Brad Westfall
bradwestfall

Jan 31st, 2020
Public Workshops

useEffect(fn, []) is not the new componentDidMount()

Tags: React hooks useEffect

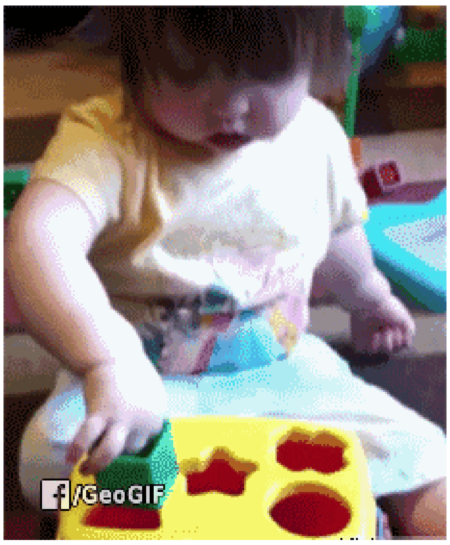
They're almost the same. But there's actually just enough of a difference to possibly get you into trouble -- especially if you're refactoring from classes.

• • •

We often times do some setup when the component first mounts like a network call or a subscription. We have taught ourselves to think in terms of "moments in time" with things like `componentDidMount()`, `componentDidUpdate()`, and `componentWillUnmount()`. It's natural to take that prior knowledge of React and to seek 1:1 equivalents in hooks. I did it myself and I think everyone does at first. Often times I'll hear in my workshops...

> *"What is the hooks equivalent to [some lifecycle method]?"*

The quick answer is that hooks are a paradigm shift from thinking in terms of "lifecycles and time" to thinking in terms of "state and synchronization with DOM". Trying to take the old paradigm and apply it to hooks just doesn't work out very well and can hold you back.



But this answer is filled with jargon and without a deeper more tangible explanation, they are quickly forgotten.

When developers start learning hooks having come from classes, they tend to think "I need to run some code once when we mount, like how `componentDidMount()` works. Ah, I see that `useEffect` with an empty dependency array does just that. Okay I know how all this works..."

This way of thinking gets us into trouble in a few ways:

1. They're actually mechanically different, so you might not get what you expect if consider them the same (which we talk about below).
2. Thinking in terms of time, like "call my side effect once on mount" can hinder your learning of hooks.
3. Refactoring from classes to hooks will not mean you simply replace your componentDidMount's with `useEffect(fn, [])`.

They run at different times

First, let's talk about the timing of each. `componentDidMount` runs after the component mounts. As the docs say, if you set state immediately (synchronously) then React knows how to trigger an extra render and use the second render's response as the initial UI so the user doesn't see a flicker. Imagine you need to read the width of a DOM element with `componentDidMount` and want to update state to reflect something about the width. Imagine this sequence of events:

1. Component renders for the first time.
2. The return value of `render()` is used to mount new DOM.
3. `componentDidMount` fires and sets state immediately (not in an async callback)
4. The state change means `render()` is called again and returns new JSX which replaces the previous render.
5. The browser only shows the second render to avoid flicker.

See Example

It's nice that this is how it works for when we need it. But most the time we don't need this pre-optimized approach because we're doing asynchronous network calls and then setting state after the paint to the screen.

`componentDidMount` and `useEffect` run after the mount. However `useEffect` runs **after the paint** has been committed to the screen as opposed to before. This means you would get a flicker if you needed to read from the DOM, then synchronously set state to make new UI.

How do get the old behavior back when we need it?

`useLayoutEffect` was designed to have the same timing as `componentDidMount`. So `useLayoutEffect(fn, [])` is a much closer match to `componentDidMount()` than `useEffect(fn, [])` -- at least from a timing standpoint.

Does that mean we should be using `useLayoutEffect` instead?

Probably not.

If you **do** want to avoid that flicker by synchronously setting state, then use `useLayoutEffect`. But since those are rare cases, you'll want to use `useEffect` most of the time.

"Capturing" Props and State

Asynchronous code is inevitable in React apps. When our async code resolves, the values of our props and state might be a little confusing though. Let's imagine we had some async code that when resolved needs to know what the state is for count:

```
class App extends React.Component {
  state = {
    count: 0
  };

  componentDidMount() {
    longResolve().then(() => {
      alert(this.state.count);
    });
  }

  render() {
    return (
      <div>
        <button
          onClick={() => {
            this.setState(state => ({ count: state.count + 1 }));
          }}
        >
          Count: {this.state.count}
        </button>
      </div>
    );
  }
}
```

You can play with the code here.

When the page loads, you have three seconds to click the button a few times before the `longResolve` promise resolves. Then an alert will tell you what the current value is for `count`. With this class component, if you click five times you'll get `5` in the alert.

Now let's refactor to hooks. Here's what we might come up with:

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

  useEffect(() => {
    longResolve().then(() => {
      alert(count);
    });
  }, []);

  return (
    <div>
      <button
        onClick={() => {
          setCount(count + 1);
        }}
      >
        Count: {count}
      </button>
    </div>
  );
}
```

See it in action.

It might seem like a good "class-to-hooks" refactor, but the behavior is different. In this example, no matter how many times you click the button before the resolve, you'll get an alert of `0`.

The difference is that `useEffect` "captures" the value of count when the effect gets created. When we give the effect function to `useEffect`, it lingers around in memory where it only knows that `count` was `0` when it was created (due to closure). It doesn't matter how much time goes by and what `count` changes to in that time, the nature of closure is to keep knowledge of what the closed-over values were when the closure was created - called "capturing".

With the class-based code, `componentDidMount` doesn't have closure over state so it just reads whatever the current value is.

If it help to think of it this way, imagine JavaScript stores that callback in memory like this:

```
// Hey memory, we need you to store a function. And
// when this function was created, there were some values like
// `count: 0` so remember that too.
() => {
  // Even though this isn't literally how it works, from a mental
  // model standpoint it's as if count is just a variable hard-coded
  // to 0
  const count = 0
  longResolve().then(() => {
    alert(count);
  });
}
```

• • •

Here's another example of how `useEffect` captures.

In Dan Abramov's "A Complete Guide to useEffect", the examples are similar to these and show how `setInterval` behaves the way you might expect with classes but not with hooks:

```
// The class version:
class App extends React.Component {
  state = { count: 0 }

  componentDidMount() {
    setInterval(() => {
      this.setState({ count: this.state.count + 1 })
    }, 1000);
  }
```

```
    render() {
      return <div>{this.state.count}</div>
    }
  }

  // What we think is the same logic but rewritten as hooks:
  function App() {
    const [count, setCount] = useState(0)

    useEffect(() => {
      const id = setInterval(() => {
        setCount(count + 1)
      }, 1000);
      return () => clearInterval(id)
    }, [])

    return <div>{count}</div>
  }
```

In the class-based code, the counter increments every second. In the hooks-based component it increments from `0` to `1` and then stops. But it's interesting to learn that the interval doesn't *actually* stop. The cause for the behavior is that this `useEffect` callback "captured" what it knows to be `count` when it's created. That callback always thinks `count` is `0` so therefore we continuously set the count to be `0 + 1` forever.

If this seems like this is problematic, give me a chance to explain that it's actually a good thing and might even help you avoid bugs 🤔.

Keep in mind that this isn't a lesson on how to do `setInterval` with hooks, it's more about how to adjust your mental model from classes to hooks and how `useEffect` "captures" values.

To start, we really need to understand why it's called the "dependency array" in the first place.

If your effect "depends" on it, it needs to be listed in the array.

Our code depends on the count variable from state. We should have done this all along.

```
    useEffect(() => {
      const id = setInterval(() => {
        setCount(count + 1)
      }, 1000)
      return () => clearInterval(id)
    }, [count])
```

Now the code works the way we wanted it to because we're saying we want the `useEffect` callback to run again when count changes. When this happens, the previous callback in memory (that remembered `count: 0`) will have its cleanup function called and therefore its `setInterval` will be destroyed. Then React will make a whole new callback in memory that knows `count` is 1.

```
// Hey memory, we need you to store a function...
() => {
  const count = 0
  const id = setInterval(() => {
    setCount(count + 1)
  }, 1000)
  return () => clearInterval(id)
}

// Later on when count changed...
// Hey memory, call the cleanup of that first function, then
// we need you to store another function...
() => {
  const count = 1
  const id = setInterval(() => {
    setCount(count + 1)
  }, 1000)
  return () => clearInterval(id)
}
```

If you do want to know the current value instead of the captured one, you can always use a mutable ref

• • •

I think the `setInterval` example is a great way to understand the capturing nature of `useEffect`, although it's worth pointing out that there is another API for setting state where we pass a function instead of the value. React will call the function with the existing state (just like the Class-based `setState` does):

```
useEffect(() => {
  const id = setInterval(() => {
    // When we pass a function, React calls that function with the current
    // state and whatever we return becomes the new state.
    setCount(count => count + 1)
  }, 1000)
  return () => clearInterval(id)
}, [])
```

Now since our effect is not closing over the `count` value it doesn't need to be added to the dependency array and we're not closing over it so therefore it is not "captured" as the original value when the effect function was made. Even though this is a bit of a tangent conversation, at least we know what the idea of capturing is for effects *and* that there is another way to set state -- hopefully a win-win in learning.

Is capturing good or bad?

There are some bugs that can be avoided when capturing is used instead of the current value. Take this example from Dan Abramov where he shows how capturing actually gives you an expected behavior over the class behavior which gives the current state. In the example, we can follow someone (similar to Twitter) and then quickly change profiles. When we change profiles before the network response resolves, there is a bug when the name of our recent follow is shown to us. It's a bug with the class version of "follow", but not the hooks version of "follow".

Read Dan's article on it here.

What about refactoring class-based code to hooks?

Perhaps you wrote some code like this?

```
class UserProfile extends React.Component {
  state = { user: null }

  componentDidMount() {
    getUser(this.props.uid).then(user => {
      this.setState({ user })
    })
  }

  render() {
    // ...
  }
}
```

Do you see the bug?

What would happen if the `uid` prop changes? We would not see the new user because we're not handling that change with `componentDidUpdate`. Usually, if your `componentDidMount` is doing a side effect that depends on props or state then you need a `componentDidUpdate` to handle the side effect again when props or state change. But sometimes we don't always do that and we can introduce bugs when we forget.

If you refactoring class-based code to hooks and you simply turn every `componentDidMount` into a `useEffect` with an empty dependency array, you will almost certainly have bugs in your new code. Let's say we refactor the above to be:

```
function UserProfile({ uid }) {
  const [user, setUser] = useState(null)

  useEffect(() => {
    getUser(uid).then(user => {
      setUser(user)
    })
  }, []) // buggy without `uid` in this array

  // ...
}
```

This code would work, until you have a situation where the `UserProfile` has its `uid` changed while it's mounted. If you have the extra hooks linting rules installed though, you would get a warning until you did `[uid]` as the dependency array. With that in place, the hooks version is doing what would have been a `componentDidMount` and a `componentDidUpdate` at the same time. So you see, the very question of "Is `useEffect` with an empty dependency array the new *version* of `componentDidMount`?" is a flawed question to begin with `componentDidMount` will most often not be refactored to `useEffect(fn, [])`

By the way, don't forget to do a cleanup function as well. This will prevent the bug of setting state on an unmounted component and setting stale state when the `uid` changes:

```
useEffect(() => {
  let isCurrent = true
  getUser(uid).then(user => {
    if (isCurrent) {
      setUser(user)
    }
  })
  return () => {
    isCurrent = false
  }
}, [uid])
```

Summary

Thinking in terms of "time" was how we did things with class-based components. Now we want to think in terms of "With this state, what does my UI look like?" and "when this state changes, what side effects need to be re-ran". Try to orientate around state instead of the "lifecycle timing" of your component.

15,360 claps

Newsletter RSS

While we don't have blog comments, we tweeted about this posting when it went live so we welcome your comments there:

COMMENTS

REACT
TRAINING

ReactTraining.com is the world-renowned React training company who's committed to diversity and education in the tech community.

ReactTraining

hello@reacttraining.com

Company

- Home • Workshop Conduct • Diversity • Attend a Workshop • Get Corporate Training • Online Courses
- Our Team • Blog • Newsletter

Software

- React Router • Reach UI • Remix

Resources

- React Docs • React Hooks Docs • Hooks API Reference • Hooks FAQ