cache

React 서버 컴포넌트

`cache` is only for use with React Server Components.

`cache`를 통해 가져온 데이터나 연산의 결과를 캐싱합니다.

```
const cachedFn = cache(fn);
```


레퍼런스

`cache(fn)`

컴포넌트 외부에서 `cache`를 호출해 캐싱 기능을 가진 함수의 한 버전을 만들 수 있습니다.

```
import {cache} from 'react';
import calculateMetrics from 'lib/metrics';

const getMetrics = cache(calculateMetrics);

function Chart({data}) {
  const report = getMetrics(data);
  // ...
}
```

`getMetrics` 가 처음 `data` 를 호출할 때, `getMetrics` 는 `calculateMetrics(data)` 를 호출하고 캐시에 결과를 저장합니다. `getMetrics` 가 같은 `data` 와 함께 다시 호출되면, `calculateMetrics(data)` 를 다시 호출하는 대신에 캐싱된 결과를 반환합니다.

아래 예시를 참고하세요.

매개변수

- `fn` : 결과를 저장하고 싶은 함수. `fn` 은 어떤 인수도 받을 수 있고 어떠한 결과도 반환할 수 있습니다.

반환값

`cache` 는 같은 타입 시그니처를 가진 `fn` 의 캐싱된 버전을 반환합니다. 이 과정에서 `fn` 을 호출하지 않습니다.

주어진 인수와 함께 `cachedFn` 을 호출할 때, 캐시에 캐싱된 데이터가 있는지 먼저 확인합니다. 만약 캐싱된 데이터가 있다면, 그 결과를 반환합니다. 만약 없다면, 매개변수와 함께 `fn` 을 호출하고 결과를 캐시에 저장하고 값을 반환합니다. `fn` 이 유일하게 호출되는 경우는 캐싱된 데이터가 없는 경우입니다.

중요합니다!

입력을 기반으로 반환 값 캐싱을 최적화하는 것을 *메모이제이션*이라고 합니다. `cache` 에서 반환되는 함수를 메모화된 함수라고 합니다.

주의 사항

- React will invalidate the cache for all memoized functions for each server request.
- Each call to `cache` creates a new function. This means that calling `cache` with the same function multiple times will return different memoized functions that do not share the same cache.
- `cachedFn` will also cache errors. If `fn` throws an error for certain arguments, it will be cached, and the same error is re-thrown when `cachedFn` is called with those same arguments.
- `cache` is for use in Server Components only.

사용법

고비용 연산 캐싱하기

반복 작업을 피하기 위해 `cache` 를 사용하세요.

```
import {cache} from 'react';
import calculateUserMetrics from 'lib/user';

const getUserMetrics = cache(calculateUserMetrics);

function Profile({user}) {
  const metrics = getUserMetrics(user);
  // ...
}

function TeamReport({users}) {
  for (let user in users) {
    const metrics = getUserMetrics(user);
    // ...
  }
  // ...
}
```

같은 `user` 객체가 `Profile`과 `TeamReport`에서 렌더링될 때, 두 컴포넌트는 작업을 공유하고, `user`를 위한 `calculateUserMetrics`를 한 번만 호출합니다.

If the same `user` object is rendered in both `Profile` and `TeamReport`, the two components can share work and only call `calculateUserMetrics` once for that `user`.

Assume `Profile` is rendered first. It will call `getUserMetrics`, and check if there is a cached result. Since it is the first time `getUserMetrics` is called with that `user`, there will be a cache miss. `getUserMetrics` will then call `calculateUserMetrics` with that `user` and write the result to cache.

When `TeamReport` renders its list of `users` and reaches the same `user` object, it will call `getUserMetrics` and read the result from cache.

If `calculateUserMetrics` can be aborted by passing an `AbortSignal`, you can use `cacheSignal()` to cancel the expensive computation if React has finished rendering. `calculateUserMetrics` may already handle cancellation internally by using `cacheSignal` directly.

주의하세요!

다른 메모화된 함수를 호출하면 다른 캐시에서 읽습니다.

같은 캐시에 접근하기 위해선, 컴포넌트는 반드시 같은 메모화된 함수를 호출해야 합니다.

```
// Temperature.js
import {cache} from 'react';
import {calculateWeekReport} from './report';

export function Temperature({cityData}) {
  // 🚩 Wrong: 컴포넌트에서 `cache`를 호출하면 각 렌더링에 대해 `getWeekReport`가 생
  const getWeekReport = cache(calculateWeekReport);
  const report = getWeekReport(cityData);
  // ...
}
```

```
// Precipitation.js
import {cache} from 'react';
import {calculateWeekReport} from './report';

// 🚩 Wrong: `getWeekReport`는 `Precipitation` 컴포넌트에서만 적용할 수 있습니다.
const getWeekReport = cache(calculateWeekReport);

export function Precipitation({cityData}) {
  const report = getWeekReport(cityData);
  // ...
}
```

위의 예시에서, `Precipitation` 와 `Temperature` 는 각각 `cache` 를 호출하여 자체 캐시 조회를 통해 새로운 메모화된 함수를 만들어 냅니다. 두 컴포넌트가 같은 `cityData` 를 렌더링한다면, `calculateWeekReport` 를 호출하는 반복 작업을 하게 됩니다.

게다가, `Temperature` 는 컴포넌트가 렌더링될 때마다 어떤 캐시 공유도 허용하지 않는 새로운 메모화된 함수 를 생성하게 됩니다.

캐시 사용을 늘리고 작업을 줄이기 위해서 두 컴포넌트는 같은 캐시에 접근하는 같은 메모화된 함수를 호출해야 합니다. 대신, 컴포넌트끼리 `import` 할 수 있는 전용 모듈에 메모화된 함수를 정의하세요.

```
// getWeekReport.js
import {cache} from 'react';
import {calculateWeekReport} from './report';

export default cache(calculateWeekReport);
```

```
// Temperature.js
import getWeekReport from './getWeekReport';

export default function Temperature({cityData}) {
  const report = getWeekReport(cityData);
  // ...
}
```

```
// Precipitation.js
import getWeekReport from './getWeekReport';

export default function Precipitation({cityData}) {
  const report = getWeekReport(cityData);
  // ...
}
```

Here, both components call the same memoized function exported from `./getWeekReport.js` to read and write to the same cache.

데이터의 스냅샷 공유하기

To share a snapshot of data between components, call `cache` with a data-fetching function like `fetch`. When multiple components make the same data fetch, only one request is made and the data returned is cached and shared across components. All components refer to the same snapshot of data across the server render.

```
import {cache} from 'react';
import {fetchTemperature} from './api.js';

const getTemperature = cache(async (city) => {
  return await fetchTemperature(city);
});

async function AnimatedWeatherCard({city}) {
  const temperature = await getTemperature(city);
  // ...
}

async function MinimalWeatherCard({city}) {
  const temperature = await getTemperature(city);
  // ...
}
```

If `AnimatedWeatherCard` and `MinimalWeatherCard` both render for the same `city`, they will receive the same snapshot of data from the `memoized function`.

`AnimatedWeatherCard` 와 `MinimalWeatherCard` 가 다른 `city` 를 `getTemperature` 의 인수로 받게 된다면, `fetchTemperature` 는 두 번 호출되고 호출마다 다른 데이터를 받게됩니다.

`city` 가 캐시 키Key처럼 동작하게 됩니다.

중요합니다!

`Asynchronous rendering` is only supported for Server Components.

```
async function AnimatedWeatherCard({city}) {
  const temperature = await getTemperature(city);
  // ...
}
```

To render components that use asynchronous data in Client Components, see `use()` documentation.

사전에 데이터 받아두기

긴 실행 시간이 소요되는 데이터 가져오기를 캐싱하면, 컴포넌트를 렌더링하기 전에 비동기 작업을 시작할 수 있습니다.

```
const getUser = cache(async (id) => {
  return await db.user.query(id);
});

async function Profile({id}) {
  const user = await getUser(id);
  return (
    <section>
      <img src={user.profilePic} />
```

```
      <h2>{user.name}</h2>
    </section>
  );
}

function Page({id}) {
  // ✅ Good: 사용자 데이터 가져오기를 시작합니다.
  getUser(id);
  // ... 몇몇의 계산 작업들
  return (
    <>
      <Profile id={id} />
    </>
  );
}
```

`Page`를 렌더링할 때, 컴포넌트는 `getUser`를 호출하지만, 반환된 데이터를 사용하지 않는다는 점에 유의하세요. 이 초기 `getUser` 호출은 페이지가 다른 계산 작업을 수행하고 자식을 렌더링하는 동안 발생하는, 비동기 데이터베이스 쿼리를 시작합니다.

`Profile`을 렌더링할 때, `getUser`를 다시 호출합니다. 초기 `getUser` 호출이 이미 사용자 데이터에 반환되고 캐싱되었다면, `Profile`이 해당 데이터를 요청하고 기다릴 때, 다른 원격 프로시저 호출 없이 쉽게 캐시에서 읽어올 수 있습니다. 초기 데이터 요청이 완료되지 않은 경우, 이 패턴으로 데이터를 미리 로드하면 데이터를 받아올 때 생기는 지연이 줄어듭니다.

자세히 살펴보기

비동기 작업 캐싱하기

자세히 보기

주의하세요!

컴포넌트 외부에서 메모화된 함수를 사용하면 캐시가 사용되지 않습니다.

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

const getUser = cache(async (userId) => {
  return await db.user.query(userId);
});

// 🚩 Wrong: 컴포넌트 외부에서 메모화된 함수를 호출하면 메모화하지 않습니다.
getUser('demo-id');

async function DemoProfile() {
  // ✅ Good: `getUser`는 메모화 됩니다.
  const user = await getUser('demo-id');
  return <Profile user={user} />;
}
```

React는 컴포넌트에서 메모화된 함수의 캐시 접근만 제공합니다. 컴포넌트 외부에서 `getUser`를 호출하면 여전히 함수를 실행하지만, 캐시를 읽거나 업데이트하지는 않습니다.

This is because cache access is provided through a context which is only accessible from a component.

자세히 살펴보기

`cache`, `memo`, `useMemo` 중 언제 어떤 걸 사용해야 하나요?

자세히 보기

문제 해결

동일한 인수로 함수를 호출해도 메모된 함수가 계속 실행됩니다

앞서 언급된 주의 사항들을 확인하세요.

- 다른 메모화된 함수를 호출하면 다른 캐시에서 읽습니다.
- 컴포넌트 외부에서 메모화된 함수를 사용하면 캐시가 사용되지 않습니다.

위의 어느 것도 해당하지 않는다면, React가 캐시에 무엇이 존재하는지 확인하는 방식에 문제가 있을 수 있습니다.

인자가 원시 값(객체, 함수, 배열 등)이 아니라면, 같은 객체 참조를 넘겼는지 확인하세요.

메모화된 함수 호출 시, React는 입력된 인자값을 조회해 결과가 이미 캐싱되어 있는지 확인합니다. React는 인수들의 얕은 동등성을 사용해 캐시 히트가 있는지를 결정합니다.

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

const calculateNorm = cache((vector) => {
  // ...
});

function MapMarker(props) {
  // 🚩 Wrong: 인자가 매 렌더링마다 변경되는 객체입니다.
  const length = calculateNorm(props);
  // ...
}

function App() {
  return (
    <>
      <MapMarker x={10} y={10} z={10} />
      <MapMarker x={10} y={10} z={10} />
    </>
  );
}
```

이 경우 두 `MapMarker` 는 동일한 작업을 수행하고 동일한 값인 `{x: 10, y: 10, z:10}` 와 함께 `calculateNorm` 를 호출하는 듯 보입니다. 객체에 동일한 값이 포함되어 있더라도 각 컴포넌트가 자체 프로퍼티 객체를 생성하므로, 동일한 객체 참조가 아닙니다.

React는 입력에서 `Object.is` 를 호출해 캐시 히트가 있는지 확인합니다.

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

const calculateNorm = cache((x, y, z) => {
  // ...
});

function MapMarker(props) {
  // ✅ Good: 메모화 함수에 인자로 원시값 제공하기
  const length = calculateNorm(props.x, props.y, props.z);
  // ...
}

function App() {
  return (
    <>
      <MapMarker x={10} y={10} z={10} />
      <MapMarker x={10} y={10} z={10} />
    </>
  );
}
```

이 문제를 해결하는 한 가지 방법은 벡터 차원을 `calculateNorm` 에 전달하는 것입니다. 차원 자체가 원시 값이기 때문에 가능합니다.

다른 방법은 벡터 객체를 컴포넌트의 프로퍼티로 전달하는 방법입니다. 두 컴포넌트 인스턴스에 동일한 객체를 전달해야 합니다.

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

const calculateNorm = cache((vector) => {
  // ...
});
```

```
function MapMarker(props) {
  // ✅ Good: 동일한 `vector` 객체를 넘겨줍니다.
  const length = calculateNorm(props.vector);
  // ...
}

function App() {
  const vector = [10, 10, 10];
  return (
    <>
      <MapMarker vector={vector} />
      <MapMarker vector={vector} />
    </>
  );
}
```


이전
addTransitionType

다음
cacheSignal