Proceso de captura y propagacion(bubbling) de los eventos en javascript

Samuel Alberto Solis Baldenegro 19100257

Propagación y captura Vamos a empezar con un ejemplo.

Este manejador está asignado a `<div>`, pero también se ejecuta si haces clic a cualquier elemento anidado como `<em>` ó `<code>`:

```
<div onclick="alert('¡El manejador!')"> <em>Si haces clic en<code>EM</code>, el manejador en <code>DIV</code> es ejecutado.</em> </div>
```

Propagación

El principio de propagación es simple.

Cuando un evento ocurre en un elemento, este primero ejecuta los manejadores que tiene asignados, luego los manejadores de su padre, y así hasta otros ancestros.

Digamos que tenemos 3 elementos anidados `FORM > DIV > P` con un manejador en cada uno de ellos:

```
<style> body * { margin: 10px; border: 1px solid blue; } </style>
```

```
<form onclick="alert('form')">FORM <div onclick="alert('div')">DIV <p onclick="alert('p')">P</p> </div> </form>
```

event.target

Un manejador en un elemento padre siempre puede obtener los detalles sobre dónde realmente ocurrió el evento.

El elemento anidado más profundo que causó el evento es llamado elemento objetivo, accesible como `event.target`

Nota la diferencia de `this (=event.currentTarget`):

`event.target` – e\Es el elemento "objetivo" que inició el evento, no cambia a través de todo el proceso de propagación. `this` – es el elemento "actual", el que tiene un manejador ejecutándose en el momento. Por ejemplo, si tenemos un solo manejador `form.onclick`, este puede atrapar todos los clicks dentro del formulario. No importa dónde el clic se hizo, se propaga hasta el `<form>` y ejecuta el manejador.

En el manejador `form.onclick`:

`this` (=`event.currentTarget`) es el elemento `<form>`, porque el manejador se ejecutó en él. `event.target` es el elemento actual dentro de el formulario al que se le hizo clic.

Veremos un ejemplo este es el html

```
!DOCTYPE HTML> <html>

<head> <meta charset="utf-8"> <link rel="stylesheet" href="example.css"> </head>

<body> Un clic muestra ambos, el <code>event.target</code> y <code>this</code> para comparar:

<form id="form">FORM <div>DIV <p>P</p> </div> </form>

<script src="script.js"></script> </body> </html>
```

Aqui vemos el css `form { background-color: green; position: relative; width: 150px; height: 150px; text-align: center; cursor: pointer; }`

`div { background-color: blue; position: absolute; top: 25px; left: 25px; width: 100px; height: 100px; }`

`p { background-color: red; position: absolute; top: 25px; left: 25px; width: 50px; height: 50px; line-height: 50px; margin: 0; }`

`body { line-height: 25px; font-size: 16px; }`

El script.js `form.onclick = function(event) { event.target.style.backgroundColor = 'yellow';`

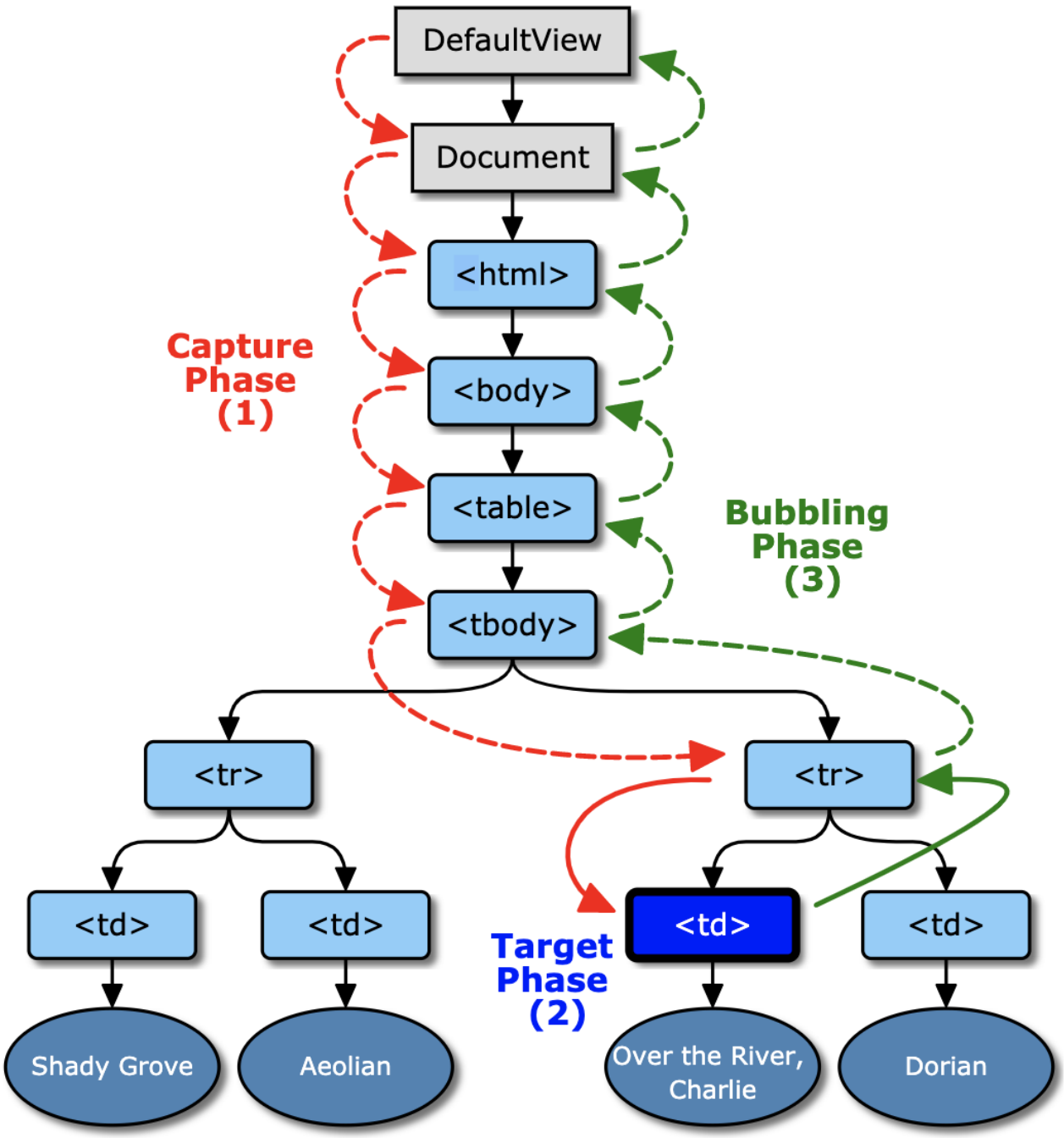
// chrome needs some time to paint yellow `setTimeout(() => { alert("target = " + event.target.tagName + ", this=" + this.tagName); event.target.style.backgroundColor = '' }, 0); };`

Captura

Hay otra fase en el procesamiento de eventos llamada “captura”. Es raro usarla en código real, pero a veces puede ser útil.

El estándar de eventos del DOM describe 3 fases de la propagación de eventos:

Fase de captura – el evento desciende al elemento. Fase de objetivo – el evento alcanza al elemento. Fase de propagación – el evento se propaga hacia arriba del elemento.



Se explica así: por un clic en `<td>` el evento va primero a través de la cadena de ancestros hacia el elemento (fase de captura), luego alcanza el objetivo y se desencadena ahí (fase de objetivo), y por último va hacia arriba (fase de propagación), ejecutando los manejadores en su camino.

Antes solo hablamos de la propagación porque la fase de captura es raramente usada. Normalmente es invisible a nosotros.

Los manejadores agregados usando la propiedad `on<event>` ó usando atributos HTML ó `addEventListener(event, handler)` con dos argumentos no ejecutarán la fase de captura, únicamente ejecutarán la 2da y 3ra fase.

Para atrapar un evento en la fase de captura, necesitamos preparar la opción `capture` como `true` en el manejador:

`elem.addEventListener(..., {capture: true})` // o, solo "true" es una forma más corta de {capture: true} `elem.addEventListener(..., true)` Hay dos posibles valores para la opción `capture`:

Si es `false` (por defecto), entonces el manejador es preparado para la fase de propagación. Si es `true` , entonces el manejador es preparado para la fase de captura.

Cuando ocurre un evento, el elemento más anidado dónde ocurrió se reconoce como el "elemento objetivo" (`event.target`).

- Luego el evento se mueve hacia abajo desde el documento raíz hacia `event.target`, llamando a los manejadores en el camino asignados con `addEventListener(..., true)` (`true` es una abreviación para `{capture: true}`).
- Luego los manejadores son llamados en el elemento objetivo mismo.
- Luego el evento se propaga desde `event.target` hacia la raíz, llamando a los manejadores que se asignaron usando `on<event>`, atributos HTML y `addEventListener` sin el 3er argumento o con el 3er argumento `false/{capture:false}`. Cada manejador puede acceder a las propiedades del objeto event:
- `event.target` – el elemento más profundo que originó el evento.
- `event.currentTarget` (=`this`) – el elemento actual que maneja el evento (el que tiene al manejador en él)
- `event.eventPhase` – la fase actual (captura=1, objetivo=2, propagación=3). Cualquier manejador de evento puede detener el evento al llamar `event.stopPropagation()`, pero no es recomendado porque no podemos realmente asegurar que no lo necesitaremos más adelante, quizá para completar diferentes cosas.