# use

`use` 는 Promise나 Context와 같은 데이터를 참조하는 React API입니다.

```
const value = use(resource);
```



---

## 레퍼런스

### `use(resource)`

컴포넌트에서 Promise나 Context와 같은 데이터를 참조하려면 `use` 를 사용하세요.

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

function MessageComponent({ messagePromise }) {
  const message = use(messagePromise);
  const theme = use(ThemeContext);
  // ...
```

다른 React Hook과 달리 `use` 는 `if` 와 같은 조건문과 반복문 내부에서 호출할 수 있습니다. 다만, 다른 React Hook과 같이 `use` 는 컴포넌트 또는 Hook에서만 호출해야 합니다.

Promise와 함께 호출될 때 `use` API는 `Suspense` 및 Error Boundary와 통합됩니다. `use` 에 전달된 Promise가 대기Pending하는 동안 `use` 를 호출하는 컴포넌트는 *Suspend*됩니다. `use` 를 호출하는 컴포넌트가 Suspense 경계로 둘러싸여 있으면 Fallback이 표시됩니다. Promise가 리졸브되면 Suspense Fallback은 `use` API가 반환한 컴포넌트로 대체됩니다. `use` 에 전달된 Promise가 Reject되면 가장 가까운 Error Boundary의 Fallback이 표시됩니다.

아래 예시를 참고하세요.

### 매개변수

- `resource` : 참조하려는 데이터입니다. 데이터는 Promise나 Context일 수 있습니다.

### 반환값

`use` Hook은 Promise나 Context에서 참조한 값을 반환합니다.

### 주의 사항

- `use` API는 컴포넌트나 Hook 내부에서 호출되어야 합니다.
- 서버 컴포넌트에서 데이터를 가져올 때는 `use` 보다 `async` 및 `await` 을 사용합니다. `async` 및 `await` 은 `await` 이 호출된 시점부터 렌더링을 시작하는 반면, `use` 는 데이터가 리졸브된 후 컴포넌트를 리렌더링합니다.
- 클라이언트 컴포넌트에서 Promise를 생성하는 것보다 서버 컴포넌트에서 Promise를 생성하여 클라이언트 컴포넌트에 전달하는 것이 좋습니다. 클라이언트 컴포넌트에서 생성된 Promise는 렌더링할 때마다 다시 생성됩니다. 서버 컴포넌트에서 클라이언트 컴포넌트로 전달된 Promise는 리렌더링 전반에 걸쳐 안정적입니다. 예시를 확인하세요.

---

# 사용법

## `use` 를 사용하여 Context 참조하기

Context가 `use` 에 전달되면 `useContext` 와 유사하게 작동합니다. `useContext` 는 컴포넌트의 최상위 수준에서 호출해야 하지만, `use` 는 `if` 와 같은 조건문이나 `for` 와 같은 반복문 내부에서 호출할 수 있습니다. `use` 는 유연하므로 `useContext` 보다 선호됩니다.

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

function Button() {
  const theme = use(ThemeContext);
  // ...
```

`use` 는 전달한 `Context` 의 `Context Value` 를 반환합니다. Context 값을 결정하기 위해 React 는 컴포넌트 트리를 탐색하고 **위에서 가장 가까운 Context Provider**를 찾습니다.

Context를 `Button` 에 전달하려면 `Button` 또는 상위 컴포넌트 중 하나를 Context Provider로 래핑합니다.

```
function MyPage() {
  return (
    <ThemeContext value="dark">
      <Form />
    </ThemeContext>
  );
}

function Form() {
  // ... 버튼 렌더링 ...
}
```

Provider와 `Button` 사이에 얼마나 많은 컴포넌트가 있는지는 중요하지 않습니다. `Form` 내부의 *어느 곳이든* `Button` 이 `use(ThemeContext)` 를 호출하면 `"dark"` 를 값으로 받습니다.

`useContext` 와 달리, `use` 는 `if` 와 같은 조건문과 반복문 내부에서 호출할 수 있습니다.

```
function HorizontalRule({ show }) {
  if (show) {
    const theme = use(ThemeContext);
    return <hr className={theme} />;
  }
  return false;
}
```

`use` 는 `if` 내부에서 호출되므로 Context에서 조건부로 값을 참조할 수 있습니다.

### 주의하세요!

`useContext` 와 마찬가지로, `use(context)` 는 항상 이를 호출하는 컴포넌트의 **위쪽에서** 가장 가까운 Context Provider를 찾습니다. 위쪽으로 탐색하며, `use(context)` 를 호출하는 컴포넌트 내부의 Context Provider는 고려하지 **않습니다**.

App.js

다운로드 새로고침 Clear 포크

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

const ThemeContext = createContext(null);

export default function MyApp() {
  return (
    <ThemeContext value="dark">
      <Form />
    </ThemeContext>
  )
}
```

자세히 보기

## 서버에서 클라이언트로 데이터 스트리밍하기

서버 컴포넌트 에서 클라이언트 컴포넌트 로 Promise Prop을 전달하여 서버에서 클라이언트로 데이터를 스트리밍할 수 있습니다.

```
import { fetchMessage } from './lib.js';
import { Message } from './message.js';

export default function App() {
  const messagePromise = fetchMessage();
  return (
    <Suspense fallback={<p>waiting for message...</p>}>
      <Message messagePromise={messagePromise} />
    </Suspense>
  );
}
```

클라이언트 컴포넌트 는 Prop으로 받은 Promise 를 `use` API에 전달합니다. 클라이언트 컴포넌트 는 서버 컴포넌트가 처음에 생성한 Promise 에서 값을 읽을 수 있습니다.

```
// message.js
'use client';

import { use } from 'react';

export function Message({ messagePromise }) {
  const messageContent = use(messagePromise);
  return <p>Here is the message: {messageContent}</p>;
}
```

Message 는 Suspense 로 래핑되어 있으므로 Promise가 리졸브될 때까지 Fallback이 표시됩니다. Promise가 리졸브되면 `use` Hook이 값을 참조하고 Message 컴포넌트가 Suspense

Fallback을 대체합니다.

message.js

새로고침 Clear 포크

```
"use client";

import { use, Suspense } from "react";

function Message({ messagePromise }) {
  const messageContent = use(messagePromise);
  return <p>Here is the message: {messageContent}</p>;
}

export function MessageContainer({ messagePromise }) {
  return (
    <Suspense fallback={<p>⌛ Downloading message...</p>}>
```

자세히 보기

## 중요합니다!

서버 컴포넌트에서 클라이언트 컴포넌트로 Promise를 전달할 때 리졸브된 값이 직렬화 가능해야 합니다. 함수는 직렬화할 수 없으므로 Promise의 리졸브 값이 될 수 없습니다.

자세히 살펴보기

**Promise를 서버 컴포넌트에서 처리해야 하나요, 아니면 클라이언트 컴포넌트에서 처리해야 하나요?**

자세히 보기

## 거부된 Promise 처리하기

경우에 따라 `use`에 전달된 Promise가 거부될 수 있습니다. 거부된 프로미스를 처리하는 방법은 2가지가 존재합니다.

1. Error Boundary를 사용하여 오류 표시하기
2. `Promise.catch`로 대체 값 제공하기

### 주의하세요!

`use`는 `try-catch` 블록에서 호출할 수 없습니다. `try-catch` 블록 대신 컴포넌트를 Error Boundary로 래핑하거나, Promise의 `catch` 메서드를 사용하여 대체 값을 제공해야 합니다.

### Error Boundary를 사용하여 오류 표시하기

Promise가 거부될 때 오류를 표시하고 싶다면 Error Boundary를 사용합니다. Error Boundary를 사용하려면 `use` API 를 호출하는 컴포넌트를 Error Boundary로 래핑합니다. `use` 에 전달된 Promise가 거부되면 Error Boundary에 대한 Fallback이 표시됩니다.

message.js

새로고침 Clear 포크

```
"use client";

import { use, Suspense } from "react";
import { ErrorBoundary } from "react-error-boundary";

export function MessageContainer({ messagePromise }) {
  return (
    <ErrorBoundary fallback={<p>⚠️Something went wrong</p>}>
      <Suspense fallback={<p>⌛Downloading message...</p>}>
        <Message messagePromise={messagePromise} />
      </Suspense>
    </ErrorBoundary>
```

자세히 보기

### `Promise.catch` 로 대체 값 제공하기

`use`에 전달된 Promise가 거부될 때 대체 값을 제공하려면 Promise의 `catch` 메서드를 사용합니다.

```
import { Message } from './message.js';

export default function App() {
  const messagePromise = new Promise((resolve, reject) => {
    reject();
  }).catch(() => {
    return "no new message found.";
  });

  return (
    <Suspense fallback={<p>waiting for message...</p>}>
      <Message messagePromise={messagePromise} />
    </Suspense>
  );
}
```

Promise의 `catch` 메서드를 사용하려면 Promise 객체에서 `catch`를 호출합니다. `catch`는 오류 메시지를 인수로 받는 함수를 인수로 받습니다. `catch`에 전달된 함수가 반환하는 값은 모두 Promise의 리졸브 값으로 사용됩니다.

---

# 문제 해결

## “Suspense Exception: This is not a real error!”

React 컴포넌트 또는 Hook 함수 외부에서, 혹은 `try - catch` 블록에서 `use`를 호출하고 있는 경우입니다. `try - catch` 블록 내에서 `use`를 호출하는 경우 컴포넌트를 Error Boundary로 래핑하거나 Promise의 `catch`를 호출하여 오류를 발견하고 Promise를 다른 값으로 리졸브합니다. 이러한 예시들을 확인하세요.

```
function MessageComponent({messagePromise}) {
  function download() {
    // ❌ `use`를 호출하는 함수가 컴포넌트나 Hook이 아닙니다.
    const message = use(messagePromise);
```

```
  // ...
```

대신, 컴포넌트 클로저 외부에서 `use`를 호출하세요. 여기서 `use`를 호출하는 함수는 컴포넌트 또는 Hook입니다.

```
function MessageComponent({messagePromise}) {
  // ✅ `use`를 컴포넌트에서 호출하고 있습니다.
  const message = use(messagePromise);
  // ...
```


이전
startTransition

다음
experimental_taintObjectReference



Meta Open Source

Copyright © Meta Platforms, Inc

uwu?

감사의 말

약관