WMNWA 2210

WEB APP DEVELOPMENT

React 11 | (Other elements of) Top-Level API

CREATIVE MEDIA EDUCATION

ΠΕΡΙΕΧΟΜΕΝΑ

Περιεχόμενα

- Components
  - React.memo
- Suspense
  - React.lazy
  - React.Suspense
- Hooks
  - useCallback
  - useMemo
  - useRef

TOP-LEVEL API

Top-Level API

Introduction

__React__ is the entry point to the __React__ library. If you load __React__ from a __<script>__ tag, these top-level APIs are available on the __React__ global.

If you use ES6 with __npm__, you can write __import React from 'react'__. If you use ES5 with __npm__, you can write __var React = require('react')__.

COMPONENTS

Components

React.memo

```
const MyComponent = React.memo(function MyComponent(props) {
  /* render using props */
});
```

React.memo *is a higher order component.*

If your component renders the same result given the same props, you can wrap it in a call to ***React.memo*** *for a performance boost in some cases by memoizing the result. This means that* ***React*** *will skip rendering the component, and reuse the last rendered result.*

React.memo *only checks for* ***prop*** *changes. If your function component wrapped in React.memo has a* ***useState****,* ***useReducer*** *or* ***useContext*** *Hook in its implementation, it will still rerender when state or context change.*

Components

React.memo (cont.)

*By default **React.memo** will only shallowly compare complex objects in the props object. If you want control over the comparison, you can also provide a custom comparison function as the second argument.*

```
function MyComponent(props) { /* render using props */ }

function areEqual(prevProps, nextProps) {
  /*
  return true if passing nextProps to render would return
  the same result as passing prevProps to render,
  otherwise return false
  */
}

export default React.memo(MyComponent, areEqual);
```

This method only exists as a performance optimization. Do not rely on it to "prevent" a render, as this can lead to bugs.

Suspense

React.lazy

React.lazy() *lets you define a component that is loaded dynamically. This helps reduce the bundle size to delay loading components that aren't used during the initial render.*

```
const SomeComponent = React.lazy(() => import('./SomeComponent'));
```

Note that rendering lazy components requires that there's a ***<React.Suspense>*** *component higher in the rendering tree. This is how you specify a loading indicator.*

Note

Using ***React.lazy*** *with dynamic import requires Promises to be available in the* ***JS*** *environment. This requires a polyfill on IE11 and below.*

Suspense

React.Suspense

React.Suspense *lets you specify the loading indicator in case some components in the tree below it are not yet ready to render. Today, lazy loading components is the only use case supported by* ***<React.Suspense>****:*

```
// This component is loaded dynamically
const OtherComponent = React.lazy(() => import('./OtherComponent'));

function MyComponent() {
  return (
    // Displays <Spinner> until OtherComponent loads
    <React.Suspense fallback={<Spinner />}>
      <div>
        <OtherComponent />
      </div>
    </React.Suspense>
  );
}
```

Hooks

useCallback

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

Returns a memoized callback.

Pass an inline callback and an array of dependencies. ***useCallback*** *will return a memoized version of the callback that only changes if one of the dependencies has changed.*

This is useful when passing callbacks to optimized child components that rely on reference equality to prevent unnecessary renders (e.g. ***shouldComponentUpdate****).*

useCallback(fn, deps) *is equivalent to* ***useMemo(() => fn, deps)****.*

Hooks

useMemo

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

Returns a memoized value.

Pass a "create" function and an array of dependencies. ***useMemo*** *will only recompute the memoized value when one of the dependencies has changed. This optimization helps to avoid expensive calculations on every render.*

Remember that the function passed to ***useMemo*** *runs during rendering. Don't do anything there that you wouldn't normally do while rendering. For example, side effects belong in* ***useEffect****, not* ***useMemo****.*

If no array is provided, a new value will be computed on every render.

Hooks

useMemo

*You may rely on **useMemo** as a performance optimization, not as a semantic guarantee.*

*In the future, **React** may choose to "forget" some previously memoized values and recalculate them on next render, e.g. to free memory for offscreen components.*

*Write your code so that it still works without **useMemo** — and then add it to optimize performance.*

Hooks

useRef

```
const refContainer = useRef(initialValue);
```

__useRef__ returns a mutable ref object whose __.current__ property is initialized to the passed argument (__initialValue__).

The returned object will persist for the full lifetime of the component.

Essentially, __useRef__ is like a "box" that can hold a mutable value in its __.current__ property.

CREATIVE MEDIA EDUCATION

Hooks

useRef (cont.)

A common use case is to access a child imperatively:

```
function TextInputWithFocusButton() {
  const inputEl = useRef(null);

  const onButtonClick = () => {
    // `current` points to the mounted text input element
    inputEl.current.focus();
  };

  return (
    <>
      <input ref={inputEl} type="text" />
      <button onClick={onButtonClick}>Focus the input</button>
    </>
  );
}
```

Hooks

useRef (cont.)

*You might be familiar with refs primarily as a way to access the DOM. If you pass a ref object to **React** with **<div ref={myRef} />**, **React** will set its .current property to the corresponding DOM node whenever that node changes.*

*However, **useRef()** is useful for more than the **ref** attribute. It's handy for keeping any mutable value around similar to how you'd use instance fields in classes.*

Hooks

useRef (cont.)

*This works because **useRef()** creates a plain **JavaScript** object. The only difference between **useRef()** and creating a **{current: ...}** object yourself is that **useRef** will give you the same ref object on every render.*

*Keep in mind that **useRef** doesn't notify you when its content changes. Mutating the **.current** property doesn't cause a re-render. If you want to run some code when React attaches or detaches a ref to a DOM node, you may want to use a callback ref instead.*

Χρήσιμα links

React Top-Level API – React
https://reactjs.org/docs/react-api.html

Memoization - Wikipedia
https://en.wikipedia.org/wiki/Memoization

Lazy loading (and preloading) components in React …
https://medium.com/hackernoon/lazy-loading-and-pre…

Lists and Keys – React
https://reactjs.org/docs/lists-and-keys.html

Extra info

Reconciliation – React
https://reactjs.org/docs/reconciliation.html

Reference Identity in Javascript — React/Performanc…
https://medium.com/@jvcjunior/reference-identity-in-ja…

A React Rendering Misconception
https://thoughtbot.com/blog/react-rendering-misconc…

How to Memoize with React.useMemo()
https://dmitripavlutin.com/react-usememo-hook/

One simple trick to optimize React re-renders
https://kentcdodds.com/blog/optimize-react-re-renders

Before You memo() — Overreacted
https://overreacted.io/before-you-memo/

THANK YOU!