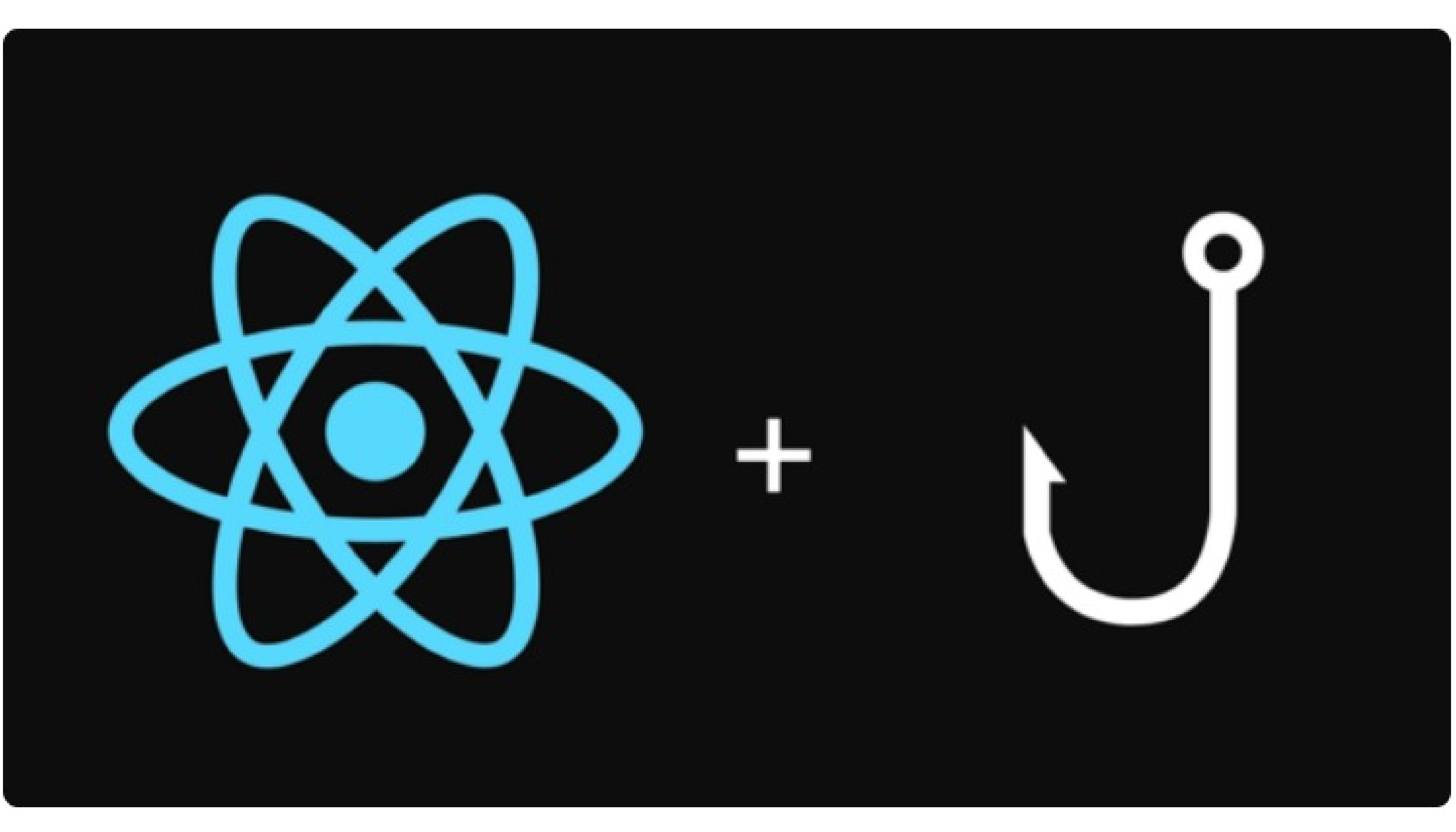

Entendiendo el hook useEffect

...

Brais Cao González
Software Engineer at Xing (New Work SE)
Fecha de publicación: 11 de oct. de 2021
Seguir

Cuando hablamos de React, posiblemente el segundo hook más usado sea useEffect, un hook que, en esencia, **sirve para realizar acciones como efecto secundario de ciertas situaciones.** Y es por esto que su nombre es useEffect, porque es un 'side effect'.

Entender esto es algo muy importante para saber usarlo bien, y para entender que aunque se utiliza para hacer lo que antes se hacía en los "antiguos" lifecycle methods de los componentes de clase de React, no es equivalente a estos y puede usarse para más cosas o puede causarte más problemas si no entiendes sus peculiaridades.

En este artículo entraré a profundizar en todos los aspectos de este hook para que puedas usarlo en tus aplicaciones sin problemas y que sea un salvavidas en vez de un incordio 😬

¿Cómo funciona useEffect?

Pasemos a ver como funciona el useEffect más básico, para no entrar en detalles de implementación, simplemente ejecutaremos un log dentro.

```
useEffect(() => {
  console.log('useEffect se lanza!');
})
```

Bien, este es el useEffect más simple que puedes hacer, un useEffect sin array de dependencias (más adelante veremos qué es esto y como usarlo correctamente) sin función de limpieza (también lo veremos más adelante) y con un simple console.log dentro.

Ahora, **¿cuándo y por qué se ejecuta lo que hay dentro de él?**

Pues bien, lo que hay dentro del hook **se va a ejecutar cuando el componente se monta y cada vez que el componente se rerenderice**, (profundizaremos más sobre esta situación en un próximo artículo dedicado a rerenders y rendimiento), y cada una de estas veces, el momento exacto en el que se ejecuta su contenido será **justo después de que el componente haya terminado de renderizarse por completo**. Esto es así para que la ejecución del efecto no interfiera con el renderizado de la UI y se corra el riesgo de bloquear este proceso aunque sea durante una mínima cantidad de tiempo.

Para los que hayais usado componentes de clase (o class components en inglés) este comportamiento sería similar al de los métodos "componentDidMount" y "componentDidUpdate" combinados.

Pero, ¿y si no quiero que se ejecute en cada rerender?, ¿y si solo quiero que se ejecute al montarse o como respuesta a un evento concreto?

El array de dependencias de useEffect

Pues para estos casos entra en juego el array de dependencias de useEffect.

Y ¿qué demonios es el array de dependencias?, pues es un array que se le pasa como parámetro a useEffect y en el que listamos aquellas variables a cuyos cambios queremos que se ejecute el efecto. Aquí entramos más en la parte de efecto secundario que mencioné al principio 😉

Veamos un par de ejemplos

Ejemplo 1:

```
useEffect(() => {
   console.log('useEffect se lanza solo cuando el componente se monta');
},[])
```

Un array de dependencias vacío, ¿y esto que significa?, pues significa que no se va a ejecutar como respuesta al cambio de ninguna variable, pero aun se va a ejecutar cuando el componente se monte, por tanto, **un useEffect con el array de dependencias vacío solo se ejecutará cuando el componente se monte.** Este comportamiento es similar al método "componentDidMount" de los componentes de clase.

Ejemplo 2:

```
useEffect(() => {
   console.log(count);
},[count])
```

Un array de dependencias con dependencias, es decir, que el efecto se ejecutará cada vez que una de ellas cambie, en este caso cada vez que la variable "count" cambie de valor, volveremos a ver el console.log en la consola de nuestro navegador.

Por tanto, se lanzará como efecto secundario de que "count" haya cambiado 🤯

Este comportamiento es muy parecido al de "componentDidUpdate" con el parámetro prevState y comparándolo con el state actual.

Esto es útil en infinidad de ocasiones, imagina que quieres hacer un fetch a una API cada vez que una variable "id" cambie, pues un useEffect con "id" como dependencia y dentro el fetch (más adelante veremos un ejemplo).

También es muy importante que todas las variables que haya dentro de la función que pasamos a useEffect, estén en el array de dependencias, para evitar que haya variables cambiando de valor sin que la función de dentro del useEffect muestre su valor actual y muestre el anterior, debido a lo que conocemos como un **closure**

Función de limpieza de useEffect

Ya entrando en la recta final de la explicación de este hook, no pensabas que un simple hook diese para tanto, ¿verdad?.

Si anteriormente habías usado class components, te estarás preguntando dónde está la alternativa a "componentWillUnmount", bien, esto es un poco particular, y depende del array de dependencias, valga la redundancia.

useEffect puede hacer uso de lo que se conoce como "función de limpieza", que **es una función que se ejecuta cuando el componente se desmonta, pero también cuando las dependencias cambian y antes de que el efecto se vuelva a ejecutar**, o si es un useEffect sin array de dependencias, en cada rerender.

Esto se usa mucho por ejemplo en situaciones en las que queremos anular una suscripción o purgar algún código asíncrono que no queramos que se ejecute cuando el componente ya está desmontado o no queremos que se ejecute más de una vez.

Esta función **tiene que ser estrictamente el único valor devuelto por la función que pasamos al useEffect**.

Ejemplo 1:

```
useEffect(() => {
   console.log('Componente se monta');

   return () => {
     console.log('Componente se desmonta');
   }

},[])
```

En este ejemplo, cuando el componente se monta, se ejecuta el primer log, y cuando el componente se desmonta, se ejecuta la función de limpieza, que es la función devuelta por la función que pasamos al useEffect, y se ejecutará el segundo log.

Este es un comportamiento muy parecido a "componentWillUnmount".

Ejemplo 2:

```
useEffect(() => {
  const id = setInterval(function log() {
    console.log(`Count es: ${count}`);
  }, 2000);



  //Esta es la función de limpieza
  return () => {
    clearInterval(id);
  }
}, [count]);
```

En este ejemplo, llamamos a la función de limpieza antes de que el efecto se vaya a relanzar después de un rerender, es decir, si la variable count cambia, y el efecto se va a volver a ejecutar, antes de eso, se ejecuta la función que realiza un clearInterval y después de eso se ejecutará el efecto que hará el nuevo setInterval, evitando así que se nos acumulen los contadores en cada rerender.

Llamadas asíncronas dentro de useEffect

Y por culpa de lo anterior, y del hecho de que la función que pasamos dentro del useEffect solo puede devolver una cosa, y esa cosa solo puede ser una función de limpieza, si queremos hacer una llamada asíncrona dentro de un useEffect tenemos que hacer esto:

```
useEffect(() => {
   const getMovie = async() => {
     const movie = await axios(
      `https://myapisuperchula/api/v1/movies?id=${id}`,
     );

     setMovie(movie);
   }

   getMovie();

},[id])
```

Y ¿por qué tenemos que hacer esto y no podemos hacer algo como esto?:

```
useEffect(async() => {
     const movie = await axios(
      `https://myapisuperchula/api/v1/movies?id=${id}`,
     );

     setMovie(movie);

},[id])
```

Pues bien, si hacemos esto React se va a quejar y nos va a decir que la función de useEffect debe devolver o una función de limpieza o nada, ¿y acaso estamos devolviendo algo en este ejemplo?

Pues sí, estamos devolviendo una promesa, TODAS las funciones async devuelven una promesa que envuelve el propio valor de retorno, en este caso undefined. Así que useEffect se está encontrando con que el valor devuelto por la función es una promesa, y se espera que sea o una función o nada.

Y por eso **tenemos que declarar la función dentro del useEffect, a parte, y después invocarla**.

Y bien, si has llegado hasta aquí te doy las gracias por dedicar tu tiempo a leer mi post y espero que hayas aprendido algo nuevo o que conozcas a alguien a quien le pueda ser útil lo que he intentado explicar aquí.