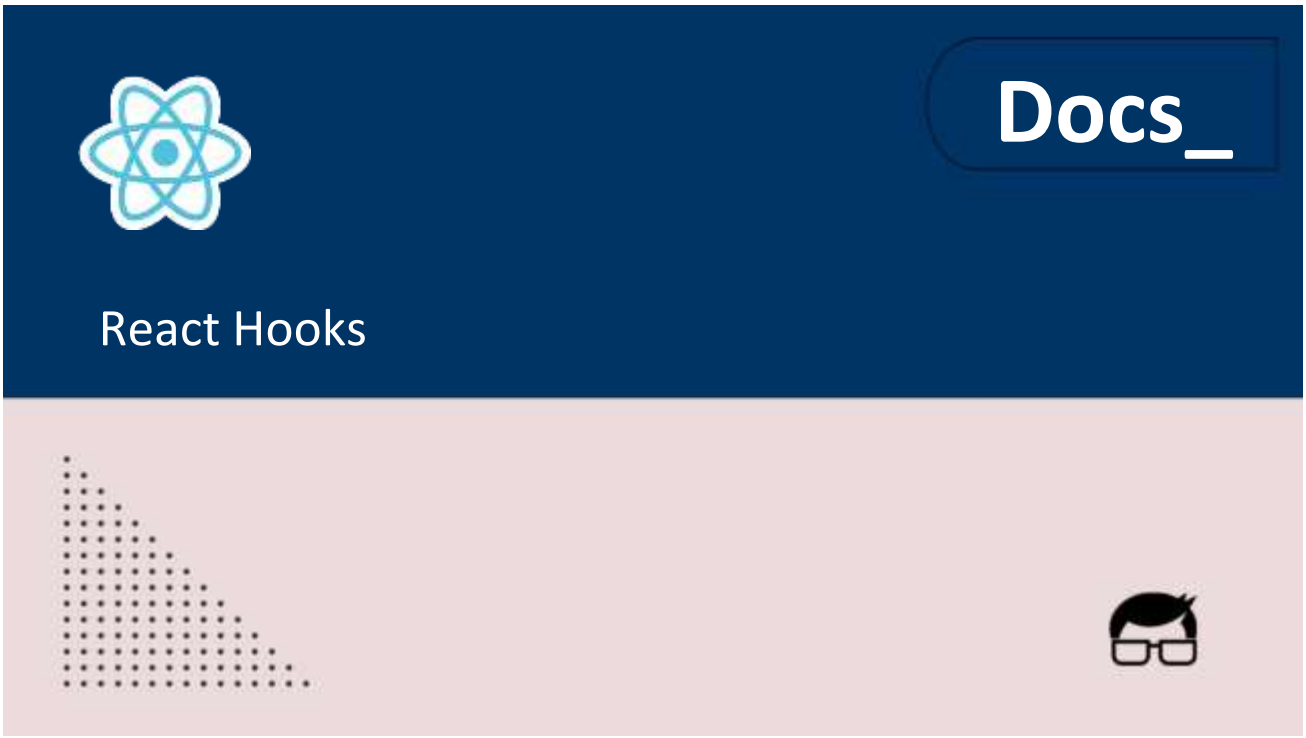

Se agregaron Hooks a React en la versión 16.8.

Los hooks permiten que los componentes de funciones tengan acceso al estado y otras características de React.

Debido a esto, los componentes de clase generalmente ya no son necesarios.

Aunque los Hooks generalmente reemplazan los componentes de las clases, no hay planes para eliminar las clases de React.

¿Qué es un hook?

Supón que tienes un super robot que puede hacer muchas cosas. Pero a veces, quieres que tu robot pueda hacer cosas especiales que no vienen con el robot por defecto.

En React, esos poderes especiales son como los Hooks. Son herramientas que te permiten dar a tus componentes habilidades que no tienen por defecto. Por ejemplo, si quisieras que tu robot pudiera recordar la cantidad de golosinas que ha recogido, podrías usar un superpoder especial (o Hook) llamado useState. Este Hook le da a tu robot una pequeña caja de memoria donde puede almacenar y recordar cosas.

Si quisieras que tu robot realizara una acción cada vez que pasa algo, como bailar cada vez que se encuentra una golosina, podrías usar otro superpoder especial (o Hook) llamado useEffect.

Así que los Hooks son como superpoderes para tus componentes en React. Te permiten hacer cosas realmente geniales que de otro modo no podrías hacer.

Ejemplo:

```
import React, { useState } from "react";
import ReactDOM from "react-dom/client";

function FavoriteColor() {
  const [color, setColor] = useState("red");

  return (
    <>
      <h1>My favorite color is {color}!</h1>
      <button
        type="button"
        onClick={() => setColor("blue")}
      >Blue</button>
      <button
        type="button"
        onClick={() => setColor("red")}
      >Red</button>
      <button
        type="button"
        onClick={() => setColor("pink")}
```

Debes importar Hooks de react.

Aquí estamos usando hook useState para realizar un seguimiento del estado de la aplicación.

El estado generalmente se refiere a los datos de la aplicación o las propiedades que deben rastrearse.

Reglas de Hooks

Hay 3 reglas para los ganchos:

- Los ganchos solo se pueden llamar dentro de los componentes de la función React.
- Los ganchos solo se pueden llamar en el nivel superior de un componente.
- Los ganchos no pueden ser condicionales

Nota: los ganchos no funcionarán en los componentes de la clase React.

Hooks personalizados

Si tiene una lógica con estado que debe reutilizarse en varios componentes, puede crear sus propios Hooks personalizados.

Existen varios hooks incorporados en React, pero los más comunes son:

useState

Este hook te permite añadir y manejar el estado en tus componentes funcionales. Te permite declarar una variable de estado y una función para actualizarla. Puedes pensar en el estado como datos internos que pueden cambiar y afectar cómo se renderiza y se comporta tu componente. Aquí tienes un ejemplo:

```
import React, { useState } from 'react';

function Counter() {
 const [count, setCount] = useState(0);

 const increment = () => {
  setCount(count + 1);
 };

 return (
  <div>
   <p>Contador: {count}</p>
   <button onClick={increment}>Incrementar</button>
  </div>
 );
}
```

En este ejemplo, useState se utiliza para añadir un estado llamado count al componente Counter. count es la variable de estado y setCount es la función que se utiliza para actualizarla. Al hacer clic en el botón, se llama a increment, que incrementa el valor de count en uno.

Imagina que tienes una caja mágica en la que puedes guardar juguetes. Con useState, puedes tener una caja mágica en tu programa de computadora donde puedes guardar un juguete especial y siempre saber cuál es.

Digamos que tienes una pelota y quieres guardarla en tu caja mágica. Con useState, puedes crear una caja llamada cajaPelota y poner tu pelota dentro de ella. Además, siempre puedes saber qué juguete hay dentro de la caja, sin importar cuántas veces la abras.

También puedes hacer algo divertido con useState. Puedes jugar con tu pelota y cambiarla por otro juguete en cualquier momento. Si quieres cambiar tu pelota por un coche, simplemente sacas el coche y lo metes en la caja. ¡Es así de fácil!

useState te permite tener tu propia caja mágica en tu programa de computadora para guardar y cambiar cosas importantes mientras juegas con ellas. Es una forma de tener recuerdos y objetos especiales dentro de tu programa, ¡como si tuvieras una caja mágica real!

Ejemplo

```
import React, { useState } from 'react';

function ToyBox() {
  const [toy, setToy] = useState('Pelota');

  const changeToy = () => {
    if (toy === 'Pelota') {
    setToy('Coche');
    } else {
      setToy('Pelota');
    }
  };

  return (
    <div>
      <p>Juguete actual: {toy}</p>
      <button onClick={changeToy}>Cambiar juguete</button>
    </div>
  );
}
```

useEffect

Este hook te permite realizar efectos secundarios en tus componentes funcionales. Los efectos secundarios son acciones que no están directamente relacionadas con la renderización del componente, como hacer una petición a una API, suscribirse a eventos o modificar el DOM. Puedes pensar en useEffect como una manera de decirle a React que ejecute cierto código después de que el componente se haya renderizado o cada vez que ciertas variables cambien. Aquí tienes un ejemplo:

```
import React, { useState, useEffect } from 'react';

function DataFetcher() {
 const [data, setData] = useState([]);

 useEffect(() => {
  fetchData();
 }, []);

 async function fetchData() {
  const response = await fetch('https://api.example.com/data');
  const data = await response.json();
  setData(data);
 }

 return (
  <div>
   {data.map((item) => (
    <p key={item.id}>{item.name}</p>
   ))}
  </div>
 );
}
```

En este ejemplo, useEffect se utiliza para realizar una petición a una API cuando el componente se monta por primera vez (usando un arreglo vacío como segundo argumento). Una vez que los datos se obtienen, se actualiza el estado con setData. La función fetchData se declara dentro del componente y se utiliza para realizar la petición.

Estos son solo dos ejemplos de los hooks más comunes en React. También existen otros hooks, como useContext para acceder al contexto de la aplicación, useRef para mantener referencias a elementos del DOM, useReducer para gestionar el estado más complejo y useCallback para memorizar funciones, entre otros. Los hooks son una forma poderosa de añadir funcionalidad a tus componentes funcionales de manera más sencilla y fácil de entender.

Imagina que tienes una mascota especial, como un perrito, y necesitas cuidarlo y darle cosas. El efecto es como una lista de cosas que debes hacer para asegurarte de que tu perrito esté feliz y saludable.

Con useEffect, puedes hacer una lista de cosas que debes hacer cuando algo cambie. Por ejemplo, si tu perrito tiene hambre, le das comida. Si está aburrido, le das un juguete para que juegue. Si está cansado, lo llevas a dormir.

El useEffect es como una "receta" que te dice qué hacer en diferentes situaciones. Si algo cambia, como el hambre de tu perrito, puedes revisar tu lista de efectos y hacer lo que se necesita para asegurarte de que tu perrito esté contento.

Ejemplo

```
import React, { useState, useEffect } from 'react';
function DogCare() {
 const [hungry, setHungry] = useState(true);

 useEffect(() => {
  if (hungry) {
   console.log("Le diste comida a tu perrito ");
   setHungry(false);
  }
 }, [hungry]);

 const feedDog = () => {
  setHungry(true);
 };

 return (
  <div>
   <p>¿Tu perrito tiene hambre? {hungry ? 'Sí' : 'No'}</p>
   <button onClick={feedDog}>Darle comida</button>
  </div>
 );
}
```

En este ejemplo, el componente DogCare tiene una variable de estado llamada hungry que representa si tu perrito tiene hambre o no. Cuando haces clic en el botón "Darle comida", se actualiza el estado a true, lo que activa el efecto definido en useEffect.

El efecto revisa si tu perrito tiene hambre y, si es así, muestra un mensaje en la consola indicando que le diste comida y actualiza el estado hungry a false.

Así que, imagina que tienes esta lista de tareas especial llamada useEffect para cuidar a tu perrito. Cuando tu perrito tiene hambre y necesita comida, revisas la lista de efectos y le das de comer. De esta manera, te aseguras de que tu perrito esté siempre bien alimentado y feliz. ¡Es como tener una lista mágica de cuidado para tu perrito en tu programa de computadora!

useContext

Este hook te permite acceder al contexto de la aplicación creado con el componente Context.Provider. El contexto te permite compartir datos entre componentes sin tener que pasar props manualmente a través de múltiples niveles de componentes. Puedes utilizar useContext para consumir los valores proporcionados por el contexto. Aquí tienes un ejemplo:

```
import React, { useContext } from 'react';

const ThemeContext = React.createContext('light');

function App() {
  return (
    <ThemeContext.Provider value="dark">
      <ChildComponent />
    </ThemeContext.Provider>
  );
}

function ChildComponent() {
  const theme = useContext(ThemeContext);

  return <p>Tema actual: {theme}</p>;
}
```

En este ejemplo, ChildComponent accede al valor proporcionado por el contexto ThemeContext utilizando useContext. En este caso, el valor del tema es 'dark'.

Imagina que estás en una fiesta de disfraces y cada persona tiene un superpoder especial. useContext es como una gran fiesta donde todas las personas pueden compartir su superpoder con los demás.

Cada persona tiene una etiqueta en su disfraz que dice cuál es su superpoder. Por ejemplo, alguien puede tener el superpoder de volar, otro puede tener el superpoder de ser muy fuerte y otro puede tener el superpoder de ser invisible.

Con useContext, puedes ir a cualquier persona en la fiesta y preguntarle por su superpoder. Ellos te dirán cuál es y podrás usarlo para hacer cosas increíbles. Por ejemplo, si alguien tiene el superpoder de volar, ¡puedes volar con ellos por el cielo!

useContext te permite preguntar a las personas en la fiesta cuál es su superpoder y usarlo para hacer cosas emocionantes juntos. Es como un gran intercambio de superpoderes donde todos pueden compartir y disfrutar de las habilidades especiales de los demás. ¡Es muy divertido!

Ejemplo

```
import React, { useContext } from 'react';

const SuperpowerContext = React.createContext('');

function Person() {
  const superpower = useContext(SuperpowerContext);

  return <p>Mi superpoder es: {superpower}</p>;
}

function SuperpowerParty() {
  return (
    <SuperpowerContext.Provider value="Volar">
      <div>
        <h2>Fiesta de Superpoderes</h2>
        <Person />
      </div>
    </SuperpowerContext.Provider>
  );
}

export default SuperpowerParty;
```

En este ejemplo, tenemos un componente llamado Person que utiliza useContext para acceder al superpoder proporcionado por el contexto SuperpowerContext. En el componente SuperpowerParty, envolvemos el componente Person con el SuperpowerContext.Provider y establecemos el valor del superpoder como "Volar".

Cuando renderizamos el componente SuperpowerParty, el componente Person accede al superpoder proporcionado por el contexto y muestra "Mi superpoder es: Volar".

Así que, imagina que estás en una fiesta de disfraces donde cada persona tiene un superpoder especial. Con useContext, puedes preguntar a cada persona cuál es su superpoder y descubrir cosas asombrosas. Es como un intercambio mágico de superpoderes donde todos pueden compartir y disfrutar de las habilidades especiales de los demás en la fiesta. ¡Es una fiesta muy divertida y llena de superpoderes!

useRef

Este hook te permite mantener una referencia mutable a un elemento del DOM o a un valor en componentes funcionales. Puedes utilizarlo para acceder directamente a elementos del DOM o para mantener valores que no causen la re-renderización del componente. Aquí tienes un ejemplo:

```
import React, { useRef } from 'react';

function TextInput() {
  const inputRef = useRef(null);

  const handleClick = () => {
    inputRef.current.focus();
  };

  return (
    <div>
      <input type="text" ref={inputRef} />
      <button onClick={handleClick}>Enfocar</button>
    </div>
  );
}
```

En este ejemplo, inputRef se utiliza para mantener una referencia al elemento input y luego se utiliza en handleClick para enfocar el campo de texto cuando se hace clic en el botón.

Imagina que tienes una varita mágica especial que puedes usar para recordar algo importante. useRef es como esa varita mágica que te ayuda a recordar cosas en tu programa de computadora.

Cuando tienes algo importante que necesitas recordar, puedes usar useRef para guardarlo en un lugar especial. Es como escribir una nota en un papel y ponerlo en tu bolsillo mágico.

Después, cuando necesitas recordar lo que guardaste, solo tienes que sacarlo de tu bolsillo mágico y leerlo. La varita mágica te ayuda a mantener ese recuerdo y puedes acceder a él en cualquier momento.

useRef te permite tener tu propia varita mágica en tu programa de computadora para recordar cosas importantes. Es una forma de guardar y acceder a información especial cuando la necesitas. ¡Es como tener una varita mágica para recordar cosas en tu programa!

Ejemplo

```
import React, { useRef } from 'react';

function MagicTrick() {
  const magicNote = useRef('');

  const performTrick = () => {
    magicNote.current = '¡Aparece un conejo de la chistera!';
    console.log(magicNote.current);
  };

  return (
    <div>
      <button onClick={performTrick}>¡Realizar el truco de magia!</button>
    </div>
  );
}
```

En este ejemplo, el componente MagicTrick tiene una vara mágica llamada magicNote creada con useRef. Cuando haces clic en el botón "¡Realizar el truco de magia!", se asigna el mensaje "¡Aparece un conejo de la chistera!" a magicNote.current y se muestra en la consola.

Imagina que tienes esta varita mágica llamada useRef en tu programa de computadora. Puedes usarla para guardar algo importante, como un mensaje mágico. Cuando quieras recordar ese mensaje, simplemente tomas la varita mágica y lees lo que hay dentro. ¡Es como tener una varita mágica para recordar cosas especiales en tu programa de computadora!

useReducer

Este hook te permite gestionar el estado más complejo en tus componentes funcionales utilizando un patrón similar al de Redux. Puedes utilizarlo para mantener y actualizar el estado de forma más estructurada y predecible. Aquí tienes un ejemplo:

```
import React, { useReducer } from 'react';

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 Counter() {
  const [state, dispatch] = useReducer(reducer, initialState);

  const increment = () => {
    dispatch({ type: 'increment' });
  };

  const decrement = () => {
    dispatch({ type: 'decrement' });
  };

  return (
    <div>
      <p>Contador: {state.count}</p>
      <button onClick={increment}>Incrementar</button>
      <button onClick={decrement}>Decrementar</button>
    </div>
  );
}
```

En este ejemplo, useReducer se utiliza para gestionar el estado del contador. reducer es una función que toma el estado actual y una acción, y devuelve el nuevo estado basado en la acción. dispatch se utiliza para enviar acciones al reducer y actualizar el estado.

Recuerda que estos son solo algunos ejemplos de los hooks disponibles en React. También existen otros hooks como useCallback para memorizar funciones, useMemo para memorizar valores computados, useEffect con

dependencias para ejecutar código cuando ciertas variables cambien, y muchos más. Cada hook tiene su propio propósito y te permite agregar funcionalidad adicional a tus componentes funcionales de manera sencilla y declarativa.

useCallback

Este hook te permite memorizar una función para evitar que se vuelva a crear en cada renderizado del componente. Puedes utilizarlo para optimizar el rendimiento de tus componentes y evitar la creación innecesaria de funciones en cada render. Aquí tienes un ejemplo:

```
import React, { useState, useCallback } from 'react';

function Counter() {
 const [count, setCount] = useState(0);

 const increment = useCallback(() => {
  setCount(count + 1);
 }, [count]);

 return (
  <div>
   <p>Contador: {count}</p>
   <button onClick={increment}>Incrementar</button>
  </div>
 );
}
```

En este ejemplo, increment se memoiza utilizando useCallback y solo se volverá a crear cuando count cambie. Esto evita que se cree una nueva función en cada renderizado del componente.

useMemo: Este hook te permite memorizar un valor computado para evitar recálculos innecesarios en cada renderizado del componente. Puedes utilizarlo para optimizar el rendimiento cuando tengas cálculos costosos o dependencias complejas. Aquí tienes un ejemplo:

```
import React, { useState, useMemo } from 'react';

function ExpensiveCalculation() {
 const [count, setCount] = useState(0);

 const expensiveValue = useMemo(() => {
  // Cálculo costoso
  let result = 0;
  for (let i = 0; i < count; i++) {
   result += i;
  }
  return result;
 }, [count]);

 return (
  <div>
   <p>Valor calculado: {expensiveValue}</p>
   <button onClick={() => setCount(count + 1)}>Incrementar</button>
  </div>
 );
}
```

En este ejemplo, expensiveValue se calcula utilizando useMemo y solo se volverá a calcular cuando countcambie. Esto evita el cálculo repetido del valor costoso en cada renderizado.

Estos son solo dos ejemplos adicionales de los hooks en React. Otros hooks útiles incluyen useContext para acceder al contexto de la aplicación, useLayoutEffect para realizar efectos secundarios sincronizados con el renderizado, useImperativeHandle para personalizar la instancia expuesta de un componente y muchos más. Loshooks te brindan una forma flexible y poderosa de agregar funcionalidad a tus componentes funcionales en React.