



# DOM和BOM

DOM (Document Object Model, 文档对象模型) 用于将文档中对象按照树形结构组织起来, 用户能够方便的通过DOM获取元素、遍历元素、修改和创建元素、设置元素属性、进行事件监听和处理。

DOM用于将HTML文档描述成文档对象, 该对象被组织成由一个个节点 (Node) 组成的树形结构。树中节点有元素节点、属性节点、文本节点等。节点之间存在父子、兄弟等层级关系, 每个节点有自己的名称、类型以及其他一些属性。同时DOM提供了一系列节点操作相关的API, 用于访问和处理元素。

在最初的JavaScript语言中, DOM用于处理事件和文档的有限功能, 被称为“DOM 0级”, 但当时并没有独立的标准, 只在HTML 4规范中进行了部分描述。1998年10月, DOM 1级成为W3C推荐标准, DOM 2级在2000年发布, 更新发布了包括DOM事件、样式等内容; 2004年DOM 3级新增了Xpath等内容。

# 一. DOM

## 1、DOM的作用

**获取节点：**获取文档的节点，例如获取文档的某节点的属性，遍历列表节点的内容，获得表单的输入值等。

DOM获取节点后，还可以对节点的属性进行改变。

```
<label for="username-input">用户名: </label><input type="text" id="username-input" />
<script>
| let userNameInput = document.getElementById('username-input')
</script>
```

**监听事件：**监听事件并做出响应，例如用户的单击事件、改变窗口的大小、表单提交等操作。

```
<label for="username-input">用户名: </label><input type="text" id="username-input" />
<button id="validate-btn">验证用户名</button>
<script>
| document.getElementById('validate-btn').onclick = function () {
| | //事件处理
| }
</script>
```

DOM树，是用于将HTML文档组织成一个树形结构。下边图形即为一个DOM树。

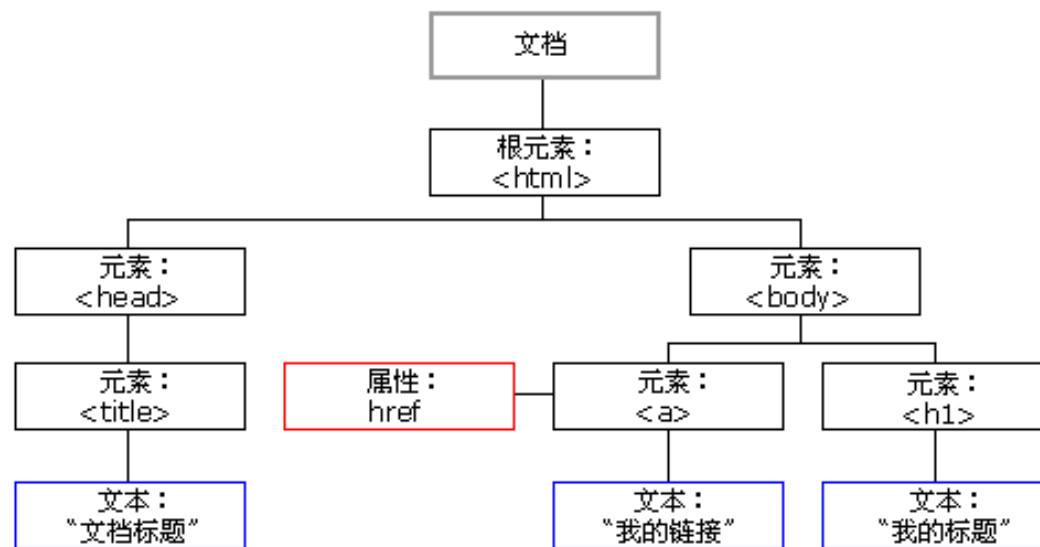
每个节点表示一个Node对象，可以通过nodeType属性来表明节点类型，常见的节点类型有以下几种：

(1) Document类型：表示整个HTML页面文档，document对象也是window对象的一个属性，可以作为全局对象来访问。

(2) Element类型：元素节点，通过访问其id、title等属性，可以获得该节点的相关属性信息。

(3) Attr类型：属性节点。

(4) Text类型：纯文本节点，不包含HTML代码。注意除了元素内文本内容可以生成Text类型节点外，标签换行处也可以生成一个空白Text类型节点。



## HTML DOM

- HTML DOM 是 HTML 的标准对象模型和编程接口。它定义了：
- 作为对象的 HTML 元素
- 所有 HTML 元素的属性
- 访问所有 HTML 元素的方法
- 所有 HTML 元素的事件

换言之：HTML DOM 是关于如何获取、更改、添加或删除 HTML 元素的标准。

## 获取元素

对象类型	方法	说明
Document对象	getElementById()	通过元素id属性获取元素
Document对象	getElementsByName()	通过元素name属性获取元素
Document对象、Element对象	getElementsByTagName()	通过标签名获取元素
Document对象、Element对象	getElementsByClassName()	通过CSS类名获取元素
Document对象、Element对象	querySelectorAll()	通过选择器获取元素
Document对象、Element对象	querySelector()	通过选择器获取第一个元素

## 获取元素

1、`getElementById()`，Document对象提供的，通过元素的id属性获取元素，返回值是object类型。id属性的唯一性，所以获取单一元素时推荐使用这种方法。但并不是所有元素都有id。

```
<ul id="myList">
  <li id="m1">首页</li>
  <li id="m2">企业介绍</li>
  <li id="m3">联系我们</li>
</ul>
<script>
  var ul = document.getElementById('myList')
  console.log(ul)
</script>
```

效率非常高！

2、`getElementsByName()`，Document对象提供的，通过元素的name属性获取元素，返回值为NodeList对象。name属性以及`getElementsByName()`常用于表单元素。

```
<form id="registerForm">
  <input type="checkbox" name="boy" />
  <input type="checkbox" name="boy" />
  <input type="checkbox" name="boy" />
</form>
<script>
  var list = document.getElementsByName('boy')
  console.log(typeof list)
</script>
```

```
▼ NodeList(3) [input, input, input] ⓘ
  ▶ 0: input
  ▶ 1: input
  ▶ 2: input
    length: 3
  ▶ [[Prototype]]: NodeList
```

object

# DOM操作

## 案例-生成随机验证码

```
<div id="code"></div>
<button class="btn" onclick="getRandomCode()">生成随机码</button>
<script>
  function getRandomCode() {
    let characters = 'abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ1234567890'
    let strCode = ''
    for (let i = 0; i < 4; i++) {
      let codePos = Math.floor(Math.random() * characters.length)
      strCode += characters[codePos]
    }
    document.getElementById('code').innerHTML = strCode
  }
</script>
```



## 获取元素

3、`getElementsByName()`，`Document`对象和`Element`对象都有提供，用于获取指定标签名的元素，返回值为`HTMLCollection`。

(1) `Document`对象的`getElementByTagName()`方法

```
<ul id="myList">
  <li id="m1">首页</li>
  <li id="m2">企业介绍</li>
  <li id="m3">联系我们</li>
</ul>

<script>
  var ul = document.getElementById('myList')
  var list = ul.getElementsByTagName('li')
  console.log(list)
</script>
```

强调：

- 1、可用在任意父元素上
- 2、不仅查直接子节点，而且查所有子代节点
- 3、返回一个动态集合，即使只找到一个元素，也返回集合，必须用[0]，取出唯一元素。

注意：`getElementsByTagName()`方法返回`NodeList`对象，而`getElementsByName()`方法返回`HTMLCollection`对象。它们的相同点在于里面都包含了所需元素的集合；不同点在于`NodeList`对象中包含了`entries`、`foreach`、`item`、`keys`和`values`方法，而`HTMLCollection`对象只有一个`namedItem`方法；而且`HTMLCollection`只包含HTML元素，而`NodeList`还包括文本、属性等。

## 获取元素

4、`getElementsByClassName()`，`Document`对象和`Element`对象都有提供，可以根据CSS类名获取元素，返回值为`HTMLCollection`。参数可以是一个类名，也可以是多个类名，多个类名的时候采用空格分隔。比如“`class-a class-b`”。类名先后顺序无关。有兼容性问题：IE9+。

```
<div id="news">
  <p class="mainTitle">新闻标题1</p>
  <p class="subTitle">新闻标题2</p>
  <p class="mainTitle">新闻标题3</p>
</div>
<script>
  var div = document.getElementById('news')
  var list = div.getElementsByClassName('mainTitle')
  console.log(list)
</script>
```

## 获取元素

5、 `querySelectorAll()` 和 `querySelector()`，通过选择器获取元素。参数是选择器， `querySelectorAll()` 返回符合选择器的所有节点的 `NodeList`， `querySelector()` 返回第一个匹配的元素。

```
<ul id="task-list-wrap">
  <li class="done delay">任务1</li>
  <li class="done">任务2</li>
  <li class="doing">任务3</li>
  <li class="waiting">任务4</li>
  <li class="waiting delay">任务5</li>
  <li class="doing delay">任务6</li>
  <li class="doing">任务7</li>
</ul>
<a href="task.html">任务要求</a>
<a href="task.html">计划进度</a>
<a href="team.html">团队</a>
<script type="text/javascript">
  console.log(document.querySelector('#task-list-wrap'))
  console.log(document.querySelectorAll('.delay'))
  console.log(document.querySelector('.delay'))
  console.log(document.querySelectorAll('[href="task.html"]'))
</script>
```

```
▼ <ul id="task-list-wrap">
  ▶ <li class="done delay">...</li>
  ▶ <li class="done">...</li>
  ▶ <li class="doing">...</li>
  ▶ <li class="waiting">...</li>
  ▶ <li class="waiting delay">...</li>
  ▶ <li class="doing delay">...</li>
  ▶ <li class="doing">...</li>
</ul>
```

```
▼ NodeList(3) [li.done.delay, li.waiting.delay, li.doing.delay] ⓘ
  ▶ 0: li.done.delay
  ▶ 1: li.waiting.delay
  ▶ 2: li.doing.delay
    length: 3
  ▶ [[Prototype]]: NodeList
```

```
▼ <li class="done delay">
  ::marker
  "任务1"
</li>
```

```
▼ NodeList(2) [a, a] ⓘ
  ▶ 0: a
  ▶ 1: a
    length: 2
  ▶ [[Prototype]]: NodeList
```

## 遍历元素

在DOM操作中经常会需要查找某个元素的其他相关元素，比如它的父元素节点、兄弟元素节点或者子结点，可以通过遍历树的方式来遍历文档结构。DOM树中的Node对象包括Document对象、Element对象等，都提供了一些属性来进行遍历操作。

属性	功能	说明
parentNode	当前节点的父节点	元素节点的父节点也是元素节点
childNodes	当前节点的直接子节点	如果只想取得元素节点，使用children属性
firstChild	当前节点的第一个子节点	如果只想取得元素节点，使用firstElementChild属性
lastChild	当前节点的最后一个子节点	如果只想取得元素节点，使用lastElementChild属性
previousSibling	当前节点的前一个兄弟节点	如果只想取得元素节点，使用previousElementSibling属性
nextSibling	当前节点的下一个兄弟节点	如果只想取得元素节点，使用nextElementSibling属性
nodeType	节点类型	1、ELEMENT_NODE 元素节点；2、ATTRIBUTE_NODE 属性节点；3、TEXT_NODE 文本节点
nodeName	节点名	元素节点：返回大写标签名；属性节点：返回属性名；文本节点：返回#text
nodeValue	节点值	元素节点：null；属性节点：返回属性值；文本节点：返回文本内容

## 遍历元素

```
<ul id="list">
  <li>html</li>
  <li>css</li>
  <li>js</li>
</ul>
<script>
  let ul = document.getElementById('list')
  console.log(ul.childNodes)
  console.log(ul.childNodes.length)
  console.log(ul.children)
  console.log(ul.childElementCount)
  let lis = ul.children
  for (let i = 0; i < lis.length; i++) {
    console.log(`元素节点类型${lis[i].nodeType}, 元素节点值
    ${lis[i].nodeName}`)
  }
</script>
```

▼ NodeList(7) ⓘ

▶ 0: text

▶ 1: li

▶ 2: text

▶ 3: li

▶ 4: text

▶ 5: li

▶ 6: text

length: 7

▶ [[Prototype]]: NodeList

7

▼ HTMLCollection(3) ⓘ

▶ 0: li

▶ 1: li

▶ 2: li

length: 3

▶ [[Prototype]]: HTMLCollection

3

③ 元素节点类型1, 元素节点值LI

## 访问元素属性

元素属性分为内置属性和自定义属性。如src、href等都是内置属性。

### 1、访问元素内置属性

对于内置属性，可以通过属性名对属性进行值的获取和设定，例如：

```

<script type="text/javascript">
  let banner = document.getElementById('banner')
  console.log(banner.id, banner.alt, banner.src)
  banner.src = 'img-banner-new.png'
</script>
```

## 访问元素属性

### 2、访问自定义属性

HTML5支持通过“data-”设置自定义属性。对于自定义属性，可以通过`getAttribute()`和`setAttribute()`方法来获取和设置。

```
<div id="product-123" data-product-id="123" data-product-name="web" data-product-status="unread"></div>
<script type="text/javascript">
  let prod = document.getElementById('product-123')
  let productObj = {
    id: prod.getAttribute('data-product-id'),
    name: prod.getAttribute('data-product-name')
  }
  prod.setAttribute('data-product-status', 'read')
  console.log(productObj)
  console.log(prod.getAttribute('data-product-status'))
</script>
```

`hasAttribute()`：判断属性是否存在；

`removeAttribute()`：删除属性。

► `{id: '123', name: 'web'}`

read

## 访问和修改文档内容

可以通过innerHTML属性完成，该属性可以是包含HTML的字符串。例如将数组fruits内容生成一个列表：

```
<ul id="fruit-list-wrap"></ul>
<script type="text/javascript">
  const fruits = ['Apple', 'Orange', 'Grape']
  let htmlStr = ''
  for (let i = 0; i < fruits.length; i++) {
    htmlStr += `<li>${fruits[i]}</li>`
  }
  document.getElementById('fruit-list-wrap').innerHTML = htmlStr
</script>
```

- Apple
- Orange
- Grape

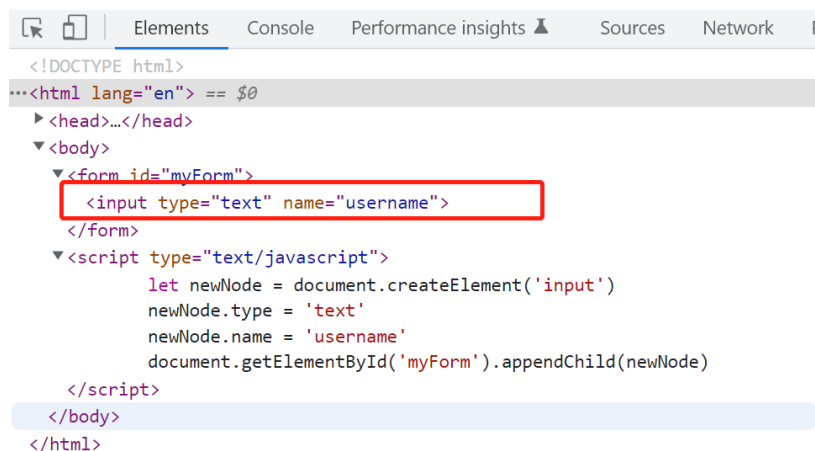


## 创建和删除元素

### 1、创建元素

分两个步骤：首先通过createElement()方法创建元素，然后通过appendChild()方法添加新节点。例如在id为“myForm”的表单中添加一个文本框，代码如下：

```
<form id="myForm"></form>
<script type="text/javascript">
  let newNode = document.createElement('input')
  newNode.type = 'text'
  newNode.name = 'username'
  document.getElementById('myForm').appendChild(newNode)
</script>
```



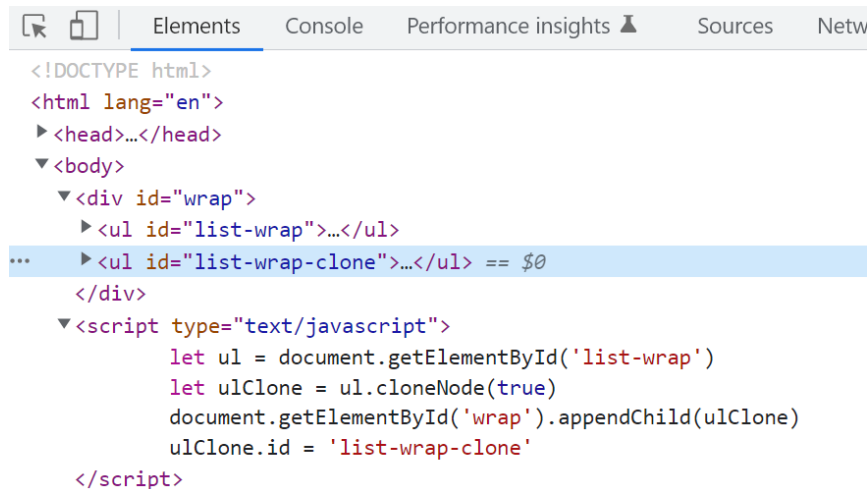
```
<!DOCTYPE html>
<html lang="en"> == $0
  <head>...</head>
  <body>
    <form id="myForm">
      <input type="text" name="username">
    </form>
    <script type="text/javascript">
      let newNode = document.createElement('input')
      newNode.type = 'text'
      newNode.name = 'username'
      document.getElementById('myForm').appendChild(newNode)
    </script>
  </body>
</html>
```

## 创建和删除元素

### 2、克隆节点

可以通过`cloneNode()`进行节点的复制，如果参数为`true`则克隆本节点及其所有后代节点，如果参数为`false`则只克隆节点本身。例如：

```
<div id="wrap">
  <ul id="list-wrap">
    <li>hello</li>
  </ul>
</div>
<script type="text/javascript">
  let ul = document.getElementById('list-wrap')
  let ulClone = ul.cloneNode(true)
  document.getElementById('wrap').appendChild(ulClone)
  ulClone.id = 'list-wrap-clone'
</script>
```



```
Elements Console Performance insights Sources Netw
<!DOCTYPE html>
<html lang="en">
  <head>...</head>
  <body>
    <div id="wrap">
      <ul id="list-wrap">...</ul>
      <ul id="list-wrap-clone">...</ul> == $0
    </div>
    <script type="text/javascript">
      let ul = document.getElementById('list-wrap')
      let ulClone = ul.cloneNode(true)
      document.getElementById('wrap').appendChild(ulClone)
      ulClone.id = 'list-wrap-clone'
    </script>
```

## 创建和删除元素

### 3、删除元素

可以通过`removeChild()`方法进行节点的删除，例如：

```
parentNode.removeChild(childrenNode); //先定位父节点，再删除子结点
```

```
someNode.parentNode.removeChild(someNode); //无须定位父节点，直接删除子结点
```

## 访问和修改样式

### 1、style属性

通过元素的style属性可以访问和修改元素的样式。例如，修改p元素的样式，代码如下：

```
p.style.fontSize= "18px" ; //设置字号，带单位px
```

```
p.style.backgroundColor = "orange" ; // 设置背景颜色
```

属性名：去横线，变驼峰

```
background-color => backgroundColor
```

```
list-style-type => listStyleType
```

### 2、cssText属性

在要修改多个属性的时候，如果分别修改各个属性会导致浏览器不断重新处理或绘制页面，产生页面回流（reflow），使用cssText则只会触发一次，提高页面性能，例如：

```
p.style.cssText= "font-size:18px; background-color:orange;"
```

## 访问和修改样式

### 3、getComputedStyle() 方法

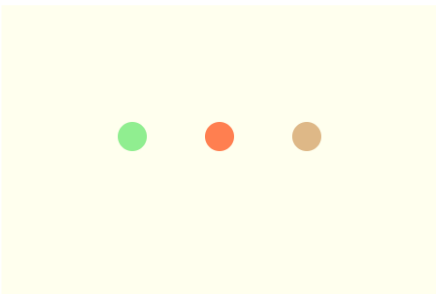
style属性只能获取行内样式，使用getComputedStyle()方法，可以获得一个元素的全部样式。

```
<div id="wrap" style="font-size: 18px; width: 200px"></div>
<script type="text/javascript">
  let wrap = document.getElementById('wrap')
  console.log(wrap.style.width)
  console.log(wrap.style.height)
  console.log(wrap.style.fontSize)
  console.log(wrap.style.backgroundColor)
  let computedStyle = window.getComputedStyle(wrap, null)
  console.log(computedStyle.width)
  console.log(computedStyle.height)
  console.log(computedStyle.fontSize)
  console.log(computedStyle.backgroundColor)
</script>
```

200px
18px
200px
300px
18px
rgb(255, 165, 0)
>

# DOM操作

例子：定义点，包括位置、颜色等属性，并创建3个点的对象，将其用绝对定位在页面上显示出来。



```
<body>
  <div id="box"></div>
  <script>
    let box = document.getElementById('box')
    function Point(x, y, color) {
      this.x = x
      this.y = y
      this.color = color
    }
    Point.prototype.createDiv = function () {
      this.div = document.createElement('div')
      box.appendChild(this.div)
      //设置div样式
      this.div.style.width = 20 + 'px'
      this.div.style.height = 20 + 'px'
      this.div.style.borderRadius = 10 + 'px' //用圆角边框样式生成点
      this.div.style.backgroundColor = this.color
      this.div.style.position = 'absolute'
      this.div.style.left = this.x + 'px'
      this.div.style.top = this.y + 'px'
    }
    //用构造函数创建点对象
    let p1 = new Point(80, 80, 'lightgreen')
    let p2 = new Point(140, 80, 'coral')
    let p3 = new Point(200, 80, 'burlywood')
    //用原型方法创建元素，添加到页面
    p1.createDiv()
    p2.createDiv()
    p3.createDiv()
  </script>
</body>
```

## 事件概述

各种鼠标、键盘操作或者浏览器自身加载资源等状态，都是通过事件（Event）机制来实现JavaScript和HTML的交互。事件相关概念：

1. 事件目标（Event Target）：指的是触发此事件的对象，Window、Document、Element是比较常见的事件目标，比如某一个按钮、输入框等；
2. 事件类型（Event Type）：指具体发生了什么事，单击事件类型是click，鼠标移动事件类型是mousemove；
3. 事件监听程序（Event Listener）：是指响应时间的处理函数；
4. 事件对象（Event Object）：是指包含事件详细信息对象；
5. 事件流模型：是事件的传播方式，由于事件目标常常是逐层嵌套在其他元素内部的，所以事件传播方式按照传播方向分两种，一种是自内向外的冒泡，另一种是自外向内的捕获。

## 事件类型

- 文档事件、用户界面事件、鼠标事件、键盘事件、表单事件等。
- 文档事件：包含HTML文档自身关于文档加载、错误异常的一些事件，比如load、unload、error事件等；
- 用户界面事件：包括页面的focus事件、blur事件、resize事件、scroll事件等；
- 鼠标事件：包括click事件、mousedown事件、mousemove事件、mouseup事件、mousewheel(鼠标滚轮)事件等
- 键盘事件：包括keydown事件、keyup事件、keypress(键盘按下下一个字符)事件等；
- 表单事件：包括focus事件、blur事件、input(输入框发生输入时)事件、submit事件等。

鼠标事件	触发条件
onclick	鼠标点击左键触发
onmouseover	鼠标经过触发
onmouseout	鼠标离开触发
onfocus	获得鼠标焦点触发
onblur	失去鼠标焦点触发
onmousemove	鼠标移动触发
onmouseup	鼠标弹起触发
onmousedown	鼠标按下触发



## 事件处理

如果需要事件发生时响应该事件，那么需要对这个事件注册一个处理函数，也就是事件监听程序。

- HTML行内注册

最简单的方式就是对元素的事件属性进行行内赋值，事件属性名是在事件名前面加上“on”。比如click事件，onclick就是click的事件属性名称。

```
<button onclick="showInfo()">点击一下</button>
<script>
  function showInfo() {
    alert('这是一个警告框!')
  }
</script>
```

## 事件处理

- JavaScript 传统方式注册事件

在JavaScript中注册事件处理函数，将函数名赋值给元素的事件属性：

```
<button id="btn">点击一下</button>
<script>
  let btn = document.getElementById('btn')
  btn.onclick = show
  function show() {
    alert('Hello~')
  }
</script>
```

也可以使用匿名函数，比如：

```
<button id="btn">点击一下</button>
<script>
  let btn = document.getElementById('btn')
  btn.onclick = function () {
    alert('Hello~')
  }
</script>
```

要删除指定的事件处理函数，只需要把事件处理属性设置为null即可。

```
btn.onclick=null
```

## 事件处理

- JavaScript 事件侦听方式注册事件

上述方法一个事件只能注册一个函数，否则后边的会覆盖前边的注册，可以通过`addEventListener()`进行事件的监听，对一个事件注册多个处理函数，同时还可以应用`removeEventListener()`方法来取消监听，但此时事件处理函数不能为匿名函数。

```
<button id="btn">点击</button>
<script>
  let btn = document.getElementById('btn')
  btn.addEventListener(
    'click',
    function () {
      alert('Hello~')
    },
    false
  )
</script>
```

```
<button id="btn">点击</button>
<script>
  let btn = document.getElementById('btn')
  btn.addEventListener('click', validateUsername, false)
  btn.addEventListener('click', validatePassword, false)
  btn.removeEventListener('click', validatePassword, false) // 删除事件监听
</script>
```

## 事件处理

### 传统注册方式

- 利用on开头的事件onclick
- 行内注册以及`btn.onclick=function() {}`
- 特点：注册事件的唯一性
- 同一个元素同一个事件只能设置一个处理函数，最后注册的处理函数将会覆盖前面注册的处理函数

### 方法监听注册方式

W3C标准推荐方式

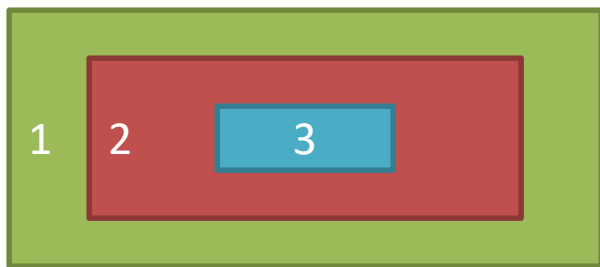
`addEventListener()`，是一个方法

IE9之前的不支持

特点：同一个元素同一个事件可以注册多个监听器，按注册顺序依次执行

## 事件流模型

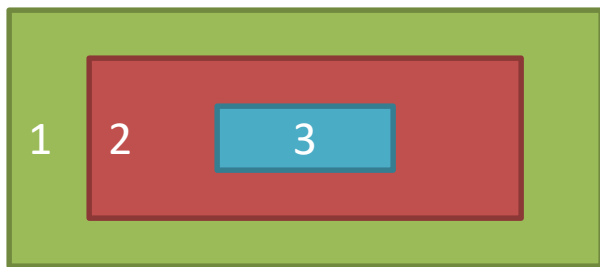
事件流模型分为冒泡和捕获两种。在早期IE浏览器上是冒泡方式，事件从直接触发事件的节点接收，然后逐级向上传播到document；而Netscape提出了捕获的事件流模型，事件是从document开始接收，向下传播到具体节点。比如在一个页面中有多层嵌套的数个节点，如图。当用户单击3时，也可以认为用户单击了1和2. 事件冒泡的响应顺序为3-2-1. 而事件捕获的方式刚好相反，是1-2-3.



## 事件流模型

### 1、事件冒泡

事件从最具体、嵌套层级最深的节点开始接收，然后逐级向上传播，这种称为事件冒泡，目前所有浏览器都支持事件冒泡。  
验证事件冒泡：



2、如果事件是先从最上层节点开始接收，逐级传递到最具体的节点，这种方式称为捕获。

addEventListener方式注册事件处理函数，第三个参数改为true

## 事件对象

当事件发生时浏览器会产生事件对象event，可以在事件处理函数中定义参数接收event对象，它提供了事件类型、目标事件相关的信息，以及一些组织冒泡、取消默认行为等方法。

常用成员

对象	说明
type	事件类型，如load、click等
target	事件目标：直接触发此事件的元素
currentTarget	事件目标：直接元素或者是事件传播中的父元素
eventPhase	返回之间传播的当前阶段，取值为：1 捕获阶段；2 目标阶段；3 冒泡阶段
clientX、clientY	鼠标坐标，相对于浏览器可视区
screenX、screenY	鼠标坐标，相对于屏幕
pageX、pageY	鼠标坐标，相对于页面
preventDefault()	不要执行与事件关联的默认动作
stopPropagation()	阻止事件冒泡

## 事件对象

应用event对象完成表单验证，当单击“提交”按钮，或者键盘抬起Enter键时，验证用户名是否为空，如果是就给出提示，并取消表单提交行为。

```
<form action="" id="form">
  <label for="my-input">用户名: </label><input type="text" name="username" id="my-input" />
  <input type="submit" value="提交" />
</form>
<script>
  document.getElementById('my-input').onkeyup = function (event) {
    if (event.keyCode == 13) {
      document.getElementById('form').submit() //等同于按下submit按钮
    }
  }
  //表单提交事件处理函数
  document.getElementById('form').onsubmit = function (event) {
    if (document.getElementById('my-input').value === '') {
      alert('用户名不能为空')
      event.preventDefault() //用户名为空取消默认提交表单行为
    }
  }
</script>
```



# DOM事件

## 事件对象

### this关键字

事件触发时，事件处理函数中可以使用this关键字表示触发事件的当前对象。例如单击div盒子，使其里边的文字字体变大。

```
<style>
  #my-div {
    margin: 100px auto;
    width: 300px;
    height: 300px;
    background-color: orange;
    font-size: 16px;
  }
</style>
</head>

<body>
  <div id="my-div">Hello</div>
  <script>
    let myDiv = document.getElementById('my-div')
    myDiv.onclick = function () {
      this.style.fontSize = '20px'
      this.style.backgroundColor = 'lightblue'
    }
  </script>
</body>
```

# DOM事件

例：

实现标签页内容切换。鼠标指针悬停在标签标题上时，标题高亮显示，下方内容区域随之切换。



```
<div class="tab">
  <ul class="tab-title">
    <li class="active">标签1</li>
    <li>标签2</li>
    <li>标签3</li>
  </ul>
  <div class="tab-content">
    <div class="item">内容1</div>
    <div class="item">内容2</div>
    <div class="item">内容3</div>
  </div>
</div>
```

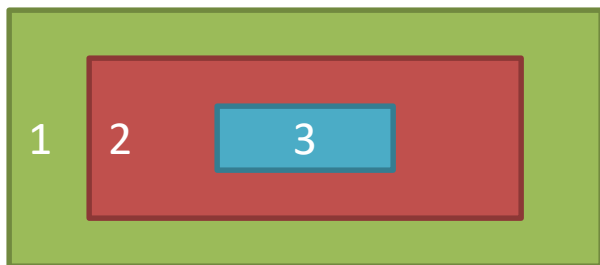
```
<script>
  let tabTitle = document.querySelector('.tab-title')
  let lis = tabTitle.querySelectorAll('li')
  let items = document.querySelectorAll('.item')
  window.onload = function () {
    // 默认显示第一项内容
    items[0].style.display = 'block'
  }
  for (let i = 0; i < lis.length; i++) {
    // 标题添加自定义属性data-index
    lis[i].setAttribute('data-index', i)
    lis[i].onmouseover = function () {
      // 鼠标指针悬停，所有标题取消高亮
      for (let i = 0; i < lis.length; i++) {
        lis[i].className = ''
      }
      // 当前标题高亮显示
      this.className = 'active'
      // 当前标题下标，对应着内容下标
      let dataIndex = this.getAttribute('data-index')
      // 所有内容隐藏
      for (let i = 0; i < items.length; i++) {
        items[i].style.display = 'none'
      }
      // 显示当前标题对应的内容
      items[dataIndex].style.display = 'block'
    }
  }
</script>
```

# DOM事件

target和currentTarget属性：

在事件冒泡中，可以用event对象中的target和currentTarget属性来详细查看事件当前正在处理的元素和目标阶段处理的元素，target是指获取事件的目标，currentTarget是指其事件处理程序当前正在处理事件的那个元素。另外也可以通过stopPropagation()方法来阻止冒泡。

比如：查看事件冒泡时的事件目标。



# DOM事件

## 事件委托:

事件委托是利用事件冒泡机制将事件处理函数注册在上层元素上，以便对下层元素进行统一处理。

原理：不再单独给每个子结点设置事件监听器，而是把这个事件监听器设置在父节点上，然后利用冒泡原理影响设置每个子结点。

- DOM查找
- DOM修改
- DOM添加
- DOM删除

```
<ul class="ul-wrap">
  <li>DOM查找</li>
  <li>DOM修改</li>
  <li>DOM添加</li>
  <li>DOM删除</li>
</ul>

<script>
  let ulWrap = document.querySelector('.ul-wrap')
  ulWrap.addEventListener(
    'click',
    function (e) {
      alert('DOM操作~')
    },
    false
  )
</script>
```

# DOM事件

e.target 和 this 的区别：

this 是事件绑定的元素， 这个函数的调用者（绑定这个事件的元素）

e.target 是事件触发的元素。

```
<ul class="ul-wrap">
  <li>DOM查找</li>
  <li>DOM修改</li>
  <li>DOM添加</li>
  <li>DOM删除</li>
</ul>
<script>
  let ulWrap = document.querySelector('.ul-wrap')
  ulWrap.addEventListener(
    'click',
    function (e) {
      alert('DOM操作~')
      console.log(e.target)
      console.log(this)
    },
    false
  )
</script>
```

---

▶ <li>...</li>

---

▶ <ul class="ul-wrap">...</ul>

[Violation] 'click' handler took 1560ms

>

## 什么是 BOM

BOM (Browser Object Model) 即**浏览器对象模型**，它提供了独立于内容而与**浏览器窗口进行交互的对象**，其核心对象是 window。

BOM 由一系列相关的对象构成，并且每个对象都提供了很多方法与属性。

BOM 缺乏标准，JavaScript 语法的标准化组织是 ECMA，DOM 的标准化组织是 W3C，BOM 最初是 Netscape 浏览器标准的一部分。

### DOM

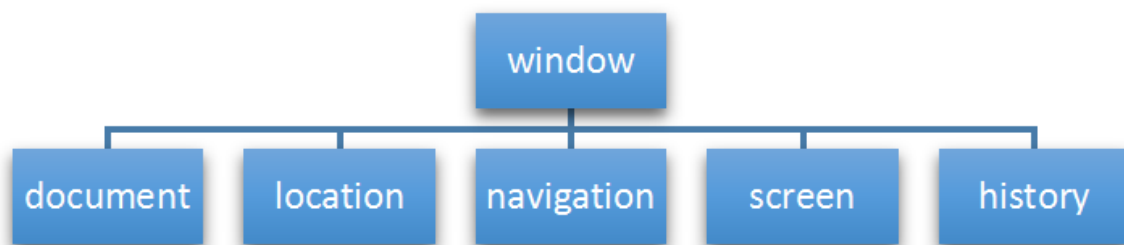
- 文档对象模型
- DOM 就是把「**文档**」当做一个「**对象**」来看待
- DOM 的顶级对象是 **document**
- DOM 主要学习的是操作页面元素
- DOM 是 W3C 标准规范

### BOM

- 浏览器对象模型
- 把「**浏览器**」当做一个「**对象**」来看待
- BOM 的顶级对象是 **window**
- BOM 学习的是浏览器窗口交互的一些对象
- BOM 是浏览器厂商在各自浏览器上定义的，兼容性较差

## BOM 的构成

BOM 比 DOM 更大，它包含 DOM。DOM的document对象也是window对象的属性。



**window 对象是浏览器的顶级对象**，它具有双重角色。

1. 它是 JS 访问浏览器窗口的一个接口。
2. 它是一个全局对象。定义在全局作用域中的变量、函数都会变成 window 对象的属性和方法。

在调用的时候可以省略 window，前面学习的对话框都属于 window 对象方法，如 alert()、prompt() 等。

**注意：** window下的一个特殊属性 window.name

window代表整个浏览器窗口

history封装当前窗口打开后，成功访问过的历史url记录

navigation封装浏览器配置信息

document封装当前正在加载的网页内容

location封装了当前窗口正在打开的url地址

screen封装了屏幕的信息

获取当前窗口大小:

完整窗口大小: window.outerWidth/outerHeight, 包含标题栏、工具栏的完整宽度、高度

文档显示区大小: window.innerWidth/innerHeight, 工作区的宽度、高度

**window**对象的常用方法:

属性	说明
open()	打开一个新的浏览器窗口。例如: let mWin=window.open(" <a href="https://www.baidu.com">https://www.baidu.com</a> ")
close()	关闭浏览器窗口。mWin.close()
alert()	显示警告框, 包含一段消息和一个确认按钮
prompt()	显示对话框, 提示用户输入
confirm()	显示对话框, 包含一段消息、确认按钮和取消按钮



## window对象的常见事件

### 1、窗口加载事件

```
window.onload=function(){}
```

或者

```
window.addEventListener( "load" , function(){});
```

window.onload()是窗口（页面）加载事件，当文档内容完全加载完会触发该事件（包括图像、脚本文件、CSS文件等）就调用处理函数。

注意：

- 1、有了window.onload()，就可以把JS代码写在页面元素的上方，因为onload是等页面内容全部加载完毕，再去执行处理函数。
- 2、 window.onload()传统注册方式，只能写一次，如果有多个，会以最后一个window.onload为准。
- 3、如果使用addEventListener则没有限制。

## window对象的常见事件

### 1、窗口加载事件

```
document.addEventListener( 'DOMContentLoaded' ,function(){})
```

DOMContentLoaded事件触发时，仅当DOM加载完成，不包括样式表、图片、flash等等。

IE9以上才支持

如果页面图片比较多的话，从用户访问到onload事件触发可能需要比较长的时间，交互效果就不能实现，必然影响用户的体验，此时用DOMContentLoaded事件比较合适。

## window对象的常见事件

### 2、调整窗口大小事件

```
window.onresize=function(){} 
```

```
window.addEventListener( "resize" , function(){});
```

window.onresize是调整窗口大小加载事件，当触发时就调用的处理函数

注意：

- 1、只要窗口大小发生像素变化，就会触发这个事件。
- 2、经常利用这个事件去完成响应式布局。我们可以用window.innerWidth去获得当前屏幕的宽度

# 定时器

## 两种定时器

window 对象提供了 2 个非常好用的方法-**定时器**。

- setTimeout()
- setInterval()

### 1、setTimeout() 定时器

```
window.setTimeout(调用函数, [延迟的毫秒数]);
```

setTimeout() 方法用于设置一个定时器，该定时器在定时器到期后执行调用函数。

#### **注意：**

1. window 可以省略。
2. 这个调用函数可以**直接写函数**，**或者写函数名**或者采取字符串 **‘函数名()’** 三种形式。第三种不推荐。
3. 延迟的毫秒数省略默认是 0，如果写，必须是毫秒。
4. 因为定时器可能有很多，所以我们经常给定时器赋值一个标识符。

# 定时器

## 两种定时器

### 1、setTimeout() 定时器

```
window.setTimeout(调用函数, [延迟的毫秒数]);
```

- setTimeout() 里的这个调用函数称为回调函数 callback
- 普通函数是按照代码顺序直接调用。
- 而这个函数，需要等待时间，时间到了才去调用这个函数，因此称为回调函数。
- 简单理解：回调，就是回头调用的意思。上一件事干完，再回头再调用这个函数。
- 例如前边遇到的element.onclick = function(){} 或者 element.addEventListener("click", fn); 里面的函数也是回调函数。

## 5秒后自动关闭广告

① 核心思路：5秒之后，就把这个广告隐藏起来

② 用定时器setTimeout

```
<body>
  
  <script>
    let shuke = document.querySelector('.shuke');
    setTimeout(function() {
      shuke.style.display = 'none';
    }, 5000);
  </script>
</body>
```

# 定时器

## 停止 setTimeout() 定时器

```
window.clearTimeout(timeoutID)
```

clearTimeout() 方法取消了先前通过调用 setTimeout() 建立的定时器。

### 注意：

1. window 可以省略。
2. 里面的参数就是定时器的标识符。

```
<body>
  <script>
    window.onload = function () {
      let btn = document.querySelector('button')
      let timer = setTimeout(function () {
        console.log('Hello~')
      }, 5000)
      btn.addEventListener('click', function () {
        clearTimeout(timer)
      })
    }
  </script>
  <button>点击停止</button>
</body>
```

# 定时器

## 2、setInterval() 定时器

```
window.setInterval(回调函数, [间隔的毫秒数]);
```

setInterval() 方法重复调用一个函数，每隔这个时间，就去调用一次回调函数。

### 注意：

1. window 可以省略。
2. 这个调用函数可以**直接写函数**，**或者写函数名**或者采取字符串 '**函数名()**' 三种形式。
3. 间隔的毫秒数省略默认是 0，如果写，必须是毫秒，表示每隔多少毫秒就自动调用这个函数。
4. 因为定时器可能有很多，所以我们经常给定时器赋值一个标识符。
5. 第一次执行也是间隔毫秒数之后执行，之后每隔毫秒数就执行一次。



## 秒杀倒计时



- ① 倒计时是不断变化的，因此需要用定时器（setInterval）
- ② 三个红色盒子里面分别存放时分秒
- ③ 三个红色盒子利用innerHTML 放入计算的小时分钟秒数

# 定时器

## 停止 setInterval() 定时器

```
window.clearInterval(intervalID);
```

clearInterval() 方法可以取消先前通过调用 setInterval() 建立的定时器。

### 注意:

1. window 可以省略。
2. 里面的参数就是定时器的标识符。

```
<button class="begin">开始</button>
<button class="stop">取消</button>
<script>
  let beginBtn = document.querySelector('.begin')
  let stopBtn = document.querySelector('.stop')
  let timer
  beginBtn.addEventListener('click', function () {
    timer = setInterval(function () {
      console.log(111)
    }, 1000)
  })
  stopBtn.addEventListener('click', function () {
    clearInterval(timer)
  })
</script>
```

# 定时器

使用定时器，完成简单的轮播图。一共三张图片，每隔3秒自动切换下一张。单击“手动切换”可以直接切换到下一张图片。



手动切换



手动切换



手动切换

# 定时器

使用定时器，完成简单的轮播图。一共三张图片，每隔3秒自动切换下一张。单击“手动切换”可以直接切换到下一张图片。

```
<div>
  
  <button>手动切换</button>
</div>
<script>
  window.onload = goNextImage //页面加载完就执行轮播
  let imgBox = document.querySelector('#img-box')
  let btn = document.querySelector('button') //手动切换按钮
  btn.onclick = goNextImage //单击按钮手动切换图片
  setInterval(goNextImage, 3000)
  let i = 0
  function goNextImage() {
    i++
    imgBox.src = `images/${i}.jpg`
    if (i == 3) i = 0
  }
</script>
```

# 定时器

## 发送短信

点击按钮后，该按钮60秒之内不能再次点击，防止重复发送短信

验证码	还剩下53秒
-----	--------

- ① 按钮点击之后，会禁用 disabled 为true
- ② 同时按钮里面的内容会变化，注意 button 里面的内容通过 innerHTML修改
- ③ 里面秒数是有变化的，因此需要用到定时器
- ④ 定义一个变量，在定时器里面，不断递减
- ⑤ 如果变量为0 说明到了时间，我们需要停止定时器，并且复原按钮初始状态。

```
<div>
  <input type="text" placeholder="验证码" />
  <button>发送</button>
</div>
<script>
  let btn = document.querySelector('button')
  let time = 59
  btn.addEventListener('click', function () {
    btn.disabled = true
    let clocktimer = setInterval(function () {
      if (time == 0) {
        clearInterval(clocktimer)
        btn.disabled = false
        btn.innerHTML = '发送'
        time = 59
      } else {
        btn.innerHTML = `还剩下${time}秒`
        time--
      }
    }, 1000)
  })
</script>
```

# JS执行机制

## JS是单线程

JavaScript 语言的一大特点就是单线程，也就是说，同一个时间只能做一件事。

单线程就意味着，所有任务需要排队，前一个任务结束，才会执行后一个任务。这样所导致的问题是：如果 JS 执行的时间过长，这样就会造成页面的渲染不连贯，导致页面渲染加载阻塞的感觉。

```
console.log(1);  
setTimeout(function () {  
  console.log(3);  
}, 1000);  
console.log(2);
```

```
console.log(1);  
setTimeout(function () {  
  console.log(3);  
}, 0);  
console.log(2);
```

## 同步和异步

为了解决这个问题，利用多核 CPU 的计算能力，HTML5 提出 Web Worker 标准，允许 JavaScript 脚本创建多个线程。于是，JS 中出现了同步和异步。

### 1、同步

前一个任务结束后再执行后一个任务，程序的执行顺序与任务的排列顺序是一致的、同步的。

### 2、异步

在做一件事情时，因为这件事情会花费很长时间，在做这件事的同时，你还可以去处理其他事情。

二者本质区别： 这条流水线上各个流程的执行顺序不同。

# JS执行机制

## 同步和异步

### 1、同步任务

同步任务都在主线程上执行，形成一个**执行栈**。

### 2、异步任务

JS 的异步是通过回调函数实现的。

一般而言，异步任务有以下三种类型：

1、普通事件，如 click、resize 等

2、资源加载，如 load、error 等

3、定时器，包括 setInterval、setTimeout 等

异步任务相关回调函数添加到**任务队列**中（任务队列也称为消息队列）。

执行栈

```
console.log(1)
setTimeout(fn, 0)
console.log(2)
```

任务队列

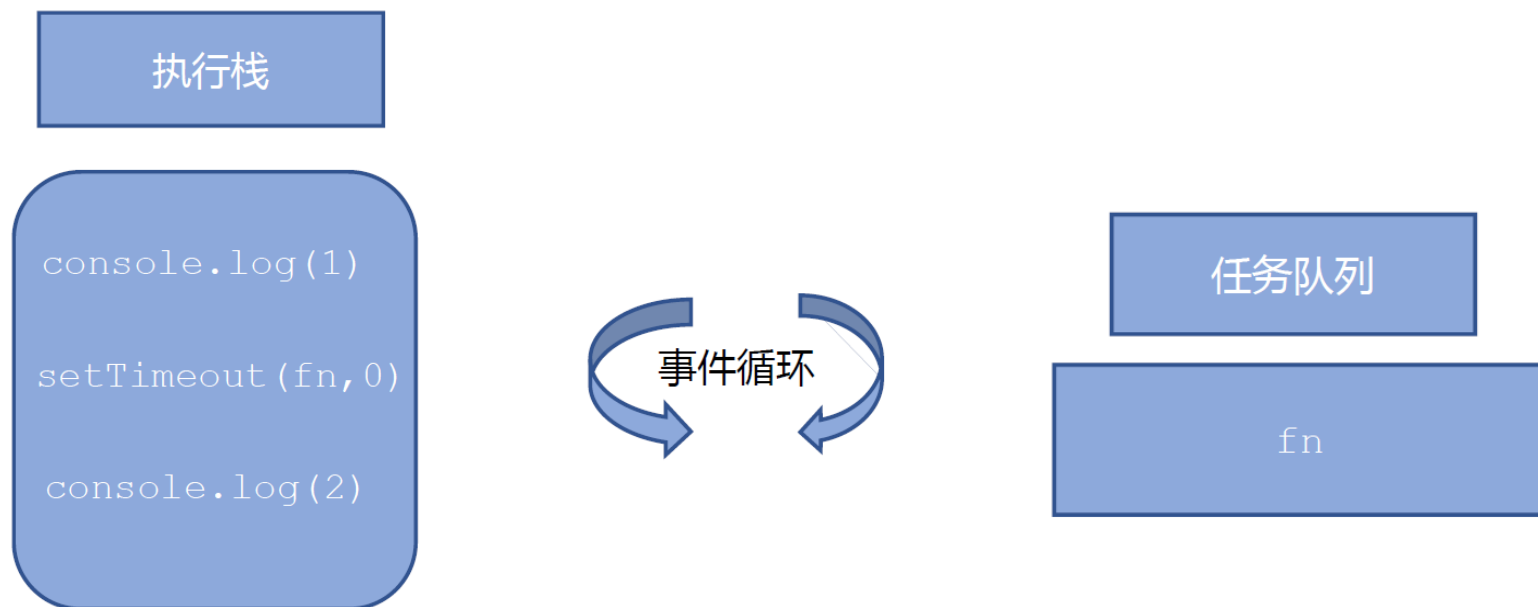
fn



# JS执行机制

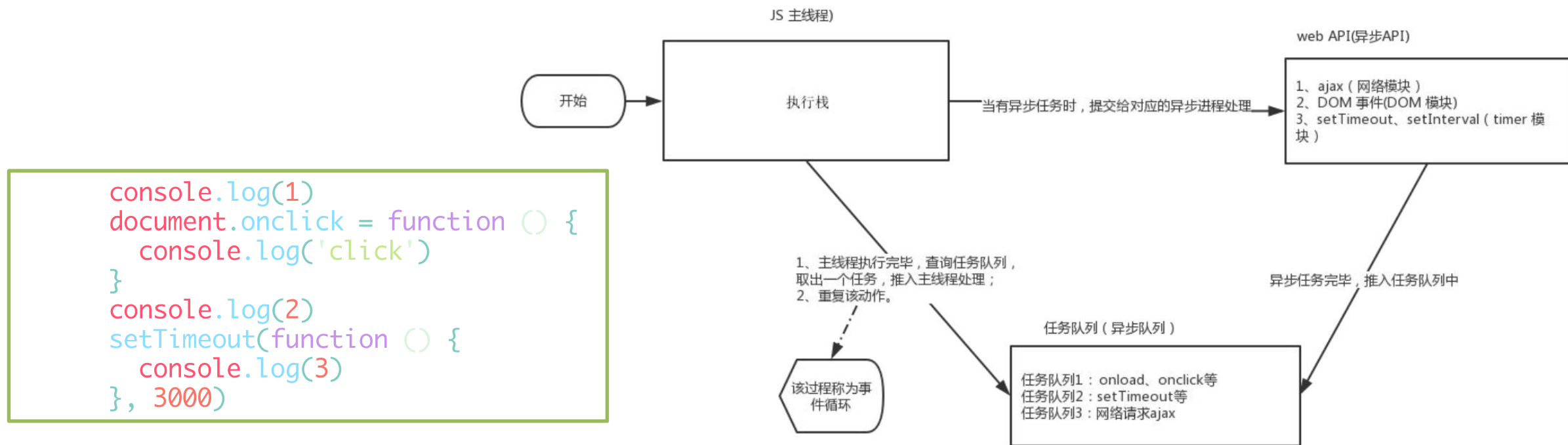
## 同步和异步

1. 先执行**执行栈中的同步任务**。
2. 异步任务（回调函数）放入任务队列中。
3. 一旦执行栈中的所有同步任务执行完毕，系统就会按次序读取**任务队列**中的异步任务，于是被读取的异步任务结束等待状态，进入执行栈，开始执行。



# JS执行机制

## 同步和异步



# JS执行机制

## 同步和异步

- (1) 所有同步任务都在主线程上执行，形成一个执行栈（execution context stack）。
- (2) 主线程之外，还存在一个"任务队列"（task queue）。只要异步任务有了运行结果，就在"任务队列"之中放置一个事件（回调函数callback）。
- (3) 一旦"执行栈"中的所有同步任务执行完毕，系统就会读取"任务队列"，看看里面有哪些事件。那些对应的异步任务，于是结束等待状态，进入执行栈，开始执行。
- (4) 主线程不断重复上面的第三步。

由于主线程不断的重复获得任务、执行任务、再获取任务、再执行，所以这种机制被称为**事件循环（event loop）**。

## location对象

location对象包含当前页面的URL信息。并且可以用于解析 URL。常用成员如下所示：

示例URL为：<http://www.eaxmpleurl.com:8080#uid=1001>。

成员	功能	说明
href	当前页面URL	返回 <a href="http://www.eaxmpleurl.com:8080#uid=1001">http://www.eaxmpleurl.com:8080#uid=1001</a>
host	主机名及端口号	返回 <a href="http://www.eaxmpleurl.com:8080">http://www.eaxmpleurl.com:8080</a>
hostname	主机名	返回 <a href="http://www.eaxmpleurl.com">http://www.eaxmpleurl.com</a>
port	端口号	返回8080
hash	锚，即#号之后的部分	返回#uid=1001
assign()	重定向页面	比如location.assign(" <a href="https://www.baidu.com">https://www.baidu.com</a> ")
reload()	重新加载当前页面，相当于刷新	比如location.reload(" <a href="https://www.baidu.com">https://www.baidu.com</a> ")
replace()	用新的页面替换当前页面，不可后退	比如location.replace(" <a href="https://www.baidu.com">https://www.baidu.com</a> ")

## location对象

例：5s之后自动跳转到指定页面。利用前边学习到的定时器做倒计时效果，跳转到指定页面用location对象的href成员。

```
<div></div>
<script>
  let div = document.querySelector('div')
  let times = 4
  setInterval(function () {
    if (times == 0) {
      location.href = 'https://www.baidu.com'
    } else {
      div.innerHTML = `当前页面内容逃走啦! ${times}秒之后跳转到百度首页`
      times--
    }
  }, 1000)
</script>
```

## navigator对象

navigator对象包含了浏览器的相关信息，比如浏览器的名称、版本、操作系统等，其中userAgent返回这些信息。

userAgent: "Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/107.0.0.0 Safari/537.36"

下边这段代码可以判断用户用什么终端打开网页，从而实现跳转

```
if ((navigator.userAgent.match(/(phone|pad|pod|iPhone|iPod|ios|iPad|Android|Mobile|BlackBerry|IEMobile|MQQBrowser|JUC|Fennec|wOSBrowser|BrowserNG|WebOS|Symbian|Windows Phone)/i))) {  
    window.location.href = "";    //手机  
} else {  
    window.location.href = "";    //电脑  
}
```

## history对象

history对象包含浏览历史URL的相关信息。

成员	功能	说明
length	当前窗口浏览历史列表的数量	history.length
back()	后退，相当于单击浏览器“后退”按钮	history.back()
forward()	前进，相当于单击浏览器“前进”按钮	history.forward()
go()	跳转	history.go(-2) 后退两个页面