React Hooks

Show & Tell - May 2019

Recap

React works by giving it a "description of the DOM" that we want to see in the "real DOM".

When we want to change the DOM, we use the state and lifecycle APIs of the class component.

This approach has some downsides.

Disclaimer

The React team, have been **extremely** strong in stressing that by introducing Hooks, they **do not recommend** rewriting your entire application to Hooks.

We don't have much React, so this isn't really relevant for us.

Motivation

From the Hooks documentation:

- Classes confuse both people and machines
- Complex components become hard to understand
- It’s hard to reuse stateful logic between components

Let's have a look at these in more detail.

Classes confuse both people and machines

The `this` keyword is famously counter-intuitive to both newer and more experienced programmers.

Class inheritance isn't *really* used by React - it's just a convenient way to inject the API into our components. The React team do not recommend creating multiple levels of inheritance.

Worse, classes are hard to fully optimise: for example class method names cannot be minified.

Complex components become hard to understand

```
class WindowWidth extends React.Component {
  constructor(props) {
    super(props)
    this.state = {
      width: null
    }
  }
  componentDidMount() {
    window.addEventListener('resize', this.handleResize)
  }
  componentWillUnmount() {
    window.removeEventListener('resize', this.handleResize)
  }
  handleResize = () => {
    this.setState({ width: window.innerWidth })
  }
  render() {
    return <div>Window width: {this.state.width}</div>
  }
}
```

It’s hard to reuse stateful logic between components

```
class ProjectData extends React.Component {
  constructor(props) {
    super(props)
    this.state = {
      data: null,
      err: null
    }
  }
  componentDidMount() {
    doFetch()
  }
  componentDidUpdate(prevProps) {
    if (this.props.id !== prevProps.id) {
      doFetch()
    }
  }
  doFetch = () => {
    fetch(this.props.url)
      .then(res => res.json())
      .then(data => this.setState({ data }))
      .catch(err => this.setState({ err }))
  }
  render() {
```

Hooks Are Here To Help

State Hook

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

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

  const handleClick = () => setCount(count + 1)

  return (
    <div>
      <p>You clicked {count} times</p>
      <button onClick={handleClick}>Click me</button>
    </div>
  )
}
```

Effect Hook

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

function WindowWidth() {
  const [width, setWidth] = useState(window.innerWidth)

  const handleResize = () => setWidth(window.innerWidth)

  useEffect(() => {
    window.addEventListener('resize', handleResize)
    return () => {
      window.removeEventListener('resize', handleResize)
    }
  })

  return <div>Window width: {width}</div>
}
```

Custom Hooks

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

function useWindowWidth() {
  const [width, setWidth] = useState(window.innerWidth)

  const handleResize = () => setWidth(window.innerWidth)

  useEffect(() => {
    window.addEventListener('resize', handleResize)
    return () => {
      window.removeEventListener('resize', handleResize)
    }
  })

  return width
}

export default useWindowWidth
```

Custom Hooks (cont.)

```
import useWindowWidth from './path/to/custom/hook'

function MyComponent() {
  const windowWidth = useWindowWidth()

  return <div>The window width is: {windowWidth}</div>
}
```

Custom Hooks (cont.)

```
$ npm install @rehooks/window-size
```

```
import useWindowWidth from '@rehooks/window-size'

function MyComponent() {
  const windowWidth = useWindowWidth()

  return <div>The window width is: {windowWidth}</div>
}
```

Gotchas

Rules of Hooks

Hooks are JavaScript functions, but they impose two additional rules:

- Only call Hooks at the top level. Don’t call Hooks inside loops, conditions, or nested functions
- Only call Hooks from React function components. Don’t call Hooks from regular JavaScript functions.

There is an eslint rule to guard against breaking these rules.

Closures

JavaScript *closes over* values in function scope. This can occasionally produce unexpected behaviour in hooks: Broken example

There is fix however, based around using a React ref: Fixed example

Hooks and (some) async APIs

Some async APIs (like `setInterval`) can be quite "aggressive", so can create some confusing bugs.

This (somewhat long) video walks through a example of this problem and it's solution.

The key is that if part of the component's state depends on other parts of state then another hook `useReducer` is helpful. This blog post might be useful too.

Thanks!

What questions do you have?

You can see the source for this presentation here