



React Hooks



Motivación HOOKS

Los Hooks resuelven una amplia variedad de problemas aparentemente desconectados en React que hemos encontrado durante más de cinco años de escribir y mantener decenas de miles de componentes.

Ya sea que estés aprendiendo React, usándolo diariamente o incluso prefieras una librería diferente con un modelo de componentes similar, es posible que reconozcas algunos de estos problemas.

- Es difícil reutilizar la lógica de estado entre componentes
- Los componentes complejos se vuelven difíciles de entender
- Las clases confunden tanto a las personas como a las máquinas



Reglas de los HOOKS

Los Hooks son funciones JS, que deben cumplir dos requisitos

Sólo pueden ser llamados desde el nivel superior de la función.

- No se pueden llamar desde bucles, condicionales o funciones anidadas

Sólo se pueden llamar desde Funciones de Componente de React, o desde otros Hooks



useState. this.setState()

```
class ExampleUseState extends React.Component {  
  constructor(props) {  
    super(props);  
    this.state = { count: 0};  
  }  
  
  render() {  
    return (  
      <div>  
        <div>{this.count}</div>  
        <button onClick={() => this.setState({count: this.state.count + 1})}>Increment</button>  
      </div>  
    );  
  }  
}
```



useState. this.setState()

Sirve para mantener un estado local al componente entre repintados y ofrecernos el valor más reciente en cada repintado, además de ofrecernos un mecanismo para actualizar dicho estado.

Permite mantener una variable de estado en un componente función como haríamos con this.state en clases.

```
const [state, setState] = useState(initialState)
```

```
const ExampleUseState = () => {  
  const [count, setCount] = React.useState(0);  
  return (  
    <div>  
      <div>{count}</div>  
      <button onClick={() => setCount(count + 1)}>Increment</button>  
    </div>  
  );  
};
```



useState. this.setState()

Su único argumento es el valor inicial que tendrá su estado, a diferencia de las clases no tiene porqué ser un objeto.

Este argumento puede ser una función, que se ejecutará solo la primera vez.

Este argumento puede ser una función, que se ejecutará solo la primera vez

```
const ExampleUseState = () => {  
  const [count, setCount] = React.useState(0);  
  return (  
    <div>  
      <div>{count}</div>  
      <button onClick={() => setCount(count + 1)}>Increment</button>  
    </div>  
  );  
};
```



useState. this.setState()

La llamada a la función updater (setState) con el mismo valor no hará un repintado.

Podemos usar tantos useStates dentro de un componente como queramos/necesitemos.

Devuelve una pareja de valores, el valor de la variable y su función actualizadora.

La función actualizadora re-pintará el componente cada vez que se invoque con un nuevo valor.

```
const ExampleUseState = () => {  
  const [count, setCount] = React.useState(0);  
  return (  
    <div>  
      <div>{count}</div>  
      <button onClick={() => setCount(count + 1)}>Increment</button>  
    </div>  
  );  
};
```



useState. this.setState()

La función actualizadora puede recibir una función, cuyo argumento es el estado actual.

A diferencia de los componentes clases, si el valor del estado es una clase, esta no se combina y la variable de estado tendrá ese nuevo valor durante su repintado, no la combinación.

```
const ExampleUseState1 = () => {  
  const initialValue = () => {  
    console.log('ExampleUseState', 'Instanciando valor inicial', 5);  
    return 5;  
  };  
  const [state, setState] = React.useState(initialValue);  
  console.log('Repintando con cambio de state', state);  
  return (  
    <div>  
      <div>{state}</div>  
      <button onClick={() => setState(state => state + 1)}>Increment</button>  
    </div>  
  );  
};
```




useEffect. componentDidMount, componentDidUpdate, componentWillUnmount

```
class ExampleUseEffect extends React.Component {
  constructor(props) {
    super(props);
    this.state = { count: 0 };
  }

  componentDidMount() {
    document.title = 'Effect count' + this.count;
  }

  componentDidUpdate() {
    document.title = 'Effect count' + this.count;
  }

  render() {
    return (
      <div>
        UseEffect, Ejemplo 1
        <button onClick={() => this.setState({count: this.state.count + 1})}>Change title</button>
      </div>
    );
  }
}
```



useEffect. componentDidMount, componentDidUpdate, componentWillUnmount

Sirve para ejecutar modificaciones sobre el DOM que no pueden ser realizadas durante el re-pintado.

Le indica a React que tiene que hacer después de haber pintado el DOM.

React recordará la función que le pasemos y la ejecutará después del repintado del DOM.

```
const ExampleUseEffect = () => {  
  const [count, setCount] = React.useState(0);  
  React.useEffect(  
    () => {  
      document.title = 'Effect count' + count;  
    }  
  );  
  return (  
    <div>  
      UseEffect, Ejemplo 1  
      <button onClick={() => setCount(count + 1)}>Change title</button>  
    </div>  
  );  
};
```



useEffect.

`componentDidMount`, `componentDidUpdate`,
`componentWillUnmount`

Se ejecutará después del primer repintado y tras cada repintado, garantizando que el DOM ya se haya repintado.

A diferencia de **cdm** y **cdu**, no bloqueará el repintado del DOM, es un proceso asíncrono.

Es un escape hacia el código imperativo.

React ejecuta los efectos justo después de repintar el DOM.

Los efectos tienen acceso a las **props** y **states** del componente.

Los efectos, por defecto, se ejecutan en cada repintado, incluyendo el primero.

Puedes usar tantos efectos como necesites dentro de un componente

useEffect. componentDidMount, componentDidUpdate, componentWillUnmount

El efecto puede devolver, de forma opcional, una función de "limpiado".

La función de limpieza se ejecutará en cada repintado del componente antes de ejecutar el nuevo efecto, imitando el comportamiento de unmount con la diferencia de que se ejecuta en cada repintado

```
class ExampleUseEffect2 extends React.Component {
  constructor(props) {
    super(props);
    this.state = { isOnline: null };
    this.handleStatusChange = this.handleStatusChange.bind(this);
  }

  componentDidMount() {
    ChatAPI.subscribeToFriendStatus(this.props.friend.id, this.handleStatusChange);
  }

  componentWillUnmount() {
    ChatAPI.unsubscribeFromFriendStatus(this.props.friend.id, this.handleStatusChange);
  }

  handleStatusChange(status) {
    this.setState({isOnline: status.isOnline});
  }


  render() {
    if (this.state.isOnline === null) {
      return 'Loading...';
    }
    return this.state.isOnline ? 'Online' : 'Offline';
  }
}
```

```
export const ExampleUseEffect3 = (props) => {
  const [isOnline, setIsOnline] = useState(null);

  useEffect(() => {
    function handleStatusChange(status) {
      setIsOnline(status.isOnline);
    }

    ChatAPI.subscribeToFriendStatus(props.friend.id, handleStatusChange);
    // Specify how to clean up after this effect:
    return function cleanup() {
      ChatAPI.unsubscribeFromFriendStatus(props.friend.id, handleStatusChange);
    };
  }, [props.id]);

  if (isOnline === null) {
    return 'Loading...';
  }
  return isOnline ? 'Online' : 'Offline';
};
```



```
class ExampleUseEffect2 extends React.Component {
  constructor(props) {
    super(props);
    this.state = { isOnline: null };
    this.handleStatusChange = this.handleStatusChange.bind(this);
  }

  componentDidMount() {
    ChatAPI.subscribeToFriendStatus(this.props.friend.id, this.handleStatusChange);
  }

  componentWillUnmount() {
    ChatAPI.unsubscribeFromFriendStatus(this.props.friend.id, this.handleStatusChange);
  }

  handleStatusChange(status) {
    this.setState({isOnline: status.isOnline});
  }

  render() {
    if (this.state.isOnline === null) {
      return 'Loading...';
    }
    return this.state.isOnline ? 'Online' : 'Offline';
  }
}
```



```
export const ExampleUseEffect3 = (props) => {
  const [isOnline, setIsOnline] = useState(null);

  useEffect(() => {
    function handleStatusChange(status) {
      setIsOnline(status.isOnline);
    }

    ChatAPI.subscribeToFriendStatus(props.friend.id, handleStatusChange);
    // Specify how to clean up after this effect:
    return function cleanup() {
      ChatAPI.unsubscribeFromFriendStatus(props.friend.id, handleStatusChange);
    };
  }, [props.id]);

  if (isOnline === null) {
    return 'Loading...';
  }
  return isOnline ? 'Online' : 'Offline';
};
```



useEffect. componentDidMount, componentDidUpdate, componentWillUnmount

Podemos controlar cuándo debe ejecutarse un efecto usando el array de dependencias.

Se especifica como segundo argumento de `useEffect` y es opcional.

Si omitimos el array, el efecto se ejecutará en cada repintado.

El efecto se ejecutará siempre que el array de dependencias cambie con respecto a la ejecución anterior.

Se considera que la dependencia ha cambiado aplicando estrictamente igualdad `'==='`.

Un array de dependencias vacío solo ejecutará el efecto al montar el componente y su limpieza se hará al desmontarse.

Este mecanismo permite replicar los comportamientos de **cdm**, **cdu** y **cwu** en una sola función



useEffect.

```
class ExampleUseEffect3 extends React.Component {
  constructor(props) {
    super(props);
    this.state = { isOnline: null };
    this.handleStatusChange = this.handleStatusChange.bind(this);
  }

  componentDidMount() {
    ChatAPI.subscribeToFriendStatus(this.props.friend.id, this.handleStatusChange);
  }

  componentDidUpdate(prevProps, prevState) {
    if(this.props.friend.id !== prevProps.friend.id) {
      ChatAPI.unsubscribeFromFriendStatus(prevProps.id, this.handleStatusChange);
      ChatAPI.subscribeToFriendStatus(this.props.friend.id, this.handleStatusChange);
    }
  }

  componentWillUnmount() {
    ChatAPI.unsubscribeFromFriendStatus(this.props.friend.id, this.handleStatusChange);
  }

  handleStatusChange(status) {
    this.setState({isOnline: status.isOnline});
  }

  render() {
    if (this.state.isOnline === null) {
      return 'Loading...';
    }
    return this.state.isOnline ? 'Online' : 'Offline';
  }
}
```



```
export const ExampleUseEffect3 = (props) => {
  const [isOnline, setIsOnline] = useState(null);

  useEffect(() => {
    function handleStatusChange(status) {
      setIsOnline(status.isOnline);
    }

    ChatAPI.subscribeToFriendStatus(props.friend.id, handleStatusChange);
    // Specify how to clean up after this effect:
    return function cleanup() {
      ChatAPI.unsubscribeFromFriendStatus(props.friend.id, handleStatusChange);
    };
  }, [props.id]);

  if (isOnline === null) {
    return 'Loading...';
  }

  return isOnline ? 'Online' : 'Offline';
};
```

useContext. static contextType y MyContext.Consumer

Crea un acceso directo entre el nodo **PADRE** y cualquier nodo **DESCENDENTE** que permite enviar valores sin repintar los componentes intermedios.

El componente hijo se repinta cada vez que el valor pasado al padre cambia.

```
// Ejemplo típico tópico es compartir el theme de Material-UI
const defaultValue = { color: 'red' };
const MaterialUIContext = React.createContext(defaultValue);
```

```
const ExampleUseContextClient = () => {
  const theme = React.useContext(MaterialUIContext);
  return (
    <div color={theme.color}>Hello World</div>
  );
};
```

```
const ExampleUseContext = () => {
  const theme = {color: 'blue'};
  return (
    <MaterialUIContext.Provider value={theme}>
      <div>
        <div>
          <div>
            <ExampleUseContextClient />
          </div>
        </div>
      </div>
    </MaterialUIContext.Provider>
  );
};
```



useReducer.

useState es una personalización de **useReducer**.

Es la alternativa preferible para gestionar lógicas compuestas de estados complejos.

Recibe la función reducer y el estado inicial.

Devuelve el estado calculado y la función dispatch.

La función dispatch permite iniciar nuevos cálculos del estado.

Acepta un tercer argumento (fnLazy), es la función de inicializado en diferido del estado
`state = fnLazy(initValue)`



useReducer.

```
const initialState = {count: 0};  
function reducer(state, action) {  
  switch (action.type) {  
    case 'increment':  
      return {count: state.count + 1};  
    case 'decrement':  
      return {count: state.count - 1};  
    default:  
      throw new Error();  
  }  
}
```

```
function ExampleUseReducer() {  
  const [state, dispatch] = React.useReducer(reducer, initialState);  
  return (  
    <div>  
      Count: {state.count}  
      <button onClick={() => dispatch({type: 'decrement'})}>-</button>  
      <button onClick={() => dispatch({type: 'increment'})}>+</button>  
    </div>  
  );  
}
```



useCallback.

Recibe un callback y un arreglo de dependencias.

useCallback devolverá una versión memorizada del callback que solo cambia si una de las dependencias ha cambiado.

```
const memoizedCallback = useCallback(  
  () => {  
    doSomething(a, b);  
  },  
  [a, b],  
);
```



useMemo.

Recibe una función de “cálculo” y un arreglo de dependencias.

useMemo solo volverá a calcular el valor memorizado cuando una de las dependencias haya cambiado.

Esta optimización ayuda a evitar cálculos costosos en cada render.

```
const memoizedValue = useMemo(() => computeExpensiveValue(a, b), [a, b]);
```



useRef.

useRef devuelve un objeto ref mutable cuya propiedad **.current** se inicializa con el argumento pasado (initialValue).

El objeto devuelto se mantendrá persistente durante la vida completa del componente.

```
function TextInputWithFocusButton() {  
  const inputEl = useRef(null);  
  const onClick = () => {  
    // `current` apunta al elemento de entrada de texto montado  
    inputEl.current.focus();  
  };  
  return (  
    <>  
      <input ref={inputEl} type="text" />  
      <button onClick={onClick}>Focus the input</button>  
    </>  
  );  
}
```

useImperativeHandle.

useImperativeHandle personaliza el valor de instancia que se expone a los componentes padres cuando se usa `ref={...}`

```
const FancyInputForwarded = (props, ref) => {
  const inputRef = React.createRef();
  React.useImperativeHandle(ref, () => {
    return {
      setFocus: () => inputRef.current.focus()
    };
  });
  return (<input ref={inputRef} />);
};
const FancyInput = React.forwardRef(FancyInputForwarded);
```

```
const ParentComponent = () => {
  const childRef = React.createRef();

  React.useEffect(() => {
    childRef.current.setFocus();
  }, []);

  return (<FancyInput ref={childRef}/>);
};
```




useLayoutEffect.

La firma es idéntica a **useEffect**, pero se dispara de forma síncrona después de todas las mutaciones del DOM.

Sirve para modificar el DOM y volver a renderizar de forma síncrona.

Las actualizaciones programadas dentro de **useLayoutEffect** se ejecutarán de forma síncrona, antes de que el navegador tenga la oportunidad de repintarse.

useDebugValue.

```
function useFriendStatus(friendID) {  
  const [isOnline, setIsOnline] = useState(null);  
  
  // ...  
  
  // Mostrar una etiqueta en DevTools junto a este Hook  
  // por ejemplo: "FriendStatus: Online"  
  useDebugValue(isOnline ? 'Online' : 'Offline');  
  
  return isOnline;  
}
```



Custom HOOKs. HOC y Render Props

Motivación: Extraer la lógica de los componentes en funciones reusables.

Equivalen a usar HOC y Render Props.

El gran beneficio es compartir lógica entre distintos componentes.

Evitar añadir componentes con sólo lógica al árbol de React. **Clean Code!!!!**

Usamos el prefijo **use-** para nombrar a nuestros HOOKS.

No tienen una firma específica, podemos pasar los parámetros que queramos y devolver lo que queramos.

Cada ejecución de un hook es completamente independiente de otras ejecuciones del mismo hook, es un elemento aislado.

Podemos llamar a tantos hooks desde nuestro hook como queramos/necesitemos.

Custom HOOKs. HOC y Render Props

```
const apiReducer = (currentState, newState) => ({...currentState, ...newState});
const initialReducerState = {isLoading: false, error: false, data: null};
const useBasicAPI = (url) => {
  const [response, setResponse] = React.useReducer(apiReducer, initialReducerState);
  React.useEffect(
    () => {
      const getRequest = async () => {
        const response = await fetch(url);
        const error = !response.ok;
        const data = await response.json();
        setResponse({data, error, isLoading: false});
      };
      setResponse({isLoading: true});
      getRequest();
    },
    [url, setResponse]
  );

  return response;
};
```

Custom HOOKs. HOC y Render Props

```
const GithubUser = ({username}) => {  
  const githubUrl = 'https://api.github.com/users/' + username;  
  const {isLoading, error, data} = useBasicAPI(githubUrl);  
  return (  
    <div>  
      <div>  
        {  
          isLoading && <div>Cargando {username}</div>  
        }  
      </div>  
      <div>  
        {  
          error && <div>Error al cargar {username}</div>  
        }  
      </div>  
      <div>  
        {  
          data && <div><img src={data.avatar_url} alt='avatar from github' /></div>  
        }  
      </div>  
    </div>  
  );  
};
```

Custom HOOKs. HOC y Render Props

Su potencia es el hecho de ser simplemente funciones que reciben y devuelven valores.

Esto hace que sea extremadamente fácil descomponer lógicas más complejas en hooks personalizados.

Nos permite aislar la lógica de pintado de la lógica funcional dentro de un componente.

```
function ChatRecipientPicker() {  
  const [recipientID, setRecipientID] = React.useState(1);  
  const isRecipientOnline = useFriendStatus(recipientID);  
  
  return (  
    <div color={isRecipientOnline ? 'green' : 'red'} />  
    <select  
      value={recipientID}  
      onChange={e => setRecipientID(Number(e.target.value))}  
    >  
      {friendList.map(friend => (  
        <option key={friend.id} value={friend.id}>  
          {friend.name}  
        </option>  
      ))}  
    </select>  
  );  
}
```