Promise?

TypeScript에서 Promise는 비동기 작업의 완료 또는 실패를 나타내는 객체로, 자바스크립트의 Promise에 타입 안전성을 더한 형태입니다.

비동기 작업을 순차적으로 실행하기 위해 JS에서는 콜백 함수를 사용합니다. 콜백 함수는 특정 로직이 끝났을 때 원하는 코드를 실행할 수 있습니다. 하지만 콜백에 또 콜백을 계속해서 호출하는 방식으로 코드를 구성하면 복잡해지고 에러 처리도 어려워집니다. 이런 단점은 Promise를 사용해서 해결할 수 있습니다.

Promise의 기본 구조와 타입 정의

```
// 제네릭 타입을 사용하여 비동기 작업의 결과 타입을 명시할 수 있습니다.

const numPromise: Promise<number> = new Promise<number>((resolve, reje
  // 비동기 작업 코드
});

const strPromise: Promise<string> = new Promise<string>((resolve, reject) =>
  // 비동기 작업 코드
});

const arrayPromise: Promise<number[]> = new Promise<number[]>((resolve,
  // 비동기 작업 코드
});

// 위의 Promise<any> 는 Promise가 비동기 작업이 완료된 후
// 최종적으로 반환할 값의 타입을 명시하는 역할을 합니다.
```

> Promise 생성자의 콜백 함수는 `resolve`와 `reject` 두 함수를 매개변수로 받습니다.

```
new Promise<T>((resolve: (successValue: T) ⇒ void, reject: (reason: any) ⇒ v
  // 비동기 코드 구현
});

// 위 코드는 아래의 js코드와 동일한 동작을 구현합니다.

const promise = new Promise(function(resolve, reject) {
  // 비동기 작업 수행
  // 성공 시: resolve(결과값)
  // 실패 시: reject(에러)
});

//  executor라 불리는 콜백 함수를 인자로 받으며,
// 이 executor는 resolve와 reject 두 함수를 매개변수로 전달받습니다.
```

Promise의 메서드

Promise의 주요 메서드는 `then`과 `catch`입니다.

```
const promise = new Promise<string>((resolve, reject) ⇒ {
  const success = true;
  if (success) {
  // resolve 함수는 promise 비동기 작업이 성공한 경우 동작합니다.
    resolve("작업 성공");
  } else {
    reject("작업 실패");
  }
});

promise
  .then((result) ⇒ {
    console.log(result); // '작업 성공' 출력
  })
  .catch((error) ⇒ {
    console.error(error); // '작업 실패' 출력
  });
```

async/await 와 차이점

더 동기적인 코드 스타일입니다. 비동기 코드를 마치 동기 코드처럼 작성할 수 있습니다.

```
async function fetchData() {
  try {
    const response = await fetch('https://api.example.com/data');
    const data = await response.json();
    console.log(data);
  } catch (error) {
    console.error('Error:', error);
  }
}
```

▼ 에러 처리에서의 차이점

Promise: `.catch()` 메서드를 사용하여 에러를 처리합니다.

async/await: 일반적인 try/catch 구문을 사용하여 에러를 처리합니다.

async/await는 Promise객체를 기반으로 작동하며, async 함수는 항상 Promise객체를 반환합니다.

즉, async/await는 Promise의 문법적 설탕(Syntactic sugar)이라고 볼 수 있습니다.

코드가 길어질수록 가독성이 좋고 흐름을 이해하기 쉽습니다.

▼ 병렬 처리에서의 차이점

Promise: `Promise.all()`, `Promise.race()` 등의 메서드를 통해 여러 Promise를 쉽게 병렬 처리할 수 있습니다.

async/await: 병렬 처리를 위해서는 추가적인 작업이 필요하며, 일반적으로 `Promise.all()`과 함께 사용합니다.

결론

실무에서는 Promise 기반 API나 라이브러리를 사용하면서, 이를 처리하는 코드는 async/await 문법으로 작성하는 패턴이 매우 일반적입니다. 특히 복잡한 비동기 로직을 다룰 때 Promise 체이닝보다 async/await를 사용하면 코드의 가독성이 크게 향상됩니다.