25 de junio de 2022

Propiedades del nodo: tipo, etiqueta y contenido

Echemos un mirada más en profundidad a los nodos DOM.

En este capítulo veremos más sobre cuáles son y aprenderemos sus propiedades más utilizadas.

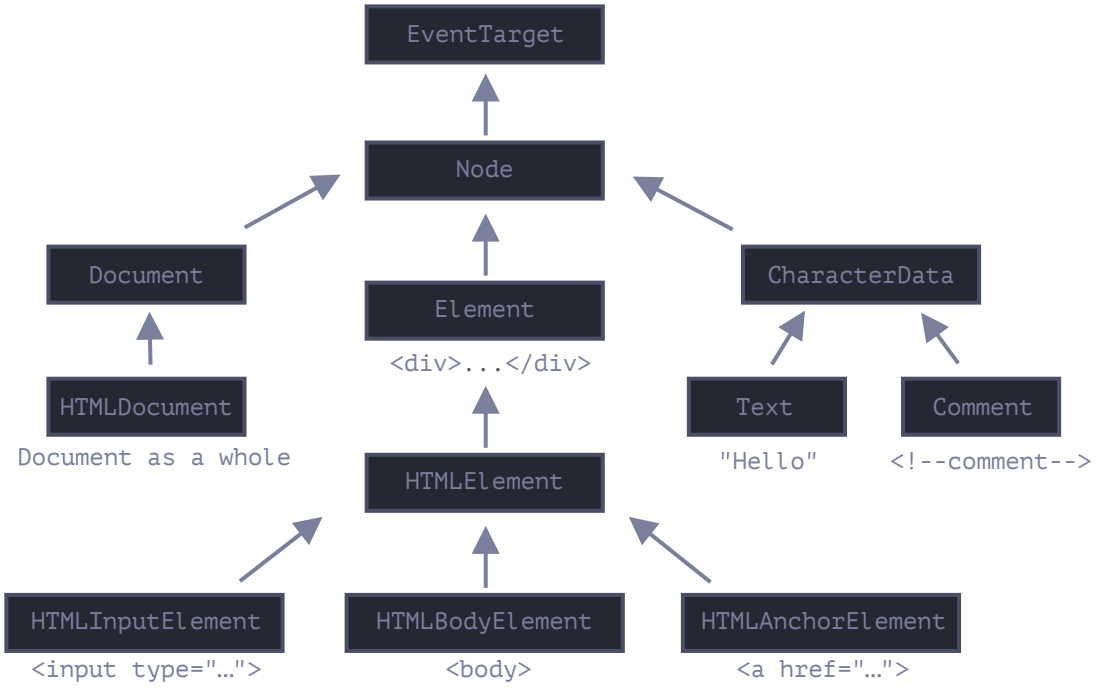
Clases de nodo DOM

Los diferentes nodos DOM pueden tener diferentes propiedades. Por ejemplo, un nodo de elemento correspondiente a la etiqueta `<a>` tiene propiedades relacionadas con el enlace, y el correspondiente a `<input>` tiene propiedades relacionadas con la entrada y así sucesivamente. Los nodos de texto no son lo mismo que los nodos de elementos. Pero también hay propiedades y métodos comunes entre todos ellos, porque todas las clases de nodos DOM forman una única jerarquía.

Cada nodo DOM pertenece a la clase nativa correspondiente.

La raíz de la jerarquía es EventTarget, que es heredada por Node, y otros nodos DOM heredan de él.

Aquí está la imagen, con las explicaciones a continuación:



Las clases son:

- EventTarget – es la clase raíz "abstracta".

  Los objetos de esta clase nunca se crean. Sirve como base, es por la que todos los nodos DOM soportan los llamados "eventos" que estudiaremos más adelante.

- Node – también es una clase "abstracta", sirve como base para los nodos DOM.

  Proporciona la funcionalidad del árbol principal: `parentNode`, `nextSibling`, `childNodes` y demás (son getters). Los objetos de la clase `Node` nunca se crean. Pero hay clases de nodos concretas que heredan de ella (y también heredan la funcionalidad de `Node`).

- Document, por razones históricas, heredado a menudo por `HTMLDocument` (aunque la última especificación no lo exige) – es el documento como un todo.

  El objeto global `document` pertenece exactamente a esta clase. Sirve como punto de entrada al DOM.

- CharacterData – una clase "abstract" heredada por:

  - Text – la clase correspondiente a texto dentro de los elementos, por ejemplo `Hello` en `<p>Hello</p>`.
  - Comment – la clase para los "comentarios". No se muestran, pero cada comentario se vuelve un miembro del DOM.

- Element – es una clase base para elementos DOM.

  Proporciona navegación a nivel de elemento como `nextElementSibling`, `children` y métodos de búsqueda como `getElementsByTagName`, `querySelector`.

  Un navegador admite no solo HTML, sino también XML y SVG. La clase `Element` sirve como base para clases más específicas: `SVGElement`, `XMLElement` (no las necesitamos aquí) y `HTMLElement`.

- Finalmente, HTMLElement – es la clase básica para todos los elementos HTML. Trabajaremos con ella la mayor parte del tiempo.

  Es heredado por elementos HTML concretos:

  - HTMLInputElement – la clase para elementos `<input>`,
  - HTMLBodyElement – la clase para los elementos `<body>`,
  - HTMLAnchorElement – la clase para elementos `<a>`,
  - ...y así sucesivamente.

Hay muchas otras etiquetas con sus propias clases que pueden tener propiedades y métodos específicos, mientras que algunos elementos, tales como `<span>`, `<section>`, `<article>`, no tienen ninguna propiedad específica entonces derivan de la clase `HTMLElement`.

Entonces, el conjunto completo de propiedades y métodos de un nodo dado viene como resultado de la cadena de herencia.

Por ejemplo, consideremos el objeto DOM para un elemento `<input>`. Pertenece a la clase HTMLInputElement.

Obtiene propiedades y métodos como una superposición de (enumerados en orden de herencia):

- `HTMLInputElement` – esta clase proporciona propiedades específicas de entrada,
- `HTMLElement` – proporciona métodos de elementos HTML comunes (y getters/setters),
- `Element` – proporciona métodos de elementos genéricos,
- `Node` – proporciona propiedades comunes del nodo DOM,
- `EventTarget` – da el apoyo para eventos (a cubrir),

- ...y finalmente hereda de `Object`, por lo que también están disponibles métodos de "objeto simple" como `hasOwnProperty`.

Para ver el nombre de la clase del nodo DOM, podemos recordar que un objeto generalmente tiene la propiedad `constructor`. Hace referencia al constructor de la clase, y `constructor.name` es su nombre:

```
alert( document.body.constructor.name ); // HTMLBodyElement
```

...O podemos simplemente usar `toString`:

```
alert( document.body ); // [object HTMLBodyElement]
```

También podemos usar `instanceof` para verificar la herencia:

```
alert( document.body instanceof HTMLBodyElement ); // true
alert( document.body instanceof HTMLElement ); // true
alert( document.body instanceof Element ); // true
alert( document.body instanceof Node ); // true
alert( document.body instanceof EventTarget ); // true
```

Como podemos ver, los nodos DOM son objetos regulares de JavaScript. Usan clases basadas en prototipos para la herencia.

Eso también es fácil de ver al generar un elemento con `console.dir(elem)` en un navegador. Allí, en la consola, puede ver `HTMLElement.prototype`, `Element.prototype` y así sucesivamente.

`console.dir(elem)` versus `console.log(elem)`

La mayoría de los navegadores admiten dos comandos en sus herramientas de desarrollo: `console.log` y `console.dir`. Envían sus argumentos a la consola. Para los objetos JavaScript, estos comandos suelen hacer lo mismo.

Pero para los elementos DOM son diferentes:

- `console.log(elem)` muestra el árbol DOM del elemento.
- `console.dir(elem)` muestra el elemento como un objeto DOM, es bueno para explorar sus propiedades.

Inténtalo en `document.body`.

IDL en la especificación

En la especificación, las clases DOM no se describen mediante JavaScript, sino con un Lenguaje de descripción de interfaz (IDL) especial, que suele ser fácil de entender.

En IDL, todas las propiedades están precedidas por sus tipos. Por ejemplo, `DOMString`, `boolean` y así sucesivamente.

Aquí hay un extracto, con comentarios:

```
// Definir HTMLInputElement
// Los dos puntos ":" significan que HTMLInputElement hereda de HTMLElement
interface HTMLInputElement: HTMLElement {
  // aquí van las propiedades y métodos de los elementos <input>

  // "DOMString" significa que el valor de una propiedad es un string
  attribute DOMString accept;
  attribute DOMString alt;
  attribute DOMString autocomplete;
  attribute DOMString value;

  // Propiedad de valor booleano (true/false)
  attribute boolean autofocus;
  ...
  // ahora el método: "void" significa que el método no devuelve ningún val
  void select();
  ...
}
```

La propiedad “nodeType”

La propiedad `nodeType` proporciona una forma “anticuada” más de obtener el “tipo” de un nodo DOM.

Tiene un valor numérico:

- `elem.nodeType == 1` para nodos de elementos,
- `elem.nodeType == 3` para nodos de texto,
- `elem.nodeType == 9` para el objeto de documento,
- hay algunos otros valores en la especificación.

Por ejemplo:

```
<body>
  <script>
  let elem = document.body;

  // vamos a examinar: ¿qué tipo de nodo es elem?
  alert(elem.nodeType); // 1 => elemento
```

```
7
8     // Y el primer hijo es...
9     alert(elem.firstChild.nodeType); // 3 => texto
10
11    // para el objeto de tipo documento, el tipo es 9
12    alert( document.nodeType ); // 9
13    </script>
14  </body>
```

En los scripts modernos, podemos usar `instanceof` y otras pruebas basadas en clases para ver el tipo de nodo, pero a veces `nodeType` puede ser más simple. Solo podemos leer `nodeType` , no cambiarlo.

Tag: nodeName y tagName

Dado un nodo DOM, podemos leer su nombre de etiqueta en las propiedades de `nodeName` o `tagName` :

Por ejemplo:

```
1  alert( document.body.nodeName ); // BODY
2  alert( document.body.tagName ); // BODY
```

¿Hay alguna diferencia entre `tagName` y `nodeName` ?

Claro, la diferencia se refleja en sus nombres, pero de hecho es un poco sutil.

- La propiedad `tagName` existe solo para los nodos `Element` .
- El `nodeName` se define para cualquier `Node` :
  - para los elementos, significa lo mismo que `tagName` .
  - para otros tipos de nodo (texto, comentario, etc.) tiene una cadena con el tipo de nodo.

En otras palabras, `tagName` solo es compatible con los nodos de elementos (ya que se origina en la clase `Element`), mientras que `nodeName` puede decir algo sobre otros tipos de nodos.

Por ejemplo, comparemos `tagName` y `nodeName` para `document` y un nodo de comentario:

```
1  <body><!-- comentario -->
2
3    <script>
4      // para comentarios
5      alert( document.body.firstChild.tagName ); // undefined (no es un elemento)
6      alert( document.body.firstChild.nodeName ); // #comment
7
8      // para documentos
9      alert( document.tagName ); // undefined (no es un elemento)
10     alert( document.nodeName ); // #document
11   </script>
12 </body>
```

Si solo tratamos con elementos, entonces podemos usar tanto `tagName` como `nodeName` – no hay diferencia.

> **El nombre de la etiqueta siempre está en mayúsculas, excepto en el modo XML**
>
> El navegador tiene dos modos de procesar documentos: HTML y XML. Por lo general, el modo HTML se usa para páginas web. El modo XML está habilitado cuando el navegador recibe un documento XML con el encabezado: `Content-Type: application/xml+xhtml` .
>
> En el modo HTML, `tagName/nodeName` siempre está en mayúsculas: es `BODY` ya sea para `<body>` o `<BoDy>` .
>
> En el modo XML, el caso se mantiene "tal cual". Hoy en día, el modo XML rara vez se usa.

innerHTML: los contenidos

La propiedad innerHTML permite obtener el HTML dentro del elemento como un string.

También podemos modificarlo. Así que es una de las formas más poderosas de cambiar la página.

El ejemplo muestra el contenido de `document.body` y luego lo reemplaza por completo:

```
<body>
  <p>Un párrafo</p>
  <div>Un div</div>

  <script>
    alert( document.body.innerHTML ); // leer el contenido actual
    document.body.innerHTML = 'El nuevo BODY!'; // reemplazar
  </script>

</body>
```

Podemos intentar insertar HTML no válido, el navegador corregirá nuestros errores:

```
<body>

  <script>
    document.body.innerHTML = '<b>prueba'; // olvidé cerrar la etiqueta
    alert( document.body.innerHTML ); // <b>prueba</b> (arreglado)
  </script>

</body>
```

> **Los scripts no se ejecutan**
>
> Si `innerHTML` inserta una etiqueta `<script>` en el documento, se convierte en parte de HTML, pero no se ejecuta.

Cuidado: “innerHTML+=” hace una sobrescritura completa

Podemos agregar HTML a un elemento usando `elem.innerHTML+="more html"` .

Así:

```
chatDiv.innerHTML += "<div>Hola<img src='smile.gif'/> !</div>";
chatDiv.innerHTML += "¿Cómo vas?";
```

Pero debemos tener mucho cuidado al hacerlo, porque lo que está sucediendo *no* es una adición, sino una sobrescritura completa.

Técnicamente, estas dos líneas hacen lo mismo:

```
elem.innerHTML += "...";
// es una forma más corta de escribir:
elem.innerHTML = elem.innerHTML + "..."
```

En otras palabras, `innerHTML+=` hace esto:

1. Se elimina el contenido antiguo.
2. En su lugar, se escribe el nuevo `innerHTML` (una concatenación del antiguo y el nuevo).

Como el contenido se “pone a cero” y se reescribe desde cero, todas las imágenes y otros recursos se volverán a cargar..

En el ejemplo de `chatDiv` arriba, la línea `chatDiv.innerHTML+="¿Cómo va?"` recrea el contenido HTML y recarga `smile.gif` (con la esperanza de que esté en caché). Si `chatDiv` tiene muchos otros textos e imágenes, entonces la recarga se vuelve claramente visible.

También hay otros efectos secundarios. Por ejemplo, si el texto existente se seleccionó con el mouse, la mayoría de los navegadores eliminarán la selección al reescribir `innerHTML` . Y si había un `<input>` con un texto ingresado por el visitante, entonces el texto será eliminado. Y así.

Afortunadamente, hay otras formas de agregar HTML además de `innerHTML` , y las estudiaremos pronto.

outerHTML: HTML completo del elemento

La propiedad `outerHTML` contiene el HTML completo del elemento. Eso es como `innerHTML` más el elemento en sí.

He aquí un ejemplo:

```
<div id="elem">Hola <b>Mundo</b></div>

<script>
  alert(elem.outerHTML); // <div id="elem">Hola <b>Mundo</b></div>
</script>
```

Cuidado: a diferencia de `innerHTML`, escribir en `outerHTML` no cambia el elemento. En cambio, lo reemplaza en el DOM.

Sí, suena extraño, y es extraño, por eso hacemos una nota aparte al respecto aquí. Echa un vistazo.

Considera el ejemplo:

```
<div>¡Hola, mundo!</div>

<script>
  let div = document.querySelector('div');

  // reemplaza div.outerHTML con <p>...</p>
  div.outerHTML = '<p>Un nuevo elemento</p>'; // (*)

  // ¡Guauu! ¡'div' sigue siendo el mismo!
  alert(div.outerHTML); // <div>¡Hola, mundo!</div> (**)
</script>
```

Parece realmente extraño, ¿verdad?

En la línea `(*)` reemplazamos `div` con `<p>Un nuevo elemento</p>`. En el documento externo (el DOM) podemos ver el nuevo contenido en lugar del `<div>`. Pero, como podemos ver en la línea `(**)`, ¡el valor de la antigua variable `div` no ha cambiado!

La asignación `outerHTML` no modifica el elemento DOM (el objeto al que hace referencia, en este caso, la variable 'div'), pero lo elimina del DOM e inserta el nuevo HTML en su lugar.

Entonces, lo que sucedió en `div.outerHTML=...` es:

- `div` fue eliminado del documento.
- Otro fragmento de HTML `<p>Un nuevo elemento</p>` se insertó en su lugar.
- `div` todavía tiene su antiguo valor. El nuevo HTML no se guardó en ninguna variable.

Es muy fácil cometer un error aquí: modificar `div.outerHTML` y luego continuar trabajando con `div` como si tuviera el nuevo contenido. Pero no es así. Esto es correcto para `innerHTML`, pero no para `outerHTML`.

Podemos escribir en `elem.outerHTML`, pero debemos tener en cuenta que no cambia el elemento en el que estamos escribiendo ('elem'). En su lugar, coloca el nuevo HTML en su lugar. Podemos obtener referencias a los nuevos elementos consultando el DOM.

nodeValue/data: contenido del nodo de texto

La propiedad `innerHTML` solo es válida para los nodos de elementos.

Otros tipos de nodos, como los nodos de texto, tienen su contraparte: propiedades `nodeValue` y `data`. Estas dos son casi iguales para uso práctico, solo hay pequeñas diferencias de especificación. Entonces usaremos `data`, porque es más corto.

Un ejemplo de lectura del contenido de un nodo de texto y un comentario:

```
<body>
  Hola
  <!-- Comentario -->
  <script>
    let text = document.body.firstChild;
    alert(text.data); // Hola

    let comment = text.nextSibling;
    alert(comment.data); // Comentario
  </script>
</body>
```

Para los nodos de texto podemos imaginar una razón para leerlos o modificarlos, pero ¿por qué comentarios?

A veces, los desarrolladores incorporan información o instrucciones de plantilla en HTML, así:

```
<!-- if isAdmin -->
  <div>¡Bienvenido, administrador!</div>
<!-- /if -->
```

...Entonces JavaScript puede leerlo desde la propiedad `data` y procesar las instrucciones integradas.

textContent: texto puro

El `textContent` proporciona acceso al *texto* dentro del elemento: solo texto, menos todas las `<tags>` .

Por ejemplo:

```
<div id="news">
  <h1>¡Titular!</h1>
  <p>¡Los marcianos atacan a la gente!</p>
</div>

<script>
  // ¡Titular! ¡Los marcianos atacan a la gente!
  alert(news.textContent);
</script>
```

Como podemos ver, solo se devuelve texto, como si todas las `<etiquetas>` fueran recortadas, pero el texto en ellas permaneció.

En la práctica, rara vez se necesita leer este tipo de texto.

Escribir en `textContent` es mucho más útil, porque permite escribir texto de "forma segura".

Digamos que tenemos un string arbitrario, por ejemplo, ingresado por un usuario, y queremos mostrarlo.

- Con `innerHTML` lo tendremos insertado "como HTML", con todas las etiquetas HTML.
- Con `textContent` lo tendremos insertado "como texto", todos los símbolos se tratan literalmente.

Compara los dos:

```
<div id="elem1"></div>
<div id="elem2"></div>

<script>
  let name = prompt("¿Cuál es tu nombre?", "<b>¡Winnie-Pooh!</b>");

  elem1.innerHTML = name;
  elem2.textContent = name;
</script>
```

1. El primer `<div>` obtiene el nombre "como HTML": todas las etiquetas se convierten en etiquetas, por lo que vemos el nombre en negrita.
2. El segundo `<div>` obtiene el nombre "como texto", así que literalmente vemos `<b>¡Winnie-Pooh!</b>` .

En la mayoría de los casos, esperamos el texto de un usuario y queremos tratarlo como texto. No queremos HTML inesperado en nuestro sitio. Una asignación a `textContent` hace exactamente eso.

La propiedad "hidden"

El atributo "hidden" y la propiedad DOM especifican si el elemento es visible o no.

Podemos usarlo en HTML o asignarlo usando JavaScript, así:

```
<div>Ambos divs a continuación están ocultos</div>

<div hidden>Con el atributo "hidden"</div>

<div id="elem">JavaScript asignó la propiedad "hidden"</div>

<script>
  elem.hidden = true;
</script>
```

Técnicamente, `hidden` funciona igual que `style="display:none"` . Pero es más corto de escribir.

Aquí hay un elemento parpadeante:

```
<div id="elem">Un elemento parpadeante</div>

<script>
  setInterval(() => elem.hidden = !elem.hidden, 1000);
</script>
```

Más propiedades

Los elementos DOM también tienen propiedades adicionales, en particular aquellas que dependen de la clase:

- `value` – el valor para `<input>`, `<select>` y `<textarea>` (`HTMLInputElement`, `HTMLSelectElement` ...).
- `href` – el "href" para `<a href="...">` (`HTMLAnchorElement`).
- `id` – el valor del atributo "id", para todos los elementos (`HTMLElement`).
- ...y mucho más...

Por ejemplo:

```
1  <input type="text" id="elem" value="value">
2
3  <script>
4    alert(elem.type); // "text"
5    alert(elem.id); // "elem"
6    alert(elem.value); // value
7  </script>
```

La mayoría de los atributos HTML estándar tienen la propiedad DOM correspondiente, y podemos acceder a ella así.

Si queremos conocer la lista completa de propiedades admitidas para una clase determinada, podemos encontrarlas en la especificación. Por ejemplo, `HTMLInputElement` está documentado en https://html.spec.whatwg.org/#htmlinputelement.

O si nos gustaría obtenerlos rápidamente o estamos interesados en una especificación concreta del navegador, siempre podemos generar el elemento usando `console.dir(elem)` y leer las propiedades. O explora las "propiedades DOM" en la pestaña Elements de las herramientas de desarrollo del navegador.

Resumen

Cada nodo DOM pertenece a una determinada clase. Las clases forman una jerarquía. El conjunto completo de propiedades y métodos proviene de la herencia.

Las propiedades principales del nodo DOM son:

`nodeType`

Podemos usarla para ver si un nodo es un texto o un elemento. Tiene un valor numérico: `1` para elementos, `3` para nodos de texto y algunos otros para otros tipos de nodos. Solo lectura.

`nodeName/tagName`

Para los elementos, nombre de la etiqueta (en mayúsculas a menos que esté en modo XML). Para los nodos que no son elementos, `nodeName` describe lo que es. Solo lectura.

`innerHTML`

El contenido HTML del elemento. Puede modificarse.

`outerHTML`

El HTML completo del elemento. Una operación de escritura en `elem.outerHTML` no toca a `elem` en sí. En su lugar, se reemplaza con el nuevo HTML en el contexto externo.

`nodeValue/data`

El contenido de un nodo que no es un elemento (text, comment). Estos dos son casi iguales, usualmente usamos `data` . Puede modificarse.

`textContent`

El texto dentro del elemento: HTML menos todas las `<tags>` . Escribir en él coloca el texto dentro del elemento, con todos los caracteres especiales y etiquetas tratados exactamente como texto. Puede insertar de forma segura texto generado por el usuario y protegerse de inserciones HTML no deseadas.

`hidden`

Cuando se establece en `true` , hace lo mismo que CSS `display:none` .

Los nodos DOM también tienen otras propiedades dependiendo de su clase. Por ejemplo, los elementos `<input>` (`HTMLInputElement`) admiten `value` , `type` , mientras que los elementos `<a>` (`HTMLAnchorElement`) admiten `href` , etc. La mayoría de los atributos HTML estándar tienen una propiedad DOM correspondiente.

Sin embargo, los atributos HTML y las propiedades DOM no siempre son iguales, como veremos en el próximo capítulo.

Tareas

Contar los descendientes

importancia: 5

Hay un árbol estructurado como `ul/li` anidado.

Escribe el código que para cada `<li>` muestra:

1. ¿Cuál es el texto dentro de él (sin el subárbol)?
2. El número de `<li>` anidados: todos los descendientes, incluidos los profundamente anidados.

Demo en nueva ventana

Abrir un entorno controlado para la tarea.

solución

¿Qué hay en nodeType?

importancia: 5

¿Qué muestra el script?

```
<html>

<body>
  <script>
    alert(document.body.lastChild.nodeType);
  </script>
</body>

</html>
```

solución

Etiqueta en comentario

importancia: 3

¿Qué muestra este código?

```
<script>
  let body = document.body;

  body.innerHTML = "<!--" + body.tagName + "-->";

  alert( body.firstChild.data ); // ¿qué hay aquí?
</script>
```

solución

¿Dónde está el "document" en la jerarquía?

importancia: 4

¿A qué clase pertenece el `document`?

¿Cuál es su lugar en la jerarquía DOM?

¿Hereda de `Node` o `Element`, o tal vez `HTMLElement`?

solución

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...)