Children

주의하세요!

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

`Children`을 사용해서 `children` Prop로 받은 JSX를 조작하고 변환할 수 있습니다.

```
const mappedChildren = Children.map(children, child =>
  <div className="Row">
    {child}
  </div>
);
```


- 레퍼런스
  - `Children.count(children)`
  - `Children.forEach(children, fn, thisArg?)`
  - `Children.map(children, fn, thisArg?)`
  - `Children.only(children)`
  - `Children.toArray(children)`
- 사용법
  - `children` 변환하기
  - 각 자식 요소에서 코드 실행하기
  - `children` 카운팅하기

    - `children` 배열로 병합하기
- 대안
    - 여러 컴포넌트 노출하기
    - Prop로 객체 배열 받기
    - 렌더링 Prop로 렌더링 커스텀하기
- 문제 해결
    - 커스텀 컴포넌트를 전달했을 때 `Children` 메서드가 렌더링 결과를 보여주지 않는 경우

레퍼런스

`Children.count(children)`

`Children.count(children)`는 `children` 데이터 구조의 자식 요소 수를 반환합니다.

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

function RowList({ children }) {
  return (
    <>
      <h1>Total rows: {Children.count(children)}</h1>
      ...
    </>
  );
}
```

아래 예시 보기.

매개변수

- `children`: 컴포넌트에서 받은 `children` Prop의 값.

반환값

`children` 내부 노드의 수.

주의 사항

- 빈 노드(`null`, `undefined` 혹은 Boolean), 문자열, 숫자, React 엘리먼트는 개별 노드로 간주합니다. 배열 자체는 개별 노드가 아니지만 배열의 자식 요소는 개별 노드로 간주합니다. **React 엘리먼트의 하위 요소는 순회하지 않습니다.** React 엘리먼트는 렌더링 되지 않으며 자식 요소를 순회하지 않습니다. Fragment 역시 순회하지 않습니다.

`Children.forEach(children, fn, thisArg?)`

`Children.forEach(children, fn, thisArg?)` 는 `children` 데이터 구조의 모든 자식 요소에 대해 특정 코드를 실행합니다.

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

function SeparatorList({ children }) {
  const result = [];
  Children.forEach(children, (child, index) => {
    result.push(child);
    result.push(<hr key={index} />);
  });
  // ...
```

아래 예시 보기.

매개변수

- `children` : 컴포넌트에서 받은 `children` Prop의 값.
- `fn` : 배열의 `forEach` 메서드 콜백처럼 각 자식 요소에서 실행할 함수. 자식 요소를 첫 번째 인수로, 인덱스를 두 번째 인수로 받습니다. 인덱스는 0에서 시작해서 호출할 때마다 증가합니다.
- **optional** `thisArg` : `fn` 함수가 호출될 때 사용될 `this` 의 값. 생략 시 `undefined` 로 간주합니다.

반환값

`Children.forEach` 는 `undefined` 를 반환합니다.

주의 사항

- 빈 노드(`null`, `undefined` 혹은 Boolean), 문자열, 숫자, React 엘리먼트는 개별 노드로 간주합니다. 배열 자체는 개별 노드가 아니지만 배열의 자식 요소는 개별 노드로 간주합니다. **React 엘리먼트의 하위 요소는 순회하지 않습니다.** React 엘리먼트는 렌더링 되지 않으며 자식 요소를 순회하지 않습니다. Fragments 역시 순회하지 않습니다.

`Children.map(children, fn, thisArg?)`

`Children.map(children, fn, thisArg?)` 은 `children` 데이터 구조에서 각 자식 요소를 매핑하거나 변환합니다.

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

function RowList({ children }) {
  return (
    <div className="RowList">
      {Children.map(children, child =>
        <div className="Row">
          {child}
        </div>
      )}
    </div>
  );
}
```

아래 예시 보기.

매개변수

- `children`: 컴포넌트에서 받은 `children` Prop의 값.
- `fn`: 베열의 `map` 메서드 콜백같은 매핑 함수. 자식 요소를 첫 번째 인수로, 인덱스를 두 번째 인수로 받습니다. 인덱스는 0에서 시작해서 호출할 때마다 증가합니다. 함수는 빈 노드(`null`, `undefined` 혹은 Boolean), 문자열, 숫자, React 엘리먼트 혹은 다른 React 노드의 배열과 같은 React 노드를 반환해야 합니다.
- **optional** `thisArg`: `fn` 함수가 호출될 때 사용될 `this` 의 값. 생략시 `undefined` 로 간주합니다.

반환값

`children` 이 `null` 이거나 `undefined` 일 땐 해당 값을 반환합니다.

그렇지 않은 경우 `fn` 함수에서 반환한 노드들로 구성된 평면 배열을 반환합니다. 반환된 배열은 `null` 과 `undefined` 를 제외하고 반환된 노드를 모두 포함합니다.

주의 사항

- 빈 노드(`null` , `undefined` 혹은 Boolean), 문자열, 숫자, React 엘리먼트는 개별 노드로 간주합니다. 배열 자체는 개별 노드가 아니지만 배열의 자식 요소는 개별 노드로 간주합니다. **React 엘리먼트의 하위 요소는 순회하지 않습니다.** React 엘리먼트는 렌더링 되지 않으며 자식 요소를 순회하지 않습니다. Fragments 역시 순회하지 않습니다.
- `fn` 에서 key를 가진 엘리먼트(혹은 엘리먼트의 배열)을 반환하는 경우, **반환된 엘리먼트의 key는 `children` 의 원본 항목의 key와 자동으로 결합됩니다.** `fn` 에서 배열로 여러 엘리먼트를 반환하는 경우, 각 엘리먼트의 key는 서로 간에만 고유하면 됩니다.

`Children.only(children)`

`Children.only(children)` 은 `children` 이 단일 React 엘리먼트인지 확인합니다.

```
function Box({ children }) {
  const element = Children.only(children);
  // ...
```

매개변수

- `children` : 컴포넌트에서 받은 `children` Prop의 값.

반환값

`children` 이 유효한 엘리먼트라면 그 엘리먼트를 반환합니다.
그렇지 않다면, 에러를 던집니다.

주의 사항

- 이 메서드는 **`children` 으로 배열 (예를 들어 `Children.map` 의 반환 값)을 넘기면 항상 예외를 발생시킵니다.** 다시 말해 `children` 은 단일 엘리먼트의 배열이 아니라 단일 React 엘리먼트여야 합니다.

`Children.toArray(children)`

`Children.toArray(children)` 은 `children` 데이터 구조로부터 배열을 생성합니다.

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

export default function ReversedList({ children }) {
  const result = Children.toArray(children);
  result.reverse();
  // ...
```

매개변수

- `children` : 컴포넌트에서 받은 `children` Prop의 값.

반환값

`children` 에 속한 엘리먼트를 평면 배열로 반환합니다.

주의 사항

- 빈 노드(`null` , `undefined` , 혹은 Boolean)는 반환된 배열에서 생략됩니다. **반환된 엘리먼트의 key는 기존 엘리먼트의 key, 중첩 수준과 위치를 기준으로 계산되므로** 배열을 평면화하더라도 동작이 변경되지 않습니다.

사용법

`children` 변환하기

`Children.map` 은 `children` Prop로 받은 JSX를 변환합니다.

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

function RowList({ children }) {
  return (
    <div className="RowList">
      {Children.map(children, child =>
        <div className="Row">
          {child}
        </div>
      )}
    </div>
  );
}
```

위 예시에서 `RowList`는 모든 자식 요소를 `<div className="Row">`로 감싸줍니다. 부모 컴포넌트가 `RowList`에 세 개의 `<p>` 태그를 `children` Prop로 넘겨준다고 가정해 봅시다.

```
<RowList>
  <p>This is the first item.</p>
  <p>This is the second item.</p>
  <p>This is the third item.</p>
</RowList>
```

위에 나온 `RowList` 구현을 통해 렌더링 된 최종 결과는 다음과 같습니다.

```
<div className="RowList">
  <div className="Row">
    <p>This is the first item.</p>
  </div>
  <div className="Row">
    <p>This is the second item.</p>
  </div>
  <div className="Row">
    <p>This is the third item.</p>
  </div>
</div>
```

`Children.map`은 `map()`을 사용해 배열을 변환하는 것과 유사하지만, `children` 데이터 구조가 *불분명하게* 취급된다는 차이가 있습니다. 즉, 배열일 수 있다고 하더라도 항상 배열이거나 다른 특정한 데이터 타입일 것이라고 가정해서는 안 됩니다. 그렇기 때문에 `children`을 변환할 때는 `Children.map`을 사용해야 합니다.

App.js RowList.js 새로고침 Clear 포크

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

export default function RowList({ children }) {
  return (
    <div className="RowList">
      {Children.map(children, child =>
        <div className="Row">
          {child}
        </div>
      )}
    </div>
  );
```

자세히 살펴보기

`children` Prop는 왜 항상 배열이 아닌가요?

자세히 보기

주의하세요!

`children` 데이터 구조는 JSX로 전달된 컴포넌트의 **렌더링 결과를 포함하지 않습니다.** 아래 예시에서 `RowList`가 받은 `children`에는 세 개가 아닌 두 개의 아이템만 포함합니다.

1. `<p>This is the first item.</p>`
2. `<MoreRows />`

그렇기 때문에 아래 예시에서는 두 개의 래퍼만 생성합니다.

App.js RowList.js 새로고침 Clear 포크

```
import RowList from './RowList.js';

export default function App() {
  return (
    <RowList>
      <p>This is the first item.</p>
      <MoreRows />
    </RowList>
  );
}

function MoreRows() {
```

자세히 보기

`children` 을 조작할 때 `<MoreRows />` 와 같은 **내부 컴포넌트의 렌더링 된 결과에 접근할 방법은 없습니다.** 그렇기 때문에 대안을 사용하는 것이 좋습니다.

각 자식 요소에서 코드 실행하기

`Children.forEach` 는 `children` 데이터 구조의 각 자식 요소를 반복합니다. 반환되는 값은 없고 배열의 `forEach` 메서드와 유사합니다. 자체 배열을 구성하는 등 커스텀 로직을 실행할 때 사용할 수 있습니다.

App.js SeparatorList.js 새로고침 Clear 포크

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

export default function SeparatorList({ children }) {
  const result = [];
  Children.forEach(children, (child, index) => {
    result.push(child);
    result.push(<hr key={index} />);
  });
  result.pop(); // Remove the last separator
  return result;
}
```

주의하세요!

위에서 언급했듯 `children` 을 조작할 때 내부 컴포넌트의 렌더링 된 결과에 접근할 방법은 없습니다. 그렇기 때문에 대안을 사용하는 것이 좋습니다.

`children` 카운팅하기

`Children.count(children)` 는 자식 요소의 수를 계산합니다.

App.js RowList.js 새로고침 Clear 포크

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

export default function RowList({ children }) {
  return (
    <div className="RowList">
      <h1 className="RowListHeader">
        Total rows: {Children.count(children)}
      </h1>
      {Children.map(children, child =>
```

```
            <div className="Row">
              {child}
```

자세히 보기

주의하세요!

위에서 언급했듯 `children`을 조작할 때 내부 컴포넌트의 렌더링 된 결과에 접근할 방법은 없습니다. 그렇기 때문에 대안을 사용하는 것이 좋습니다.

`children` 배열로 병합하기

`Children.toArray(children)`는 `children` 데이터 구조를 일반적인 자바스크립트 배열로 변경합니다. 이것을 사용해서 `filter`, `sort`, `reverse`와 같은 배열의 내장 메서드를 조작할 수 있습니다.

App.js ReversedList.js 새로고침 Clear 포크

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

export default function ReversedList({ children }) {
  const result = Children.toArray(children);
  result.reverse();
  return result;
}
```

주의하세요!

위에서 언급했듯 `children` 을 조작할 때 내부 컴포넌트의 렌더링 된 결과에 접근할 방법은 없습니다. 그렇기 때문에 대안을 사용하는 것이 좋습니다.

대안

중요합니다!

아래와 같이 사용되는 `Children` (대문자 `C`) API의 대안에 대한 설명입니다.

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

소문자 `c` 인 `children` Prop와 혼동해서는 안 됩니다. `children` Prop는 좋은 방법이고 권장되는 방식입니다.

여러 컴포넌트 노출하기

`Children` 메서드로 자식 요소를 조작하는 코드는 취약할 수 있습니다. JSX에서 컴포넌트에 자식 요소를 전달할 때 해당 컴포넌트가 개별 자식 요소를 조작하거나 변환될 수 있기 때문입니다.

가능한 한 `Children` 메서드는 사용하지 않는 것이 좋습니다. 예를 들어 `RowList` 의 각 자식 요소를 `<div className="Row">` 로 감싸려면, `Row` 컴포넌트를 내보내고 다음과 같이 각 row를 직접 감싸는 것을 권장합니다.

App.js RowList.js 새로고침 Clear 포크

```
import { RowList, Row } from './RowList.js';

export default function App() {
  return (
    <RowList>
      <Row>
        <p>This is the first item.</p>
      </Row>
      <Row>
        <p>This is the second item.</p>
      </Row>
      <Row>
```

자세히 보기

`Children.map`과 달리 이 방식은 모든 자식 요소를 자동으로 감싸지 않습니다. **그러나 `Children.map`의 앞선 예시와 달리 더 많은 컴포넌트를 추출해도 동작한다는 중요한 이점이 있습니다.**

예를 들어 `MoreRows` 컴포넌트를 직접 추출하더라도 동작합니다.

App.js RowList.js 새로고침 Clear 포크

```
import { RowList, Row } from './RowList.js';

export default function App() {
  return (
    <RowList>
      <Row>
        <p>This is the first item.</p>
      </Row>
      <MoreRows />
    </RowList>
  );
}
```

자세히 보기

`<MoreRows />` 가 단일 자식 요소이자 단일 row로 취급되기 때문에 `Children.map` 은 동작하지 않습니다.

Prop로 객체 배열 받기

Prop로 명시적으로 배열을 전달할 수 있습니다. 예를 들어, 아래 예시에서 `RowList` 는 `rows` 배열을 Prop로 받습니다.

App.js RowList.js 새로고침 Clear 포크

```
import { RowList, Row } from './RowList.js';

export default function App() {
  return (
    <RowList rows={[
      { id: 'first', content: <p>This is the first item.</p> },
      { id: 'second', content: <p>This is the second item.</p> },
      { id: 'third', content: <p>This is the third item.</p> }
    ]} />
  );
}
```

`rows` 는 일반적인 자바스크립트 배열이기 때문에, `RowList` 컴포넌트는 `map` 과 같은 내장 배열 메서드를 사용할 수 있습니다.

이 패턴은 구조화된 데이터를 자식 요소와 함께 전달할 때 더 유용합니다. 아래 예시에서 `TabSwitcher` 컴포넌트는 `tabs` Prop로 객체 배열을 받습니다.

App.js TabSwitcher.js 새로고침 Clear 포크

```
import TabSwitcher from './TabSwitcher.js';

export default function App() {
  return (
    <TabSwitcher tabs={[
      {
        id: 'first',
        header: 'First',
        content: <p>This is the first item.</p>
      },
      {
        id: 'second',
```

자세히 보기

JSX로 자식 요소를 전달할 때와 달리 이런 방식은 각 아이템에 `header` 와 같은 추가 정보를 연결할 수 있습니다. `tabs` 가 배열 구조이고 직접 다뤄지기 때문에 `Children` 메서드는 필요하지 않습니다.

렌더링 Prop로 렌더링 커스텀하기

모든 개별 항목에 대해 JSX를 생성하는 대신 JSX를 반환하는 함수를 전달하고 필요할 때 해당 함수를 호출할 수도 있습니다. 아래 예시에서 `App` 컴포넌트는 `renderContent` 함수를 `TabSwitcher` 컴포넌트에 전달합니다. `TabSwitcher` 컴포넌트는 선택된 탭에 대해서만 `renderContent` 를 호출합니다.

App.js TabSwitcher.js 새로고침 Clear 포크

```
import TabSwitcher from './TabSwitcher.js';

export default function App() {
  return (
    <TabSwitcher
      tabIds={['first', 'second', 'third']}
      getHeader={tabId => {
        return tabId[0].toUpperCase() + tabId.slice(1);
      }}
      renderContent={tabId => {
        return <p>This is the {tabId} item.</p>;
      }}
```

`renderContent` 와 같이 사용자 인터페이스의 일부를 어떻게 렌더링할지 정의하는 Prop를 *렌더링 Prop*라고 합니다. 하지만 특별한 것은 아닙니다. 단지 일반적인 함수의 Prop일 뿐입니다.

렌더링 Prop은은 함수이므로 정보를 전달할 수 있습니다.
아래 예시에서 `RowList` 컴포넌트는 각 row의 `id` 와 `index` 를 `renderRow` 에 렌더링 Prop로 전달하고, `index` 가 짝수인 row를 강조합니다.

App.js RowList.js 새로고침 Clear 포크

```
import { RowList, Row } from './RowList.js';

export default function App() {
  return (
    <RowList
      rowIds={['first', 'second', 'third']}
      renderRow={(id, index) => {
        return (
          <Row isHighlighted={index % 2 === 0}>
            <p>This is the {id} item.</p>
          </Row>
        );
```

자세히 보기

부모 컴포넌트와 자식 컴포넌트가 자식 요소를 직접 조작하지 않고도 효과적으로 협력할 수 있는 좋은 예시입니다.

문제 해결

커스텀 컴포넌트를 전달했을 때 `Children` 메서드가 렌더링 결과를 보여주지 않는 경우

`RowList` 에 두 개의 자식 요소를 아래와 같이 전달했다고 가정해 봅시다.

```
<RowList>
  <p>First item</p>
  <MoreRows />
</RowList>
```

`RowList` 내부에서 `Children.count(children)` 를 실행시킨다면 결과는 `2` 입니다. `MoreRows` 가 10개의 다른 요소를 렌더링하거나 `null` 을 반환하더라도, `Children.count(children)` 는 `2` 입니다. `RowList` 관점에서는 그것이 받은 JSX만 고려할 뿐 `MoreRows` 컴포넌트의 내부는 고려하지 않습니다.

이런 제약때문에 컴포넌트를 추출하기 어려울 수 있습니다. 그렇기 때문에 `Children` 대신 대안을 사용하는 것이 좋습니다.

이전
레거시 React API

다음
cloneElement