Hello, React Hooks!

안희종 (heejongahn@gmail.com)
2019-02-09

Table of Contents

- Topic: **React Hooks**

Motivation

- **React Hooks 프로포절은 왜 나왔을까?**

Motivation

- React Class Component가 가진 세 가지 문제점 (from 공식 문서):

Motivation

- React Class Component가 가진 세 가지 문제점 (from 공식 문서):
  - 컴포넌트 간 상태가 있는 로직을 재활용하기 어렵다*It's hard to reuse stateful logic between components*

Motivation

- React Class Component가 가진 세 가지 문제점 (from 공식 문서):
  - 컴포넌트 간 상태가 있는 로직을 재활용하기 어렵다*It's hard to reuse stateful logic between components*
  - 복잡한 컴포넌트는 이해하기 어려워진다*Complex components become hard to understand*

Motivation

- React Class Component가 가진 세 가지 문제점 (from 공식 문서):
  - 컴포넌트 간 상태가 있는 로직을 재활용하기 어렵다*It's hard to reuse stateful logic between components*
  - 복잡한 컴포넌트는 이해하기 어려워진다*Complex components become hard to understand*
  - 사람과 기계 모두에게 클래스는 헷갈리는 개념이다*Classes confuse both people and machines*

컴포넌트 간 상태가 있는 로직을 재활용하기 어렵다

- 상태가 있는 로직을 재활용하기 위한 그간의 솔루션: HoC, Render Props

컴포넌트 간 상태가 있는 로직을 재활용하기 어렵다

- 상태가 있는 로직을 재활용하기 위한 그간의 솔루션: HoC, Render Props
- 문제점
  - 컴포넌트 구조에 제약이 발생

컴포넌트 간 상태가 있는 로직을 재활용하기 어렵다

- 상태가 있는 로직을 재활용하기 위한 그간의 솔루션: HoC, Render Props
- 문제점
  - 컴포넌트 구조에 제약이 발생
  - 로직 재활용을 위한 너무 많은 래퍼 엘리먼트 → Wrapper Hell

컴포넌트 간 상태가 있는 로직을 재활용하기 어렵다

```
▼ <Unknown>
  ▼ <t debug={false} errorMessage="">
    ▼ <o>
      ▼ <t>
        ▼ <t>
          ▼ <Router>
            ▼ <RouterContext>
              ▼ <Apollo(Connect(Apollo(n)))>
                ▼ <t fetchPolicy="network-only" errorPolicy="ignore" ssr={false} displayName="Apollo(Connect(Apollo(n)))"
                  skip={false} warnUnhandledError={true}>
                  ▼ <Connect(Apollo(n)) authLoading={false} isAuthenticated={2539615}>
                    ▼ <Apollo(n) authLoading={false} isAuthenticated={2539615}>
                      ▼ <t errorPolicy="ignore" ssr={false} displayName="Apollo(n)" skip={false} warnUnhandledError={true}>
                        ▼ <n authLoading={false} isAuthenticated={2539615} userLoading={false}>
                          ▼ <Connect(Apollo(t)) authLoading={false} isAuthenticated={2539615} userLoading={false}>
                            ▶ <Apollo(t) authLoading={false} isAuthenticated={2539615} userLoading={false} isMobile={false
                              } isOnline={true} lang="id" popUp={false} searchModalOpen={false} sessionId={2539615} xdevice
                              ="">…</Apollo(t)> == $r
                            </Connect(Apollo(t))>
                          </n>
                        </t>
                      </Apollo(n)>
                    </Connect(Apollo(n))>
                  </t>
                </Apollo(Connect(Apollo(n)))>
              </RouterContext>
            </Router>
          </t>
        </t>
      </o>
    </t>
  </Unknown>
```

https://medium.com/@jackyef/react-hooks-why-we-should-embrace-it-86e408663ad6

컴포넌트 간 상태가 있는 로직을 재활용하기 어렵다

Wrapper Hell

https://medium.com/@jackyef/react-hooks-why-we-should-embrace-it-86e408663ad6

복잡한 컴포넌트는 이해하기 어려워진다

- 특정 로직(데이터 받아오기, 타이머, …)에 관하여:

복잡한 컴포넌트는 이해하기 어려워진다

- 특정 로직(데이터 받아오기, 타이머, ...)에 관하여:
  - 최초 등록: `componentDidMount`

복잡한 컴포넌트는 이해하기 어려워진다

- 특정 로직(데이터 받아오기, 타이머, ...)에 관하여:
  - 최초 등록: `componentDidMount`
  - 싱크 맞추기: `componentDidUpdate`

복잡한 컴포넌트는 이해하기 어려워진다

- 특정 로직(데이터 받아오기, 타이머, ...)에 관하여:
  - 최초 등록: `componentDidMount`
  - 싱크 맞추기: `componentDidUpdate`
  - 해지: `componentWillUnmount`

복잡한 컴포넌트는 이해하기 어려워진다

- 특정 로직(데이터 받아오기, 타이머, ...)에 관하여:
  - 최초 등록: `componentDidMount`
  - 싱크 맞추기: `componentDidUpdate`
  - 해지: `componentWillUnmount`
- 하나의 로직에 관련된 코드가 여러 메소드에 나뉘어 존재

복잡한 컴포넌트는 이해하기 어려워진다

- 특정 로직(데이터 받아오기, 타이머, ...)에 관하여:
  - 최초 등록: `componentDidMount`
  - 싱크 맞추기: `componentDidUpdate`
  - 해지: `componentWillUnmount`
- 하나의 로직에 관련된 코드가 여러 메소드에 나뉘어 존재
- 한 컴포넌트에 엮인 “특정 로직”의 갯수가 점점 늘어난다면?

복잡한 컴포넌트는 이해하기 어려워진다

서로 연관된 코드는 떨어지는 반면,

전혀 상관 없는 코드가 같은 메소드 내에 자리하곤 합니다.

이는 쉽게 버그와 비일관성을 초래합니다.

Mutually related code that changes together gets split apart,

but completely unrelated code ends up combined in a single method.

This makes it too easy to introduce bugs and inconsistencies.

복잡한 컴포넌트는 이해하기 어려워진다

서로 연관된 코드는 떨어지는 반면,

전혀 상관 없는 코드가 같은 메소드 내에 자리하곤 합니다.

이는 쉽게 **버그**와 **비일관성**을 초래합니다.

Mutually related code that changes together gets split apart,

but completely unrelated code ends up combined in a single method.

This makes it too easy to introduce bugs and inconsistencies.

사람과 기계 모두에게 클래스는 헷갈리는 개념이다

- 사람의 입장

사람과 기계 모두에게 클래스는 헷갈리는 개념이다

- 사람의 입장
  - 자바스크립트의 `class`는 처음 배우는 사람과 객체 지향 언어 경험이 있는 사람 모두에게 헷갈릴 여지가 많다 (How does `this` work?)

사람과 기계 모두에게 클래스는 헷갈리는 개념이다

- 사람의 입장
  - 자바스크립트의 `class`는 처음 배우는 사람과 객체 지향 언어 경험이 있는 사람 모두에게 헷갈릴 여지가 많다 (How does `this` work?)
  - 장황한 문법, 배워야 할 또 하나의 개념

사람과 기계 모두에게 클래스는 헷갈리는 개념이다

- 사람의 입장
  - 자바스크립트의 `class`는 처음 배우는 사람과 객체 지향 언어 경험이 있는 사람 모두에게 헷갈릴 여지가 많다 (How does `this` work?)
  - 장황한 문법, 배워야 할 또 하나의 개념
- 기계의 입장

사람과 기계 모두에게 클래스는 헷갈리는 개념이다

- 사람의 입장
  - 자바스크립트의 `class`는 처음 배우는 사람과 객체 지향 언어 경험이 있는 사람 모두에게 헷갈릴 여지가 많다 (How does `this` work?)
  - 장황한 문법, 배워야 할 또 하나의 개념
- 기계의 입장
  - 기계가 최적화하기에 어려운 코드 생성

사람과 기계 모두에게 클래스는 헷갈리는 개념이다

- 사람의 입장
  - 자바스크립트의 `class`는 처음 배우는 사람과 객체 지향 언어 경험이 있는 사람 모두에게 헷갈릴 여지가 많다 (How does `this` work?)
  - 장황한 문법, 배워야 할 또 하나의 개념
- 기계의 입장
  - 기계가 최적화하기에 어려운 코드 생성
  - Poor Minification, Flaky & unreliable Hot Reload

Hooks to the rescue!

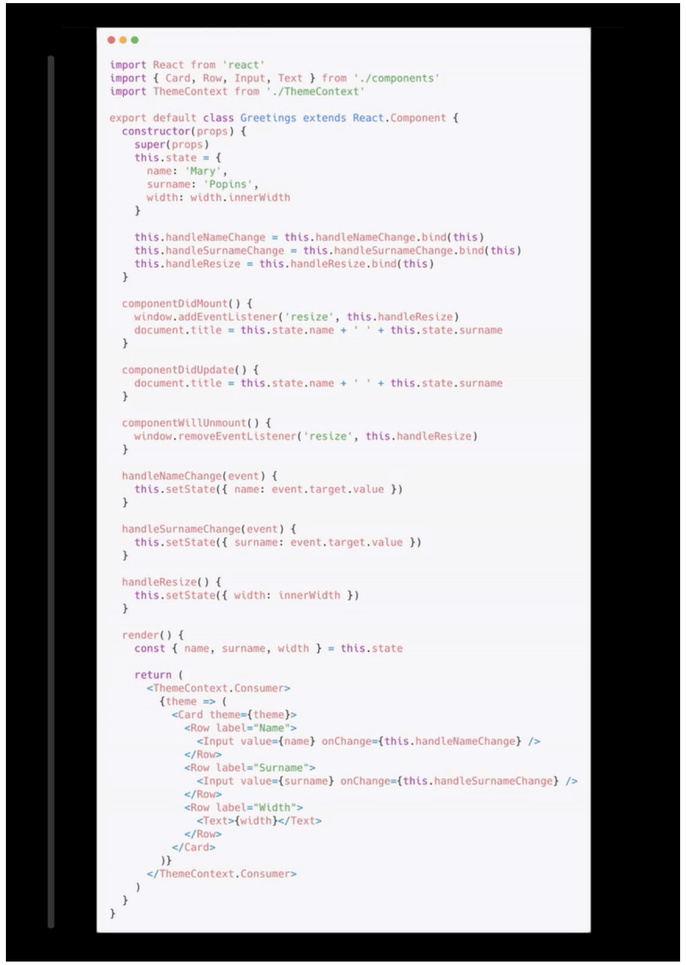
래퍼 엘리먼트 없는

로직 재활용

Hooks to the rescue!

래퍼 엘리먼트 없는

로직 재활용



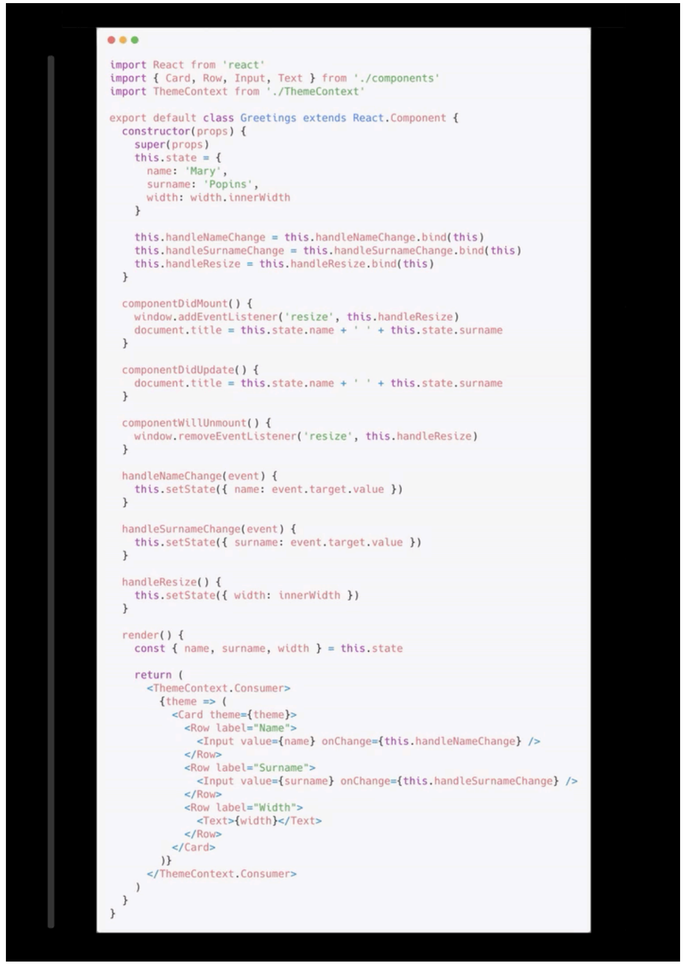
```
import React from 'react'
import { Card, Row, Input, Text } from './components'
import ThemeContext from './ThemeContext'

export default class Greetings extends React.Component {
  constructor(props) {
    super(props)
    this.state = {
      name: 'Mary',
      surname: 'Popins',
      width: width.innerWidth
    }

    this.handleNameChange = this.handleNameChange.bind(this)
    this.handleSurnameChange = this.handleSurnameChange.bind(this)
    this.handleResize = this.handleResize.bind(this)
  }

  componentDidMount() {
    window.addEventListener('resize', this.handleResize)
    document.title = this.state.name + ' ' + this.state.surname
  }

  componentDidUpdate() {
    document.title = this.state.name + ' ' + this.state.surname
  }

  componentWillUnmount() {
    window.removeEventListener('resize', this.handleResize)
  }

  handleNameChange(event) {
    this.setState({ name: event.target.value })
  }

  handleSurnameChange(event) {
    this.setState({ surname: event.target.value })
  }

  handleResize() {
    this.setState({ width: innerWidth })
  }

  render() {
    const { name, surname, width } = this.state

    return (
      <ThemeContext.Consumer>
        {theme => (
          <Card theme={theme}>
            <Row label="Name">
              <Input value={name} onChange={this.handleNameChange} />
            </Row>
            <Row label="Surname">
              <Input value={surname} onChange={this.handleSurnameChange} />
            </Row>
            <Row label="Width">
              <Text>{width}</Text>
            </Row>
          </Card>
        )}
      </ThemeContext.Consumer>
    )
  }
}
```

https://twitter.com/prchdk/status/1056960391543062528

Hooks to the rescue!

래퍼 엘리먼트 없는

로직 재활용



```
import React from 'react'
import { Card, Row, Input, Text } from './components'
import ThemeContext from './ThemeContext'

export default class Greetings extends React.Component {
  constructor(props) {
    super(props)
    this.state = {
      name: 'Mary',
      surname: 'Popins',
      width: width.innerWidth
    }

    this.handleNameChange = this.handleNameChange.bind(this)
    this.handleSurnameChange = this.handleSurnameChange.bind(this)
    this.handleResize = this.handleResize.bind(this)
  }

  componentDidMount() {
    window.addEventListener('resize', this.handleResize)
    document.title = this.state.name + ' ' + this.state.surname
  }

  componentDidUpdate() {
    document.title = this.state.name + ' ' + this.state.surname
  }

  componentWillUnmount() {
    window.removeEventListener('resize', this.handleResize)
  }

  handleNameChange(event) {
    this.setState({ name: event.target.value })
  }

  handleSurnameChange(event) {
    this.setState({ surname: event.target.value })
  }

  handleResize() {
    this.setState({ width: innerWidth })
  }

  render() {
    const { name, surname, width } = this.state

    return (
      <ThemeContext.Consumer>
        {theme => (
          <Card theme={theme}>
            <Row label="Name">
              <Input value={name} onChange={this.handleNameChange} />
            </Row>
            <Row label="Surname">
              <Input value={surname} onChange={this.handleSurnameChange} />
            </Row>
            <Row label="Width">
              <Text>{width}</Text>
            </Row>
          </Card>
        )}
      </ThemeContext.Consumer>
    )
  }
}
```

No more classes,

Just functions

https://twitter.com/prchdk/status/1056960391543062528

Usage

- React Hooks는 Class Component의 문제점을 해결하기 위해 나왔다.
- **React Hooks, 어떻게 사용할 수 있을까?**

Usage

- **Built-in Hooks**
  - `useState`
  - `useEffect`
  - `useRef`
  - `useContext`
- **Custom Hooks**

Built-in Hook: useState

```
class Example extends React.Component {
  constructor(props) {
    super(props);
    this.state = {
      count: 0
    };
  }

  render() {
    return (
      <div>
        <p>You clicked {this.state.count} times</p>
        <button onClick={() => this.setState({ count: this.state.count + 1 })}>
          Click me
        </button>
      </div>
    );
  }
}
```

초기값

값 읽기

값 쓰기

Built-in Hook: useState

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

function Example() {
  // Declare a new state variable, which we'll call "count"
  const [count, setCount] = useState(0);

  return (
    <div>
      <p>You clicked {count} times</p>
      <button onClick={() => setCount(count + 1)}>
        Click me
      </button>
    </div>
  );
}
```

초기값

값 읽기

값 쓰기

Built-in Hook: useState

```
function ExampleWithManyStates() {
  // Declare multiple state variables!
  const [age, setAge] = useState(42);
  const [fruit, setFruit] = useState('banana');
  const [todos, setTodos] = useState([{ text: 'Learn Hooks' }]);
```

필요하다면 여러 state 필드의 값과 세터를
여러 번의 useState 호출로 분리해서 사용할 수도 있다

Built-in Hook: `useState`

Built-in Hook: `useEffect`

```
class Example extends React.Component {
  constructor(props) {
    super(props);
    this.state = {
      count: 0
    };
  }

  componentDidMount() {
    document.title = `You clicked ${this.state.count} times`;
  }

  componentDidUpdate() {
    document.title = `You clicked ${this.state.count} times`;
  }

  render() {
    return (
      <div>
        <p>You clicked {this.state.count} times</p>
        <button onClick={() => this.setState({ count: this.state.count + 1 })}>
          Click me
        </button>
      </div>
    );
  }
}
```

- 사이드 이펙트를 수행하는 코드
  - 데이터 불러오기
  - 이벤트 구독
  - DOM 조작
  - ...
- 보통 클래스 컴포넌트의 `componentDidMount`, `componentDidUpdate` 라이프사이클 메소드에 정의

Built-in Hook: useEffect

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

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

  useEffect(() => {
    document.title = `You clicked ${count} times`;
  });

  return (
    <div>
      <p>You clicked {count} times</p>
      <button onClick={() => setCount(count + 1)}>
        Click me
      </button>
    </div>
  );
}
```

- useEffect에 인자로 넘긴 함수는 매 렌더시 **DOM 업데이트가 끝난 이후** 실행된다

Built-in Hook: `useEffect` (w/ cleanup)

```
class FriendStatus extends React.Component {
  constructor(props) {
    super(props);
    this.state = { isOnline: null };
    this.handleStatusChange = this.handleStatusChange.bind(this);
  }

  componentDidMount() {
    ChatAPI.subscribeToFriendStatus(
      this.props.friend.id,
      this.handleStatusChange
    );
  }

  componentWillUnmount() {
    ChatAPI.unsubscribeFromFriendStatus(
      this.props.friend.id,
      this.handleStatusChange
    );
  }

  handleStatusChange(status) {
    this.setState({
      isOnline: status.isOnline
    });
  }

  render() {
    if (this.state.isOnline === null) {
      return 'Loading...';
    }
    return this.state.isOnline ? 'Online' : 'Offline';
  }
}
```

API 구독 등록

API 구독 해지

- 클래스 컴포넌트의 구독 신청, 타이머 등록 등 로직의 뒷처리: `componentWillUnmount` 함수에서 일반적으로 처리

Built-in Hook: `useEffect` (w/ cleanup)

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

function FriendStatus(props) {
  const [isOnline, setIsOnline] = useState(null);

  function handleStatusChange(status) {
    setIsOnline(status.isOnline);
  }

  useEffect(() => {
    ChatAPI.subscribeToFriendStatus(props.friend.id, handleStatusChange);
    // Specify how to clean up after this effect:
    return function cleanup() {
      ChatAPI.unsubscribeFromFriendStatus(props.friend.id, handleStatusChange);
    };
  });

  if (isOnline === null) {
    return 'Loading...';
  }
  return isOnline ? 'Online' : 'Offline';
}
```

구독 등록:
DOM 업데이트가 끝날 때마다 실행

구독 해지:
새 렌더가 시작되기 전, 혹은
컴포넌트 언마운트 시 실행

- `useEffect` 사용 시, 인자로 넘긴 함수의 반환값으로 뒷처리를 위한 함수를 명시

Built-in Hook: `useEffect` (optimization)

```
componentDidUpdate(prevProps, prevState) {
  if (prevState.count !== this.state.count) {
    document.title = `You clicked ${this.state.count} times`;
  }
}
```

```
useEffect(() => {
  document.title = `You clicked ${count} times`;
}, [count]; // Only re-run the effect if count changes
```

두 번째 인자로 의존성 배열을 넘겨 해당 의존성이 변경된 경우에만 호출할 수 있다.
`componentDidMount`에 해당하는 시점에 딱 한 번만 호출되길 원한다면 의존성 배열로 `[]` 를 넘긴다.

Built-in Hook: `useEffect`

Built-in Hook: `useRef`

```
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>
    </>
  );
}
```

- 렌더 간 보존되는 가변 상태를 저장하는 데 사용
- `React.createRef`와 유사한 용도로 사용 가능하지만, DOM node reference에만 용도가 제한되진 않는다.

Built-in Hook: `useContext`

`useContext`

```
const context = useContext(Context);
```

Accepts a context object (the value returned from `React.createContext`) and returns the current context value, as given by the nearest context provider for the given context.

When the provider updates, this Hook will trigger a rerender with the latest context value.

Other Built-in Hooks

- `useCallback`

Other Built-in Hooks

- `useCallback`
- `useMemo`

Other Built-in Hooks

- `useCallback`
- `useMemo`
- `useLayoutEffect`

Other Built-in Hooks

- `useCallback`
- `useMemo`
- `useLayoutEffect`
- `...`

Other Built-in Hooks

- `useCallback`
- `useMemo`
- `useLayoutEffect`
- ...
- https://reactjs.org/docs/hooks-reference.html

Custom Hooks

- Hooks = A way to reuse stateful logic (inside function components)

Custom Hooks

- Hooks = A way to reuse stateful logic (inside function components)
- 함수 본문 중 임의의 부분을 새로운 함수로 뽑아낼 수 있는 것과 마찬가지로,
  상태를 갖는 임의의 로직을 Hook의 형태로 뽑아내 재사용할 수 있음

Custom Hooks

- Hooks = A way to reuse stateful logic (inside function components)
- 함수 본문 중 임의의 부분을 새로운 함수로 뽑아낼 수 있는 것과 마찬가지로,
  상태를 갖는 임의의 로직을 Hook의 형태로 뽑아내 재사용할 수 있음
- 커스텀 훅 = 이름이 “use”로 시작하고, 다른 훅을 호출할 수 있는 자바스크립트 함수
  A custom Hook is a JavaScript function whose name starts with “use” and that may call other Hooks. 출처

Rule of Hooks (link)

- 훅은 **함수의 최상위**에서만 호출하세요
  *Only call Hooks **at the top level**.*

Rule of Hooks (link)

- 훅은 **함수의 최상위**에서만 호출하세요
  *Only call Hooks **at the top level**.*

  - 반복문, 조건문, 감싸진 함수에서 호출해서는 안 됩니다
    Don’t call Hooks inside loops, conditions, or nested functions

Rule of Hooks (link)

- 훅은 **함수의 최상위**에서만 호출하세요
  *Only call Hooks **at the top level**.*

  - 반복문, 조건문, 감싸진 함수에서 호출해서는 안 됩니다
    Don’t call Hooks inside loops, conditions, or nested functions

- 훅은 **리액트 함수 컴포넌트 내에서만** 호출하세요
  *Only call Hooks **from React function components**.*

Rule of Hooks (link)

- 훅은 **함수의 최상위**에서만 호출하세요
  *Only call Hooks **at the top level**.*

  - 반복문, 조건문, 감싸진 함수에서 호출해서는 안 됩니다
    Don’t call Hooks inside loops, conditions, or nested functions

- 훅은 **리액트 함수 컴포넌트 내에서만** 호출하세요
  *Only call Hooks **from React function components**.*

  - 보통의 자바스크립트 함수 내에서 호출해서는 안 됩니다
    Don’t call Hooks from regular JavaScript functions.

Rule of Hooks (link)

- 훅은 **함수의 최상위**에서만 호출하세요
  *Only call Hooks **at the top level**.*

  - 반복문, 조건문, 감싸진 함수에서 호출해서는 안 됩니다
    Don’t call Hooks inside loops, conditions, or nested functions

- 훅은 **리액트 함수 컴포넌트 내에서만** 호출하세요
  *Only call Hooks **from React function components**.*

  - 보통의 자바스크립트 함수 내에서 호출해서는 안 됩니다
    Don’t call Hooks from regular JavaScript functions.

- 이 두 규칙을 제약하는 린터도 존재

Custom Hooks example: `useInterval`

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

function useInterval(callback, delay) {
  const savedCallback = useRef();

  // Remember the latest callback.
  useEffect(() => {
    savedCallback.current = callback;
  });

  // Set up the interval.
  useEffect(() => {
    function tick() {
      savedCallback.current();
    }
    if (delay !== null) {
      let id = setInterval(tick, delay);
      return () => clearInterval(id);
    }
  }, [delay]);
}
```

https://overreacted.io/making-setinterval-declarative-with-react-hooks/

Custom Hooks example: `useSpring`

Either: overwrite values to change the animation

If you re-render the component with changed props, the animation will update.

```
const props = useSpring({ opacity: toggle ? 1 : 0 })
```

Or: pass a function that returns values, and update using "set"

You will get a `set(Values)` function back, use it to update the animation. This will not cause the component to render, which can be generally more performant. It also prevents configs from being re-created on every render. Handling updates like this is extremely useful for fast-occuring updates, like event streams, mousemoves, etc.

```
const [props, set] = useSpring(() => ({ opacity: 1 }))
// ...
  <div onClick={() => set({ opacity: toggle ? 1 : 0 })} />
```

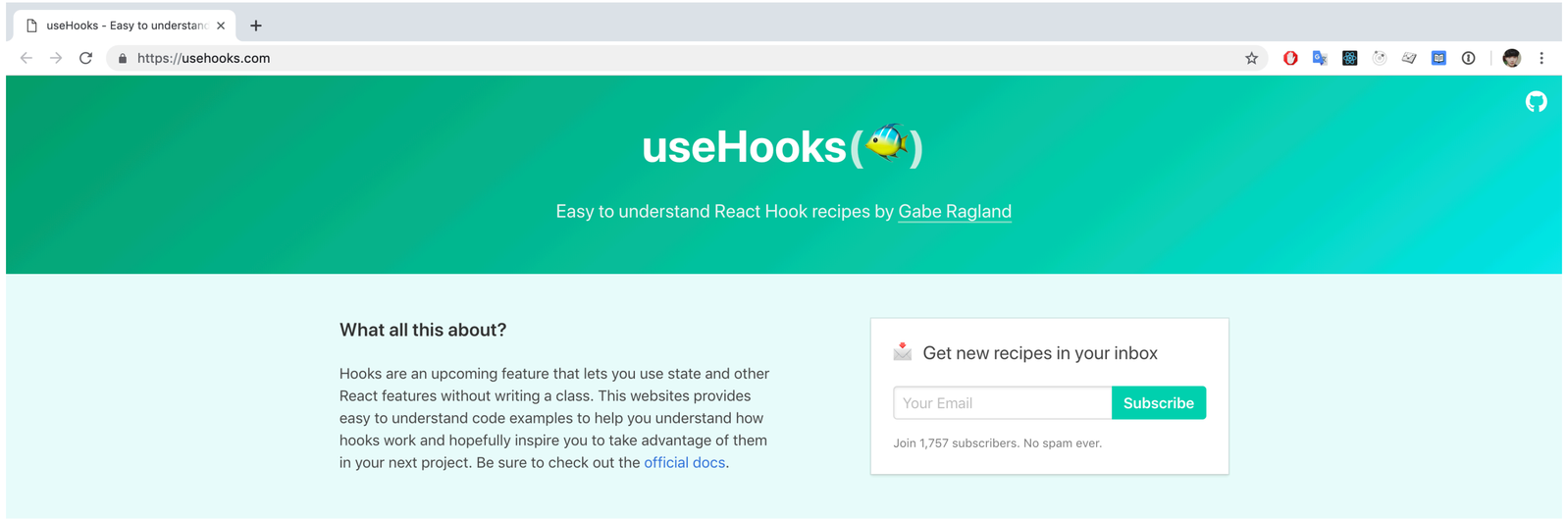
Finally: distribute animated props among the view

The return value is an object containing animated props.

```
return <animated.div style={props}>i will fade</animated.div>
```

http://react-spring.surge.sh/hooks

… and more Custom Hook examples



useMedia

This hook makes it super easy to utilize media queries in your component logic. In our example below we render a different number of columns depending on which media query matches the current screen width, and then distribute images amongst the columns in a way that limits column height difference (we don't want one column way longer than the rest).

You could create a hook that directly measures screen width instead of using media queries, but this method is nice because it makes it easy to share media queries between JS and your stylesheet. See it in action in the CodeSandbox Demo.

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

function App() {
  const columnCount = useMedia(
    // Media queries
    ['(min-width: 1500px)', '(min-width: 1000px)', '(min-width: 600px)'],
    // Column counts (relates to above media queries by array index)
    [5, 4, 3],
    // Default column count
```

https://usehooks.com/

Internal

- React Hooks는 Class Component의 문제점을 해결하기 위해 나왔다.
- React Hooks는 빌트인 훅과 커스텀 훅 둘로 나뉘며, (요래요래) 사용할 수 있다.
- **React Hooks는 내부적으로 어떻게 동작할까?**

Rule of Hooks (revisited)

- 훅은 **함수의 최상위**에서만 호출하세요
  *Only call Hooks **at the top level**.*

  - 반복문, 조건문, 감싸진 함수에서 호출해서는 안 됩니다
    Don’t call Hooks inside loops, conditions, or nested functions

- 훅은 **리액트 함수 컴포넌트 내에서만** 호출하세요
  *Only call Hooks **from React function components**.*

  - 보통의 자바스크립트 함수 내에서 호출해서는 안 됩니다
    Don’t call Hooks from regular JavaScript functions.

- 이 두 규칙을 제약하는 린터도 존재

Rule of Hooks (revisited)

- 훅은 **함수의 최상위**에서만 호출하세요
  *Only call Hooks **at the top level**.*

  - 반복문, 조건문, 감싸진 함수에서 호출해서는 안 됩니다
    Don’t call Hooks inside loops, conditions, or nested functions.

  - 오직 **리액트 함수 컴포넌트 내**에서만 호출하세요
    *Only call Hooks **from React function components**.*

  - 보통의 자바스크립트 함수 내에서 호출해서는 안 됩니다
    Don’t call Hooks from regular JavaScript functions.

- 이 두 규칙을 제약하는 린터도 존재

yeah but why?

Internal: Pseudocode ver.

```
// Pseudocode
let hooks, i;
function useState() {
  i++;
  if (hooks[i]) {
    // Next renders
    return hooks[i];
  }
  // First render
  hooks.push(...);
}

// Prepare to render
i = -1;
hooks = fiber.hooks || [];
// Call the component
YourComponent();
// Remember the state of Hooks
fiber.hooks = hooks;
```

https://overreacted.io/react-as-a-ui-runtime/

```
let hooks = null;

export function useHook() {
  hooks.push(hookData);
}

function reactsInternalRenderAComponentMethod(component) {
  hooks = [];
  component();
  let hooksForThisComponent = hooks;
  hooks = null;
}
```

https://mobile.twitter.com/jamiebuilds/status/1055538414538223616

From “Rule of Hooks” doc

- 그렇다면 리액트는 어떤 상태가 어떤 `useState` 호출에 대응하는지
  어떻게 알 수 있을까? *So how does React know which state corresponds to which `useState` call?*

From “Rule of Hooks” doc

- 그렇다면 리액트는 어떤 상태가 어떤 `useState` 호출에 대응하는지
  어떻게 알 수 있을까? *So how does React know which state corresponds to which `useState` call?*

- 정답은 바로 **리액트가 훅의 호출 순서에 의존한다**는 것입니다.
  *The answer is that **React relies on the order in which Hooks are called.***

From “Rule of Hooks” doc

- 그렇다면 리액트는 어떤 상태가 어떤 `useState` 호출에 대응하는지
  어떻게 알 수 있을까? *So how does React know which state corresponds to which `useState` call?*

- 정답은 바로 **리액트가 훅의 호출 순서에 의존한다**는 것입니다.
  *The answer is that **React relies on the order in which Hooks are called.***

- 이것이 바로 훅이 컴포넌트의 최상위 레벨에서 호출되어야만 하는 이유입니다.
  This is why Hooks must be called on the top level of our components.

From “Rule of Hooks” doc

- 그렇다면 리액트는 어떤 상태가 어떤 `useState` 호출에 대응하는지
  어떻게 알 수 있을까? *So how does React know which state corresponds to which `useState` call?*

- 정답은 바로 **리액트가 훅의 호출 순서에 의존한다**는 것입니다.
  *The answer is that **React relies on the order in which Hooks are called.***

- 이것이 바로 훅이 컴포넌트의 최상위 레벨에서 호출되어야만 하는 이유입니다.
  This is why Hooks must be called on the top level of our components.

- 왜 호출 순서에 의존하나요? 👉 See Why Do React Hooks Rely on Call Order?

From “Rule of Hooks” doc

- 그렇다면 리액트는 어떤 상태가 어떤 `useState` 호출에 대응하는지
  어떻게 알 수 있을까? *So how does React know which state corresponds to which `useState` call?*

- 정답은 바로 **리액트가 훅의 호출 순서에 의존한다**는 것입니다.
  *The answer is that **React relies on the order in which Hooks are called.***

- 이것이 바로 훅이 컴포넌트의 최상위 레벨에서 호출되어야만 하는 이유입니다.
  This is why Hooks must be called on the top level of our components.

- 왜 호출 순서에 의존하나요? 👉 See Why Do React Hooks Rely on Call Order?

  - **tl;dr**: 각 훅 호출을 독립적으로 만들고, 인자를 받을 수 있으면서도
    이름 충돌을 막고 문법적 간결성을 유지할 수 있도록.

더 자세히 알고 싶다면…

- Under the hood of React’s hooks system
- React as a UI Runtime
- React hooks: not magic, just arrays
- react-reconciler/src/ReactFiberHooks.js

Summary

- React Hooks는 CC(Class Component)의 문제점을 해결하고자 생겨났다.
  Hooks를 사용해 기존엔 CC가 필요했던 작업을 함수 컴포넌트로도 수행할 수 있다.

Summary

- React Hooks는 CC(Class Component)의 문제점을 해결하고자 생겨났다.
  Hooks를 사용해 기존엔 CC가 필요했던 작업을 함수 컴포넌트로도 수행할 수 있다.

- React는 `useState`, `useEffect` 등 프리미티브로서의 빌트인 훅을 제공한다.
  프로그래머는 이런 빌트인 훅을 사용, 임의의 로직을 커스텀 훅으로 정의할 수 있다.

Summary

- React Hooks는 CC(Class Component)의 문제점을 해결하고자 생겨났다.
  Hooks를 사용해 기존엔 CC가 필요했던 작업을 함수 컴포넌트로도 수행할 수 있다.

- React는 `useState`, `useEffect` 등 프리미티브로서의 빌트인 훅을 제공한다.
  프로그래머는 이런 빌트인 훅을 사용, 임의의 로직을 커스텀 훅으로 정의할 수 있다.

- React Hooks의 동작은 Hook의 호출 순서에 의존한다.
  따라서, 모든 Hook은 Rules of Hook을 만족하는 방식으로 정의 및 호출되어야 한다.

Summary

References (1/2)

- General
  - Introducing Hooks
  - A comment on RFC: React Hooks
  - React as a UI Runtime
- Motivation
  - React Today and Tomorrow and 90% Cleaner React With Hooks

References (2/2)

- Usage
  - Making Sense of React Hooks
  - Making setInterval Declarative with React Hooks
- Internal
  - Why Do React Hooks Rely on Call Order?
  - React hooks: not magic, just arrays
  - Under the hood of React’s hooks system
  - react-reconciler/src/ReactFiberHooks.js