



Component



주의하세요!

컴포넌트를 클래스 대신 함수로 정의하는 것을 추천합니다. [マイグレーション方法を確認하세요。](#)

Component는 [자바스크립트 클래스](#)로 정의된 React 컴포넌트의 기본 클래스입니다. React에서 클래스 컴포넌트를 계속 지원하지만, 새 코드에서는 사용하지 않는 것을 추천합니다.

```
class Greeting extends Component {
  render() {
    return <h1>Hello, {this.props.name}!</h1>;
  }
}
```

- [레퍼런스](#)

- [Component](#)
- [context](#)
- [props](#)
- [state](#)
- [constructor\(props\)](#)
- [componentDidCatch\(error, info\)](#)
- [componentDidMount\(\)](#)
- [componentDidUpdate\(prevProps, prevState, snapshot?\)](#)

- `componentWillMount()`
- `componentWillReceiveProps(nextProps)`
- `componentWillUpdate(nextProps, nextState)`
- `componentWillUnmount()`
- `forceUpdate(callback?)`
- `getSnapshotBeforeUpdate(prevProps, prevState)`
- `render()`
- `setState(nextState, callback?)`
- `shouldComponentUpdate(nextProps, nextState, nextContext)`
- `UNSAFE_componentWillMount()`
- `UNSAFE_componentWillReceiveProps(nextProps, nextContext)`
- `UNSAFE_componentWillUpdate(nextProps, nextState)`
- `static contextType`
- `static defaultProps`
- `static getDerivedStateFromError(error)`
- `static getDerivedStateFromProps(props, state)`
- 사용법
 - 클래스 컴포넌트 정의하기
 - 클래스 컴포넌트에 state 추가하기
 - 클래스 컴포넌트에 생명주기 메서드 추가하기
 - Error Boundary로 렌더링 오류 포착하기
- 대안
 - 클래스에서 함수로 간단한 컴포넌트 마이그레이션하기
 - state가 있는 컴포넌트를 클래스에서 함수로 마이그레이션하기
 - 생명주기 메서드가 있는 컴포넌트를 클래스에서 함수로 마이그레이션하기
 - context가 있는 컴포넌트를 클래스에서 함수로 마이그레이션하기

레퍼런스

Component

React 컴포넌트를 클래스로 정의하려면, 내장 Component 클래스를 확장하고 [render 메서드](#)를 정의하세요.

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

class Greeting extends Component {
  render() {
    return <h1>Hello, {this.props.name}!</h1>;
  }
}
```

render 메서드만 필수고 다른 메서드는 선택 사항입니다.

[아래의 더 많은 예시를 확인하세요.](#)

context

클래스 컴포넌트의 [context](#)는 `this.context`로 사용할 수 있습니다. [static contextType](#)를 사용하여 어떤 context를 받을지 지정해야만 사용할 수 있습니다.

클래스 컴포넌트는 한 번에 하나의 context만 읽을 수 있습니다.

```
class Button extends Component {
  static contextType = ThemeContext;

  render() {
    const theme = this.context;
    const className = 'button-' + theme;
    return (
      <button className={className}>
        {this.props.children}
      </button>
    );
  }
}
```

▣ 중요합니다!

클래스 컴포넌트에서 `this.context` 를 읽는 것은 함수 컴포넌트에서 `useContext` 와 같습니다.

마이그레이션 방법을 확인하세요.

props

클래스 컴포넌트에 전달되는 `props`는 `this.props` 로 사용할 수 있습니다.

```
class Greeting extends Component {
  render() {
    return <h1>Hello, {this.props.name}!</h1>;
  }
}

<Greeting name="Taylor" />
```

▣ 중요합니다!

클래스 컴포넌트에서 `this.props` 를 읽는 것은 함수 컴포넌트에서 `props`를 선언하는 것과 같습니다.

마이그레이션 방법을 확인하세요.

state

클래스 컴포넌트의 state는 this.state로 사용할 수 있습니다. state 필드는 반드시 객체여야 합니다. state를 직접 변경하지 마세요. state를 변경하려면 새 state로 setState를 호출하세요.

```
class Counter extends Component {  
  state = {  
    age: 42,  
  };  
  
  handleAgeChange = () => {  
    this.setState({  
      age: this.state.age + 1  
    });  
  };  
  
  render() {  
    return (  
      <>  
        <button onClick={this.handleAgeChange}>  
          Increment age  
        </button>  
        <p>You are {this.state.age}.</p>  
      </>  
    );  
  }  
}
```

▣ 중요합니다!

클래스 컴포넌트에서 state를 정의하는 것은 함수 컴포넌트에서 useState를 호출하는 것과 같습니다.

マイ그레이션 방법을 확인하세요.

constructor(props)

클래스 컴포넌트가 마운트(화면에 추가됨)되기 전에 `constructor`가 실행됩니다. 일반적으로 `constructor`는 React에서 두 가지 목적으로만 사용됩니다. `state`를 선언하고 클래스 메서드를 클래스 인스턴스에 [바인딩](#)할 수 있습니다.

```
class Counter extends Component {  
  constructor(props) {  
    super(props);  
    this.state = { counter: 0 };  
    this.handleClick = this.handleClick.bind(this);  
  }  
  
  handleClick() {  
    // ...  
  }  
}
```

최신 자바스크립트 문법을 사용한다면 `constructor`는 거의 필요하지 않습니다. 대신 최신 브라우저와 [Babel](#)과 같은 도구에서 모두 지원되는 [공용 클래스 필드 문법](#)을 사용하여 위의 코드를 다시 작성할 수 있습니다.

```
class Counter extends Component {  
  state = { counter: 0 };  
  
  handleClick = () => {  
    // ...  
  }  
}
```

`constructor`는 부수 효과 또는 구독을 포함하면 안됩니다.

매개변수

- `props`: 컴포넌트의 초기 `props`.

반환값

`constructor`는 아무것도 반환하면 안 됩니다.

주의 사항

- constructor에서 부수 효과 또는 구독을 실행하지 마세요. 대신 `componentDidMount` 를 사용하세요.
- constructor 내부에서는 다른 명령어보다 `super(props)` 를 먼저 호출해야 합니다. 그렇게 하지 않으면, constructor가 실행되는 동안 `this.props` 는 `undefined` 가 되어 혼란스럽고 버그가 발생할 수 있습니다.
- constructor는 `this.state` 를 직접 할당할 수 있는 유일한 위치입니다. 다른 모든 메서드에서는 `this.setState()` 를 대신 사용해야 합니다. constructor에서 `setState` 를 호출하지 마십시오.
- 서버 렌더링을 사용할 때, constructor는 서버에서 역시 실행되고, 뒤이어 `render` 메서드도 실행됩니다. 그러나 `componentDidMount` 또는 `componentWillUnmount` 와 같은 수명 주기 메서드는 서버에서 실행되지 않습니다.
- [Strict 모드](#)가 설정되면 React는 개발 중인 constructor 를 두 번 호출한 다음 인스턴스 중 하나를 삭제합니다. 이를 통해 constructor 외부로 옮겨져야 하는 우발적인 부수 효과를 파악할 수 있습니다.

▣ 중요합니다!

함수 컴포넌트에 `constructor` 와 정확히 동등한 것은 없습니다. 함수 컴포넌트에 `state` 를 선언하려면 `useState` 를 호출하세요. 초기 `state` 를 다시 계산하지 않으려면 [함수를 useState 에 전달하세요](#).

`componentDidCatch(error, info)`

`componentDidCatch` 를 정의하면, 일부 자식 컴포넌트(먼 자식을 포함)가 에러를 발생시킬 때 React가 이를 호출합니다. 이를 통해 운영 중인 에러 보고 서비스에 에러를 기록할 수 있습니다.

일반적으로, 에러에 대한 응답으로 State를 업데이트하고 사용자에게 에러 메시지를 표시할 수 있는 `static getDerivedStateFromError` 와 함께 사용합니다. 이런 여러 메서드를 사용하는 컴포넌트를 *Error Boundary*라고 합니다.

[예시를 확인하세요](#).

매개변수

- `error`: 발생한 에러입니다. 실제로, 보통은 `에러`의 인스턴스가 되지만 JavaScript에서 문자열 또는 `null`을 포함한 어떤 값이든 `throw` 할 수 있기 때문에 보장되지 않습니다.
- `info`: 에러에 대한 추가 정보를 포함하는 객체입니다. 이것의 `componentStack` 필드는 모든 부모 컴포넌트의 이름과 출처 위치뿐만 아니라 에러를 `throw`한 컴포넌트의 `stack trace`을 포함합니다. 프로덕션에서, 컴포넌트의 이름은 최소화됩니다. 프로덕션 에러 보고를 설정하면 일반 JavaScript 에러 스택과 동일한 방법으로 소스맵을 사용하여 컴포넌트 스택을 디코딩할 수 있습니다.

반환값

`componentDidCatch` 는 아무것도 반환하면 안 됩니다.

주의 사항

- 과거에는 UI를 업데이트하고 대체 에러 메세지를 표시하기 위해 `setState` 를 `componentDidCatch` 안에서 호출하는 것이 일반적이었습니다. 이는 `static getDerivedStateFromError` 를 정의하기 위해 더 이상 사용되지 않습니다.
- React의 프로덕션과 개발 빌드는 `componentDidCatch` 가 에러를 처리하는 방식이 약간 다릅니다. 개발에서는, 에러는 `window` 까지 버블링될 것이며, 이는 `window.onerror` 또는 `window.addEventListener('error', callback)` 가 `componentDidCatch` 에 의해 탐지된 에러를 가로챈다는 것을 의미합니다. 대신 프로덕션에서, 에러는 버블링되지 않을 것이며, 이는 어떤 상위의 에러 핸들러가 `componentDidCatch` 에 의해 명시적으로 탐지되지 않은 에러만을 수신하는 것을 의미합니다.

▣ 중요합니다!

함수 컴포넌트에 `componentDidCatch` 와 직접적으로 동등한 것은 아직 없습니다. 클래스 컴포넌트 생성을 피하려면, 위와 같이 `ErrorBoundary` 컴포넌트를 하나 작성하여 앱 전체에 사용합니다. 그렇지 않으면, 그것을 해주는 `react-error-boundary package`를 사용할 수 있습니다.

componentDidMount()

componentDidMount 메서드를 정의하면 구성 요소가 화면에 추가 (마운트) 될 때 React가 호출합니다. 이것은 데이터 가져오기를 시작하거나, 구독을 설정하거나, DOM 노드를 조작하는 일반적인 장소입니다.

componentDidMount 를 구현하면 일반적으로 버그를 방지하기 위해 다른 생명주기 메서드를 구현해야 합니다. 예를 들어, componentDidMount 가 일부 state나 props를 읽는 경우 변경 사항을 처리하기 위해 componentDidUpdate 를 구현하고 componentDidMount 가 수행하던 작업을 정리하기 위해 componentWillUnmount 도 구현해야 합니다.

```
class ChatRoom extends Component {
  state = {
    serverUrl: 'https://localhost:1234'
  };

  componentDidMount() {
    this.setupConnection();
  }

  componentDidUpdate(prevProps, prevState) {
    if (
      this.props.roomId !== prevProps.roomId ||
      this.state.serverUrl !== prevState.serverUrl
    ) {
      this.destroyConnection();
      this.setupConnection();
    }
  }

  componentWillUnmount() {
    this.destroyConnection();
  }

  // ...
}
```

더 많은 예시를 확인하세요.

매개변수

componentDidMount 는 매개변수를 사용하지 않습니다.

반환값

componentDidMount 는 아무것도 반환하면 안 됩니다.

주의 사항

- Strict 모드가 켜져 있으면 개발 중인 React가 componentDidMount 를 호출한 다음 componentWillMount 를 호출하고 componentDidMount 를 다시 호출합니다. 이를 통해 componentWillMount 를 구현하는 것을 잊었거나 로직이 componentDidMount 가 수행하는 작업을 완전히 “미러링”하지 않는 경우를 알 수 있습니다.
- componentDidMount 에서 setState 를 즉시 호출할 수 있지만, 가능하면 피하는 것이 가장 좋습니다. 이는 추가 렌더링을 일으키지만 브라우저가 화면을 업데이트하기 전에 일어납니다. 이 경우 render 가 두 번 호출되더라도 사용자는 중간 state 를 볼 수 없습니다. 이 패턴은 종종 성능 문제를 발생시키므로 주의하여 사용하십시오. 대부분의 경우 constructor 에서 초기 state 를 대신 할당할 수 있습니다. 그러나 모달이나 툴팁과 같이 크기나 위치에 따라 달라지는 것을 렌더링하기 전에 DOM 노드를 측정해야 하는 경우에는 필요할 수 있습니다.

▣ 중요합니다!

많은 사용 사례에서 componentDidMount , componentDidUpdate 및 componentWillMount 를 클래스 컴포넌트에 함께 정의하는 것은 함수 컴포넌트에서 useEffect 를 호출하는 것과 같습니다. 드물지만 브라우저 페인트 전에 코드를 실행하는 것이 중요한 경우에는 useLayoutEffect 를 사용하는 것이 더 적합합니다.

マイ그레이션 방법을 확인하세요.

componentDidUpdate(prevProps, prevState, snapshot?)

`componentDidUpdate` 메서드를 정의하면 컴포넌트가 업데이트된 `props` 또는 `state`로 다시 렌더링된 직후 React가 이 메서드를 호출합니다. 이 메서드는 초기 렌더링에 호출되지 않습니다.

업데이트 후 DOM을 조작하는 데 사용할 수 있습니다. 또한 현재 `props`를 이전 `props`와 비교하는 한 네트워크 요청을 수행하는 일반적인 장소이기도 합니다(예: `props`가 변경되지 않은 경우 네트워크 요청이 필요하지 않을 수 있습니다). 일반적으로 `componentDidMount` 및 `componentWillUnmount` 와 함께 사용합니다.

```
class ChatRoom extends Component {
  state = {
    serverUrl: 'https://localhost:1234'
  };

  componentDidMount() {
    this.setupConnection();
  }

  componentDidUpdate(prevProps, prevState) {
    if (
      this.props.roomId !== prevProps.roomId ||
      this.state.serverUrl !== prevState.serverUrl
    ) {
      this.destroyConnection();
      this.setupConnection();
    }
  }

  componentWillUnmount() {
    this.destroyConnection();
  }

  // ...
}
```

더 많은 예시를 확인하세요.

매개변수

- `prevProps`: 업데이트 이전의 `props`. `prevProps` 와 `this.props` 를 비교하여 변경된 내용을 확인합니다.

- prevState: 업데이트 전 state. prevState 를 `this.state` 와 비교하여 변경된 내용을 확인합니다.
- snapshot: `getSnapshotBeforeUpdate` 를 구현한 경우, snapshot 에는 해당 메서드에서 반환한 값이 포함됩니다. 그렇지 않으면 `undefined` 가 됩니다.

반환값

`componentDidUpdate` 는 아무것도 반환하지 않아야 합니다.

주의 사항

- `shouldComponentUpdate` 가 정의되어 있으면 `componentDidUpdate` 가 호출되지 않고 `false` 를 반환합니다.
- `componentDidUpdate` 내부의 로직은 일반적으로 `this.props` 를 `prevProps` 와 비교하고 `this.state` 를 `prevState` 와 비교하는 조건으로 래핑해야 합니다. 그렇지 않으면 무한 루프가 생성될 위험이 있습니다.
- `componentDidUpdate` 에서 `setState` 를 즉시 호출할 수도 있지만 가능하면 피하는 것이 가장 좋습니다. 추가 렌더링이 트리거되지만 브라우저가 화면을 업데이트하기 전에 발생합니다. 이렇게 하면 이 경우 `render` 가 두 번 호출되더라도 사용자에게 중간 state가 표시되지 않습니다. 이 패턴은 종종 성능 문제를 일으키지만 드물게 모달이나 툴팁처럼 크기나 위치에 따라 달라지는 것을 렌더링하기 전에 DOM 노드를 측정해야 하는 경우에 필요할 수 있습니다.

▣ 중요합니다!

많은 사용 사례에서 `componentDidMount`, `componentDidUpdate` 및 `componentWillUnmount` 를 클래스 컴포넌트에 함께 정의하는 것은 함수 컴포넌트에서 `useEffect` 를 호출하는 것과 같습니다. 드물지만 브라우저 페인트 전에 코드를 실행하는 것이 중요한 경우에는 `useLayoutEffect` 를 사용하는 것이 더 적합합니다.

マイグ레이션 방법을 확인하세요.

componentWillMount()

➊ 더 이상 사용되지 않습니다!

이 API의 이름이 `componentWillMount`에서 `UNSAFE_componentWillMount`로 변경되었습니다. 이전 이름은 더 이상 사용되지 않습니다. 향후 React의 주요 버전에서는 새로운 이름만 작동합니다.

컴포넌트를 자동으로 업데이트하려면 [rename-unsafe-lifecycles codemod](#)를 실행하세요.

`componentWillReceiveProps(nextProps)`

➊ 더 이상 사용되지 않습니다!

이 API의 이름이 `componentWillReceiveProps`에서 `UNSAFE_componentWillReceiveProps`로 변경되었습니다. 이전 이름은 더 이상 사용되지 않습니다. 향후 React의 주요 버전에서는 새로운 이름만 작동합니다.

컴포넌트를 자동으로 업데이트하려면 [rename-unsafe-lifecycles codemod](#)를 실행하세요.

`componentWillUpdate(nextProps, nextState)`

➊ 더 이상 사용되지 않습니다!

이 API의 이름이 `componentWillUpdate`에서 `UNSAFE_componentWillUpdate`로 변경되었습니다. 이전 이름은 더 이상 사용되지 않습니다. 향후 React의 주요 버전에서는 새로운 이름만 작동합니다.

컴포넌트를 자동으로 업데이트하려면 [rename-unsafe-lifecycles codemod](#)를 실행하세요.

componentWillUnmount()

componentWillUnmount 메서드를 정의하면 React는 컴포넌트가 화면에서 제거 (마운트 해제) 되기 전에 이 메서드를 호출합니다. 이는 데이터 불러오기를 취소하거나 구독을 제거하는 일반적인 장소입니다.

componentWillUnmount 내부의 로직은 [componentDidMount](#) 내부의 로직을 “미러링”해야 합니다. 예를 들어 componentDidMount 가 구독을 설정하면 componentWillUnmount 는 해당 구독을 정리해야 합니다. componentWillUnmount 의 정리 로직이 일부 props나 state를 읽는 경우, 일반적으로 이전 props나 state에 해당하는 리소스(예: 구독)를 정리하기 위해 [componentDidUpdate](#) 도 구현해야 합니다.

```
class ChatRoom extends Component {
  state = {
    serverUrl: 'https://localhost:1234'
  };

  componentDidMount() {
    this.setupConnection();
  }

  componentDidUpdate(prevProps, prevState) {
    if (
      this.props.roomId !== prevProps.roomId ||
      this.state.serverUrl !== prevState.serverUrl
    ) {
      this.destroyConnection();
      this.setupConnection();
    }
  }

  componentWillUnmount() {
    this.destroyConnection();
  }
}
```

```
// ...  
}
```

더 많은 예시를 확인하세요.

매개변수

`componentWillUnmount` 는 어떤 매개변수도 받지 않습니다.

반환값

`componentWillUnmount` 는 아무것도 반환하지 않아야 합니다.

주의 사항

- `Strict 모드`가 켜져 있으면 개발 시 React는 `componentDidMount` 를 호출한 다음 즉시 `componentWillUnmount` 를 호출한 다음 `componentDidMount` 를 다시 호출합니다. 이렇게 하면 `componentWillUnmount` 를 구현하는 것을 잊어버렸거나 그 로직이 `componentDidMount` 의 동작을 완전히 “미러링”하지 않는지 확인할 수 있습니다.

▣ 중요합니다!

많은 사용 사례에서 `componentDidMount`, `componentDidUpdate` 및 `componentWillUnmount` 를 클래스 컴포넌트에 함께 정의하는 것은 함수 컴포넌트에서 `useEffect` 를 호출하는 것과 같습니다. 드물지만 브라우저 페인트 전에 코드를 실행하는 것이 중요한 경우에는 `useLayoutEffect` 를 사용하는 것이 더 적합합니다.

マイグ레이션 방법을 확인하세요.

forceUpdate(callback?)

컴포넌트를 강제로 다시 렌더링합니다.

일반적으로는 필요하지 않습니다. 컴포넌트의 `render` 메서드가 `this.props`, `this.state` 또는 `this.context`에서만 읽는 경우, 컴포넌트 내부 또는 부모 중 하나에서 `setState`를 호출하면 자동으로 다시 렌더링됩니다. 하지만 컴포넌트의 `render` 메서드가 외부 데이터 소스로부터 직접 읽어오는 경우, 데이터 소스가 변경될 때 사용자 인터페이스를 업데이트하도록 React에 지시해야 합니다. 이것이 바로 `forceUpdate` 가 할 수 있는 일입니다.

`forceUpdate`의 모든 사용을 피하고 `render`에서 `this.props`와 `this.state`로부터만 읽도록 하세요.

매개변수

- **optional callback**: 지정한 경우 React는 업데이트가 커밋된 후 사용자가 제공한 callback을 호출합니다.

반환값

`forceUpdate`은 아무것도 반환하지 않습니다.

주의 사항

- `forceUpdate`를 호출하면 React는 `shouldComponentUpdate`를 호출하지 않고 다시 렌더링합니다.

▣ 중요합니다!

외부 데이터 소스를 읽는 것과 `forceUpdate`를 사용하여 변경된 내용에 따라 클래스 컴포넌트를 다시 렌더링하도록 강제하는 것은 함수 컴포넌트에서 `useSyncExternalStore`로 대체되었습니다.

getSnapshotBeforeUpdate(prevProps, prevState)

`getSnapshotBeforeUpdate`를 구현하면 React가 DOM을 업데이트하기 바로 전에 호출합니다. 이를 통해 컴포넌트가 잠재적으로 변경되기 전에 DOM에서 일부 정보(예: 스크롤 위치)를 캡처할

수 있습니다. 이 생명주기 메서드가 반환하는 모든 값은 `componentDidUpdate`에 매개변수로 전달됩니다.

예를 들어 업데이트 중에 스크롤 위치를 유지해야 하는 채팅 스레드와 같은 UI에서 이 메서드를 사용할 수 있습니다.

```
class ScrollingList extends React.Component {
  constructor(props) {
    super(props);
    this.listRef = React.createRef();
  }

  getSnapshotBeforeUpdate(prevProps, prevState) {
    // 목록에 새 항목을 추가하고 있나요?
    // 나중에 스크롤을 조정할 수 있도록 스크롤 위치를 캡처합니다.
    if (prevProps.list.length < this.props.list.length) {
      const list = this.listRef.current;
      return list.scrollHeight - list.scrollTop;
    }
    return null;
  }

  componentDidUpdate(prevProps, prevState, snapshot) {
    // 스냅샷 값이 있으면 방금 새 항목을 추가한 것입니다.
    // 새 항목이 기존 항목을 시야 밖으로 밀어내지 않도록 스크롤을 조정합니다.
    // (여기서 스냅샷은 getSnapshotBeforeUpdate에서 반환된 값입니다.)
    if (snapshot !== null) {
      const list = this.listRef.current;
      list.scrollTop = list.scrollHeight - snapshot;
    }
  }

  render() {
    return (
      <div ref={this.listRef}>{/* ...contents... */}</div>
    );
  }
}
```

위의 예시에서는 `getSnapshotBeforeUpdate`에서 직접 `scrollHeight` 속성을 읽는 것이 중요합니다. `render`, `UNSAFE_componentWillReceiveProps` 또는 `UNSAFE_componentWillUpdate`가 호출되는 시점과 React가 DOM을 업데이트하는 시점 사이에 잠재적인 시간 간격이 있기 때문에 이러한 여러 메서드에서 이(역주: `scrollHeight`)를 읽는 것은 안전하지 않습니다.

매개변수

- `prevProps`: 업데이트 이전의 `props`. `prevProps` 와 `this.props` 를 비교하여 변경된 내용을 확인합니다.
- `prevState`: 업데이트 전 `state`. `prevState` 를 `this.state` 와 비교하여 변경된 내용을 확인합니다.

반환값

원하는 유형의 스냅샷 값 또는 `null` 을 반환해야 합니다. 반환한 값은 `componentDidUpdate`의 세 번째 인자로 전달됩니다.

주의 사항

- `shouldComponentUpdate` 가 정의되어 있으면 `getSnapshotBeforeUpdate` 가 호출되지 않고 `false` 를 반환합니다.

▣ 중요합니다!

현재로서는 함수 컴포넌트에 대한 `getSnapshotBeforeUpdate` 와 동등한 함수가 없습니다. 이 사용 사례는 매우 드물지만, 이 기능이 필요한 경우 현재로서는 클래스 컴포넌트를 작성해야 합니다.

render()

`render` 메서드는 클래스 컴포넌트에서 유일하게 필요한 메서드입니다.

`render` 메서드는 화면에 표시할 내용을 지정해야 합니다, 예를 들어

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

class Greeting extends Component {
  render() {
    return <h1>Hello, {this.props.name}!</h1>;
  }
}
```

React는 언제든 render 를 호출할 수 있으므로 특정 시간에 실행된다고 가정하면 안 됩니다. 일 반적으로 render 메서드는 JSX를 반환해야 하지만 몇 가지 (문자열과 같은) 다른 반환 유형이 지원됩니다. 반환된 JSX를 계산하기 위해 render 메서드는 this.props, this.state 및 this.context 를 읽을 수 있습니다.

render 메서드는 순수 함수로 작성해야 합니다. 즉, props, state 및 context가 동일한 경우 동일한 결과를 반환해야 합니다. 또한 (구독 설정과 같은) 부수 효과를 포함하거나 브라우저 API와 상호작용하면 안 됩니다. 부수 효과는 이벤트 핸들러나 componentDidMount 와 같은 메서드에서 발생해야 합니다.

매개변수

- render : 어떤 매개변수도 받지 않습니다.

반환값

render 는 유효한 모든 React 노드를 반환할 수 있습니다. 여기에는 <div />, 문자열, 숫자, portals, 빈 노드(null, undefined, true, false) 및 React 노드의 배열과 같은 React 엘리먼트가 포함됩니다.

주의 사항

- render 는 props, state, context의 순수한 함수로 작성되어야 합니다. 부수 효과가 없어야 합니다.
- shouldComponentUpdate 가 정의되고 false 를 반환하면 render 가 호출되지 않습니다.
- Strict 모드가 켜져 있으면 React는 개발 과정에서 render 를 두 번 호출한 다음 결과 중 하나를 버립니다. 이렇게 하면 render 메서드에서 제거해야 하는 우발적인 부수 효과를 알아차릴 수 있습니다.

- render 호출과 후속 componentDidMount 또는 componentDidUpdate 호출 사이에는 일대 일 대응이 없습니다. render 호출 결과 중 일부는 유익할 때 React에 의해 버려질 수 있습니다.

setState(nextState, callback?)

setState 를 호출하여 React 컴포넌트의 state를 업데이트합니다.

```
class Form extends Component {  
  state = {  
    name: 'Taylor',  
  };  
  
  handleNameChange = (e) => {  
    const newName = e.target.value;  
    this.setState({  
      name: newName  
    });  
  }  
  
  render() {  
    return (  
      <>  
        <input value={this.state.name} onChange={this.handleNameChange} />  
        <p>Hello, {this.state.name}.</p>  
      </>  
    );  
  }  
}
```

setState 는 컴포넌트 state에 대한 변경 사항을 큐에 넣습니다. 이 컴포넌트와 그 자식이 새로운 state로 다시 렌더링해야 한다는 것을 React에게 알려줍니다. 이것이 상호작용에 반응하여 사용자 인터페이스를 업데이트하는 주요 방법입니다.

❗ 주의하세요!

`setState` 를 호출해도 이미 실행 중인 코드의 현재 state는 변경되지 않습니다.

```
function handleClick() {
  console.log(this.state.name); // "Taylor"
  this.setState({
    name: 'Robin'
  });
  console.log(this.state.name); // Still "Taylor"!
}
```

오로지 다음 렌더링부터 `this.state` 가 반환할 내용에만 영향을 줍니다.

`setState` 에 함수를 전달할 수도 있습니다. 이 함수를 사용하면 이전 state를 기반으로 state를 업데이트할 수 있습니다.

```
handleIncreaseAge = () => {
  this.setState(prevState => {
    return {
      age: prevState.age + 1
    };
  });
}
```

이 작업을 수행할 필요는 없지만 동일한 이벤트 중에 state를 여러 번 업데이트하려는 경우 유용합니다.

매개변수

- `nextState`: 객체 또는 함수 중 하나입니다.
 - 객체를 `nextState` 로 전달하면 `this.state` 에 얕게(shallowly) 병합됩니다.
 - 함수를 `nextState` 로 전달하면 업데이터 함수로 취급됩니다. 이 함수는 순수해야 하고, `pending state`와 `props`를 인자로 받아야 하며, `this.state` 에 얕게(shallowly) 병합할 객체를 반환해야 합니다. React는 업데이터 함수를 큐에 넣고 컴포넌트를 다시 렌더링합니다.

다. 다음 렌더링 중에 React는 큐에 있는 모든 업데이터를 이전 state에 적용하여 다음 state를 계산합니다.

- **optional** callback: 지정한 경우 React는 업데이트가 커밋된 후 사용자가 제공한 callback 을 호출합니다.

반환값

setState 는 아무것도 반환하지 않습니다.

주의 사항

- setState 를 컴포넌트를 업데이트하는 즉각적인 명령이 아닌 요청으로 생각하세요. 여러 컴포넌트가 이벤트에 반응하여 state를 업데이트하면 React는 업데이트를 batch하고 이벤트가 끝날 때 단일 패스로 함께 다시 렌더링합니다. 드물게 특정 state 업데이트를 강제로 동기화하여 적용해야 하는 경우, `flushSync` 로 래핑할 수 있지만, 이 경우 성능이 저하될 수 있습니다.
- setState 는 `this.state` 를 즉시 업데이트하지 않습니다. 따라서 setState 를 호출한 직후 `this.state` 를 읽는 것은 잠재적인 위험이 될 수 있습니다. 대신, 업데이트가 적용된 후에 실행되도록 보장되는 `componentDidUpdate` 또는 setState callback 인자를 사용하십시오. 이전 state를 기반으로 state를 설정해야 하는 경우 위에서 설명한 대로 함수를 nextState 에 전달할 수 있습니다.

▣ 중요합니다!

클래스 컴포넌트에서 setState 를 호출하는 것은 함수 컴포넌트에서 `set` 함수를 호출하는 것과 유사합니다.

マイ그레이션 방법을 확인하세요.

shouldComponentUpdate(nextProps, nextState, nextContext)

shouldComponentUpdate 를 정의하면 React가 이를 호출하여 재렌더링을 건너뛸 수 있는지 여부를 결정합니다.

직접 작성을 원하는 것이 확실하다면, `this.props` 를 `nextProps` 와, `this.state` 를 `nextState` 와 비교하고 `false` 를 반환하여 React에 업데이트를 건너뛸 수 있음을 알릴 수 있습니다.

```
class Rectangle extends Component {
  state = {
    isHovered: false
  };

  shouldComponentUpdate(nextProps, nextState) {
    if (
      nextProps.position.x === this.props.position.x &&
      nextProps.position.y === this.props.position.y &&
      nextProps.size.width === this.props.size.width &&
      nextProps.size.height === this.props.size.height &&
      nextState.isHovered === this.state.isHovered
    ) {
      // 변경된 사항이 없으므로 다시 렌더링할 필요가 없습니다.
      return false;
    }
    return true;
  }

  // ...
}
```

새로운 `props`나 `state`가 수신되면 렌더링하기 전에 React가 `shouldComponentUpdate` 를 호출합니다. 기본값은 `true` 입니다. 이 메서드는 초기 렌더링이나 `forceUpdate` 가 사용될 때는 호출되지 않습니다.

매개변수

- `nextProps`: 컴포넌트가 렌더링할 다음 `props`입니다. `nextProps` 와 `this.props` 를 비교하여 무엇이 변경되었는지 확인합니다.
- `nextState`: 컴포넌트가 렌더링할 다음 `state`입니다. `nextState` 와 `this.state` 를 비교하여 무엇이 변경되었는지 확인합니다.
- `nextContext`: 컴포넌트가 렌더링할 다음 `context`입니다. `nextContext` 를 `this.context` 와 비교하여 변경된 내용을 확인합니다. `static contextType (modern)` 을 지정한 경우에만

사용할 수 있습니다.

반환값

컴포넌트를 다시 렌더링하려면 `true` 를 반환합니다. 이것이 기본 동작입니다.

React에 재렌더링을 건너뛸 수 있음을 알리려면 `false` 를 반환합니다.

주의 사항

- 이 메서드는 오직 성능 최적화를 위해서만 존재합니다. 이 메서드 없이 컴포넌트가 중단되는 경우 먼저 그 문제를 해결하세요.
- `shouldComponentUpdate` 를 직접 작성하는 대신 `PureComponent` 를 사용하는 것을 고려하세요. `PureComponent` 는 `props`와 `state`를 얕게(shallowly) 비교하여 필요한 업데이트를 건너뛸 가능성을 줄여줍니다.
- `shouldComponentUpdate` 에서 완전 일치(deep equality) 검사를 하거나 `JSON.stringify` 를 사용하는 것은 권장하지 않습니다. 이는 성능을 예측할 수 없고 모든 `prop`과 `state`의 데이터 구조에 의존적이게 합니다. 최상의 경우 애플리케이션에 몇 초씩 멈추는 현상이 발생하고 최악의 경우 애플리케이션이 충돌할 위험이 있습니다.
- `false` 를 반환해도 자식 컴포넌트들에서 그들의 `state`가 변경될 때 다시 렌더링되는 것을 막지는 못합니다.
- `false` 를 반환한다고 해서 컴포넌트가 다시 렌더링되지 않는다는 보장은 없습니다. React는 반환값을 힌트로 사용하지만 다른 이유로 컴포넌트를 다시 렌더링하는 것이 합리적일 경우 여전히 렌더링을 선택할 수 있습니다.

▣ 중요합니다!

`shouldComponentUpdate` 로 클래스 컴포넌트를 최적화하는 것은 `memo` 로 함수 컴포넌트를 최적화하는 것과 유사합니다. 함수 컴포넌트는 `useMemo` 를 통해 더 세분화된 최적화도 제공합니다.

UNSAFE_componentWillMount()

`UNSAFE_componentWillMount` 를 정의하면 React는 `constructor` 바로 뒤에 이를 호출합니다. 이 메서드는 역사적인 이유로만 존재하며 새로운 코드에서 사용하면 안 됩니다. 대신 다른 대안을 사용하세요.

- `state`를 초기화하려면 `state` 를 클래스 필드로 선언하거나 `constructor` 내에서 `this.state` 를 설정하세요.
- 부수 효과를 실행하거나 구독을 설정해야 하는 경우 해당 로직을 `componentDidMount` 로 옮기세요.

안전하지 않은 생명주기에서 벗어나 마이그레이션한 사례를 확인하세요.

매개변수

`UNSAFE_componentWillMount` 는 어떠한 매개변수도 받지 않습니다.

반환값

`UNSAFE_componentWillMount` 는 아무것도 반환하지 않아야 합니다.

주의 사항

- 컴포넌트가 `static getDerivedStateFromProps` 또는 `getSnapshotBeforeUpdate` 를 구현하는 경우 `UNSAFE_componentWillMount` 가 호출되지 않습니다.
- 이름과는 달리, 앱이 `Suspense` 와 같은 최신 React 기능을 사용하는 경우 `UNSAFE_componentWillMount` 는 컴포넌트가 마운트될 것을 보장하지 않습니다. 렌더링 시도가 일시 중단되면(예를 들어 일부 자식 컴포넌트의 코드가 아직 로드되지 않았기 때문에) React는 진행 중인 트리를 버리고 다음 시도에서 컴포넌트를 처음부터 구성하려고 시도합니다. 이것이 바로 이 메서드가 “안전하지 않은” 이유입니다. 마운팅에 의존하는 코드(예: 구독 추가)는 `componentDidMount` 로 이동해야 합니다.
- `UNSAFE_componentWillMount` 는 `서버 렌더링` 중에 실행되는 유일한 생명주기 메서드입니다. 모든 실용적인 용도로 볼 때 `constructor` 와 동일하므로 이러한 유형의 로직에는 `constructor` 를 대신 사용해야 합니다.

▣ 중요합니다!

클래스 컴포넌트 내 UNSAFE_componentWillMount 내부에서 `setState` 를 호출하여 state를 초기화하는 것은 함수 컴포넌트에서 해당 state를 `useState` 에 초기 state로 전달하는 것과 동일합니다.

UNSAFE_componentWillReceiveProps(nextProps, nextContext)

UNSAFE_componentWillReceiveProps 를 정의하면 컴포넌트가 새로운 props를 수신할 때 React가 이를 호출합니다. 이 메서드는 역사적인 이유로만 존재하며 새로운 코드에서 사용하면 안 됩니다. 대신 다른 방법을 사용하세요.

- props 변경에 대한 응답으로 **부수 효과를 실행**(예: 데이터 가져오기, 애니메이션 실행, 구독 재초기화)해야 하는 경우 해당 로직을 `componentDidUpdate` 로 옮기세요.
- **props가 변경될 때만 일부 데이터를 다시 계산하지 않아야** 하는 경우 대신 `memoization helper`를 사용하세요.
- **props가 변경될 때 일부 state를 “초기화” 해야** 하는 경우, 컴포넌트를 `완전히 제어`하거나 `key`로 `완전히 제어하지 않도록` 만드는 것이 좋습니다.
- **props가 변경될 때 일부 state를 “조정” 해야** 하는 경우 렌더링 중에 props만으로 필요한 모든 정보를 계산할 수 있는지 확인하세요. 계산할 수 없는 경우 `static getDerivedStateFromProps` 를 대신 사용하세요.

안전하지 않은 생명주기에서 벗어나 마이그레이션한 사례를 확인하세요.

매개변수

- `nextProps`: 컴포넌트가 부모 컴포넌트로부터 받으려는 다음 props입니다. `nextProps` 와 `this.props` 를 비교하여 무엇이 변경되었는지 확인합니다.
- `nextContext`: 컴포넌트가 가장 가까운 공급자(provider)로부터 받으려는 다음 props입니다. `nextContext` 를 `this.context` 와 비교하여 변경된 내용을 확인합니다. `static contextType (modern)` 을 지정한 경우에만 사용할 수 있습니다.

반환값

UNSAFE_componentWillReceiveProps 는 아무것도 반환하지 않아야 합니다.

주의 사항

- 컴포넌트가 `static getDerivedStateFromProps` 또는 `getSnapshotBeforeUpdate` 를 구현하는 경우 `UNSAFE_componentWillReceiveProps` 가 호출되지 않습니다.
- 이름과는 달리, 앱이 `Suspense` 와 같은 최신 React 기능을 사용하는 경우 `UNSAFE_componentWillReceiveProps` 는 컴포넌트가 해당 `props`를 수신 할 것을 보장하지 않습니다. 렌더링 시도가 일시 중단되면(예를 들어 일부 자식 컴포넌트의 코드가 아직 로드되지 않았기 때문에) React는 진행 중인 트리를 버리고 다음 시도 중에 컴포넌트를 처음부터 다시 구성하려고 시도합니다. 다음 렌더링을 시도할 때쯤이면 `props`가 달라져 있을 수 있습니다. 이것이 바로 이 메서드가 “안전하지 않은” 이유입니다. 커밋된 업데이트에 대해서만 실행되어야 하는 코드(예: 구독 재설정)는 `componentDidUpdate` 로 이동해야 합니다.
- `UNSAFE_componentWillReceiveProps` 는 컴포넌트가 지난번과 다른 `props`를 받았다는 것을 의미하지 않습니다. `nextProps` 와 `this.props` 를 직접 비교하여 변경된 사항이 있는지 확인해야 합니다.
- React는 마운트하는 동안 초기 `props`와 함께 `UNSAFE_componentWillReceiveProps` 를 호출하지 않습니다. 컴포넌트의 일부 `props`가 업데이트될 경우에만 이 메서드를 호출합니다. 예를 들어, 일반적으로 같은 컴포넌트 내부에서 `setState` 를 호출해도 `UNSAFE_componentWillReceiveProps` 가 트리거되지 않습니다.

▣ 중요합니다!

클래스 컴포넌트에서 `UNSAFE_componentWillReceiveProps` 내부의 `setState` 를 호출하여 `state`를 “조정”하는 것은 함수 컴포넌트에서 렌더링 중에 `useState` 에서 `set` 함수를 호출하는 것과 동일합니다.

`UNSAFE_componentWillUpdate(nextProps, nextState)`

`UNSAFE_componentWillUpdate` 를 정의하면 React는 새 `props`나 `state`로 렌더링하기 전에 이를 호출합니다. 이 메서드는 역사적인 이유로만 존재하며 새로운 코드에서 사용하면 안 됩니다. 대신 다른 대안을 사용하세요.

- props나 state 변경에 대한 응답으로 부수 효과(예: 데이터 가져오기, 애니메이션 실행, 구독 재초기화)를 실행해야 하는 경우, 해당 로직을 `componentDidUpdate`로 이동하세요.
- 나중에 `componentDidUpdate`에서 사용할 수 있도록 DOM에서 일부 정보(예: 현재 스크롤 위치를 저장하기 위해)를 읽어야 하는 경우, 대신 `getSnapshotBeforeUpdate` 내부에서 읽습니다.

안전하지 않은 생명주기에서 벗어나 마이그레이션한 사례를 확인하세요.

매개변수

- `nextProps`: 컴포넌트가 렌더링할 다음 `props`입니다. `nextProps`와 `this.props`를 비교하여 무엇이 변경되었는지 확인합니다.
- `nextState`: 컴포넌트가 렌더링할 다음 `state`입니다. `nextState`와 `this.state`를 비교하여 무엇이 변경되었는지 확인합니다.

반환값

`UNSAFE_componentWillUpdate`는 아무것도 반환하지 않아야 합니다.

주의 사항

- `shouldComponentUpdate`가 정의된 경우 `UNSAFE_componentWillUpdate`는 호출되지 않으며 `false`를 반환합니다.
- 컴포넌트가 `static getDerivedStateFromProps` 또는 `getSnapshotBeforeUpdate`를 구현하는 경우 `UNSAFE_componentWillUpdate`가 호출되지 않습니다.
- `componentWillUpdate` 중에 `setState`를 호출하는 것(또는 Redux 액션을 dispatch하는 것과 같이 `setState`가 호출되도록 하는 모든 메서드)은 지원되지 않습니다.
- 이름과는 달리, 앱이 `Suspense`와 같은 최신 React 기능을 사용하는 경우 `UNSAFE_componentWillUpdate`는 컴포넌트가 업데이트될 것을 보장하지는 않습니다. 렌더링 시도가 일시 중단되면(예를 들어 일부 하위 컴포넌트의 코드가 아직 로드되지 않았기 때문에) React는 진행 중인 트리를 버리고 다음 시도에서 컴포넌트를 처음부터 새로 구성하려고 시도합니다. 다음 렌더링 시도 시에는 `props`와 `state`가 달라질 수 있습니다. 이것이 바로 이 메서드가 “안전하지 않은” 이유입니다. 커밋된 업데이트에 대해서만 실행되어야 하는 코드(예: 구독 재설정)는 `componentDidUpdate`로 이동해야 합니다.
- `UNSAFE_componentWillUpdate`는 컴포넌트가 지난번과 다른 `props`나 `state`를 받았다는 것을 의미하지 않습니다. `nextProps`를 `this.props`와, `nextState`를 `this.state`와 직접 비교하여 변경된 사항이 있는지 확인해야 합니다.

- React는 마운트하는 동안 초기 props와 state와 함께 UNSAFE_componentWillUpdate 를 호출하지 않습니다.

▣ 중요합니다!

함수 컴포넌트에는 UNSAFE_componentWillUpdate 와 직접적으로 대응하는 것이 없습니다.

static contextType

클래스 컴포넌트에서 `this.context` 를 읽으려면 읽어야 하는 context를 지정해야 합니다. static contextType 으로 지정하는 context는 이전에 `createContext` 로 생성한 값이어야 합니다.

```
class Button extends Component {  
  static contextType = ThemeContext;  
  
  render() {  
    const theme = this.context;  
    const className = 'button-' + theme;  
    return (  
      <button className={className}>  
        {this.props.children}  
      </button>  
    );  
  }  
}
```

▣ 중요합니다!

클래스 컴포넌트에서 `this.context`를 읽는 것은 함수 컴포넌트에서 `useContext`와 같습니다.

マイグ레이션 방법을 확인하세요.

static defaultProps

`static defaultProps`를 정의하여 클래스의 기본 `props`을 설정할 수 있습니다. `undefined`와 누락된 `props`에는 사용되지만 `null` `props`에는 사용되지 않습니다.

예를 들어, `color` `prop`의 기본값을 'blue'로 정의하는 방법은 다음과 같습니다.

```
class Button extends Component {
  static defaultProps = {
    color: 'blue'
  };

  render() {
    return <button className={this.props.color}>click me</button>;
  }
}
```

`color` `props`이 제공되지 않거나 `undefined`인 경우 기본적으로 'blue'로 설정됩니다.

```
<>
/* this.props.color is "blue" */
<Button />

/* this.props.color is "blue" */
<Button color={undefined} />

/* this.props.color is null */
<Button color={null} />
```

```
/* this.props.color is "red" */
<Button color="red" />
</>
```

▣ 중요합니다!

클래스 컴포넌트에서 `defaultProps` 를 정의하는 것은 함수 컴포넌트에서 `default values`를 사용하는 것과 유사합니다.

`static getDerivedStateFromError(error)`

`static getDerivedStateFromError` 를 정의하면 렌더링 도중 자식 컴포넌트(멀리 떨어진 자식 포함)가 에러를 `throw` 할 때 React가 이를 호출합니다. 이렇게 하면 UI를 지우는 대신 오류 메시지를 표시할 수 있습니다.

일반적으로 일부 분석 서비스에 오류 보고서를 보낼 수 있는 `componentDidCatch` 와 함께 사용합니다. 이러한 메서드가 있는 컴포넌트를 *Error Boundary* 라고 합니다.

[예시를 확인하세요.](#)

매개변수

- `error : throw` 된 오류입니다. 실제로는 일반적으로 `Error` 의 인스턴스가 되지만, 자바스크립트에서는 문자열이나 심지어 `null` 을 포함한 모든 값을 `throw` 할 수 있으므로 보장되지는 않습니다.

반환값

`static getDerivedStateFromError` 는 컴포넌트에 오류 메시지를 표시하도록 지시하는 `state` 를 반환해야 합니다.

주의 사항

- `static getDerivedStateFromError` 는 순수 함수여야 합니다. 예를 들어 분석 서비스를 호출하는 등의 부수 효과를 수행하려면 `componentDidCatch` 도 구현해야 합니다.

▣ 중요합니다!

함수 컴포넌트에서 `static getDerivedStateFromError`에 대해 직접적으로 동등한 것은 아직 없습니다. 클래스 컴포넌트를 만들지 않으려면 위와 같이 하나의 `ErrorBoundary` 컴포넌트를 작성하고 앱 전체에서 사용하세요. 또는 이를 수행하는 `react-error-boundary` package를 사용하세요.

static getDerivedStateFromProps(props, state)

`static getDerivedStateFromProps`를 정의하면 React는 초기 마운트 및 후속 업데이트 모두에서 `render`를 호출하기 바로 전에 이를 호출합니다. `state`를 업데이트하려면 객체를 반환하고, 아무것도 업데이트하지 않으려면 `null`을 반환해야 합니다.

이 메서드는 시간이 지남에 따라 `props`의 변경에 따라 `state`가 달라지는 [드문 사용 사례](#)를 위해 존재합니다. 예를 들어, 이 `Form` 컴포넌트는 `userID` `props`가 변경되면 `email` `state`를 재설정합니다.

```
class Form extends Component {
  state = {
    email: this.props.defaultEmail,
    prevUserID: this.props.userID
  };

  static getDerivedStateFromProps(props, state) {
    // 현재 사용자가 변경될 때마다,
    // 해당 사용자와 연결된 state의 모든 부분을 재설정합니다.
    // 이 간단한 예시에서는 이메일만 해당됩니다.
    if (props.userID !== state.prevUserID) {
      return {
        prevUserID: props.userID,
        email: props.defaultEmail
      };
    }
    return null;
  }
}
```

```
// ...  
}
```

이 패턴을 사용하려면 props의 이전 값(예: userID)을 state(예: prevUserID)로 유지해야 한다는 점에 유의하세요.

⚠ 주의하세요!

state를 파생하면 코드가 장황해지고 컴포넌트에 대해 생각하기 어려워집니다. [더 간단한 대안에 익숙해지도록 하세요.](#)

- props 변경에 대한 응답으로 부수 효과(예: 데이터 불러오기 또는 애니메이션)를 수행해야 하는 경우, 대신 `componentDidUpdate` 메서드를 사용하세요.
- props가 변경될 때만 일부 데이터를 다시 계산하려면 [memoization helper](#)를 대신 사용하세요.
- prop이 변경될 때 일부 state를 “초기화” 하려면 컴포넌트를 완전히 제어하거나 `key`를 사용해 완전히 제어하지 않도록 만드는 것이 좋습니다.

매개변수

- `props`: 컴포넌트가 렌더링할 다음 props입니다.
- `state`: 컴포넌트가 렌더링할 다음 state입니다.

반환값

`static getDerivedStateFromProps` 는 state를 업데이트할 객체를 반환하거나, 아무것도 업데이트하지 않으면 `null`을 반환합니다.

주의 사항

- 이 메서드는 원인에 관계없이 모든 렌더링에서 호출됩니다. 이는 부모가 다시 렌더링을 일으킬 때만 발동하고 로컬 `setState`의 결과가 아닐 때만 발동하는 [UNSAFE_componentWillReceiveProps](#)와는 다릅니다.

- 이 메서드에는 컴포넌트 인스턴스에 대한 액세스 권한이 없습니다. 원하는 경우 클래스 정의 외부 컴포넌트 props 및 state의 순수 함수를 추출하여 static getDerivedStateFromProps 와 다른 클래스 메서드 사이에 일부 코드를 재사용할 수 있습니다.

▣ 중요합니다!

클래스 컴포넌트에서 static getDerivedStateFromProps 를 구현하는 것은 함수 컴포넌트에서 렌더링하는 동안 useState 에서 set 함수를 호출하는 것과 동일합니다.

사용법

클래스 컴포넌트 정의하기

React 컴포넌트를 클래스로 정의하려면 기본 제공 Component 클래스를 확장하고 render 메서드를 정의합니다,

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

class Greeting extends Component {
  render() {
    return <h1>Hello, {this.props.name}!</h1>;
  }
}
```

React는 화면에 표시할 내용을 파악해야 할 때마다 render 메서드를 호출합니다. 보통은 JSX를 반환합니다. render 메서드는 순수 함수여야 합니다. JSX만 계산해야 합니다.

함수 컴포넌트와 마찬가지로 클래스 컴포넌트는 부모 컴포넌트로부터 props로 정보를 받는 것 가능합니다. 하지만 props를 읽는 문법은 다릅니다. 예를 들어, 부모 컴포넌트가 <Greeting name="Taylor" /> 를 렌더링하는 경우, this.props.name 과 같이 this.props 에서 name prop을 읽을 수 있습니다.

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

class Greeting extends Component {
  render() {
    return <h1>Hello, {this.props.name}!</h1>;
  }
}

export default function App() {
  return (
    <>
      <Greeting name="Sara" />
    </>
  );
}
```

▼ 자세히 보기

클래스 컴포넌트 내부에서는 Hooks(`use` 로 시작하는 함수, 예를 들어 `useState`)가 지원되지 않습니다.

 주의하세요!

컴포넌트를 클래스 대신 함수로 정의하는 것을 추천합니다. [マイグ레이션 方法を確認하세요.](#)

클래스 컴포넌트에 state 추가하기

클래스에 `state`를 추가하려면 `state`라는 프로퍼티에 객체를 할당합니다. `state`를 업데이트하려면 `this.setState`를 호출합니다.

App.js

↳ 다운로드 ⌂ 새로고침 ✖ Clear ⌂ 포크

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

export default class Counter extends Component {
  state = {
    name: 'Taylor',
    age: 42,
  };

  handleNameChange = (e) => {
    this.setState({
      name: e.target.value
    });
  };
}
```

▼ 자세히 보기

💡 주의하세요!

컴포넌트를 클래스 대신 함수로 정의하는 것을 추천합니다. [マイグレーション方法を確認하세요。](#)

클래스 컴포넌트에 생명주기 메서드 추가하기

클래스에서 정의할 수 있는 몇 가지 특별한 메서드가 있습니다.

`componentDidMount` 메서드를 정의하면 컴포넌트가 화면에 추가 (*마운트*) 될 때 React가 이를 호출합니다. 컴포넌트가 `props`나 `state` 변경으로 인해 다시 렌더링되면 React는 `componentDidUpdate`를 호출합니다. 컴포넌트가 화면에서 제거 (*마운트 해제*) 된 후 React는 `componentWillUnmount`를 호출합니다.

`componentDidMount`을 구현하는 경우 일반적으로 버그를 피하기 위해 세 가지 생명주기를 모두 구현해야 합니다. 예를 들어 `componentDidMount` 가 `state`나 `props`를 읽었다면 해당 변경 사항을 처리하기 위해 `componentDidUpdate`도 구현해야 하고, `componentDidMount` 가 수행하던 작업을 정리하기 위해 `componentWillUnmount`도 구현해야 합니다.

예를 들어 이 ChatRoom 컴포넌트는 채팅 연결을 `props` 및 `state`와 동기화하여 유지합니다:

App.js ChatRoom.js chat.js

↪ 새로고침 × Clear ⌂ 포크

```
import { Component } from 'react';
import { createConnection } from './chat.js';

export default class ChatRoom extends Component {
  state = {
    serverUrl: 'https://localhost:1234'
  };
}
```

```
componentDidMount() {  
  this.setupConnection();  
}
```

▼ 자세히 보기

[Strict 모드](#)가 켜져 있을 때 개발할 때 React는 componentDidMount 를 호출하고, 즉시 componentWillUnmount 를 호출한 다음, componentDidMount 를 다시 호출합니다. 이렇게 하면 componentWillUnmount 를 구현하는 것을 잊어버렸거나 그 로직이 componentDidMount 의 동작을 완전히 “미러링”하지 않는지 알 수 있습니다.

⚠ 주의하세요!

컴포넌트를 클래스 대신 함수로 정의하는 것을 추천합니다. [マイグレーション方法を確認하세요.](#)

Error Boundary로 렌더링 오류 포착하기

기본적으로 애플리케이션이 렌더링 도중 에러를 발생시키면 React는 화면에서 해당 UI를 제거합니다. 이를 방지하기 위해 UI의 일부를 *Error Boundary*로 감싸면 됩니다. *Error Boundary*는 에러가 발생한 부분 대신 오류 메시지와 같은 Fallback UI를 표시할 수 있는 특수 컴포넌트입니다.

▣ 중요합니다!

Error boundaries do not catch errors for:

- Event handlers ([learn more](#))
- Server side rendering
- Errors thrown in the error boundary itself (rather than its children)
- Asynchronous code (e.g. `setTimeout` or `requestAnimationFrame` callbacks);
an exception is the usage of the `startTransition` function returned by the
`useTransition` Hook. Errors thrown inside the transition function are caught
by error boundaries ([learn more](#))

Error Boundary 컴포넌트를 구현하려면 오류에 대한 응답으로 State를 업데이트하고 사용자에게 오류 메시지를 표시할 수 있는 `static getDerivedStateFromError`를 제공해야 합니다. 또한 선택적으로 `componentDidCatch`를 구현하여 분석 서비스에 오류를 기록하는 등의 추가 로직을 추가할 수도 있습니다.

With `captureOwnerStack` you can include the Owner Stack during development.

```
import * as React from 'react';

class ErrorBoundary extends React.Component {
  constructor(props) {
    super(props);
    this.state = { hasError: false };
  }

  static getDerivedStateFromError(error) {
    // state를 업데이트하여 다음 렌더링에 fallback UI가 표시되도록 합니다.
    return { hasError: true };
  }
}
```

```

componentDidCatch(error, info) {
  logErrorToMyService(
    error,
    // Example "componentStack":
    //   in ComponentThatThrows (created by App)
    //   in ErrorBoundary (created by App)
    //   in div (created by App)
    //   in App
    info.componentStack,
    // Warning: `captureOwnerStack` is not available in production.
    React.captureOwnerStack(),
  );
}

render() {
  if (this.state.hasError) {
    // 사용자 지정 fallback UI를 렌더링할 수 있습니다.
    return this.props.fallback;
  }

  return this.props.children;
}
}

```

그런 다음 컴포넌트 트리의 일부를 래핑할 수 있습니다.

```

<ErrorBoundary fallback={<p>Something went wrong</p>}>
  <Profile />
</ErrorBoundary>

```

Profile 또는 그 하위 컴포넌트가 오류를 발생시키면 ErrorBoundary 가 해당 오류를 “포착”하고 사용자가 제공한 오류 메시지와 함께 fallback UI를 표시한 다음 프로덕션 오류 보고서를 오류 보고 서비스에 전송합니다.

모든 컴포넌트를 별도의 Error Boundary로 묶을 필요는 없습니다. [Error Boundary의 세분화](#)를 고려할 때는 오류 메시지를 표시하는 것이 적절한 위치를 고려하세요. 예를 들어 메시징 앱의 경우 Error Boundary를 대화 목록 주위에 위치시키는 것이 좋습니다. 또한 모든 개별 메시지 주위에 위

치시키는 것도 좋습니다. 하지만 모든 아바타 주위에 Boundary를 위치시키는 것은 적절하지 않습니다.

▣ 중요합니다!

현재 Error Boundary를 함수 컴포넌트로 작성할 수 있는 방법은 없습니다. 하지만 Error Boundary 클래스를 직접 작성할 필요는 없습니다. 예를 들어 [react-error-boundary](#) 를 대신 사용할 수 있습니다.

대안

클래스에서 함수로 간단한 컴포넌트 마이그레이션하기

일반적으로 [컴포넌트를 함수로 대신 정의합니다.](#)

예를 들어 이 Greeting 클래스 컴포넌트를 함수로 변환한다고 가정해 보겠습니다.

App.js

↳ 다운로드 ⌂ 새로고침 ✖ Clear ⌂ 포크

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

class Greeting extends Component {
  render() {
    return <h1>Hello, {this.props.name}!</h1>;
  }
}

export default function App() {
  return (
    <>
      <Greeting name="Sara" />
    </>
  );
}
```

▼ 자세히 보기

Greeting이라는 함수를 정의합니다. 여기로 render 함수의 본문을 이동합니다.

```
function Greeting() {  
  // ... render 메서드의 코드를 여기로 옮깁니다 ...  
}
```

this.props.name 대신 구조 분해 문법을 사용하여 name prop을 정의하고 직접 읽습니다.

```
function Greeting({ name }) {  
  return <h1>Hello, {name}!</h1>;  
}
```

다음은 전체 예시입니다.

App.js

↳ 다운로드 ⌂ 새로고침 ✖ Clear ☰ 포크

```
function Greeting({ name }) {  
  return <h1>Hello, {name}!</h1>;  
}  
  
export default function App() {
```

```
return (
  <>
  <Greeting name="Sara" />
  <Greeting name="Cahal" />
  <Greeting name="Edite" />
</>
```

state가 있는 컴포넌트를 클래스에서 함수로 마이그레이션하기

이 Counter 클래스 컴포넌트를 함수로 변환한다고 가정해 봅시다.

App.js

↳ 다운로드 ⌂ 새로고침 ✖ Clear ✎ 포크

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

export default class Counter extends Component {
  state = {
    name: 'Taylor',
    age: 42,
  };

  handleNameChange = (e) => {
    this.setState({
      name: e.target.value
    });
  };
}
```

name: e.target.value

▼ 자세히 보기

필요한 **state** 변수가 있는 함수를 선언하는 것으로 시작하세요.

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

function Counter() {
  const [name, setName] = useState('Taylor');
  const [age, setAge] = useState(42);
  // ...
}
```

다음으로 이벤트 핸들러를 변환합니다.

```
function Counter() {
  const [name, setName] = useState('Taylor');
  const [age, setAge] = useState(42);

  function handleNameChange(e) {
    setName(e.target.value);
  }
}
```

```
function handleAgeChange() {
  setAge(age + 1);
}

// ...
```

마지막으로, `this` 으로 시작하는 모든 레퍼런스를 컴포넌트에서 정의한 변수 및 함수로 바꿉니다. 예를 들어, `this.state.age` 를 `age` 로 바꾸고 `this.handleNameChange` 를 `handleNameChange` 로 바꿉니다.

다음은 완전히 변환된 컴포넌트입니다.

App.js

↳ 다운로드 ⌂ 새로고침 ✕ Clear ☒ 포크

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

export default function Counter() {
  const [name, setName] = useState('Taylor');
  const [age, setAge] = useState(42);

  function handleNameChange(e) {
    setName(e.target.value);
  }

  function handleAgeChange() {
    setAge(age + 1);
  }
}
```

▼ 자세히 보기

생명주기 메서드가 있는 컴포넌트를 클래스에서 함수로 마이그레이션하기

생명주기 메서드가 있는 ChatRoom 클래스 컴포넌트를 함수로 변환한다고 가정해 보겠습니다.

App.js ChatRoom.js chat.js

↪ 새로고침 X Clear ⌂ 포크

```
import { Component } from 'react';
import { createConnection } from './chat.js';

export default class ChatRoom extends Component {
  state = {
    serverUrl: 'https://localhost:1234'
  };

  componentDidMount() {
    this.setupConnection();
  }
}
```

▼ 자세히 보기

먼저 `componentWillUnmount` 가 `componentDidMount` 와 반대 동작을 하는지 확인합니다. 위의 예시에서는 `componentDidMount` 가 설정한 연결을 끊습니다. 이러한 로직이 누락된 경우 먼저 추가하세요.

다음으로, `componentDidUpdate` 메서드가 `componentDidMount` 에서 사용 중인 `props` 및 `state`의 변경 사항을 처리하는지 확인합니다. 위의 예시에서 `componentDidMount` 는 `this.state.serverUrl` 과 `this.props.roomId` 를 읽는 `setupConnection` 을 호출합니다. 이 때문에 `componentDidUpdate` 는 `this.state.serverUrl` 과 `this.props.roomId` 가 변경되었는지 확인하고, 변경된 경우 연결을 재설정합니다. `componentDidUpdate` 로직이 누락되었거나 모든 관련 `props` 및 `state`의 변경 사항을 처리하지 않는 경우 먼저 해당 로직을 수정하세요.

위의 예시에서, 생명주기 메서드 내부의 로직은 컴포넌트를 React 외부의 시스템(채팅 서버)에 연결합니다. 컴포넌트를 외부 시스템에 연결하려면 [이 로직을 하나의 Effect로 설명하세요](#).

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

function ChatRoom({ roomId }) {
  const [serverUrl, setServerUrl] = useState('https://localhost:1234');

  useEffect(() => {
    const connection = createConnection(serverUrl, roomId);
    connection.connect();
    return () => {
      connection.disconnect();
    };
  }, [serverUrl, roomId]);

  // ...
}
```

이 `useEffect` 호출은 위의 생명주기 메서드의 로직과 동일합니다. 생명주기 메서드가 서로 관련이 없는 여러 가지 작업을 수행하는 경우, [이를 여러 개의 독립적인 Effect로 분할하세요](#). 다음은 완전한 예시입니다.

```
import { useState, useEffect } from 'react';
import { createConnection } from './chat.js';

export default function ChatRoom({ roomId }) {
  const [serverUrl, setServerUrl] = useState('https://localhost:1234');

  useEffect(() => {
    const connection = createConnection(serverUrl, roomId);
    connection.connect();
    return () => {
      connection.disconnect();
    };
}
```

▼ 자세히 보기

▣ 중요합니다!

컴포넌트가 외부 시스템과 동기화되지 않는 경우 Effect가 필요하지 않을 수 있습니다.

context가 있는 컴포넌트를 클래스에서 함수로 마이그레이션하기

이 예시에서 Panel 및 Button 클래스 컴포넌트는 `this.context`에서 context를 읽습니다.

App.js

↳ 다운로드 ⌂ 새로고침 ✖ Clear ☰ 포크

```
import { createContext, Component } from 'react';

const ThemeContext = createContext(null);

class Panel extends Component {
  static contextType = ThemeContext;

  render() {
    const theme = this.context;
    const className = 'panel-' + theme;
    return (
      <section className={className}>
```

▼ 자세히 보기

함수 컴포넌트로 변환할 때는 `this.context`를 `useContext` 호출로 바꿔주세요.

```
import { createContext, useContext } from 'react';

const ThemeContext = createContext(null);

function Panel({ title, children }) {
  const theme = useContext(ThemeContext);
  const className = 'panel-' + theme;
  return (
    <section className={className}>
      <h1>{title}</h1>
      {children}
    </section>
  );
}
```

▼ 자세히 보기

이전

< [cloneElement](#)

다음

[createElement](#) >



Copyright © Meta Platforms, Inc

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

