React useRef() Hook Explained in 3 Steps

Updated April 2, 2023
reactuserefhookdomelement
60 Comments

In this post you'll learn how to use `React.useRef()` hook to create persisted mutable values (also known as references or refs), as well as access DOM elements.

Before I go on, let me recommend something to you.

The path to becoming proficient in React isn't easy... but fortunately with a good teacher you can shortcut.

Take the course "React Front To Back Course" by Brad Traversy to improve your React skills in a fun and practical way. Use the coupon code DMITRI and get your 20% discount!

Table of Contents

1. Mutable values

`useRef(initialValue)` is a built-in React hook that accepts one argument as the initial value and returns a *reference* (aka *ref*). A reference is an object having a special property `current`.

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

function MyComponent() {
  const initialValue = 0;
  const reference = useRef(initialValue);

  const someHandler = () => {
    // Access reference value:
    const value = reference.current;

    // Update reference value:
    reference.current = newValue;
  };

  // ...
}
```

`reference.current` accesses the reference value, and `reference.current = newValue` updates the reference value. Pretty simple.

There are 2 rules to remember about references:

1. The value of the reference is *persisted* (remains unchanged) between component re-renderings;
2. Updating a reference *doesn't trigger a component re-rendering*.

Now, let's see how to use `useRef()` in practice.

1.1 Use case: logging button clicks

The component `LogButtonClicks` uses a reference to store the number of clicks on a button:

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

function LogButtonClicks() {
  const countRef = useRef(0);

  const handle = () => {
    countRef.current++;
    console.log(`Clicked ${countRef.current} times`);
  };

  console.log('I rendered!');

  return <button onClick={handle}>Click me</button>;
}
```

Open the demo.

`const countRef = useRef(0)` creates a reference `countRef` initialized with `0`.

When the button is clicked, `handle` callback is invoked and the reference value is incremented: `countRef.current++`. Then the reference value is logged to the console.

Updating the reference value `countRef.current++` doesn't trigger component re-rendering. This is demonstrated by the fact that `'I rendered!'` is logged to the console just once, at initial rendering, and no re-rendering happens when the reference is updated.

Now a reasonable question: what's the main difference between reference and state?

Reference and state diff

Let's reuse the component `LogButtonClicks` from the previous section, but this time use `useState()` hook to count the number of button clicks:

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

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

  const handle = () => {
    const updatedCount = count + 1;
    console.log(`Clicked ${updatedCount} times`);
    setCount(updatedCount);
  };

  console.log('I rendered!');

  return <button onClick={handle}>Click me</button>;
}
```

Open the demo.

Open the demo and click the button. Each time you click, you will see in the console the message `'I rendered!'` — meaning that each time the state is updated, the component re-renders.

So, the 2 main differences between reference and state:

1. Updating a reference doesn't trigger re-rendering, while updating the state makes the component re-render;
2. The reference update is synchronous (the updated reference value is available right away), while the state update is asynchronous (the state variable is updated after re-rendering).

From a higher point of view, references store infrastructure data of side-effects, while the state stores information that is directly rendered on the screen.

1.2 Use case: implementing a stopwatch

You can store inside a reference infrastructure data of side effects: timer ids, socket ids, etc.

The component `Stopwatch` uses `setInterval(callback, time)` timer function to increase each second the counter of a stopwatch. The timer id is stored in a reference `timerIdRef`:

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

function Stopwatch() {
  const timerIdRef = useRef(0);
  const [count, setCount] = useState(0);

  const startHandler = () => {
    if (timerIdRef.current) { return; }
    timerIdRef.current = setInterval(() => setCount(c => c+1),
1000);
  };

  const stopHandler = () => {
    clearInterval(timerIdRef.current);
    timerIdRef.current = 0;
  };

  useEffect(() => {
    return () => clearInterval(timerIdRef.current);
  }, []);

  return (
    <div>
      <div>Timer: {count}s</div>
      <div>
        <button onClick={startHandler}>Start</button>
        <button onClick={stopHandler}>Stop</button>
      </div>
    </div>
  );
}
```

Open the demo.

`startHandler()` function, which is invoked when the *Start* button is clicked, starts the timer and saves the timer id in the reference `timerIdRef.current = setInterval(...)`.

To stop the stopwatch user clicks *Stop* button. The *Stop* button handler `stopHandler` accesses the timer id from the reference and stops the timer `clearInterval(timerIdRef.current)`.

Additionally, if the component unmounts while the stopwatch is active, the cleanup function of `useEffect()` is going to stop the timer too.

In the stopwatch example, the reference was used to store the infrastructure data — the active timer id.

Side challenge: can you improve the stopwatch by adding a Reset button? Share your solution in a comment below!

2. Accessing DOM elements

Another useful application of the `useRef()` hook is to access DOM elements directly. This is performed in 3 steps:

1. Define the reference to access the element `const elementRef = useRef();`
2. Assign the reference to `ref` attribute of the element: `<div ref={elementRef}></div>`;
3. After mounting, `elementRef.current` points to the DOM element.

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

function AccessingElement() {
  const elementRef = useRef();

   useEffect(() => {
    const divElement = elementRef.current;
    console.log(divElement); // logs <div>I'm an element</div>
  }, []);

  return (
    <div ref={elementRef}>
      I'm an element
    </div>
  );
}
```

Open the demo.

2.1 Use case: focusing on an input

You would need to access DOM elements, for example, to focus on the input field when the component mounts.

To make it work you'll need to create a reference to the input, assign the reference to `ref` attribute of the tag, and after mounting call the special method `element.focus()` on the element.

Here's a possible implementation of the `<InputFocus>` component:

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

function InputFocus() {
  const inputRef = useRef();

  useEffect(() => {
    inputRef.current.focus();
  }, []);

  return (
    <input
      ref={inputRef}
      type="text"
    />
  );
}
```

Open the demo.

`const inputRef = useRef()` creates a reference to hold the input element.

`inputRef` is then assigned to `ref` attribute of the input field: `<input ref={inputRef} type="text" />`.

React then, after mounting, sets `inputRef.current` to be the input element. Inside the callback of `useEffect()` you can set the focus to the input programmatically: `inputRef.current.focus()`.

Tip: if you want to learn more about `useEffect()`, I highly recommend checking my post A Simple Explanation of React.useEffect().

Ref is null on initial rendering

During initial rendering, the reference supposed to hold the DOM element is empty:

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

function InputFocus() {
  const inputRef = useRef();

  useEffect(() => {
    // Logs `HTMLInputElement`
    console.log(inputRef.current);

    inputRef.current.focus();
  }, []);

  // Logs `undefined` during initial rendering
  console.log(inputRef.current);

  return <input ref={inputRef} type="text" />;
}
```

Open the demo.

During initial rendering React still determines the output of the component, so there's no DOM structure created yet. That's why `inputRef.current` evaluates to `undefined` during initial rendering.

`useEffect(callback, [])` hook invokes the callback right after mounting when the input element has already been created in DOM.

`callback` function of the `useEffect(callback, [])` is the right place to access `inputRef.current` because it is guaranteed that the DOM is constructed.

3. Updating references restriction

The function scope of the functional component should either calculate the output or invoke hooks.

That's why updating a reference (as well as updating state) shouldn't be performed inside the immediate scope of the component's function.

The reference must be updated either inside a `useEffect()` callback or inside handlers (event handlers, timer handlers, etc).

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

function MyComponent({ prop }) {
  const myRef = useRef(0);

  useEffect(() => {
    myRef.current++; // Good!

    setTimeout(() => {
      myRef.current++; // Good!
    }, 1000);
  }, []);

  const handler = () => {
    myRef.current++; // Good!
  };

  myRef.current++; // Bad!

  if (prop) {
    myRef.current++; // Bad!
  }

  return <button onClick={handler}>My button</button>;
}
```

4. Summary

`useRef()` hook creates references.

Calling `const reference = useRef(initialValue)` with the initial value returns a special object named reference. The reference object has a property `current`: you can use this property to read the reference value `reference.current`, or update `reference.current = newValue`.

Between the component re-renderings, the value of the reference is persisted.

Updating a reference, contrary to updating state, doesn't trigger component re-rendering.

References can also access DOM elements. Assign the reference to `ref` attribute of the element you'd like to access: `<div ref={reference}>Element</div>` — and the element is available at `reference.current` after the component mounting.

Want to improve your React knowledge further? Follow A Simple Explanation of React.useEffect().

Challenge: write a custom hook `useEffectSkipFirstRender()` *that works as* `useEffect()`*, only that it doesn't invoke the callback after initial rendering (Hint: you need to use* `useRef()`*). Share your solution in a comment below!*