Promise 응용하기

타입스크립트 해적단

Resolve Reject

✅ 기본 문법

```js
Promise.resolve(value);
```

- `value` : Promise로 감싸고 싶은 값

🔍 동작 설명

- `value`가 **Promise 객체**인 경우:
  - 해당 Promise를 그대로 반환합니다.
- `value`가 **비-Promise 값**인 경우:
  - 해당 값을 가진 **resolve된 Promise**를 반환합니다.

📘 예제

1. 일반 값으로 resolve

```js
Promise.resolve(42).then((result) => {
  console.log(result); // 42
});
```

2. 이미 있는 Promise 전달

```js
const existingPromise = new Promise((res) => res("OK"));

const p = Promise.resolve(existingPromise);
p.then(console.log); // OK
```

Resolve Reject

✅ 기본 문법

```js
Promise.reject(reason);
```

- `reason` : 거부(reject)되는 이유, 일반적으로 에러 객체나 에러 메시지를 전달합니다.

🔍 동작 설명

- `Promise.reject()` 는 **즉시 실패한(Promise가 reject된)** 상태의 Promise를 반환합니다.
- `.catch()` 나 `.then(_, rejectCallback)` 으로 에러를 처리할 수 있습니다.

📘 예제

1. 기본 사용

```js
Promise.reject("Something went wrong")
  .catch((error) => {
    console.error(error); // Something went wrong
  });
```

2. 에러 객체와 함께 사용

```js
Promise.reject(new Error("Invalid input"))
  .catch((err) => {
    console.error(err.message); // Invalid input
  });
```

기본 예제 1

```javascript
// 1초 후에 완료되는 Promise
function delay(ms, value) {
  return new Promise((resolve) => {
    setTimeout(() => {
      console.log(`완료: ${value}`);
      resolve(value);
    }, ms);
  });
}

// 사용 예시
delay(1000, "첫 번째").then(() => {
  console.log("첫 번째 끝");
});
```

기본 예제 1

- delay(ms, value) 함수는 Promise를 반환
- setTimeout을 사용하여 ms 밀리초 뒤에 실행됨
- 지정된 시간이 지나면:
  - console.log("완료: ...") 출력
  - resolve(value) 호출 → Promise가 fulfilled(이행됨) 상태가 되며 value를 결과값으로 전달

```javascript
// 1초 후에 완료되는 Promise
function delay(ms, value) {
  return new Promise((resolve) => {
    setTimeout(() => {
      console.log(`완료: ${value}`);
      resolve(value);
    }, ms);
  });
}

// 사용 예시
delay(1000, "첫 번째").then(() => {
  console.log("첫 번째 끝");
});
```

```
완료: 첫 번째
첫 번째 끝
```

기본 예제 2

```
delay(1000, "A")
  .then(() => delay(500, "B"))
  .then(() => delay(800, "C"))
  .then(() => {
    console.log("모두 끝!");
  });
```

기본 예제 2

- 1초 후 A
- 0.5초 후 B
- 0.8초 후 C
- 모두 끝!

```
delay(1000, "A")
  .then(() => delay(500, "B"))
  .then(() => delay(800, "C"))
  .then(() => {
    console.log("모두 끝!");
  });
```

```
완료: A
완료: B
완료: C
모두 끝!
```

직렬 예제

```
function runSerialWithThen() {
  const items = ["하나", "둘", "셋"];

  items
    .reduce((promiseChain, item) => {
      return promiseChain.then(() => delay(1000, item));
    }, Promise.resolve()) // 초기값: 즉시 완료되는 Promise
    .then(() => {
      console.log("끝");
    });
}

runSerialWithThen();
```

병렬 예제

```
async function runParallel() { const items = ["X", "Y", "Z"];
await Promise.all(items.map((item) => delay(1000, item)));
console.log("끝"); }
```

병렬 예제

- X, Y, Z의 순서는 보장되지 않음. 세 개가 동시에 시작하므로, 거의 동시에 출력되지만 자바스크립트 엔진이나 스케줄러 상황에 따라 순서는 달라질 수 있음.

```
async function runParallel() { const items = ["X", "Y", "Z"];
await Promise.all(items.map((item) => delay(1000, item)));
console.log("끝"); }
```

직렬 예제

```
function runSerialWithThen() {
  const items = ["하나", "둘", "셋"];

  items
    .reduce((promiseChain, item) => {
      return promiseChain.then(() => delay(1000, item));
    }, Promise.resolve()) // 초기값: 즉시 완료되는 Promise
    .then(() => {
      console.log("끝");
    });
}

runSerialWithThen();
```

직렬 예제

- Promise.resolve() → 시작점이 되는 "빈" Promise
- reduce를 이용해 각 item에 대해 체인을 이어붙임
- 첫 번째 아이템 "하나"는 1초 후 완료
- 그 다음 "둘"은 그 뒤에 1초 후 완료
- 마지막 "셋"도 이어서 실행
- 마지막 .then()에서 “끝” 출력

```
function runSerialWithThen() {
  const items = ["하나", "둘", "셋"];

  items
    .reduce((promiseChain, item) => {
      return promiseChain.then(() => delay(1000, item));
    }, Promise.resolve()) // 초기값: 즉시 완료되는 Promise
    .then(() => {
      console.log("끝");
    });
}

runSerialWithThen();
```

🧪 실습 문제: 순차 실행되는 비동기 작업 래퍼 만들기

문제 설명

비동기 작업을 순차적으로 실행해야 할 경우가 있습니다. 예를 들어, 저장 요청을 서버에 순서대로 보내야 할 때입니다.

이번 과제에서는 주어진 `mutateAsync` 함수를 기반으로, **항상 1개씩만 순서대로 실행되도록 보장하는 래퍼 함수를 만들어보세요.**

요구사항

- `createSerialMutation(mutateAsync)` 함수는 객체를 반환해야 합니다.
- 반환된 객체는 `mutateSerial(vars)` 메서드를 포함해야 합니다.
- `mutateSerial`이 여러 번 빠르게 호출되어도 **항상 순차적으로 실행**되어야 합니다.
- 각 호출은 Promise를 반환해야 하며, **성공/실패는 그대로 전달**되어야 합니다.
- 이전 작업이 실패하더라도 다음 작업은 **정상적으로 실행**되어야 합니다.

테스트용 비동기 함수

다음은 실제 서버 대신 사용하는 `setTimeout` 기반의 가짜 저장 함수입니다:

js 코드 복사

```
function fakeSave(input, ms = 200) {
  return new Promise((resolve, reject) => {
    setTimeout(() => {
      resolve({ ok: true, input });
    }, ms);
  });
}
```

기대 결과

다음 코드로 테스트할 때:

js 코드 복사

```
const saveSerial = createSerialMutation(fakeSave);

saveSerial
  .mutateSerial({ text: 'A' })
  .then(r => console.log('A', r));

saveSerial
  .mutateSerial({ text: 'B' })
  .then(r => console.log('B', r));

saveSerial
  .mutateSerial({ text: 'C' })
  .then(r => console.log('C', r));
```

콘솔 출력 순서는 항상 다음과 같아야 합니다:

yaml 코드 복사

```
A { ok: true, input: { text: 'A' } }
B { ok: true, input: { text: 'B' } }
C { ok: true, input: { text: 'C' } }
```

응용 (React Query)

```ts
'use client';

import { useCallback, useRef } from 'react';
import {
  useMutation,
  UseMutationOptions,
  UseMutationResult,
} from '@tanstack/react-query';

/**
 * 직렬화: mutateAsync 호출을 큐에 연결하여 항상 1개씩 순차 실행
 * - runningRef는 이전 작업의 Promise를 기억하고, 새 작업을 그 뒤에 연결
 * - 앞선 실패가 뒤에 영향 주지 않도록 catch로 체인 정리
 */
export function useSerialMutation<TData, TError, TVars>(
  mutationOptions: UseMutationOptions<TData, TError, TVars>
): UseMutationResult<TData, TError, TVars> & {
  mutateSerial: (vars: TVars) => Promise<TData>;
} {
  const base = useMutation<TData, TError, TVars>(mutationOptions);
  const runningRef = useRef<Promise<unknown> | null>(null);

  const mutateSerial = useCallback(
    async (vars: TVars) => {
      const run = async () => base.mutateAsync(vars);

      // 기존 체인이 있으면 그 뒤에 연결
      runningRef.current = (runningRef.current ?? Promise.resolve())
        .catch(() => {}) // 이전 실패로 체인이 끊기지 않게
        .then(run);

      // 타입 보정을 위해 as unknown as Promise<TData>
      return runningRef.current as unknown as Promise<TData>;
    },
    [base]
  );

  return Object.assign(base, { mutateSerial });
}
```