数是ES6引入到JavaScript中的，是一种新式的匿名函数的写法，类似于其他语言中Lambda函数。箭头函数和传统函数有很多的不同，例如作用域、语法写法等等。

一、传统函数的定义

1. 普通函数定义

下面是一个sum函数的定义，可以返回两个参数之和。

```
1 function sum(a, b) {  
2     return a + b  
3 }
```

对于传统函数，你甚至可以在定义之前调用该函数

```
1 sum(1, 2)  
2  
3 function sum(a, b) {  
4     return a + b  
5 }
```

你可以通过函数名，打印出其函数申明

```
1 console.log(sum)
```

输出结果如下：

```
1 f sum(a, b) {  
2     return a + b  
3 }
```

函数表达式通常可以有个名字，但是也可以是匿名的，意味着函数是没有名字的。

2、匿名函数

下面例子是sum函数的匿名申明方式：

```
1  const sum = function (a, b) {  
2      return a + b  
3  }
```

这时我们将匿名函数赋值给了sum变量，如果这时候在定义之前调用会导致错误：

```
1  sum(1, 2)  
2  
3  const sum = function (a, b) {  
4      return a + b  
5  }
```

错误返回：Uncaught ReferenceError: Cannot access 'sum' before initialization

我们也可以把sum打印出来：

```
1  const sum = function (a, b) {  
2      return a + b  
3  }  
4  
5  console.log(sum)
```

打印出来的结果如下：

```
1  f (a, b) {  
2      return a + b  
3  }
```

sum是一个匿名函数，而不是一个有名字的函数。

二、箭头函数

箭头函数和普通函数有非常多的不同，**箭头函数没有自己的this绑定，没有prototype，不能被用做构造函数。同时，箭头函数可以作为普通函数提供更为简洁的写法。**

1、箭头函数定义

写法非常简洁：

```
const sum = (a, b) => { return a + b }
```

可以简化函数声明，并且提供隐含返回值：

```
const sum = (a, b) => a + b
```

当只有一个参数的时候，可以省略()，写成如下形式：

```
const square = x => x * x
```

2、this绑定

在JavaScript中，this往往是一个比较复杂诡异的事情。在JavaScript中有bind、apply、call等方法会影响this所指定的对象。

箭头函数的this是语义层面的，因此，箭头函数中的this是由上下文作用域决定的。

下面的例子解释this在普通函数和箭头函数的区别：

```
1  const printNumbers = {  
2    phrase: 'The current value is:',  
3    numbers: [1, 2, 3, 4],  
4  
5    loop() {  
6      this.numbers.forEach(function (number) {  
7        console.log(this.phrase, number)  
8      })  
9    },  
10 }
```

你也许会期望loop函数会打印出文本和对应的数字，然后真正返回的内容其实是undefined：

```
1  printNumbers.loop()
```

输出：

```
1  undefined 1  
2  undefined 2  
3  undefined 3  
4  undefined 4
```

在上面的例子中，this.phrase代表了undefined，说明在forEach方法中的匿名函数中的this并不会指向printNumber对象。这是因为普通函数并不是通过上下文作用域来决定this的值，而是通过实际调用函数的对象来决定的。

在老版本的JavaScript，你可以通过bind方法来显示的绑定this。使用bind的方式如下：

```
1  const printNumbers = {  
2    phrase: 'The current value is:',  
3    numbers: [1, 2, 3, 4],  
4  
5    loop() {  
6      // 将外部的printNumber对象绑定到内部forEach函数中的'this'  
7      this.numbers.forEach(  
8        function (number) {  
9          console.log(this.phrase, number)  
10         }.bind(this),  
11      )  
12    },  
13  }  
14  
15  printNumbers.loop()
```

这将输出正确的结果：

```
1 The current value is: 1
2 The current value is: 2
3 The current value is: 3
4 The current value is: 4
```

箭头函数提供了一个更为优雅的解决方案，由于其this的指向是由上下文作用域来决定的，因此它会指向printNumbers对象：

```
1 const printNumbers = {
2   phrase: 'The current value is:',
3   numbers: [1, 2, 3, 4],
4
5   loop() {
6     this.numbers.forEach((number) => {
7       console.log(this.phrase, number)
8     })
9   },
10 }
11
12 printNumbers.loop()
```

上面箭头函数在循环中可以很好的使用，但是作为对象方法却会造成问题。如下例：

```
1 const printNumbers = {
2   phrase: 'The current value is:',
3   numbers: [1, 2, 3, 4],
4
5   loop: () => {
6     this.numbers.forEach((number) => {
7       console.log(this.phrase, number)
8     })
9   },
10 }
11 复制代码
```

调用loop方法

```
1 printNumbers.loop()
2 复制代码
```

结果会出现如下错误：

```
1 Uncaught TypeError: Cannot read property 'forEach' of undefined
2 复制代码
```

这是因为在loop箭头函数声明的时候，this指向并不会是printNumbers对象，而是外部的Window。而Window上面并没有numbers字段，从而导致错误。因此，对象方法一般使用传统方法。

3、箭头函数没有Prototype和构造函数

在JavaScript中的Function或者Class中都会有一个Prototype属性，这个可以用于对象拷贝或者继承。

```
1 function myFunction() {  
2     this.value = 5  
3 }  
4  
5 // Log the prototype property of myFunction  
6 console.log(myFunction.prototype)  
7 复制代码
```

输出：

```
1 {constructor: f}
```

但是，箭头函数是不存在Prototype属性，我们可以尝试打印出箭头函数的Prototype。

```
1 const myArrowFunction = () => {}  
2  
3 // Attempt to log the prototype property of myArrowFunction  
4 console.log(myArrowFunction.prototype)
```

输出：

```
1 undefined
```

同时，由于没有Prototype属性，因此也没法通过new进行实例化。

```
1 const arrowInstance = new myArrowFunction()  
2 console.log(arrowInstance)
```

输出错误：

```
1 Uncaught TypeError: myArrowFunction is not a constructor
```

总结：

箭头函数始终是匿名的，没有Prototype或者构造函数，无法用new进行实例化，并且是通过语义上下文来决定this指向。

函数参数

JavaScript 函数对参数的值没有进行任何的检查。

显式参数(Parameters)与隐式参数(Arguments)

在先前的教程中，我们已经学习了函数的显式参数：

```
1  functionName(parameter1, parameter2, parameter3) {
2      // 要执行的代码.....
3  }
```

函数显式参数在函数定义时列出。

函数隐式参数在函数调用时传递给函数真正的值。

- JavaScript 函数定义显式参数时没有指定数据类型。
- JavaScript 函数对隐式参数没有进行类型检测。
- JavaScript 函数对隐式参数的个数没有进行检测。

ES5 中如果函数在调用时未提供隐式参数，参数会默认设置为：**undefined**

有时这是可以接受的，但是建议最好为参数设置一个默认值：

```
1  function myFunction(x, y) {
2      if (y === undefined) {
3          y = 0;
4      }
5  }
```

ES6 函数可以自带参数

ES6 支持函数带有默认参数，就判断 `undefined` 和 `||` 的操作：

```
1  function myFunction(x, y = 10) {
2      // y is 10 if not passed or undefined
3      return x + y;
4  }
5
6  myFunction(0, 2) // 输出 2
7  myFunction(5); // 输出 15, y 参数的默认值
```

arguments 对象

JavaScript 函数有个内置的对象 `arguments` 对象。

`argument` 对象包含了函数调用的参数数组。

```
1  //arguments[0]~arguments[5]分别对应参数值
2  function findMax(){ ... }
3  x = findMax(1, 123, 500, 115, 44, 88);
```

函数调用

JavaScript 函数有 4 种调用方式。

每种方式的不同在于 **this** 的初始化。

this 关键字

一般而言，在JavaScript中，**this**指向函数执行时的当前对象。

调用 JavaScript 函数

在之前的章节中我们已经学会了如何创建函数。

函数中的代码在函数被调用后执行。

```
1 function myFunction(a, b) {  
2     return a * b;  
3 }  
4 //myFunction() 和 window.myFunction() 是一样的:  
5 myFunction(10, 2);           // myFunction(10, 2) 返回 20  
6 window.myFunction(10, 2);    // window.myFunction(10, 2) 返回 20
```

以上函数不属于任何对象。但是在 JavaScript 中它始终是默认的全局对象。

在 HTML 中默认的全局对象是 HTML 页面本身，所以函数是属于 HTML 页面。

在浏览器中的页面对象是浏览器窗口(window 对象)。以上函数会自动变为 window 对象的函数。

全局对象

当函数没有被自身的对象调用时 **this** 的值就会变成全局对象。

在 web 浏览器中全局对象是浏览器窗口 (window 对象)。

该实例返回 **this** 的值是 window 对象：

```
1 function myFunction() {  
2     return this;  
3 }  
4 myFunction();           // 返回 window 对象
```

函数作为全局对象调用，会使 **this** 的值成为全局对象。使用 window 对象作为一个变量容易造成程序崩溃。

函数作为方法调用

在 JavaScript 中你可以将函数定义为对象的方法。

以下实例创建了一个对象 (**myObject**)，对象有两个属性 (**firstName** 和 **lastName**)，及一个方法 (**fullName**)：

```
1 var myObject = {  
2     firstName:"John",  
3     lastName: "Doe",  
4     fullName: function () {  
5         return this.firstName + " " + this.lastName;  
6     }  
7 }  
8 myObject.fullName();           // 返回 "John Doe"
```

fullName 方法是一个函数。函数属于对象。**myObject** 是函数的所有者。

this对象, 拥有 JavaScript 代码。实例中 **this** 的值为 **myObject** 对象。

测试以下! 修改 **fullName** 方法并返回 **this** 值:

```
1  var myObject = {
2      firstName: "John",
3      lastName: "Doe",
4      fullName: function () {
5          return this;
6      }
7  }
8  myObject.fullName();           // 返回 [object Object] (所有者对象)
```

构造函数调用函数

如果函数调用前使用了 **new** 关键字, 则是调用了构造函数。

这看起来就像创建了新的函数, 但实际上 JavaScript 函数是重新创建的对象:

```
1  // 构造函数:
2  function myFunction(arg1, arg2) {
3      this.firstName = arg1;
4      this.lastName = arg2;
5  }
6
7  // This creates a new object
8  var x = new myFunction("John", "Doe");
9  x.firstName;                   // 返回 "John"
```

构造函数的调用会创建一个新的对象。新对象会继承构造函数的属性和方法。

闭包

JavaScript 变量可以是局部变量或全局变量。

私有变量可以用到闭包。

所有函数都能访问全局变量。

实际上, 在 JavaScript 中, 所有函数都能访问它们上一层的作用域。

JavaScript 支持嵌套函数。嵌套函数可以访问上一层的函数变量。

该实例中, 内嵌函数 **plus()** 可以访问父函数的 **counter** 变量:

```
1  function add() {
2      var counter = 0;
3      function plus() {counter += 1;}
4      plus();
5      return counter;
6  }
```

如果我们能在外部访问 `plus()` 函数，这样就能解决计数器的困境。

我们同样需要确保 `counter = 0` 只执行一次。

我们需要闭包。

如何实现闭包???

待续

类

类是用于创建对象的模板。

我们使用 `class` 关键字来创建一个类，类体在一对大括号 `{}` 中，我们可以在大括号 `{}` 中定义类成员的位置，如方法或构造函数。

每个类中包含了一个特殊的方法 `constructor()`，它是类的构造函数，这种方法用于创建和初始化一个由 `class` 创建的对象。

创建一个类的语法格式如下：

```
1 class ClassName {
2   constructor() { ... }
3 }
```

实例：

```
1 class Runoob {
2   constructor(name, url) {
3     this.name = name;
4     this.url = url;
5   }
6 }
7
8 //以上实例创建了一个类，名为 "Runoob"。
9 //类中初始化了两个属性： "name" 和 "url"。
```

使用类

定义好类后，我们就可以使用 `new` 关键字来创建对象：

```
1 class Runoob {
2   constructor(name, url) {
3     this.name = name;
4     this.url = url;
5   }
6 }
7
8 let site = new Runoob("菜鸟教程", "https://www.runoob.com");
```

创建对象时会自动调用构造函数方法 `constructor()`。

类表达式

类表达式是定义类的另一种方法。类表达式可以命名或不命名。命名类表达式的名称是该类体的局部名称。

```
1 // 未命名/匿名类
2 let Runoob = class {
3   constructor(name, url) {
4     this.name = name;
5     this.url = url;
6   }
7 };
8 console.log(Runoob.name);
9 // output: "Runoob"
10
11 // 命名类
12 let Runoob = class Runoob2 {
13   constructor(name, url) {
14     this.name = name;
15     this.url = url;
16   }
17 };
18 console.log(Runoob.name);
19 // 输出: "Runoob2"
```

构造方法

构造方法是一种特殊的方法：

- 构造方法名为 `constructor()`。
- 构造方法在创建新对象时会自动执行。
- 构造方法用于初始化对象属性。
- 如果不定义构造方法，JavaScript 会自动添加一个空的构造方法。

类的方法

我们使用关键字 `class` 创建一个类，可以添加一个 `constructor()` 方法，然后添加任意数量的方法。

```
1 class ClassName {
2   constructor() { ... }
3   method_1() { ... }
4   method_2() { ... }
5   method_3() { ... }
6 }
```

以下实例我们创建一个 "age" 方法，用于返回网站年龄：

```
1 class Runoob {
2   constructor(name, year) {
3     this.name = name;
4     this.year = year;
5   }
6   age() {
```

```

7     let date = new Date();
8     return date.getFullYear() - this.year;
9 }
10 }
11
12 let runoob = new Runoob("菜鸟教程", 2018);
13 document.getElementById("demo").innerHTML =
14 "菜鸟教程 " + runoob.age() + " 岁了。";

```

严格模式

类声明和类表达式的主体都执行在严格模式下。比如，构造函数，静态方法，原型方法，getter 和 setter 都在严格模式下执行。

如果你没有遵循严格模式，则会出现错误：

```

1 //不能使用类未声明的变量:
2 class Runoob {
3     constructor(name, year) {
4         this.name = name;
5         this.year = year;
6     }
7     age() {
8         // date = new Date(); // 错误
9         let date = new Date(); // 正确
10        return date.getFullYear() - this.year;
11    }
12 }

```


```

> class Runoob {
  constructor(name, year) {
    this.name = name;
    this.year = year;
  }
  age() {
    date = new Date(); // 错误
    return date.getFullYear() - this.year;
  }
}

myCar = new Runoob("菜鸟教程", 2020);
document.getElementById("demo").innerHTML =
"菜鸟教程 " + myCar.age() + " 岁了。";

```

不能使用没有声明的变量



```

✖ ▶ Uncaught ReferenceError: date is not defined
    at Runoob.age (<anonymous>:7:10)
    at <anonymous>:15:17
> |

```

类关键字

关键字	描述
extends	继承一个类
static	在类中定义一个静态方法
super	调用父类的构造方法

继承

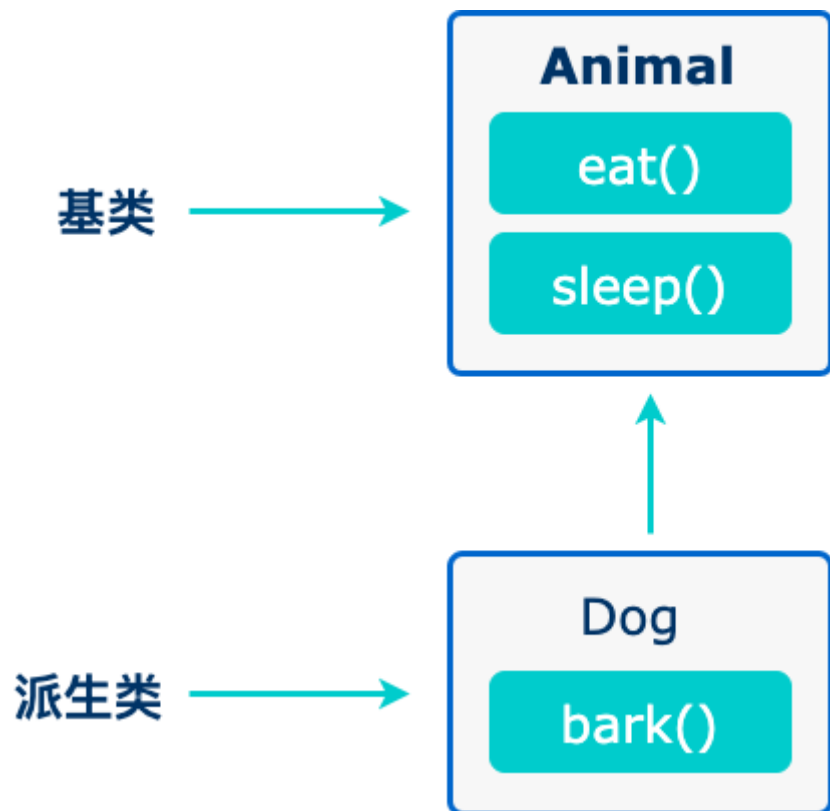
JavaScript 类继承使用 `extends` 关键字。

继承允许我们依据另一个类来定义一个类，这使得创建和维护一个应用程序变得更容易。

`super()` 方法用于调用父类的构造函数。

当创建一个类时，您不需要重新编写新的数据成员和成员函数，只需指定新建的类继承了一个已有的类的成员即可。这个已有的类称为**基类（父类）**，新建的类称为**派生类（子类）**。

继承代表了 **is a** 关系。例如，哺乳动物是动物，狗是哺乳动物，因此，狗是动物，等等。



```
1 // 基类
2 class Animal {
3     // eat() 函数
4     // sleep() 函数
5 };
6
7
8 //派生类
9 class Dog extends Animal {
10     // bark() 函数
11 };
```

以下实例创建的类 "Runoob" 继承了 "Site" 类：

```
1 class Site {
2     constructor(name) {
3         this.sitename = name;
4     }
5     present() {
6         return '我喜欢' + this.sitename;
7     }
8 }
9
10 class Runoob extends Site {
11     constructor(name, age) {
12         super(name);
13         this.age = age;
14     }
15     show() {
16         return this.present() + ', 它创建了 ' + this.age + ' 年.';
17     }
18 }
19
20 let noob = new Runoob("菜鸟教程", 5);
21 document.getElementById("demo").innerHTML = noob.show();
22 //输出: 我喜欢菜鸟教程, 它创建了 5 年。
```

super() 方法引用父类的构造方法。

通过在构造方法中调用 **super()** 方法，我们调用了父类的构造方法，这样就可以访问父类的属性和方法。

继承对于代码可复用性很有用。

getter 和 setter

类中我们可以使用 **getter** 和 **setter** 来获取和设置值，getter 和 setter 都需要在严格模式下执行。

getter 和 setter 可以使得我们对属性的操作变的很灵活。

类中添加 **getter** 和 **setter** 使用的是 **get** 和 **set** 关键字。

以下实例为 sitename 属性创建 getter 和 setter：

```
1 class Runoob {
2     constructor(name) {
3         this.sitename = name;
4     }
5     get s_name() {
6         return this.sitename;
7     }
8     set s_name(x) {
9         this.sitename = x;
10    }
11 }
12
13 let noob = new Runoob("菜鸟教程");
```

```
14
15 document.getElementById("demo").innerHTML = noob.s_name;
```

注意：即使 `getter` 是一个方法，当你想获取属性值时也不要使用括号。

`getter/setter` 方法的名称不能与属性的名称相同，在本例中属名为 `sitename`。

很多开发者在属性名称前使用下划线字符 `_` 将 `getter/setter` 与实际属性分开：

以下实例使用下划线 `_` 来设置属性，并创建对应的 `getter/setter` 方法：

```
1 class Runoob {
2   constructor(name) {
3     this._sitename = name;
4   }
5   get sitename() {
6     return this._sitename;
7   }
8   set sitename(x) {
9     this._sitename = x;
10  }
11 }
12
13 let noob = new Runoob("菜鸟教程");
14
15 document.getElementById("demo").innerHTML = noob.sitename;
```

要使用 `setter`，请使用与设置属性值时相同的语法，虽然 `set` 是一个方法，但需要不带括号：

```
1 class Runoob {
2   constructor(name) {
3     this._sitename = name;
4   }
5   set sitename(x) {
6     this._sitename = x;
7   }
8   get sitename() {
9     return this._sitename;
10  }
11 }
12
13 let noob = new Runoob("菜鸟教程");
14 noob.sitename = "RUNOOB";
15 document.getElementById("demo").innerHTML = noob.sitename;
```

提升

函数声明和类声明之间的一个重要区别在于，函数声明会提升，**类声明不会**。

你首先需要声明你的类，然后再访问它，否则类似以下的代码将抛出 `ReferenceError`：

静态方法

静态方法是使用 **`static`** 关键字修饰的方法，又叫类方法，属于类的，但不属于对象，在实例化对象之前可以通过 **类名.方法名** 调用静态方法。

静态方法不能在对象上调用，只能在类中调用。

```
1 class Runoob {
2   constructor(name) {
3     this.name = name;
4   }
5   static hello() {
6     return "Hello!!";
7   }
8 }
9
10 let noob = new Runoob("菜鸟教程");
11
12 // 可以在类中调用 'hello()' 方法
13 document.getElementById("demo").innerHTML = Runoob.hello();
14
15 // 不能通过实例化后的对象调用静态方法
16 // document.getElementById("demo").innerHTML = noob.hello();
17 // 以上代码会报错
```

实例对象调用静态方法会报错：

```
> class Runoob {
  constructor(name) {
    this.name = name;
  }
  static hello() {
    return "Hello!!";
  }
}

let noob = new Runoob("菜鸟教程");
< undefined
> noob.hello();
✖ ▶ Uncaught TypeError: noob.hello is not a function
   at <anonymous>:1:6
> |
```

实例对象不能调用静态方法

如果你想在对象 noob 中使用静态方法，可以作为一个参数传递给它：

```
1 class Runoob {
2   constructor(name) {
3     this.name = name;
4   }
5   static hello(x) {
6     return "Hello " + x.name;
7   }
8 }
9 let noob = new Runoob("菜鸟教程");
10 document.getElementById("demo").innerHTML = Runoob.hello(noob);
```

1.初始化

```
1 | npm init -y
```

package.json可以查看包

换机可以快速迁移包数据，自动安装和package.json内版本匹配的包

```
1 | npm i
```

2.安装

```
1 | npm install 名称
```

3.更新

```
1 | npm update 名称
```

4.旧版本

```
1 | npm install 名称@ban'ben'hao
```

Turf.js

[GET START | Turf.js中文网 \(fenxianglu.cn\)](#)

Turf 是可用于浏览器和Node 端的高级地理空间分析工具，Turf 提供多达 150 种的空间分析功能，你需要的都可以找到，比如数据抽稀、Tin、等高线、缓冲区、网格聚合、空间关系判断，坐标转换等等。

Turf 简单易懂，小巧灵敏，模块化，运行速度极快！

- 简约：使用 GeoJSON 的模块化、简单易懂的 JavaScript 函数
- 模块化：Turf 是一个个小模块的集合，按需引用
- 快速：利用最新的算法，前端计算

Turf 作为空间数据分析工具，处理的对象是空间数据，使用之前你需要了解什么是 GeoJSON。

GeoJSON 相关介绍：[WebGIS 标准数据格式 GeoJSON 格式介绍](#)

使用

使用场景

- WebGIS 系统中经常会涉及到空间数据的处理，计算比如计算面积、缓冲区、中心点计算这种情况我可以使用Turf 在前端进行计算、这样灵活性会更好。
- 本地环境数据进行处理，比如计算多边形的面积，点和面数据空间连接，我们就可以使用 Turf 写脚本批量完成数据操作。
- 服务端提供空间计算服务，借助Turf 可以实现很多空间数据计算服务，相对其他软件工具要简单很多。

使用方法

npm安装turf.js

```
1 | npm install @turf/turf
```

引入

```
1 | import Point from 'ol/geom/Point';
2 | import Feature from 'ol/Feature';
3 | import VectorLayer from 'ol/layer/Vector';
4 | import VectorSource from 'ol/source/Vector';
5 | import { Stroke, Style, Icon, Text as TextStyle, Circle as CircleStyle,
  | Fill } from 'ol/style.js';
6 | import Select from 'ol/interaction/Select';
7 | import { click, pointerMove } from 'ol/events/condition.js';
8 | import GeoJSON from 'ol/format/GeoJSON';
9 | //面数据用于边界修剪。如果不进行边界的修剪生成的等值面就是点集的最大边界作为边界。
10 | // import
  | {featureCollection,interpolate,isobands,flatten,randomPoint,bbox,featureEa
  | ch,buffer,intersect} from '@turf/turf';
11 |
12 | import * as turf from '@turf/turf'
```

Openlayers

```
1 | npm install ol
```

然后可以使用：

```
1 | import OlMap from 'ol/map';
2 | import OlView from 'ol/view';
3 | import OlTile from 'ol/layer/tile';
4 | import OlLayerVector from 'ol/layer/vector';
5 | import OlSourceVector from 'ol/source/vector';
```