24 de octubre de 2022

Página: DOMContentLoaded, load, beforeunload, unload

El ciclo de vida de una página HTML tiene tres eventos importantes:

- `DOMContentLoaded` – el navegador HTML está completamente cargado y el árbol DOM está construido, pero es posible que los recursos externos como `<img>` y hojas de estilo aún no se hayan cargado.
- `load` – no solo se cargó el HTML, sino también todos los recursos externos: imágenes, estilos, etc.
- `beforeunload/unload` – el usuario sale de la pagina.

Cada evento puede ser útil:

- Evento `DOMContentLoaded` – DOM está listo, por lo que el controlador puede buscar nodos DOM, inicializar la interfaz.
- Evento `load` – se cargan recursos externos, por lo que se aplican estilos, se conocen tamaños de imagen, etc.
- Evento `beforeunload` – el usuario se va: podemos comprobar si el usuario guardó los cambios y preguntarle si realmente quiere irse.
- Evento `unload` – el usuario casi se fue, pero aún podemos iniciar algunas operaciones, como enviar estadísticas.

Exploremos los detalles de estos eventos.

DOMContentLoaded

El evento `DOMContentLoaded` ocurre en el objeto `document` .

Debemos usar `addEventListener` para capturarlo:

```
document.addEventListener("DOMContentLoaded", ready);
// no "document.onDOMContentLoaded = ..."
```

Por ejemplo:

```
<script>
  function ready() {
    alert('DOM is ready');

    // la imagen aún no está cargada (a menos que se haya almacenado en caché), por lo que
    alert(`Image size: ${img.offsetWidth}x${img.offsetHeight}`);
  }

  document.addEventListener("DOMContentLoaded", ready);
</script>

<img id="img" src="https://en.js.cx/clipart/train.gif?speed=1&cache=0">
```

En el ejemplo, el controlador del evento `DOMContentLoaded` se ejecuta cuando el documento está cargado, por lo que puede ver todos los elementos, incluido el `<img>` que está después de él.

Pero no espera a que se cargue la imagen. Entonces, `alert` muestra los tamaños en cero.

A primera vista, el evento `DOMContentLoaded` es muy simple. El árbol DOM está listo – aquí está el evento. Sin embargo, hay algunas peculiaridades.

DOMContentLoaded y scripts

Cuando el navegador procesa un documento HTML y se encuentra con una etiqueta `<script>` , debe ejecutarla antes de continuar construyendo el DOM. Esa es una precaución, ya que los scripts pueden querer modificar el DOM, e incluso hacer `document.write` en él, por lo que `DOMContentLoaded` tiene que esperar.

Entonces DOMContentLoaded siempre ocurre después de tales scripts:

```
<script>
  document.addEventListener("DOMContentLoaded", () => {
```

```
    alert("DOM listo!");
  });
</script>

<script src="https://cdnjs.cloudflare.com/ajax/libs/lodash.js/4.3.0/lodash.js"></script>

<script>
  alert("Libreria cargada, linea de script ejecutada");
</script>
```

En el ejemplo anterior, primero vemos "Biblioteca cargada ..." y luego "¡DOM listo!" (se ejecutan todos los scripts).

Scripts que no bloquean DOMContentLoaded

Hay dos excepciones a esta regla:

1. Scripts con el atributo `async`, que cubriremos un poco más tarde, no bloquea el `DOMContentLoaded`.
2. Los scripts que se generan dinámicamente con `document.createElement('script')` y luego se agregan a la página web, tampoco bloquean este evento.

DOMContentLoaded y estilos

Las hojas de estilo externas no afectan a DOM, por lo que `DOMContentLoaded` no las espera.

Pero hay una trampa. Si tenemos un script después del estilo, entonces ese script debe esperar hasta que se cargue la hoja de estilo:

```
<link type="text/css" rel="stylesheet" href="style.css">
<script>
  // el script no se ejecuta hasta que se cargue la hoja de estilo
  alert(getComputedStyle(document.body).marginTop);
</script>
```

La razón de esto es que el script puede querer obtener coordenadas y otras propiedades de elementos dependientes del estilo, como en el ejemplo anterior. Naturalmente, tiene que esperar a que se carguen los estilos.

Como DOMContentLoaded espera los scripts, ahora también espera a los estilos que están antes que ellos.

Autocompletar del navegador integrado

Firefox, Chrome y Opera autocompletan formularios en `DOMContentLoaded`.

Por ejemplo, si la página tiene un formulario con nombre de usuario y contraseña, y el navegador recuerda los valores, entonces en `DOMContentLoaded` puede intentar completarlos automáticamente (si el usuario lo aprueba).

Entonces, si `DOMContentLoaded` es pospuesto por scripts de largo tiempo de carga, el autocompletado también espera. Probablemente haya visto eso en algunos sitios (si usa la función de autocompletar del navegador): los campos de inicio de sesión/contraseña no se autocompletan inmediatamente, sino con retraso hasta que la página se carga por completo. En realidad es el retraso hasta el evento `DOMContentLoaded`.

window.onload

El evento `load` en el objeto `window` se activa cuando se carga toda la página, incluidos estilos, imágenes y otros recursos. Este evento está disponible a través de la propiedad `onload`.

El siguiente ejemplo muestra correctamente los tamaños de las imágenes, porque `window.onload` espera todas las imágenes:

```
<script>
  window.onload = function() { // también puede usar window.addEventListener('load', (event
    alert('Página cargada');

    // la imagen es cargada al mismo tiempo
    alert(`Tamaño de imagen: ${img.offsetWidth}x${img.offsetHeight}`);
  };
</script>

<img id="img" src="https://en.js.cx/clipart/train.gif?speed=1&cache=0">
```

window.onunload

Cuando un visitante abandona la página, el evento `unload` se activa en `window`. Podemos hacer algo allí que no implique un retraso, como cerrar ventanas emergentes relacionadas.

La excepción notable es el envío de análisis.

Supongamos que recopilamos datos sobre cómo se usa la página: clicks del mouse, desplazamientos, áreas de página visitadas, etc.

Naturalmente, el evento `unload` sucede cuando el usuario nos deja y nos gustaría guardar los datos en nuestro servidor.

Existe un método especial `navigator.sendBeacon(url, data)` para tales necesidades, descrito en la especificación https://w3c.github.io/beacon/.

Este envía los datos en segundo plano. La transición a otra página no se retrasa: el navegador abandona la página, pero aún realiza `sendBeacon`.

Así es como se usa:

```
let analyticsData = { /* objeto con datos recopilados */ };

window.addEventListener("unload", function() {
  navigator.sendBeacon("/analytics", JSON.stringify(analyticsData));
});
```

- La solicitud se envía como POST.
- Podemos enviar no solo una cadena, sino también formularios y otros formatos, como se describe en el capítulo <info: fetch>, pero generalmente es un objeto string.
- Los datos están limitados por 64 kb.

Cuando finaliza la solicitud `sendBeacon`, es probable que el navegador ya haya abandonado el documento, por lo que no hay forma de obtener la respuesta del servidor (que suele estar vacía para análisis).

También hay una bandera `keepalive` para hacer tales solicitudes "after-page-left" en el método [fetch](info: fetch) para solicitudes de red genéricas. Puede encontrar más información en el capítulo <info: fetch-api>.

Si queremos cancelar la transición a otra página, no podemos hacerlo aquí. Pero podemos usar otro evento: `onbeforeunload`.

window.onbeforeunload

Si un visitante inició la navegación fuera de la página o intenta cerrar la ventana, el controlador `beforeunload` solicita una confirmación adicional.

Si cancelamos el evento, el navegador puede preguntar al visitante si está seguro.

Puede probarlo ejecutando este código y luego recargando la página:

```
window.onbeforeunload = function() {
  return false;
};
```

Por razones históricas, devolver una cadena no vacía también cuenta como cancelar el evento. Hace algún tiempo, los navegadores solían mostrarlo como un mensaje, pero como dice la especificación moderna, no deberían.

Aquí hay un ejemplo:

```
window.onbeforeunload = function() {
  return "Hay cambios sin guardar. ¿Salir ahora?";
};
```

El comportamiento se modificó, porque algunos webmasters abusaron de este controlador de eventos mostrando mensajes engañosos y molestos. Entonces, en este momento, los navegadores antiguos aún pueden mostrarlo como un mensaje, pero aparte de eso, no hay forma de personalizar el mensaje que se muestra al usuario.

El `event.preventDefault()` no funciona desde un manejador `beforeunload`

Esto puede sonar extraño, pero la mayoría de los navegadores ignoran `event.preventDefault()` .

Lo que significa que el siguiente código puede no funcionar:

```
window.addEventListener("beforeunload", (event) => {
  // no funciona, así que el manejador de evento no hace nada
  event.preventDefault();
});
```

En lugar de ello, en tales manejadores uno debe establecer `event.returnValue` a un string para obtener un resultado similar al pretendido en el código de arriba:

```
window.addEventListener("beforeunload", (event) => {
  // funciona, lo mismo que si devolviera desde window.onbeforeunload
  event.returnValue = "Hsy cambios sin grabar. ¿Abandonar ahora?";
});
```

readyState

¿Qué sucede si configuramos el controlador `DOMContentLoaded` después de cargar el documento?

Naturalmente, nunca se ejecutará.

Hay casos en los que no estamos seguros de si el documento está listo o no. Nos gustaría que nuestra función se ejecute cuando se cargue el DOM, ya sea ahora o más tarde.

La propiedad `document.readyState` nos informa sobre el estado de carga actual.

Hay 3 valores posibles:

- `"loading"` – el documento se está cargando.
- `"interactive"` – el documento fue leído por completo.
- `"complete"` – el documento se leyó por completo y todos los recursos (como imágenes) también se cargaron.

Entonces podemos verificar `document.readyState` y configurar un controlador o ejecutar el código inmediatamente si está listo.

Como esto:

```
function work() { /*...*/ }

if (document.readyState == 'loading') {
  // cargando todavía, esperar el evento
  document.addEventListener('DOMContentLoaded', work);
} else {
  // DOM está listo!
  work();
}
```

También existe el evento `readystatechange` que se activa cuando cambia el estado, por lo que podemos imprimir todos estos estados así:

```
// estado actual
console.log(document.readyState);

//imprimir los cambios de estado
document.addEventListener('readystatechange', () => console.log(document.readyState));
```

El evento `readystatechange` es una mecánica alternativa para rastrear el estado de carga del documento, apareció hace mucho tiempo. Hoy en día, rara vez se usa.

Veamos el flujo de eventos completo para ver si están completados.

Aquí hay un documento con `<iframe>` , `<img>` y controladores que registran eventos:

```
<script>
  log('initial readyState:' + document.readyState);

  document.addEventListener('readystatechange', () => log('readyState:' + document.readySta
```

```
  document.addEventListener('DOMContentLoaded', () => log('DOMContentLoaded'));

  window.onload = () => log('window onload');
</script>

<iframe src="iframe.html" onload="log('iframe onload')"></iframe>

<img src="https://en.js.cx/clipart/train.gif" id="img">
<script>
  img.onload = () => log('img onload');
</script>
```

El ejemplo práctico está en el sandbox.

La salida típica:

1. [1] readyState inicial: loading
2. [2] readyState: interactive
3. [2] DOMContentLoaded
4. [3] iframe onload
5. [4] img onload
6. [4] readyState: complete
7. [4] window onload

Los números entre corchetes denotan el tiempo aproximado en el que ocurre. Los eventos etiquetados con el mismo dígito ocurren aproximadamente al mismo tiempo (+ – unos pocos ms).

- `document.readyState` se convierte en `interactive` justo antes de `DOMContentLoaded` . Estas dos cosas realmente significan lo mismo.
- `document.readyState` se convierte en `complete` cuando se cargan todos los recursos (`iframe` e `img`). Aquí podemos ver que ocurre aproximadamente al mismo tiempo que `img.onload` (`img` es el último recurso) y `window.onload` . Cambiar al estado `complete` significa lo mismo que "window.onload". La diferencia es que `window.onload` siempre funciona después de todos los demás controladores `load` .

Resumen

Eventos de carga de página:

- El evento `DOMContentLoaded` se activa en el `document` cuando el DOM está listo. Podemos aplicar JavaScript a elementos en esta etapa.
  - Secuencias de comandos como `<script> ... </script>` o `<script src =" ... "> </script>` bloquean DOMContentLoaded, el navegador espera a que se ejecuten.
  - Las imágenes y otros recursos también pueden seguir cargándose.
- El evento `load` en `window` se activa cuando se cargan la página y todos los recursos. Rara vez lo usamos, porque generalmente no hay necesidad de esperar tanto.
- El evento `beforeunload` en `window` se activa cuando el usuario quiere salir de la página. Si cancelamos el evento, el navegador pregunta si el usuario realmente quiere irse (por ejemplo, tenemos cambios sin guardar).
- El evento `unload` en `window` se dispara cuando el usuario finalmente se está yendo, en el controlador solo podemos hacer cosas simples que no impliquen demoras o preguntas al usuario. Debido a esa limitación, rara vez se usa. Podemos enviar una solicitud de red con `navigator.sendBeacon` .
- `document.readyState` es el estado actual del documento, los cambios se pueden rastrear con el evento `readystatechange` :
  - `loading` – el documento esta cargando.
  - `interactive` – el documento se analiza, ocurre aproximadamente casi al mismo tiempo que `DOMContentLoaded` , pero antes.
  - `complete` – el documento y los recursos se cargan, ocurre aproximadamente casi al mismo tiempo que `window.onload` , pero antes.

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...)

A **alberto**

hace 2 años editado

Los eventos **beforeunload** y **unload** son una dupla poderosa y algo confusa.

El evento **beforeunload** se dispara cuando desde el navegador optamos por **cerrar el tab** donde se ejecuta nuestra aplicación o **cerrar el navegador**.

En ambos casos, aparece una ventana emergente tipo **confirm** controlada por el navegador y con un mensaje preestablecido que podia personalizarse anteriormente, ahora eso está limitado y, aunque lo indiques, lo ignora.

Esa ventana emergente presenta 2 botones, uno para **confirmar** y otro para **cancelar**.

Si seleccionas **cancelar**, no sucede nada, la acción queda sin efecto.

Si seleccionas **confirmar**, ocurre lo siguiente:
1) si no has configurado el evento **unload**, no sucede nada, la acción queda sin efecto.
2) si configuraste el evento 'unload' este se dispara inmediatamente y desde alli podrás realizar algunas tareas finales y liberar recursos antes de descargarse la página de memoria.

Muy interesante lo que explica articulo en la sección **window.onunload**, yo desconocía esto: **navigator.sendBeacon**, se le ve un potenciaal tremendo, lo pondré en la caja de herramientas. ;-)