Temario

- Flujo de eventos
- Eventos en react
- Eventos Sinteticos

Envío de eventos y flujo de eventos DOM

Esta sección brinda una breve descripción general del mecanismo de envío de eventos y describe cómo se propagan los eventos a través del árbol DOM. Las aplicaciones pueden enviar objetos de eventos utilizando el dispatchEvent()método, y el objeto de evento se propagará a través del árbol DOM según lo determine el flujo de eventos DOM.

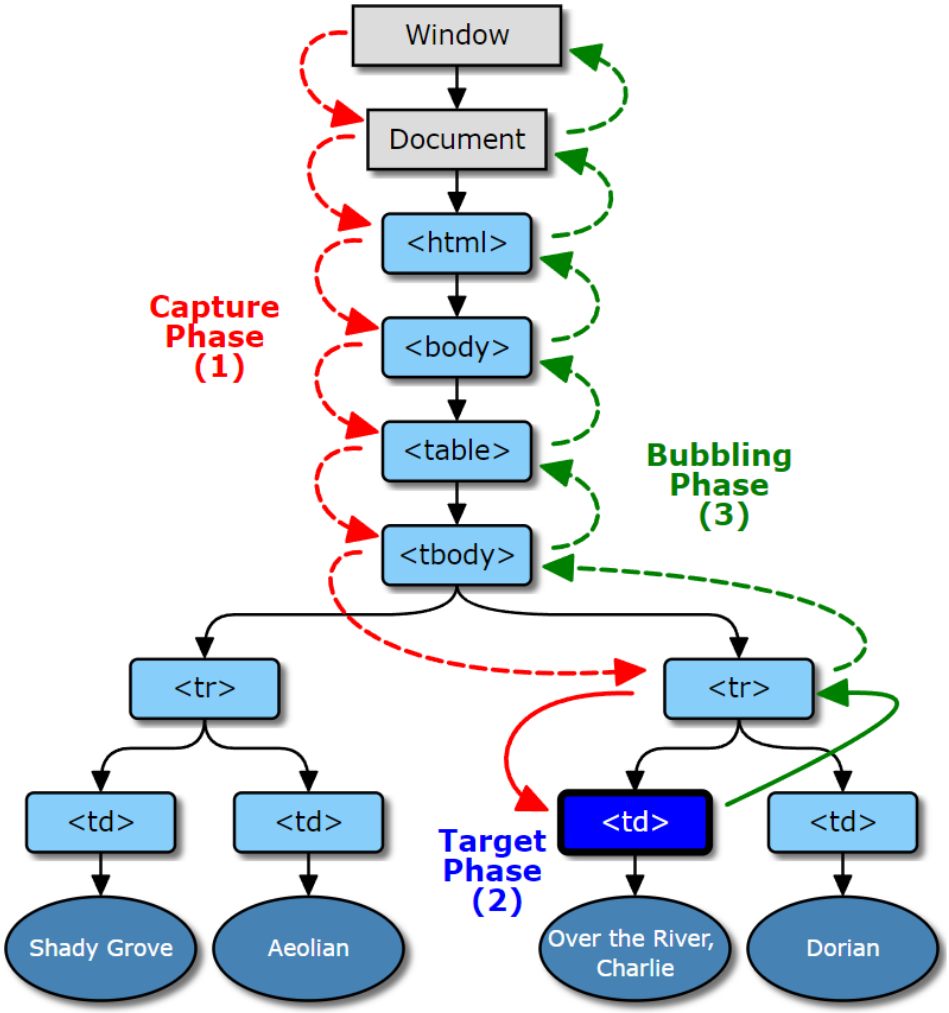

Figure 1 *Representación gráfica de un evento enviado en un árbol DOM utilizando el flujo de eventos DOM*

Los objetos de evento se envían a un objetivo de evento. Pero antes de que pueda comenzar el envío, primero se debe determinar la ruta de propagación del objeto de evento.

La ruta de propagación es una lista ordenada de objetivos de eventos actuales a través de los cuales pasa el evento. Esta ruta de propagación refleja la estructura de árbol jerárquico del documento. El último elemento de la lista es el objetivo del evento y los elementos anteriores de la lista se conocen como antepasados del objetivo, con el elemento inmediatamente anterior como el principal del objetivo.

Una vez que se ha determinado la ruta de propagación, el objeto de evento pasa por una o más fases de evento. Hay tres fases de eventos: fase de captura, fase de objetivo y fase de burbuja. Los objetos de evento completan estas fases como se describe a continuación. Se omitirá una fase si no se admite o si se detuvo la propagación del objeto de evento. Por

ejemplo, si el atributo bubbles se establece en falso, se omitirá la fase de burbuja y, si stopPropagation()se ha llamado antes del envío, se omitirán todas las fases.

La fase de captura: el objeto de evento se propaga a través de los ancestros del objetivo desde la ventana hasta el padre del objetivo. Esta fase también se conoce como la fase de captura.

La fase de destino: el objeto de evento llega al destino de evento del objeto de evento. Esta fase también se conoce como la fase en el objetivo. Si el tipo de evento indica que el evento no burbujea, el objeto de evento se detendrá después de completar esta fase.

La fase de burbuja: el objeto de evento se propaga a través de los ancestros del objetivo en orden inverso, comenzando con el padre del objetivo y terminando con la Ventana . Esta fase también se conoce como la fase burbujeante.

Acciones predeterminadas y eventos cancelables

Los eventos generalmente los envía la implementación como resultado de una acción del usuario, en respuesta a la finalización de una tarea o para señalar el progreso durante una actividad asincrónica (como una solicitud de red). Algunos eventos se pueden usar para controlar el comportamiento que la implementación puede tomar a continuación (o deshacer una acción que la implementación ya tomó). Se dice que los eventos de esta categoría son cancelables y el comportamiento que cancelan se denomina acción predeterminada. Los objetos de eventos cancelables se pueden asociar con una o más 'acciones predeterminadas'. Para cancelar un evento, llame al método preventDefault().

Un evento ***mousedown*** se envía inmediatamente después de que el usuario presiona un botón en un dispositivo señalador (generalmente un mouse). Una posible acción predeterminada tomada por la implementación es configurar una máquina de estado que permita al usuario arrastrar imágenes o seleccionar texto. La acción predeterminada depende de lo que suceda a continuación; por ejemplo, si el dispositivo señalador del usuario está sobre el texto, podría comenzar una selección de texto. Si el dispositivo señalador del usuario está sobre una imagen, podría comenzar una acción de arrastre de imagen. Evitar la acción predeterminada de un evento ***mousedown*** evita que se produzcan estas acciones.

Las acciones predeterminadas normalmente se realizan después de que se haya completado el envío del evento, pero en casos excepcionales también se pueden realizar inmediatamente antes de que se envíe el evento.

La acción predeterminada asociada con el evento click en los elementos <input type="checkbox"> alterna el valor checked del atributo IDL de ese elemento. Si se hace click cancela la acción predeterminada del evento, el valor se restaura a su estado anterior.

Cuando se cancela un evento, se omiten las acciones predeterminadas condicionales asociadas con el evento (o, como se mencionó anteriormente, si las acciones predeterminadas se llevan a cabo antes del envío, se deshace su efecto). El atributo cancelable indica si un objeto de evento es cancelable . La llamada preventDefault() detiene todas las acciones predeterminadas relacionadas de un objeto de evento. El atributo defaultPrevent indica si un evento ya ha sido cancelado (por ejemplo, por un detector de eventos anterior). Si la propia aplicación DOM inició el envío, el valor de retorno del método dispatchEvent() indica si el objeto de evento fue cancelado.

Muchas implementaciones interpretan además el valor de retorno de un detector de eventos, como el valor false, para indicar que la acción predeterminada de los eventos cancelables se cancelará (aunque los controladores window.onerror se cancelan al devolver true).

Eventos síncronos y asíncronos

Los eventos se pueden enviar de forma síncrona o asíncrona.

Los eventos que son síncronos (eventos de sincronización) se tratan como si estuvieran en una cola virtual en un modelo de primero en entrar, primero en salir, ordenados por secuencia de ocurrencia temporal con respecto a otros eventos, a cambios en el DOM y al usuario. interacción. Cada evento en esta cola virtual se retrasa hasta que el evento anterior haya completado su comportamiento de propagación o se haya cancelado. Algunos eventos de sincronización son impulsados por un dispositivo o proceso específico, como los eventos del botón del mouse. Estos eventos se rigen por los algoritmos de orden de eventos definidos para ese conjunto de eventos, y los agentes de usuario distribuirán estos eventos en el orden definido.

Los eventos que son asíncronos (eventos asíncronos) pueden enviarse a medida que se completan los resultados de la acción, sin relación con otros eventos, con otros cambios en el DOM ni con la interacción del usuario.

Durante la carga de un documento, se analiza y ejecuta un elemento de secuencia de comandos en línea. El evento load se pone en cola para que se active de forma asíncrona en el elemento de secuencia de comandos. Sin embargo, debido a que es un evento asíncrono, no se garantiza su orden en relación con otros eventos síncronos activados durante la carga del documento (como el evento DOMContentLoaded de [HTML5]).

Eventos de confianza

El agente de usuario confía en los eventos generados por el agente de usuario , ya sea como resultado de la interacción del usuario o como resultado directo de los cambios en el DOM, con privilegios que no se otorgan a los eventos generados por script a través del método createEvent(), modificado utilizando el método initEvent(), o enviado a través del método dispatchEvent(). El atributo isTrusted de los eventos de confianza tiene un valor de true, mientras que los eventos que no son de confianza tienen un valor isTrusted de atributo de false.

La mayoría de los eventos que no son de confianza no desencadenarán acciones predeterminadas, con la excepción del evento click. Este evento siempre desencadena la acción predeterminada, incluso si el atributo isTrusted lo es false (este comportamiento se mantiene por compatibilidad con versiones anteriores). Todos los demás eventos que no son de confianza se comportan como si preventDefault()se hubiera llamado al método en ese evento.

Disparadores de activación y comportamiento

Ciertos objetivos de eventos (como un enlace o un elemento de botón) pueden tener un comportamiento de activación asociado (como seguir un enlace) que las implementaciones realizan en respuesta a un disparador de activación (como hacer clic en un enlace).

Tanto HTML como SVG tienen un <a>elemento que indica un enlace. Los disparadores de activación relevantes para un <a>elemento son un evento click en el contenido de texto o imagen del elemento <a>, o un evento keydown con un valor key de atributo de clave cuando el elemento tiene el foco. El comportamiento de activación de un elemento normalmente es cambiar el contenido de la ventana al contenido del nuevo documento, en el caso de enlaces externos, o reposicionar el documento actual en relación con el nuevo ancla, en el caso de enlaces internos.

Un activador de activación es una acción del usuario o un evento que indica a la implementación que debe iniciarse un comportamiento de activación. Los desencadenantes de activación iniciados por el usuario incluyen hacer clic con el botón del mouse en un elemento activable, presionar la tecla Enter cuando un elemento activable tiene el foco o presionar una tecla que de alguna manera está vinculada a un elemento activable (una tecla de acceso rápido o tecla de acceso) incluso cuando ese elemento no lo hace. tener enfoque Disparadores de activación basados en eventos puede incluir eventos basados en temporizadores que activan un elemento a una determinada hora o después de que haya transcurrido un cierto período de tiempo, eventos de progreso después de que se haya completado una determinada acción o muchos otros eventos basados en condiciones o estados.

Manejando eventos en React

Manejar eventos en elementos de React es muy similar a manejar eventos con elementos del DOM. Hay algunas diferencias de sintaxis:

Los eventos de React se nombran usando camelCase, en vez de minúsculas.

Con JSX pasas una función como el manejador del evento, en vez de un string.

Por ejemplo, el HTML:

```
<button onclick="activateLasers()">
  Activate Lasers
</button>
```

En React es algo diferente:

```
<button onClick={activateLasers}>
  Activate Lasers
</button>
```

Otra diferencia es que en React no puedes retornar false para prevenir el comportamiento por defecto. Debes, explícitamente, llamar preventDefault. Por ejemplo, en un HTML plano, para prevenir el comportamiento por defecto de enviar un formulario, puedes escribir:

```
<form onsubmit="console.log('You clicked submit.'); return false">
  <button type="submit">Submit</button>
</form>
```

En cambio, en React, esto podría ser:

```
function Form() {
  function handleSubmit(e) {
    e.preventDefault();
    console.log('You clicked submit.');
  }

  return (
```

```
  <form onSubmit={handleSubmit}>
    <button type="submit">Submit</button>
  </form>
 );
}
```

Aquí, e es un evento sintético. React define estos eventos sintéticos acorde a las especificaciones W3C, para que no tengas que preocuparte por la compatibilidad entre distintos navegadores. Los eventos de React no funcionan exactamente igual que los eventos nativos.

Cuando estás utilizando React, generalmente no necesitas llamar addEventListener para agregar escuchadores de eventos a un elemento del DOM después de que este es creado. En cambio, solo debes proveer un manejador de eventos cuando el elemento se renderiza inicialmente.

Cuando defines un componente usando una clase de ES6, un patrón muy común es que los manejadores de eventos sean un método de la clase. Por ejemplo, este componente Toggle renderiza un botón que permite al usuario cambiar el estado entre “ENCENDIDO” y “APAGADO”:

```
class Toggle extends React.Component {
 constructor(props) {
  super(props);
  this.state = {isToggleOn: true};

  // Este enlace es necesario para hacer que `this` funcione en el callback
  this.handleClick = this.handleClick.bind(this);
 }

 handleClick() {
  this.setState(prevState => ({
   isToggleOn: !prevState.isToggleOn
  }));
 }

 render() {
```

```
    return (
      <button onClick={this.handleClick}>
        {this.state.isToggleOn ? 'ON' : 'OFF'}
      </button>
    );
  }
}
```

Tienes que tener mucho cuidado en cuanto al significado de this en los callbacks de JSX. En JavaScript, los métodos de clase no están ligados por defecto. Si olvidas ligar this.handleClick y lo pasas a onClick, this será undefined cuando se llame la función.

Esto no es un comportamiento especifico de React; esto hace parte de cómo operan las funciones JavaScript. Generalmente, si refieres un método sin usar () después de este, tal como onClick={this.handleClick}, deberías ligar ese método.

Si te molesta llamar bind, existen dos maneras de evitarlo. Si usas la sintaxis experimental campos públicos de clases, puedes usar los campos de clases para ligar los callbacks correctamente:

```
class LoggingButton extends React.Component {
  // Esta sintaxis nos asegura que `this` está ligado dentro de handleClick
  // Peligro: esto es una sintaxis *experimental*
  handleClick = () => {
    console.log('this is:', this);
  }

  render() {
    return (
      <button onClick={this.handleClick}>
        Click me
      </button>
    );
  }
}
```

Esta sintaxis está habilitada por defecto en Create React App.

Si no estás usando la sintaxis de campos públicos de clases, puedes usar una función flecha en el callback:

```
class LoggingButton extends React.Component {
 handleClick() {
  console.log('this is:', this);
 }

 render() {
  // Esta sintaxis nos asegura que `this` esta ligado dentro de handleClick
  return (
   <button onClick={() => this.handleClick()}>
    Click me
   </button>
  );
 }
}
```

El problema con esta sintaxis es que un callback diferente es creado cada vez que LogginButton es renderizado. En la mayoría de los casos, esto está bien. Sin embargo, si este callback se pasa como una propiedad a componentes más bajos, estos componentes podrían renderizarse nuevamente. Generalmente, recomendamos ligar en el constructor o usar la sintaxis de campos de clases, para evitar esta clase de problemas de rendimiento.

Pasando argumentos a escuchadores de eventos

Dentro de un bucle es muy común querer pasar un parámetro extra a un manejador de eventos. Por ejemplo, si id es el ID de una fila, cualquiera de los códigos a continuación podría funcionar:

```
<button onClick={(e) => this.deleteRow(id, e)}>Delete Row</button>

<button onClick={this.deleteRow.bind(this, id)}>Delete Row</button>
```

Las dos líneas anteriores son equivalentes, y utilizan funciones flecha y Function.prototype.bind respectivamente.

En ambos casos, el argumento e que representa el evento de React va a ser pasado como un segundo argumento después del ID. Con una función flecha, tenemos que pasarlo explícitamente, pero con bind cualquier argumento adicional es pasado automáticamente

Eventos sintéticos de React

Ya hemos visto cómo escribimos código parecido a HTML con JSX que se transforma en código HTML de verdad tras pasar por React. De una manera parecida, en React tenemos un sistema de eventos sintéticos que usaremos como si fueran normales. Aunque están diseñados para que pasen por eventos regulares, igual que JSX pasa por etiquetas HTML normales, cabe destacar que son una capa que nos proporciona React y que no son eventos reales del DOM, y por eso se llaman sintéticos.

Las funciones escuchadoras (listeners) de React se declaran de forma parecida a aquellos que no recomendábamos (en línea). Esto no es una contradicción: recordemos que en React escribimos interfaces declarativas, y esto es solo una sintaxis comprensible para asignar comportamiento. No deben declarar funciones escuchadoras así fuera de React.

Cuando escuchamos un evento, declaramos una función escuchadora (listener) que se ejecutará cuando se reciba un evento de cierto tipo. Esto es así tanto para eventos del DOM como para eventos sintéticos de React, sólo cambiaremos cómo asignamos la función al tipo de evento.

Vamos a ver un ejemplo. Queremos escuchar un evento de click desde un botón que declaramos con JSX. Escribiremos el botón (<button>texto</button>) y en un atributo onClick, añadiremos la función "escuchadora", que será la reacción. Quedará así:

```
const alertButton =
  <button onClick={ /* aquí va la función */ }>
    Pedir más información
  </button>;
```

Podríamos escribir directamente la función escuchadora como una arrow function ahí, pero no quedaría legible. Preferiremos declararla fuera y la pasaremos (sin llamarla) al atributo de JSX:

```
const onClickListener = event => {
  alert("Para más información, acuda a recepción.");
};
const alertButton = (
  <button onClick={onClickListener}>Pedir más información</button>
);
```

Ya está. Cuando hagamos clic en el botón, React se encargará de escuchar el evento y de ejecutar la función.

Naturalmente, hay más atributos para escuchar eventos a parte de onClick. Los nombres de los atributos tendrán la forma onEventoEscuchado, con cada palabra del nombre del evento que se escucha escrita con mayúsculas iniciales. Es decir, escucharemos el evento focus rellenando el atributo onFocus, el evento mouseover rellenando el atributo onMouseOver, y así sucesivamente. Pueden consultar el listado completo de atributos soportados, pero a continuación vamos a listar los más usados, como ya hicimos en la sesión de eventos:

Escuchadores de eventos de ratón:

onClick: botón izquierdo del ratón

onMouseOver: pasar el ratón sobre un elemento

onMouseOut: sacar el ratón del elemento

Escuchadores de eventos de teclado:

onKeyPress: pulsar una tecla

Escuchadores de eventos sobre elementos:

onFocus: poner el foco (seleccionar) en un elemento, por ejemplo un <input>

onBlur: quitar el foco de un elemento

onChange: al cambiar el contenido de un <input> , <textarea> o <select> (no es necesario quitar el foco del input para que se considere un cambio, al contrario que en el DOM)

Escuchadores de eventos de formularios:

onSubmit: pulsar el botón submit del formulario

onReset: pulsar el botón reset del formulario

React no puede controlar los eventos de la ventana, así que los siguientes eventos sintéticos no existen (sí existen sus correspondientes eventos de DOM):

Escuchadores de eventos de la ventana

onResize: se ha cambiado el tamaño de la ventana

onScroll: se ha hecho scroll en la ventana o un elemento