DOM&BOM

概念：DOM Document Object Model

专门操作网页内容的 API 标准 —— W3C

作用：统一所有浏览器操作网页内容的 API

场景：只要操作网页内容

包含：增删改查，事件绑定

DOM Tree

概念：内存中保存网页中所有内容的树形结构

作用：为了保存不确定深度的层级关系

内容：

1. 一个网页只有一个唯一的树根节点： document
   document 包含整个网页的内容
2. 网页中所有内容，都是 document 的后代
3. 每个节点，都是 node 类型的节点对象

> Node 类型的三个属性：
>
> | 返回值 | nodeType | nodeName | nodeValue |
> | --- | --- | --- | --- |
> | document | 9 | #document | null |
> | 元素 | 1 | 全大写标签名 | null |
> | 属性 | 2 | 属性名 | 属性值 |
> | 文本 | 3 | #text | 文本内容 |

DOM 操作的固定套路

查找触发事件的元素 -> 绑定事件 -> 查找要修改的元素 -> 修改元素

查找

> **节点树**

1. 不用查找就可直接获得的元素

```
document.documentElement   // html
document.head              // head
document.body              // body
document.form[i/id]        // form
```

2. 按节点间关系查找

> 场景：如果已经获得一个节点，找周围的节点时
>
> 2 大类
>
> 1. 父子关系
>
> `elem.parentNode` elem 的父节点
>
> `elem.childNodes` elem 的直接子节点
>
> `elem.firstChild` elem 的第一个直接子节点
>
> `elem.lastChild` elem 的最后一个直接子节点
>
> 2. 兄弟关系：
>
> `elem.previousSibling` elem 的前一个兄弟节点
>
> `elem.nextSibling` elem 的后一个兄弟节点
>
> 问题：受看不见的空字符的干扰！
>
> 解决：元素树

元素树

仅包含元素节点的树结构

父子关系

`elem.parentElement` elem 的父元素

`elem.children` elem 的直接子元素

`elem.firstElementChild` elem 的第一个直接子元素

`elem.lastElementChild` elem 的最后一个直接子元素

兄弟关系：

`elem.previousElementSibling` elem 的前一个兄弟元素

`elem.nextElementSibling` elem 的后一个兄弟元素

> 优点：不会受看不见的空字符的干扰
>
> 强调：元素树不是一颗新树，只是节点树的子集

3.按 HTML 查找

按 ID 查找

`var elem = document.getElementById("id");`

返回一个元素

如果找不到，返回 `null`

强调：只能用 document 调用

按 标签名 查找

`var elems = parent.getElementsByTagName("标签名");`

返回一个类数组对象

如果找不到，返回空集合 `[]`

强调：可在任意父元素上调用

不仅查找直接子元素，且查找所有后代子元素

按 name 属性查找

`var elems = document.getElementsByName("name");`

返回一个类数组对象

如果找不到，返回空集合

强调：只能用 document 调用

按 class 属性查找

`var elems = parent.getElementsByClassName("className");`

返回一个类数组对象

如果找不到，返回空集合

强调：可在任意父元素上调用

不仅查找直接子元素，且查找所有后代子元素

多类名只用一个 class 也能找到

强调：返回动态集合

动态集合：不实际存储数据，每次访问集合，都要重新查找 dom树

优点：首次查找速度快，不需要返回所有数据，只需要返回想要的一个数据即可

缺点：反复访问集合，会导致反复查找，效率低

> 错误遍历：for(var i = 0; i < btns.length; i ++){}
>
> 正确遍历：for(var i = 0, len = btns.length; i < len; i ++){}

问题：一次只能按一种条件查找

解决：按 选择器 查找

4. 按选择器查找

只找一个元素：

```
var elem = parent.querySelector("selector");
```

> 强调：只返回一个元素，
>
> 如果找不到，返回 null
>
> 即使找到多个元素，也只返回第一个符合条件

查找多个元素：

```
var elems = parent.querySelectorAll("selector");
```

> 强调：返回非动态集合
>
> 非动态集合：保存所有数据，即使反复访问，也不会重新查找 dom 树
>
> 如果找不到，返回空集合
>
> 不仅查找直接子元素，且查找所有后代元素
>
> 优点：即使反复访问集合，也不会反复查找 dom 树
>
> 缺点：首次查找速度慢

总结

按 HTML VS 选择器

> | | HTML | 选择器 |
> |---|---|---|
> | 返回值 | 动态集合 | 非动态集合 |
> | 效率 | 高 | 低 |
> | 易用性 | 低 | 高 |
> | 应用场景 | 仅凭一个条件就可找到想要的元素 | 查找条件复杂 |

修改

内容

原始 HTML 代码片段：elem.innerHTML

纯文本内容：elem.textContent

> 一般不用，有浏览器兼容问题，IE8 不支持

textContent VS innerHTML

> 1. 去掉内嵌的标签
> 2. 将转义字符转化为正文

表单元素的值：elem.value

属性

> **HTML 标准属性：**

核心 dom

elem.getAttribute("属性名");

elem.setAttribute("属性名", "值");

elem.hasAttribute("属性名");

elem.removeAttribute("属性名");

HTML DOM：对核心 DOM 部分常用 API 的简化

优点：简单　缺点：不是万能

elem.attribute;

elem.标准属性

特例：elem.className

> **状态属性**

enabled disabled selected checked

值时 boolean 类型

不能用 核心 dom 操作（核心DOM只能操作字符串类型的属性）

只能用 HTML DOM

补充： css3 状态伪类　:checked :disabled :selected

自定义扩展属性

概念：HTML标准中没有规定的，开发人员自定义的属性，浏览器不认识

场景：2 种

> 在客户端就近临时缓存业务数据
>
> 代替其他选择器，用于批量选择元素添加事件

样式

> 1. 内联样式

修改：elem.style.css.属性名 = "value";

优点：优先级最高，当前元素独有

获取：

错误：elem.style.css.属性名 只能获取内联样式

正确：获得计算后的样式

计算后的样式：最后应用到元素上的所有样式的和，且将相对单位计算为绝对单位

如何获得：

var style = getComputedStyle(elem);

style.fontSize ...

强调：计算后的样式 **只读**

> 1. 内部 / 外部样式表

获得样式表对象

var sheet = document.styleSheets[i];

获得样式表对象中的 cssRule

var cssRule = sheet.cssRules[i];

获得 cssRule 中的 css 属性值

cssRule.style.css.属性名 = value;

> 问题：一句话只能修改一个 css 属性
>
> 解决：用 class 批量应用一套样式

添加 / 删除

添加

> 3 步

1. 创建空对象：var a = document.createElement("a");

   => `<a></a>`
2. 设置关键属性：

   a.innerHTML = "Go To Tmooc";

   a.href = "http://tmooc.cn";

   => `<a href="http://tmooc.cn">Go To Tmooc</a>`
3. 将新元素添加到 DOM 树上

   parent.appendChild(a);

   parent.insertBefore(a, child);

   parent.replaceChild(a, child);

HTML DOM 常用对象

`Image`

代表页面上一个 img 元素

var img = new Image();

`select`

代表页面上一个 select 元素

属性：`.selectedIndex` 获得当前选中项的下标位置

`.value` 获得当前选中项的值，没有value 返回 innerHTML

`.options` 获得当前 select 下所有 option 对象

`.options.length` 获得当前 select 下 option 的个数

`.length` = `options.length`

固定套路：清空 select： sel.length = 0;

方法：`.add(option)` 问题：不支持文档片段

`.remove(i)`

`option`

代表一个 option 元素

创建：var opt = new Option(text, value);

`table`

代表一个 table 元素

管理行分组

添加：table.createTHead();

table.createTBody();

table.createTFoot();

删除：table.deleteTHead();

table.deleteTfoot();

获取：table.tHead;

table.tFoot;

table.tBodies[i];

行分组管理行

添加：行分组.insertRow[i];

解决：table.deleteRow(rowIndex);

tr.rowIndex 时 tr 在整个表中的位置

获取：行分组 .row[i]

删除：tr.deleteCell(i);

获取：tr.cells[i]

`form`

代表一个 form 元素

获取：document.forms[i/id]

属性：

.elements	获取表单中所有表单元素的集合（input button textarea select）
.elememts.length	表单元素的个数
.length	= .elements.length

方法：form.sumbit(); 手动提交表单

element

获取：var elem = form.elements[i/id/name];

更简化：如果表单元素有 name 属性

form.name

方法：

elem.focus();

elem.blur();

BOM

Browser Object Model

专门操作浏览器窗口的 API -- 没有标准

包括

window 作为 2 个角色

1. 保存所有 ES + DOM + BOM 所有 API

2. 代替 global 充当全局作用域对象

包括：

history, location, navigator, document, screen, event

使用场景：

只要操作浏览器窗口时

---

window

属性：		
窗口大小	.outerWidth / .outerHeight	窗口总大小
	.innerWidth / .innerHeight	文档显示区大小
方法：	window.scrollTo(left, top);	滚动到
	window.scrollBy(left, top);	滚动过

强调：因为网页很少横向滚动，所有 left 几乎都是 0；

打开和关闭窗口：

1. 在当前窗口打开，可后退

   `html : <a href="url" target="_self"></a>`

   `js : window.open(url, "_self")`
2. 在当前窗口打开，不可后退

   `js : location.replace(url);`
3. 在新窗口打开，可打开多个

   `html :<a href="url" target="_blank"></a>`

   `js : window.open(url, "_blank")`
4. 在新窗口打开，只能打开一个

   `html: <a href="url" target="自定义"></a>`

   `js : window.open(url, "自定义")`

   原理：每个窗口在内存中都有唯一的窗口名 name

   浏览器规定，相同窗口名的窗口只能有一个，后打开的会替换先打开的同名窗口

   内置窗口名：

   _self 自动获得当前窗口名作为新窗口

   _blank 不指定窗口名，让浏览器随机分配窗口名

关闭窗口

window.close();

history

概念：保存当前窗口打开后成功访问过的 url 的历史记录 **栈**

使用场景：在程序中前进 / 后退时

形式： history.go(n);

history.go(1); 前进一步

history.go(0); 刷新

history.go(-1); 后退一步

location

概念：保存当前窗口正在打开的 url 对象

使用场景：获取 url 的信息或跳转时

语法：location.[prop]

属性

href (完整url)	**protocol**	**host (主机名 + 端口号)**
hostname (仅主机名)	port (仅端口号)	pathname
search	hash (锚点地址)	

方法

跳转			
在当前窗口打开新链接，可后退	location.href = url;	location = url	location.assign(url) =>location.href
在当前窗口打开新链接，禁止后退	location.replace(url);		
刷新			
普通刷新	history.go(0)	F5	location.reload();

刷新			
普通刷新	history.go(0)	F5	location.reload();
强制刷新	location.reload(true);		

普通刷新：默认优先从本地缓存中获取资源，除非本地没有或资源过期，才从服务器下载资源

强制刷新：跳过本地缓存，直接从服务器下载最新资源

---

navigator

概念：封装浏览器配置信息的对象

使用场景：访问浏览器的配置信息

包括：

> .cookieEnabled：判断是否启用了 cookie
>
> cookie：客户端 **持久** 保存用户私密信息的小文件
>
> 原因：程序中的变量都是保存在内存中，都是 **临时** 的，程序退出，数据丢失
>
> 使用场景：只要希望即使程序关闭，数据也能持久保存

> plugins：封装浏览器安装插件信息的集合
>
> 插件：为浏览器添加新功能的小软件
>
> 使用场景：只要判断是否安装某插件时
>
> 语法：
>
> navigitor.plugins["Chrome PDF Viewer"] !== undefined

> userAgent: 保存浏览器名称和版本号的字符串

定时器

周期性定时器

概念：让程序每隔一段时间，反复执行某一项任务

使用场景：让程序每隔一段时间，反复执行某一项任务

语法：

> 1. 定义任务函数
> 2. 启动定时器：

> var timer = setInterval(task, 间隔);
>
> 3. 停止定时器：
>
> clearInterval(timer);
>
> > timer 定时器的序号
>
> 停止 2 种方法
>
> 1. 手动停止
> 2. 自动停止：在任务函数中，判断临界值，如果达到临界值，就自动调用 clearInterval

一次性定时器

> 概念：让程序先等待一段时间，再延迟执行一项任务
>
> 使用场景：延迟执行一项任务
>
> 语法：
>
> > 1. 定义任务函数
> > 2. 启动定时器：
> >
> > var timer = setTimeout(task, 间隔);
> >
> > 3. 停止定时器：
> >
> > clearTimeout(timer);

定时器原理

> setInterval,setTimeout 其实只是将任务函数添加到定时器对象中
>
> 当任务函数执行时，必须先加入到等待队列中，等待执行
>
> 只有当主程序所有代码执行完，才能执行等待队列中的回调函数
>
> 结论：定时器中的回调函数，一定会在主程序最后一句话执行完成后才执行

---

event

事件：浏览器自动触发的，用户手动触发的页面内容或状态的改变

事件处理函数：在事件发生时自动触发的函数

使用场景：只要事件发生时，希望执行一项任务

形式：

1. 在 HTML 中

   `<button onclick="js"><button>`

   问题：不符合内容和行为相分离的原则，不便于维护
2. 在 js 中，用赋值方式

`btn.onclick = function(){ // this => btn}`

3. 在 js 中，添加事件监听对象：

   `btn.addEventListener("click", function(){});`

   事件监听对象 = 元素 + 事件名 + 处理函数

   所有监听对象，都是添加到浏览器一个巨大的数组中保存

   当事件发生时，浏览器遍历事件监听对象数组，查找匹配的监听对象，执行处理函数

   `btn.removeEventListener("click", function(){处理函数});`

   强调：如果希望移除一个处理函数，则绑定时和移除时必须使用具名函数，找到同一个函数对象才能移除

事件模型

> 1. 捕获：由外向内记录各级父元素上绑定的处理函数
> 2. 目标触发：优先触发目标元素上的处理函数
> 3. 冒泡：由内向外依次触发各级父元素上的处理函数

事件对象

> 概念：事件发生时，自动创建的封装事件信息的对象，并提供了操作事件的 API
>
> 使用场景：获取事件信息或修改事件行为时
>
> 形式：
>
> 获取：事件对象总是作为处理函数的第一个参数传入
>
> `elem.onclick = function(e){}`
>
> API：
>
> 1. 取消冒泡 / 停止漫延： `e.stopPropagation()`
> 2. 利用冒泡：
>
>   **优化：尽量减少事件监听对象的个数**
>
>   原因：浏览器触发事件，都是用遍历监听对象数组的方式触发事件，监听事件的个数决定了遍历效率
>
>   场景：只要多个平级子元素需要绑定相同的事件处理函数时
>
>   形式：
>
>   只在父元素上绑定一次事件处理函数
>
>   =》2 个难题
>
>   1. this 不再指向子元素，而是指向父元素
>
>   解决：e.target 代替 this
>
>   e.target 能自动获取目标元素
>   2. 筛选目标元素：只有 e.target 符合要求时，才能执行操作
> 3. 取消事件 / 阻止默认行为
>
>   场景：不希望继续执行事件的默认行为时

> > 形式：`e.preventDefault();`
>
> 3 种经典场景
>
> > 1. 用 a 元素当做按钮使用时
> >
> > 问题：a 元素经常会擅自在 url 后添加锚点地址
> >
> > 解决：阻止 a 的默认行为
> >
> > 2. 表单的 submit 默认提交事件
> >
> > input button + form.onsubmit + e.preventDefault
> >
> > 3. HTML5 中拖拽 API 都要取消默认行为

鼠标坐标

> 相对于屏幕左上角：e.screenX / e.screenY
>
> 相对于文档显示区左上角：e.clientX / e.clientY
>
> 相对于触发事件的左上角：e.offsetX / e.offsetY