cloneElement

주의하세요!

`cloneElement` 사용하는 것은 일반적이지 않고 불안정한 코드를 만들 수 있습니다. 일반적으로 사용하는 대안을 살펴보세요.

`cloneElement` 를 사용하면 엘리먼트를 기준으로 새로운 React 엘리먼트를 만들 수 있습니다.

```
const clonedElement = cloneElement(element, props, ...children)
```


레퍼런스

`cloneElement(element, props, ...children)`

새로운 React 엘리먼트를 만들기 위해 `element` 를 기준으로 하고, `props` 와 `children` 을 다르게 하여 `cloneElement` 를 호출하세요.

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

// ...
const clonedElement = cloneElement(
  <Row title="Cabbage">
    Hello
  </Row>,
  { isHighlighted: true },
  'Goodbye'
);

console.log(clonedElement); // <Row title="Cabbage" isHighlighted={true}>Goodbye</Row>
```

아래에서 더 많은 예시를 볼 수 있습니다.

매개변수

- `element`: `element` 인자는 유효한 React 엘리먼트여야 합니다. 예를 들어, `<Something />` 과 같은 JSX 노드, `createElement` 로 호출해 얻은 결과물 또는 다른 `cloneElement` 로 호출해 얻은 결과물이 될 수 있습니다.

- `props`: `props` 인자는 객체 또는 `null` 이어야 합니다. `null` 을 전달하면 복제된 엘리먼트는 원본 `element.props` 를 모두 유지합니다. 그렇지 않으면 `props` 객체의 각 prop에 대해 반환된 엘리먼트는 `element.props` 의 값보다 `props` 의 값을 "우선"합니다. 나머지 `props` 는 원본 `element.props` 에서 채워집니다. `props.key` 또는 `props.ref` 를 전달하면 원본의 것을 대체합니다.

- **(선택사항)** `...children`: 0개 이상의 자식 노드가 필요합니다. React 엘리먼트, 문자열, 숫자, portals, 빈 노드 (`null`, `undefined`, `true`, `false`) 및 React 노드 배열을 포함한 모든 React 노드가 해당할 수 있습니다. `...children` 인자를 전달하지 않으면 원본 `element.props.children` 이 유지됩니다.

반환값

`cloneElement` 는 다음과 같은 프로퍼티를 가진 React 엘리먼트 객체를 반환합니다.

- `type`: `element.type` 과 동일합니다.
- `props`: `element.props` 와 전달한 `props` 를 얕게 병합한 결과입니다.
- `ref`: `props.ref` 에 의해 재정의되지 않은 경우 원본 `element.ref` 입니다.
- `key`: `props.key` 에 의해 재정의되지 않은 경우 원본 `element.key` 입니다.

일반적으로 컴포넌트에서 엘리먼트를 반환하거나 다른 엘리먼트의 자식으로 만듭니다. 엘리먼트의 프로퍼티를 읽을 수 있지만, 생성된 후에는 모든 엘리먼트의 프로퍼티를 읽을 수 없는 것처럼 취급하고 렌더링하는 것이 좋습니다.

주의 사항

- 엘리먼트를 복제해도 **원본 엘리먼트는 수정되지 않습니다.**
- **자식이 모두 정적인 경우에만** `cloneElement(element, null, child1, child2, child3)` 와 같이 **자식을 여러 개의 인자로 전달해야 합니다.** 자식이 동적으로 생성되었다면 `cloneElement(element, null, listItems)` 와 같이 전체 배열을 세 번째 인자로 전달해야 합니다. 이렇게 하면 React가 모든 동적 리스트에 대해 key가 누락되었다는 경고를 보여줍니다. 정적 리스트의 경우는 순서가 변경되지 않으므로 이 작업은 필요하지 않습니다.
- `cloneElement` 는 데이터 흐름을 추적하기 어렵기 때문에 다음 대안을 사용해 보세요.

사용법

엘리먼트의 props 재정의하기

일부 React 엘리먼트 의 props를 재정의하려면 재정의하려는 props 를 `cloneElement` 에 전달하세요.

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

// ...
const clonedElement = cloneElement(
  <Row title="Cabbage" />,
  { isHighlighted: true }
);
```

clonedElement 의 결과는 `<Row title="Cabbage" isHighlighted={true} />` 가 됩니다.

어떤 경우에 유용한지 예시를 통해 알아보도록 하겠습니다.

`children` 을 선택할 수 있는 행 목록으로 렌더링하고, 선택된 행을 변경하는 "다음" 버튼이 있는 `List` 컴포넌트를 상상해 보세요. `List` 컴포넌트는 선택된 행을 다르게 렌더링해야 하므로 전달받은 모든 `<Row>` 자식 요소를 복제합니다. 그리고 `isHighlighted: true` 또는 `isHighlighted: false` 인 `prop` 을 추가합니다.

```
export default function List({ children }) {
  const [selectedIndex, setSelectedIndex] = useState(0);
  return (
    <div className="List">
      {Children.map(children, (child, index) =>
        cloneElement(child, {
          isHighlighted: index === selectedIndex
        })
      )}
```

다음과 같이 `List` 에서 전달받은 원본 JSX가 있다고 가정합시다.

```
<List>
  <Row title="Cabbage" />
  <Row title="Garlic" />
  <Row title="Apple" />
</List>
```

자식 요소를 복제함으로써 `List` 는 모든 `Row` 안에 추가적인 정보를 전달할 수 있습니다. 결과는 다음과 같습니다.

```
<List>
  <Row
    title="Cabbage"
    isHighlighted={true}
  />
  <Row
```

```
      title="Garlic"
      isHighlighted={false}
    />
    <Row
      title="Apple"
      isHighlighted={false}
    />
  </List>
```

”다음” 버튼을 누르면 `List` 의 state가 업데이트되고 다른 행이 하이라이트 표시가 되는 것을 확인할 수 있습니다.

App.js List.js Row.js data.js 새로고침 Clear 포크

```
import { Children, cloneElement, useState } from 'react';

export default function List({ children }) {
  const [selectedIndex, setSelectedIndex] = useState(0);
  return (
    <div className="List">
      {Children.map(children, (child, index) =>
        cloneElement(child, {
          isHighlighted: index === selectedIndex
        })
      )}
      <hr />
```

자세히 보기

요약하자면, `List` 는 전달받은 `<Row />` 엘리먼트를 복제하고 추가로 들어오는 prop 또한 추가합니다.

주의하세요!

자식 요소를 복제하는 것은 앱에서 데이터가 어떻게 흘러가는지 파악하기 어렵기 때문에 다음 대안을 사용해 보세요.

대안

렌더링 prop으로 데이터를 전달하기

`cloneElement` 를 사용하는 대신에 `renderItem` 과 같은 *렌더링 prop*을 사용하는 것을 고려해 보세요. 다음 예시의 `List` 는 `renderItem` 을 prop으로 받습니다. `List` 는 모든 item에 대해 `renderItem` 을 호출하고 `isHighlighted` 를 인자로 전달합니다.

```
export default function List({ items, renderItem }) {
  const [selectedIndex, setSelectedIndex] = useState(0);
  return (
    <div className="List">
      {items.map((item, index) => {
        const isHighlighted = index === selectedIndex;
        return renderItem(item, isHighlighted);
      })}
```

`renderItem` prop은 렌더링 방법을 지정하는 prop이기 때문에 “렌더링 prop”이라고 불립니다. 예를 들어, 주어진 `isHighlighted` 값으로 `<Row>` 를 렌더링하는 `renderItem` 을 전달할 수 있습니다.

```
<List
  items={products}
  renderItem={(product, isHighlighted) =>
    <Row
      key={product.id}
      title={product.title}
      isHighlighted={isHighlighted}
    />
  }
/>
```

최종적으로 `cloneElement` 와 같은 결과가 됩니다.

```
<List>
  <Row
    title="Cabbage"
    isHighlighted={true}
  />
  <Row
    title="Garlic"
    isHighlighted={false}
  />
  <Row
    title="Apple"
    isHighlighted={false}
  />
</List>
```

하지만 `isHighlighted` 값의 출처를 명확하게 추적할 수 있습니다.

App.js List.js Row.js data.js 새로고침 Clear 포크

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

export default function List({ items, renderItem }) {
  const [selectedIndex, setSelectedIndex] = useState(0);
  return (
    <div className="List">
```

```
        {items.map((item, index) => {
          const isHighlighted = index === selectedIndex;
          return renderItem(item, isHighlighted);
        })}
        <hr />
```

자세히 보기

이러한 패턴은 더 명시적이기 때문에 `cloneElement` 보다 선호됩니다.

Context를 통해 데이터 전달하기

`cloneElement` 의 또 다른 대안으로는 Context를 통해 데이터를 전달하는 것입니다.

예를 들어, `createContext` 를 호출하여 `HighlightContext` 를 정의할 수 있습니다.

```
export const HighlightContext = createContext(false);
```

`List` 컴포넌트는 렌더링하는 모든 item을 `HighlightContext.Provider` 로 감쌀 수 있습니다.

```
export default function List({ items, renderItem }) {
  const [selectedIndex, setSelectedIndex] = useState(0);
  return (
    <div className="List">
      {items.map((item, index) => {
        const isHighlighted = index === selectedIndex;
        return (
          <HighlightContext key={item.id} value={isHighlighted}>
            {renderItem(item)}
          </HighlightContext>
        );
      })}
```

이러한 접근 방식으로 인해 `Row`는 `isHighlighted` prop을 받을 필요가 없어집니다. 대신 context를 읽습니다.

```
export default function Row({ title }) {
  const isHighlighted = useContext(HighlightContext);
  // ...
```

이에 따라 `isHighlighted`를 `<Row>`로 전달하는 것에 대해 호출된 컴포넌트가 알거나 걱정하지 않아도 됩니다.

```
<List
  items={products}
  renderItem={product =>
    <Row title={product.title} />
  }
/>
```

대신에 `List`와 `Row`는 context를 통해 하이라이팅 로직을 조정합니다.

```
import { useState } from 'react';
import { HighlightContext } from './HighlightContext.js';

export default function List({ items, renderItem }) {
  const [selectedIndex, setSelectedIndex] = useState(0);
  return (
    <div className="List">
      {items.map((item, index) => {
        const isHighlighted = index === selectedIndex;
        return (
          <HighlightContext
            key={item.id}
```

자세히 보기

context를 통해 데이터를 전달하는 것에 대하여 자세히 알아보세요.

커스텀 Hook으로 로직 추출하기

다른 접근 방식으로는 자체 hook을 통해 “비시각적인” 로직을 추출하는 것을 시도해 볼 수 있습니다. 그리고 hook에 의해서 반환된 정보를 사용하여 렌더링할 내용을 정합니다. 예를 들어 다음과 같이 `useList` 같은 커스텀 hook을 작성할 수 있습니다.

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

export default function useList(items) {
  const [selectedIndex, setSelectedIndex] = useState(0);

  function onNext() {
    setSelectedIndex(i =>
      (i + 1) % items.length
    );
  }

  const selected = items[selectedIndex];
  return [selected, onNext];
}
```

그러므로 다음과 같이 사용할 수 있습니다.

```
export default function App() {
  const [selected, onNext] = useList(products);
  return (
    <div className="List">
      {products.map(product =>
        <Row
          key={product.id}
          title={product.title}
          isHighlighted={selected === product}
        />
      )}
      <hr />
      <button onClick={onNext}>
        다음
      </button>
    </div>
  );
}
```

데이터 흐름은 명시적이지만 state는 모든 컴포넌트에서 사용할 수 있는 `useList` custom hook 내부에 있습니다.

App.js useList.js Row.js data.js 새로고침 Clear 포크

```
import Row from './Row.js';
import useList from './useList.js';
import { products } from './data.js';

export default function App() {
  const [selected, onNext] = useList(products);
  return (
    <div className="List">
      {products.map(product =>
        <Row
          key={product.id}
          title={product.title}
```

자세히 보기

이러한 접근 방식은 다른 컴포넌트 간에 해당 로직을 재사용하고 싶을 때 특히 유용합니다.

이전
Children

다음
Component

uwu?

React 학습하기

- 빠르게 시작하기
- 설치하기
- UI 표현하기
- 상호작용성 더하기
- State 관리하기
- 탈출구

API 참고서

- React APIs
- React DOM APIs

커뮤니티

- 행동 강령
- 팀 소개
- 문서 기여자
- 감사의 말

더 보기

- 블로그
- React Native
- 개인 정보 보호
- 약관