24 de octubre de 2022

Buscar: getElement*, querySelector*

Las propiedades de navegación del DOM son ideales cuando los elementos están cerca unos de otros. Pero, ¿y si no lo están? ¿Cómo obtener un elemento arbitrario de la página?

Para estos casos existen métodos de búsqueda adicionales.

document.getElementById o sólo id

Si un elemento tiene el atributo `id`, podemos obtener el elemento usando el método `document.getElementById(id)`, sin importar dónde se encuentre.

Por ejemplo:

```
<div id="elem">
  <div id="elem-content">Elemento</div>
</div>

<script>
  // obtener el elemento
  let elem = document.getElementById('elem');

  // hacer que su fondo sea rojo
  elem.style.background = 'red';
</script>
```

Existe además una variable global nombrada por el `id` que hace referencia al elemento:

```
<div id="elem">
  <div id="elem-content">Elemento</div>
</div>

<script>
  // elem es una referencia al elemento del DOM con id="elem"
  elem.style.background = 'red';

  // id="elem-content" tiene un guion en su interior, por lo que no puede ser u
  // ...pero podemos acceder a él usando corchetes: window['elem-content']
</script>
```

...Esto es a menos que declaremos una variable de JavaScript con el mismo nombre, entonces ésta tiene prioridad:

```
<div id="elem"></div>

<script>
  let elem = 5; // ahora elem es 5, no una referencia a <div id="elem">

  alert(elem); // 5
</script>
```

Por favor, no utilice variables globales nombradas por id para acceder a los elementos

Este comportamiento se encuentra descrito en la especificación, pero está soportado principalmente para compatibilidad.

El navegador intenta ayudarnos mezclando espacios de nombres (*namespaces*) de JS y DOM. Esto está bien para los scripts simples, incrustados en HTML, pero generalmente no es una buena práctica. Puede haber conflictos de nombres. Además, cuando uno lee el código de JS y no tiene el HTML a la vista, no es obvio de dónde viene la variable.

Aquí en el tutorial usamos `id` para referirnos directamente a un elemento por brevedad, cuando es obvio de dónde viene el elemento.

En la vida real `document.getElementById` es el método preferente.

El `id` debe ser único

El `id` debe ser único. Sólo puede haber en todo el documento un elemento con un `id` determinado.

Si hay múltiples elementos con el mismo id, entonces el comportamiento de los métodos que lo usan es impredecible, por ejemplo `document.getElementById` puede devolver cualquiera de esos elementos al azar. Así que, por favor, sigan la regla y mantengan el `id` único.

Sólo `document.getElementById`, no `anyElem.getElementById`

El método `getElementById` sólo puede ser llamado en el objeto `document`. Busca el `id` dado en todo el documento.

querySelectorAll

Sin duda el método más versátil, `elem.querySelectorAll(css)` devuelve todos los elementos dentro de `elem` que coinciden con el selector CSS dado.

Aquí buscamos todos los elementos `<li>` que son los últimos hijos:

```
<ul>
```

```
  <li>La</li>
  <li>prueba</li>
</ul>
<ul>
  <li>ha</li>
  <li>pasado/li>
</ul>
<script>
  let elements = document.querySelectorAll('ul > li:last-child');

  for (let elem of elements) {
    alert(elem.innerHTML); // "prueba", "pasado"
  }
</script>
```

Este método es muy poderoso, porque se puede utilizar cualquier selector de CSS.

> **También se pueden usar pseudoclases**
>
> Las pseudoclases como `:hover` (cuando el cursor sobrevuela el elemento) y `:active` (cuando hace clic con el botón principal) también son soportadas. Por ejemplo, `document.querySelectorAll(':hover')` devolverá una colección de elementos sobre los que el puntero hace hover en ese momento (en orden de anidación: desde el más exterior `<html>` hasta el más anidado).

querySelector

La llamada a `elem.querySelector(css)` devuelve el primer elemento para el selector CSS dado.

En otras palabras, el resultado es el mismo que `elem.querySelectorAll(css)[0]`, pero este último busca *todos* los elementos y elige uno, mientras que `elem.querySelector` sólo busca uno. Así que es más rápido y también más corto de escribir.

matches

Los métodos anteriores consistían en buscar en el DOM.

El elem.matches(css) no busca nada, sólo comprueba si el `elem` coincide con el selector CSS dado. Devuelve `true` o `false`.

Este método es útil cuando estamos iterando sobre elementos (como en un array) y tratando de filtrar los que nos interesan.

Por ejemplo:

```
<a href="http://example.com/file.zip">...</a>
<a href="http://ya.ru">...</a>

<script>
  // puede ser cualquier colección en lugar de document.body.children
  for (let elem of document.body.children) {
```

```
    if (elem.matches('a[href$="zip"]')) {
      alert("La referencia del archivo: " + elem.href );
    }
  }
</script>
```

closest

Los *ancestros* de un elemento son: el padre, el padre del padre, su padre y así sucesivamente. Todos los ancestros juntos forman la cadena de padres desde el elemento hasta la cima.

El método `elem.closest(css)` busca el ancestro más cercano que coincide con el selector CSS. El propio `elem` también se incluye en la búsqueda.

En otras palabras, el método `closest` sube del elemento y comprueba cada uno de los padres. Si coincide con el selector, entonces la búsqueda se detiene y devuelve dicho ancestro.

Por ejemplo:

```
<h1>Contenido</h1>

<div class="contents">
  <ul class="book">
    <li class="chapter">Capítulo 1</li>
    <li class="chapter">Capítulo 2</li>
  </ul>
</div>

<script>
  let chapter = document.querySelector('.chapter'); // LI

  alert(chapter.closest('.book')); // UL
  alert(chapter.closest('.contents')); // DIV

  alert(chapter.closest('h1')); // null (porque h1 no es un ancestro)
</script>
```

getElementsBy*

También hay otros métodos que permiten buscar nodos por una etiqueta, una clase, etc.

Hoy en día, son en su mayoría historia, ya que `querySelector` es más poderoso y corto de escribir.

Aquí los cubrimos principalmente por completar el temario, aunque todavía se pueden encontrar en scripts antiguos.

- `elem.getElementsByTagName(tag)` busca elementos con la etiqueta dada y devuelve una colección con ellos. El parámetro `tag` también puede ser un asterisco `"*"` para "cualquier etiqueta".
- `elem.getElementsByClassName(className)` devuelve elementos con la clase dada.

- `document.getElementsByName(name)` devuelve elementos con el atributo `name` dado, en todo el documento. Muy raramente usado.

Por ejemplo:

```
1  // obtener todos los divs del documento
2  let divs = document.getElementsByTagName('div');
```

Para encontrar todas las etiquetas `input` dentro de una tabla:

```
1  <table id="table">
2    <tr>
3      <td>Su edad:</td>
4
5      <td>
6        <label>
7          <input type="radio" name="age" value="young" checked> menos de 18
8        </label>
9        <label>
10         <input type="radio" name="age" value="mature"> de 18 a 50
11       </label>
12       <label>
13         <input type="radio" name="age" value="senior"> más de 60
14       </label>
15     </td>
16   </tr>
17 </table>
18
19 <script>
20   let inputs = table.getElementsByTagName('input');
21
22   for (let input of inputs) {
23     alert( input.value + ': ' + input.checked );
24   }
25 </script>
```

¡No olvides la letra `"s"` !

Los desarrolladores novatos a veces olvidan la letra `"s"` . Esto es, intentan llamar a `getElementByTagName` en vez de a `getElementsByTagName` .

La letra `"s"` no se encuentra en `getElementById` porque devuelve sólo un elemento. But `getElementsByTagName` devuelve una colección de elementos, de ahí que tenga la `"s"` .

¡Devuelve una colección, no un elemento!

Otro error muy extendido entre los desarrolladores novatos es escribir:

```
// no funciona
document.getElementsByTagName('input').value = 5;
```

Esto no funcionará, porque toma una *colección* de inputs y le asigna el valor a ella en lugar de a los elementos dentro de ella.

En dicho caso, deberíamos iterar sobre la colección o conseguir un elemento por su índice y luego asignarlo así:

```
// debería funcionar (si hay un input)
document.getElementsByTagName('input')[0].value = 5;
```

Buscando elementos `.article`:

```
<form name="my-form">
  <div class="article">Artículo</div>
  <div class="long article">Artículo largo</div>
</form>

<script>
  // encontrar por atributo de nombre
  let form = document.getElementsByName('my-form')[0];

  // encontrar por clase dentro del formulario
  let articles = form.getElementsByClassName('article');
  alert(articles.length); // 2, encontró dos elementos con la clase "article"
</script>
```

Colecciones vivas

Todos los métodos `"getElementsBy*"` devuelven una colección *viva* (*live collection*). Tales colecciones siempre reflejan el estado actual del documento y se "auto-actualizan" cuando cambia.

En el siguiente ejemplo, hay dos scripts.

1. El primero crea una referencia a la colección de `<div>`. Por ahora, su longitud es `1`.
2. El segundo script se ejecuta después de que el navegador se encuentre con otro `<div>`, por lo que su longitud es de `2`.

```
<div>Primer div</div>

```

```
<script>
  let divs = document.getElementsByTagName('div');
  alert(divs.length); // 1
</script>

<div>Segundo div</div>

<script>
  alert(divs.length); // 2
</script>
```

Por el contrario, `querySelectorAll` devuelve una colección *estática*. Es como un array de elementos fijos.

Si lo utilizamos en lugar de `getElementsByTagName`, entonces ambos scripts dan como resultado `1`:

```
<div>Primer div</div>

<script>
  let divs = document.querySelectorAll('div');
  alert(divs.length); // 1
</script>

<div>Segundo div</div>

<script>
  alert(divs.length); // 1
</script>
```

Ahora podemos ver fácilmente la diferencia. La colección estática no aumentó después de la aparición de un nuevo `div` en el documento.

Resumen

Hay 6 métodos principales para buscar nodos en el DOM:

Método	Busca por...	¿Puede llamar a un elemento?	¿Vivo?
`querySelector`	selector CSS	✓	-
`querySelectorAll`	selector CSS	✓	-
`getElementById`	`id`	-	-
`getElementsByName`	`name`	-	✓
`getElementsByTagName`	etiqueta o `'*'`	✓	✓
`getElementsByClassName`	class	✓	✓

Los más utilizados son `querySelector` y `querySelectorAll`, pero `getElementBy*` puede ser de ayuda esporádicamente o encontrarse en scripts antiguos.

Aparte de eso:

- Existe `elem.matches(css)` para comprobar si `elem` coincide con el selector CSS dado.
- Existe `elem.closest(css)` para buscar el ancestro más cercano que coincida con el selector CSS dado. El propio `elem` también se comprueba.

Y mencionemos un método más para comprobar la relación hijo-padre, ya que a veces es útil:

- `elemA.contains(elemB)` devuelve true si `elemB` está dentro de `elemA` (un descendiente de `elemA`) o cuando `elemA==elemB`.

Tareas

Buscar elementos

importancia: 4

Aquí está el documento con la tabla y el formulario.

¿Cómo encontrar?...

1. La tabla con `id="age-table"`.
2. Todos los elementos `label` dentro de la tabla (debería haber 3).
3. El primer `td` en la tabla (con la palabra "Age").
4. El `form` con `name="search"`.
5. El primer `input` en ese formulario.
6. El último `input` en ese formulario.

Abra la página table.html en una ventana separada y haga uso de las herramientas del navegador.

solución

Lección anterior Próxima lección

Compartir

Mapa del Tutorial

Comentarios

- Si tiene sugerencias sobre qué mejorar, por favor enviar una propuesta de GitHub o una solicitud de extracción en lugar de comentar.
- Si no puede entender algo en el artículo, por favor explique.

- Para insertar algunas palabras de código, use la etiqueta `<code>`, para varias líneas – envolverlas en la etiqueta `<pre>`, para más de 10 líneas – utilice una entorno controlado (sandbox) (plnkr, jsbin, codepen…)