# JavaScript

**无08 王知衡 wangzhih20@mails.tsinghua.edu.cn**

> 本教程参考MDN、菜鸟教程和上一届张凯学长、brgg的讲义

# 什么是JavaScript?

JavaScript ( JS ) 是一种具有函数优先的轻量级，解释型或即时编译型的编程语言。最早作为开发 Web 页面的脚本语言而出名，也被应用于很多非浏览器环境。

JavaScript 可以在网页上实现复杂的功能，网页展现给你的不再是简单的静态信息，而是实时的内容更新，交互式的地图，2D/3D 动画，滚动播放的视频等等。它是标准 Web 技术蛋糕的第三层，其中 HTML 和 CSS 我们已经在前面进行了详细的讲解。

JavaScript 是轻量级解释型语言。浏览器接受到 JavaScript 代码，并以代码自身的文本格式运行它。技术上，几乎所有 JavaScript 转换器都运用了一种叫做即时编译（just-in-time compiling）的技术；当 JavaScript 源代码被执行时，它会被编译成二进制的格式，使代码运行速度更快。尽管如此，JavaScript 仍然是一门解释型语言，因为编译过程发生在代码运行中，而非之前。

# 语法

## 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。

**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）。

可以用`typeof()`获取数据类型。

### 数字(Number)

JavaScript只有一种数字类型。极大或极小的数可以用科学计数法来书写。

```
var x = 1.2e5;  // 120000
```

NaN 属性是代表非数字值的特殊值。该属性用于指示某个值不是数字。可以把 Number 对象设置为该值，来指示其不是数字值。你可以使用 isNaN() 全局函数来判断一个值是否是 NaN 值。

```
var x = 1000 / "Apple";
isNaN(x); // return true
var y = 100 / "1000";
isNaN(y); // return false; y = 0.1
```

内置对象Math提供多种算数值类型和函数。

## 字符串（String）

字符串是存储字符的变量。

字符串可以是引号中的任意文本。你可以使用单引号或双引号，但必须成对。

可以在字符串字面值上使用字符串对象的所有方法——JavaScript会自动将字符串字面值转换为一个临时字符串对象，调用该方法，然后废弃掉那个临时的字符串对象，比如`'eesast'.length`

可以使用内置函数`parseInt()`（对非整数取整）和`parseFloat()`来将字符串转为Number

在ES2015中，引入了模板字符串，使用反引号（`）来代替普通字符串中的用双引号和单引号，我们对其最常见的使用就是使用占位符 ${expression} 来在其中插入表达式，例如

```
let age = 20;
console.log(`I'm ${age} yaers old.`);
```

## 空（Null）和 未定义（Undefined）

null表示空对象引用，undefined表示变量声明过但为赋值。

## 数组（Array）

JavaScript 数组的长度和元素类型都是非固定的，并且其数据在内存中也可以不连续。

数组是一个单个对象，其中包含很多值，方括号括起来，并用逗号分隔。

```
let a = [1, 2, 'a', true]; // 创建数组
let b = new Array(1, 2, 'a', true); // 与上面代码等效
let c = new Array(3); // 定义一个长度为3的空数组
console.log(a[0]); // 访问数组中的元素
```

Array具有一些常用的方法：

遍历：`forEach()`

对数组中的每个元素执行参数中的函数

```
const array = ["a", "b", "c"];
array.forEach((element) => console.log(element));
```

筛选：`filter()`

```
var words = ['spray', 'limit', 'elite', 'exuberant', 'destruction', 'present'];
const result = words.filter((word) => word.length > 6);
console.log(result);
// expected output: Array ["exuberant", "destruction", "present"]
```

## 对象（Object）

JavaScript中的对象Object与我们知道的Python中的字典比较相似。

对象由花括号分隔。在括号内部，对象的属性以名称和值对的形式 (name : value) 来定义。属性由逗号分隔：

```
var person = {
    firstName: "John",
    lastName: "Doe",
    id: 1234
}; // 比较推荐的方式
var obj = new Object();
```

对象属性有两种寻址方式：

```
name = person.lastName;
name = person["lastName"];
```

和大家刚学过的 C++类似，这里的对象也有 this 来指向了当前代码运行时的对象。

### 动态类型

在JavaScript中，当对一个变量赋值时，不需要考虑它的类型

```
let v = 100;
typeof(v); // return number
v = "abc";
typeof(v); // return string
```

## 函数（Function）

函数定义与声明： 定义函数使用 function 关键字，例如

```
function square(number) {
    return number * number;
}
```

也可以使用表达式

```
const square = function (number) {
    return number * number;
} // 本质上是将匿名函数分配为变量的值
var x = square(4); // return 16
```

箭头函数，类似于 C++ lambda，相比函数表达式具有较短的语法并且不绑定 this。箭头函数总是匿名的。

```
var a = ["Hydrogen", "Helium", "Lithium", "Beryllium"];
var a2 = a.map((s) => s.length); // [8, 6, 7, 9]
```

箭头函数的this是继承父执行上下文里面的this

```
let num = 11;
const obj1 = {
    num: 22,
    fn1: function() {
        let num = 33;
        const obj2 = {
            num: 44,
```

```
            fn2: () => {
                console.log(this.num);
            }
        }
        obj2.fn2();
    }
}
obj1.fn1(); // 22
```

这里箭头函数的执行上下文是函数fn1()，所以它就继承了fn1()的this，obj1调用的fn1，所以fn1的this指向obj1， 所以obj1.num 为 22。PS：简单对象（非函数）是没有执行上下文的!

arguments 对象：函数的实际参数会被保存在一个类似数组的 arguments 对象中。在函数内，你可以按如下方式找出传入的参数：`arguments[i];`，可通过`arguments.length`获取参数数量。

剩余参数：剩余参数语法允许将不确定数量的参数表示为数组

```
function multiply(multiplier, ...theArgs) {
    return theArgs.map((x) => multiplier * x);
}
var arr = multiply(2, 1, 2, 3);
console.log(arr); // [2, 4, 6]
```

## 变量

变量的声明方式

- `var`：声明一个变量，可选初始化一个值
- `let`：声明一个块作用域的局部变量，可选初始化一个值
- `const`：声明一个块作用域的只读常量，必须初始化一个值

如果声明了一个变量却没有对它赋值，那么这个变量的类型是`undefined`

`let`的作用域是块作用域，`var`的作用域是全局或函数作用域

```
{
    let a = 10;
    var b = 20;
}
console.log(a); // ReferenceError: a is not defined
console.log(b); // 20
```

如果你编写一个声明并初始化变量的多行 JavaScript 程序，你可以在初始化一个变量之后用 `var` 声明它，它仍然可以工作。例如：

```
a = 10;
var a;
```

这是由于变量的提升。

当你使用 `var` 时，可以根据需要多次声明相同名称的变量，但是 `let` 不能。

但是这样非常容易引起混乱，因此建议大家尽量多使用`let`，而不是`var`

## 运算符

- 仅介绍与C/C++不同的部分
- 求幂`x**2`
- 判断相等与不等：`===`（值和类型均相等）与`!==`、`==`与`!=`
- 字符串运算：`+`可以直接连接两个字符串，并同时会尝试将另一个操作数转化为string
- 解构赋值（以及rest操作符）：`...`将属性/值从对象/数组中取出/赋值给其他变量，或进行相反操作。例如

```
[a, b, ...rest] = [10, 20, 30, 40, 50];
console.log(rest); // Array [30, 40, 50]
let c = [1, 2, rest]; // Array [1, 2, [30, 40, 50]]
let d = [1, 2, ...rest]; // Array [1, 2, 30, 40, 50]
```

## 控制结构

`if ... else`, `do ... while`, `while`, `switch`与C/C++类似

介绍三种`for`：

- `for ... in`：迭代的是自定义的属性，而不是数组的元素

```
for (let property in object) {
    // do something with object property
}
```

- `for ... of`：在可迭代对象（包括Array、Map、Set、arguments等）上创建一个循环，对值的每一个独特属性调用一次迭代

```
for (let value of array) {
    // do something with value
}
```

- `forEach`：对数组中的每个元素执行一次给定的函数（前面举过例子）

## 异常处理

可以用`throw`语句抛出一个异常并且用`try...catch`语句捕获处理它。

- `throw`可以抛出任意对象，例如：

```
throw "Error"; // string type
throw 42; // number type
throw {
    toString: function () {
        return "I'm an object!";
    }
};
```

- `try ... catch`

try...catch 语句标记一块待尝试的语句，并规定一个以上的响应应该有一个异常被抛出。如果我们抛出一个异常，try...catch 语句就捕获它。换句话，如果 try 代码块中的语句（或者 try 代码块中调用的方法）一旦抛出了异常，那么执行流程会立即进入 catch 代码块。如果 try 代码块没有抛出异常，catch 代码块就会被跳过。finally 代码块总会紧跟在 try 和 catch 代码块之后执行，但会在 try 和 catch 代码块之后的其他代码之前执行。

## 模块

类似于Python中的import，JavaScript中也可以导入、导出模块。

```
import {
    name,
    draw,
    reportArea,
    reportPerimeter,
} from "/js-examples/modules/basic-modules/modules/square.js";
```

> 导入时需要加花括号，哪怕只导入一个模块（导入default除外）

```
export function draw(ctx, length, x, y, color) {
    ctx.fillStyle = color;
    ctx.fillRect(x, y, length, length);
    return {
        length: length,
        x: x,
        y: y,
        color: color,
    };
}
```

> 导出 class/function 后没有分号
>
> 也可以先声明函数/对象，再导出

模块提供了一个特殊的默认导出 `export default` 语法，以使“一个模块只做一件事”的方式看起来更好。将 `export default` 放在要导出的实体前：

```
export default class User { // 只需要添加 "default" 即可
    constructor(name) {
        this.name = name;
    }
}
```

每个文件最多只有一个`export default`。导入`default`时不需要加花括号。

# 异步

## 同步函数长时间运行时存在的问题

当我们运行同步程序时，实际上浏览器是按照我们书写代码的顺序一行一行地执行程序的。浏览器会等待代码的解析和工作，在上一行完成后才会执行下一行。这样做是很有必要的，因为每一行新的代码都是建立在前面代码的基础之上的。

调用函数的时候也是同步的，在函数返回之前，调用者必须等待函数完成其工作。

但如果运行一个耗时的同步函数，在函数运行的过程中，用户不能输入任何东西，也不能点击任何东西，或做任何其他事情（被阻塞）。因此，我们想要有一种方法，让我们的程序可以：

- 通过调用一个函数来启动一个长期运行的操作
- 让函数开始操作并立即返回，这样我们的程序就可以保持对其他事件做出反应的能力
- 当操作最终完成时，通知我们操作的结果。

这就是异步函数为我们提供的能力。

# 异步的实现

## 回调

异步 callbacks 其实就是函数，只不过是作为参数传递给那些在后台执行的其他函数. 当那些后台运行的代码结束，就调用 callbacks 函数，通知你工作已经完成，或者其他有趣的事情发生了。使用callbacks有一点老套，在一些老派但经常使用的 API 里面，你会经常看到这种风格。

```
function test(value) {
    console.log(value);
}
function main(callback, value) {
    console.log("main");
    callback(value);
}
main(test, "callback");

setTimeout( () => {
    console.log("hi");
}, 2000); // 计时器，单位为ms
```

回调函数的写法比较古老，且回调函数多层嵌套会产生一个回调地狱，因此后面很少单纯用回调来实现异步。

## Promise

在基于 Promise 的 API 中，异步函数会启动操作并返回 `Promise` 对象。然后，你可以将处理函数附加到 Promise 对象上，当操作完成时（成功或失败），这些处理函数将被执行。

举个例子，我们发送一个请求，从服务器上获得一个 JSON 文件：

```
const fetchPromise = fetch('https://mdn.github.io/learning-
area/javascript/apis/fetching-data/can-store/products.json');

console.log(fetchPromise); // Promise {<state>:"pending"}

fetchPromise.then( response => {
  console.log(`已收到响应：${response.status}`);
}); // 如果获取操作成功，Promise将调用处理函数，传入一个包含服务器响应的Response对象

console.log("已发送请求……");
```

完整的输出结果应该是这样：

```
Promise { <state>: "pending" }
已发送请求……
已收到响应：200
```

`已发送请求……` 的消息在我们收到响应之前就被输出了。与同步函数不同，`fetch()` 在请求仍在进行时返回，这使我们的程序能够保持响应性。响应显示了 `200`（OK）的状态码，意味着我们的请求成功了。

Promise 是表示异步操作完成或失败的对象。可以说，它代表了一种中间状态。 本质上，这是浏览器说“我保证尽快给您答复”的方式，因此得名“Promise”。

一个 Promise 必然处于以下几种状态之一：

- 待定（pending）：初始状态，既没有被兑现，也没有被拒绝。
- 已兑现（fulfilled）：意味着操作成功完成。
- 已拒绝（rejected）：意味着操作失败。

待定状态的 Promise 对象要么会通过一个值被兑现（fulfilled），要么会通过一个原因（错误）被拒绝（rejected）。当这些情况之一发生时，我们用 promise 的 then 方法排列起来的相关处理程序就会被调用。

`then()` 本身也会返回一个 Promise，这个 Promise 将指示 `then()` 中调用的异步函数的完成状态。这意味着我们可以链式使用Promise。

```
const fetchPromise = fetch('https://mdn.github.io/learning-
area/javascript/apis/fetching-data/can-store/products.json');

fetchPromise
  .then( response => {
    if (!response.ok) {
      throw new Error(`HTTP error: ${response.status}`);
    }
    return response.json();
  })
  .then( json => {
    console.log(json[0].name);
  });
```

接下来看一下后面的`.then()`：`.then()` 块包含一个回调函数，如果前一个操作成功，该函数将运行，并且每个回调都接收前一个成功操作的结果作为输入；每个`.then()` 块返回另一个 Promise，这意味着可以将多个`.then()` 块链接到另一个块上，这样就可以依次执行多个异步操作。如果其中任何一个`.then()` 块失败，则在末尾运行`.catch()` 块——与同步 `try...catch` 类似，`.catch()` 提供了一个错误对象，可用来报告发生的错误类型。

```
const fetchPromise = fetch('bad-scheme://mdn.github.io/learning-
area/javascript/apis/fetching-data/can-store/products.json');

fetchPromise
  .then( response => {
    if (!response.ok) {
      throw new Error(`HTTP 请求错误：${response.status}`);
    }
    return response.json();
  })
  .then( json => {
```

```
    console.log(json[0].name);
  })
  .catch( error => {
    console.error(`无法获取产品列表：${error}`);
  });
```

此外，Promise对象还有一些重要的方法

- `Promise.all(iterable)`：这个方法返回一个新的 Promise 对象，该 Promise 对象在 iterable 参数对象里所有的 Promise 对象都成功的时候才会触发成功，一旦有任何一个 iterable里面的 Promise 对象失败则立即触发该 Promise 对象的失败。

  ```
  const fetchPromise1 = fetch('https://mdn.github.io/learning-area/javascript/apis/fetching-data/can-store/products.json');
  const fetchPromise2 = fetch('https://mdn.github.io/learning-area/javascript/apis/fetching-data/can-store/not-found');
  const fetchPromise3 = fetch('https://mdn.github.io/learning-area/javascript/oojs/json/superheroes.json');

  Promise.all([fetchPromise1, fetchPromise2, fetchPromise3])
    .then( responses => {
      for (const response of responses) {
        console.log(`${response.url}：${response.status}`);
      }
    })
    .catch( error => {
      console.error(`获取失败：${error}`)
    });
  ```

- Promise.any(iterable)： 接收一个 Promise 对象的集合，当其中的一个 Promise 成功，就返回那个成功的 Promise 的值。

欲了解更多关于Promise的内容，请参见Promise参考文档。

## async和await

async/await 是基于 Promise 的语法糖，使异步代码更易于编写和阅读。通过使用它们，异步代码看起来更像是老式同步代码，因此它们非常值得学习。

`async` 关键字为我们提供了一种更简单的方法来处理基于异步 Promise 的代码。在一个函数的开头添加 `async`，就可以使其成为一个异步函数，函数会返回Promise，而不是直接返回值。

```
let hello = async () => {
    return "hello";
};

hello().then((resolved) => {
    console.log(resolved);
});

console.log(hello());
```

在异步函数中，你可以在调用一个返回 Promise 的函数之前使用 `await` 关键字。这使得代码在该点上等待，直到 Promise 被完成，这时 Promise 的响应被当作返回值，或者被拒绝的响应被作为错误抛出。

我们可以用`async/await`来重写我们的 fetch 示例。

```
async function fetchProducts() {
  try {
    // 在这一行之后，我们的函数将等待 `fetch()` 调用完成
    // 调用 `fetch()` 将返回一个“响应”或抛出一个错误
    const response = await fetch('https://mdn.github.io/learning-
area/javascript/apis/fetching-data/can-store/products.json');
    if (!response.ok) {
      throw new Error(`HTTP 请求错误：${response.status}`);
    }
    // 在这一行之后，我们的函数将等待 `response.json()` 的调用完成
    // `response.json()` 调用将返回 JSON 对象或抛出一个错误
    const json = await response.json();
    console.log(json[0].name);
  }
  catch(error) {
    console.error(`无法获取产品列表：${error}`);
  }
}

fetchProducts();
```

简单总结一下，异步是一种非顺序执行的方式，不用阻塞当前线程来等待任务处理完成，而是允许后续操作，在此任务完成后回调通知此线程。javascript 是单线程的，异步执行实际上是将异步任务加入主线程之外的事件循环中。

# 用JS调用网页中的元素

我们前面讲到过，JavaScript可以在网页上实现复杂的功能，网页展现给你的不再是简单的静态信息，而是实时的内容更新，交互式的地图，2D/3D 动画，滚动播放的视频等等。

这里先举一个简单的例子：

```
<!DOCTYPE html>
<html>

<head>
    <meta charset="utf-8">
    <title>JavaScript示例</title>
</head>

<body>
    <p>Name: 小明</p>
    <script src="script.js" defer></script>
</body>

</html>
```

```
const para = document.querySelector('p');

para.addEventListener('click', updateName);

function updateName() {
    let name = prompt('Please input a new name:');
    para.textContent = 'Name: ' + name;
}
```

那么js是如何调动页面中的元素呢?

# 文档对象模型 (DOM)

**文档对象模型 (DOM)**将 web 页面与到脚本或编程语言连接起来。通常是指 JavaScript，但将 HTML、SVG 或 XML 文档建模为对象并不是 JavaScript 语言的一部分。**DOM 模型用一个逻辑树来表示一个文档**，树的每个分支的终点都是一个节点 (node)，每个节点都包含着对象 (objects)。DOM 的方法 (methods) 让你可以用特定方式操作这个树，用这些方法你可以改变文档的结构、样式或者内容。节点可以关联上事件处理器，一旦某一事件被触发了，那些事件处理器就会被执行。

通过DOM，JavaScript能够改变页面中的HTML元素、HTML属性、CSS样式并对页面中的事件做出反应。

# 查找HTML元素

- 通过id查找HTML元素

  ```
  var x = document.getElementById("intro"); // 查找id为"intro"的元素，未找到返回null
  ```

- 通过标签名查找HTML元素

  ```
  var x=document.getElementById("main");
  var y=x.getElementsByTagName("p");
  //查找 id="main" 的元素，然后查找 id="main" 元素中的所有 <p> 元素
  ```

- 通过类名查找HTML元素

  ```
  var x=document.getElementsByClassName("intro");
  ```

# 改变HTML

## 改变HTML输出流

创建动态的HTML内容 `document.write()`

```
<!DOCTYPE html>
<html>
<body>

<script>
    document.write(Date());
</script>

</body>
</html>
```

## 改变HTML内容

使用`innerHTML`属性

```
<html>
<body>

<p id="p1">Hello World!</p>

<script>
    document.getElementById("p1").innerHTML="新文本!";
</script>

</body>
</html>
```

## 改变HTML属性

使用`.属性名`

`document.getElementById(id).attribute=新属性值`

```
<!DOCTYPE html>
<html>
<body>

<img id="image" src="smiley.gif">

<script>
    document.getElementById("image").src="landscape.jpg";
</script>

</body>
</html>
```

## 改变CSS样式

`document.getElementById(id).style.property=新样式`

```
<!DOCTYPE html>
<html>
<head>
<meta charset="utf-8">
<title>菜鸟教程(runoob.com)</title>
```

```
</head>
<body>

<p id="p1">Hello World!</p>
<p id="p2">Hello World!</p>
<script>
    document.getElementById("p2").style.color="blue";
    document.getElementById("p2").style.fontFamily="Arial";
    document.getElementById("p2").style.fontSize="larger";
</script>
<p>以上段落通过脚本修改。</p>

</body>
</html>
```

## 使用事件

我们可以在事件发生时执行JavaScript。

常见事件：

- 点击鼠标
- 页面加载完成
- 鼠标移入或移出

### 使用事件属性

```
<button onclick="displayDate()">点这里</button>
```

### 使用HTML DOM分配事件

```
<script>
document.getElementById("myBtn").onclick=function(){displayDate()};
</script>
```

### onload和onunload事件

onload 和 onunload 事件会在用户进入或离开页面时被触发。

onload 事件可用于检测访问者的浏览器类型和浏览器版本，并基于这些信息来加载网页的正确版本。
onload 和 onunload 事件可用于处理 cookie。

```
<body onload="checkCookies()">
```

### onchange事件

onchange 事件常结合对输入字段的验证来使用。

```
<input type="text" id="fname" onchange="upperCase()">
```

### onmouseover 和 onmouseout 事件

onmouseover 和 onmouseout 事件可用于在用户的鼠标移至 HTML 元素上方或移出元素时触发函数。

## 事件监听器

### addEventListener()

```
document.getElementById("myBtn").addEventListener("click", displayDate);
```

addEventListener() 方法用于向指定元素添加事件句柄。

addEventListener() 方法添加的事件句柄不会覆盖已存在的事件句柄。

你可以向一个元素添加多个事件句柄。

你可以向同个元素添加多个同类型的事件句柄，如：两个 "click" 事件。

你可以向任何 DOM 对象添加事件监听，不仅仅是 HTML 元素。如： window 对象。

addEventListener() 方法可以更简单的控制事件（冒泡与捕获）。

当你使用 addEventListener() 方法时, JavaScript 从 HTML 标记中分离开来，可读性更强， 在没有控制 HTML标记时也可以添加事件监听。

你可以使用 removeEventListener() 方法来移除事件的监听。

```
element.addEventListener(event, function, useCapture);
```

- event：事件类型（注意不要使用"on"前缀）
- function：事件触发后调用的函数
- useCapture（Boolean）：描述事件是冒泡还是捕获，可选，默认false冒泡

> 事件传递有两种方式：冒泡与捕获。
>
> 事件传递定义了元素事件触发的顺序。 如果你将 `<p>` 元素插入到 `<div>` 元素中，用户点击 `<p>` 元素, 哪个元素的 "click" 事件先被触发呢?
>
> 在 **冒泡** 中，内部元素的事件会先被触发，然后再触发外部元素，即：`<p>` 元素的点击事件先触发，然后会触发 `<div>` 元素的点击事件。
>
> 在 **捕获** 中，外部元素的事件会先被触发，然后才会触发内部元素的事件，即： `<div>` 元素的点击事件先触发 ， 然后再触发 `<p>` 元素的点击事件。

### removeEventListener()

emoveEventListener() 方法移除由 addEventListener() 方法添加的事件句柄:

```
element.removeEventListener("mousemove", myFunction);
```