[JS-deep-Dive]. 45장 프로미스

Status	완료

프로미스

프로미스의 등장

> 콜백함수로 비동기 처리를 하는것의 단점을 보완하기 위해 등장하게됨.

- 콜백 패턴
    - 비동기 함수의 처리 결과를 위해 비동기 함수 내부에서 콜백함수를 전달함
    - 비동기 처리의 성공과 실패에따라 호출될 콜백을 전달하여 에러처리

▼ 콜백 패턴의 단점

- 콜백 헬
  - 비동기 결과의 처리를 수행하는 함수가 비동기 결과를 리턴한다면, 다시 또 콜백함수를 전달하고.. 가 반복되어 콜백 지옥에 빠지게 됨
  - 콜백헬로 인해 가독성이 나빠지고 실수를 유발하게 됨
  - 비동기 함수는 콜백함수가 호출되는것을 기다리지 않고 즉시 종료되어 스택에서 제거됨. 에러는 콜스택의 아래 방향으로 전파되는데, 비동기 함수는 이미 스택에서 제거 되었으므로 `비동기 함수에서 catch 문을 사용했더라도 콜백함수에 에러를 잡아내지 못한다.`

프로미스의 생성

> ▫ promis 생성자 함수는 비동기 처리를 수행할 콜백함수를 인수로 받는다, 그리고 이 콜백함수는 비동기 처리가 성공하면 호출할 resolve와 비동기 처리가 실패하면 호출할 reject함수를 인수로 전달 받음
>
> ▫ 프로미스는 비동기 처리 상태와 처리 결과를 관리하는 객체

```
const test = new Promise((resolve, reject) => {
    const 성공 = true; // 성공 여부를 설정 (true: 성공, false: 실패

    if (성공) {
        resolve('작업 성공!');
    } else {
        reject('작업 실패!');
    }
});
```

▼ 프로미스 객체와 처리 결과의 상태

1.pending

프로미스가 생성된 직후 기본상태, 비동기 처리 진행중

2. fulfilled
   a. 비동기 처리가 성공하여 resolve 함수를 호출, 비동기 처리 결과를 값으로 가진다.
3. rejected

a. 비동기 처리가 실패하여 reject 함수 호출

b. 비동기 처리 결과인 error객체를 값으로 갖는다.

4. settled상태

   fulfilled or rejected 상태

   실패, 성공에 관련없이 pending 이 아닌 상태로, 비동기가 처리가 수행된 상태를 의미, setted 상태가 되면 더는 다른 상태로 변화할수없다.

프로미스의 후속 처리 메소드

> ▫프로미스의 상태가 pending 에서 변화하면, 결과값을 가지고 동작을 해야한다. 이를 위해 프로미스는 catch, then, finally 메소드를 제공한다.
>
> ▫후속처리 메소드에 인수로 전달할 콜백함수가 선택적으로 호출된다. 그리고 그 콜백함수에 프로미스의 처리 결과가 인수로 전달된다.

프로미스 후속처리 메소드 (then, catch, finally)

> 모든 후속 처리 프로미스는 프로미스를 반환하며, 비동기로 동작한다.

1. then

   ▫두개의 콜백함수를 인수로 전달받는다.

   ▫첫번째 함수 : resolve가 호출되었을때 처,. 이때 비동기 처리 결과를 인수로 전달 받는다

   ▫두번째 함수 : reject가 호출되었을때 처리, 에러를 인수로 전달받음

```
const test = new Promise((resolve, reject) => {
    const 성공 = true; // 성공 여부를 설정 (true: 성공, false:
    if (성공) {
        resolve('작업 성공!');
    } else {
        reject('작업 실패!');
    }
});
```

```
test.then(
    (result) => {
        // 첫 번째 콜백: 성공 시 실행
        console.log(result);
    },
    (error) => {
        // 두 번째 콜백: 실패 시 실행
        console.error(error);
    }
);
```

2. catch
   - 한개의 콜백함수를 인수로 전달받음
   - 프로미스가 rejected 상태인경우에만 호출됨
3. finally
   - 한개의 콜백 함수를 인수로 전달 , 성공이나 실패와 상관없이 무조건 한번 호출,

프로미스의 에러처리

> - then 메소드의 두번째 콜백으로 에러 처리가 가능하나, 첫번째 콜백에서의 에러는 잡아내지 못하고, 가독성이 부족
> - catch 메서드를 then 메서드 이후에 호출하면 비동기 처리에서 발생한 에러, then 메소드 에서 발생한 에러까지 모두 캐치하므로, 이방법을 권장한다.

```
const test = new Promise((resolve, reject) => {
    const 성공 = true; // 성공 여부를 설정 (true: 성공, false: 실패
    if (성공) {
        resolve('작업 성공!');
    } else {
        reject('작업 실패!');
    }
```

```
});

test.then(
    (result) => {
        console.log('성공:', result);
    })
    .catch((error) => {
        console.error('실패:', error);
    });
```

- 프로미스 체이닝

  위에서 봤듯이 promise 의 후속처리 메소드는 언제나 프로미스를 반환하므로 연속적으로 호출 할 수 있다. 이를 프로미스 체이닝 이라고 한다.

프로미스의 정적 메서드

> promise 인스턴스에서 호출하는것이 아니라, promise 객체에서 호출하는 메소드

1. Promise.resolve / Promise.reject

▫ resolve : 메소드에 이미 존재하는 값을 래핑하여 프로미스를 생성하기 위해 사용

> 비동기 작업 없이 즉시 성공 상태의 promise 를 반환할때 유용

▫ reject : 인수로 전달 받은 값을 reject하는 프로미스 생성

> 특정 상황에서 즉시 실패 상태의 promise를 반환할때 사용

▼ 메소드에 이미 존재하는 값을 래핑?

- 존재하는 값을 감싸서 프로미스 객체로 변환한다. 즉, 값을 프로미스처럼 취급함
- 특정값을 입력받아서 즉시 fulfilled 상태의 프로미스를 반환한다.
- 값을 프로미스처럼 취급해주면, 후속 메소드를 통해 비동기적으로 사용이 가능함.

2. Promise.all

> 여러 비동기 작업을 동시에 실행하고, **모든 작업이 완료된 후에 처리**가 필요할 때 활용

1. 여러개의 비동기 처리를 모두 병렬 처리할때 사용.
2. 프로미스를 요소로 갖는 배열등의 이터러블 을 인수로 전달
3. 모든 요소가 fulfilled 상태가 되면 모든 처리 결과를 배열에 저장해 새로운 프로미스를 반환
4. 하나라도 reject 상태가 되면 즉시 종료함
5. 인수로 전달받은 이터러블 요소가 프로미스가 아닌경우 promise. resolve 메소드를 통해 프로미스로 래핑함

3. Promise.race

1. 프로미스를 요소로 갖는 배열 등의 이터러블을 인수로 전달 받는다. fulfilled 상태가 된 프로미스 처리 결과를 resolve하는 새로운 프로미스를 반환한다.
2. reject상태가 되면 에러를 reject하는 새로운 프로미스를 즉시 반환함.

4. Promise.allSettled

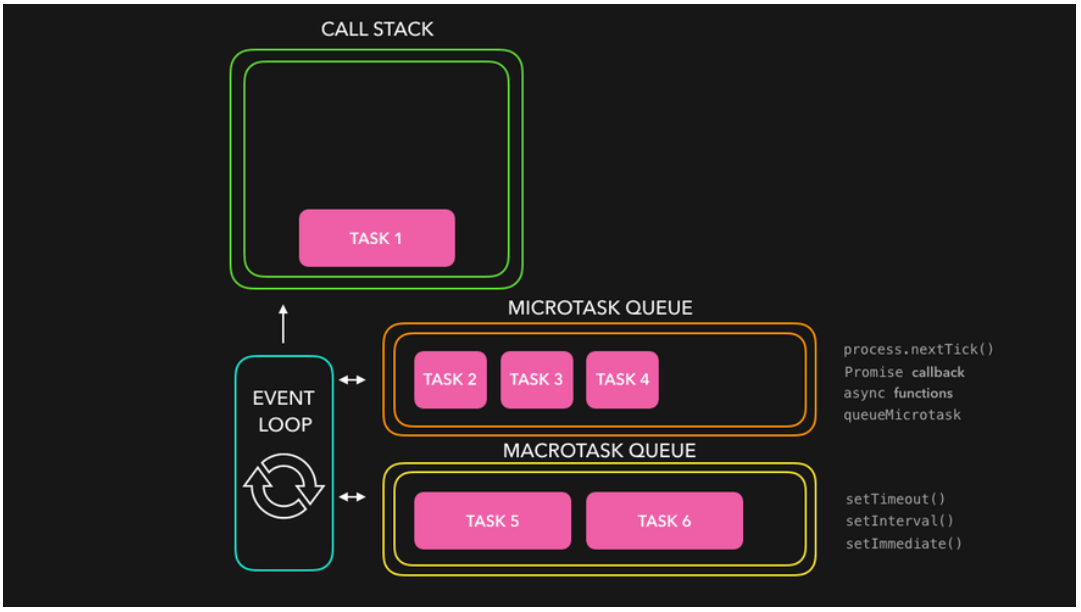
> 모든 Promise의 상태를 확인하고 싶을 때 활용

1. 프로미스를 요소로 갖는 이터러블을 인수로 전달 받는다. 전달받은 프로미스가 모두 settled 상태면 처리 결과를 배열로 반환
2. 프로미스의 처리 결과를 나타내는 객체의 배열을 갖는데, fulfilled 상태인경우에는 상태값인 status와 처리 결과값인value 프로퍼티를 갖는다
3. reject 상태인경우 처리 상태를 나타내는 status, 에러를 나타내는 reason 프로퍼티를 가진다.

마이크로 태스크 큐

> ▫마이크로 태스크 큐에는 프로미스의 후속처리 메소드의 콜백함수가 일시 저장됨.
>
> ▫그 외의 비동기 함수의 콜백함수나 이벤트 핸들러는 태스크 큐에 일시 저장됨.
>
> ▫ 마이크로 태스크 큐는 태스크 큐보다 우선순위가 높음

- 이벤트 루프의 동작
  1. 호출스택이 비었는지 지속적으로 확인
  2. 호출 스택이 비게 되면 제일 먼저 마이크로 태스크 큐를 확인하고 가장 오래된 태스크부터 꺼내서 호출스택으로 전달해 주는데, 이걸 **마이크로태스크 큐가 다 빌때까지 수행**
  3. 모든 마이크로태스크가 처리된 직후, 렌더링 작업이 필요하면 렌더링을 수행
  4. 매크로 태스크 큐를 확인
  5. 매크로 태스크 큐에서 가장 오래된 태스크 하나를 꺼내 호출 스택에 전달
  6. 위를 반복



- 매크로 태스크 큐 (=콜백큐, 이벤트 큐)
  - DOM 이벤트의 콜백, 타이머, 스크립트 로딩 등

- 마이크로 태스크 큐
  - 프로미스, 후속메소드, await , Object.observe, process등

fetch

> ▫ `fetch` 함수는 기본적으로 Promise를 반환, 따라서 별도로 `new Promise()`를 생성할 필요 없이 바로 체이닝(`.then`, `.catch`)을 사용하여 비동기 처리가 가능 한 **JavaScript API**
>
> ▫위의 명시적으로 promise 를 생성하는것보다 더 권장되는 방식
>
> ▫http 응답을 나타내는 response 객체를 래핑한 프로미스를 반환하므로 후속 처리 메소드 then 을 통해 프로미스가 resolve 한 response 객체를 전달 받을 수 있다.

```
const test = new Promise((resolve, reject) => {
    const 성공 = true; // 성공 여부를 설정 (true: 성공, false: 실패
    if (성공) {
        resolve('작업 성공!');
    } else {
        reject('작업 실패!');
    }
});

test.then(
    (result) => {
        console.log('성공:', result);
    })
    .catch((error) => {
        console.error('실패:', error);
    });
```

response 의 프로토 타입의 메소드

- json메소드

fetch 함수가 반환한 mimi 타입을 application/json인 http 바디를 받으려면 json 메소드를 사용한다.

리스폰스 객체에서 body를 취득하여 역직렬화 한다.

fetch 함수의 에러처리

- fetch 함수가 반환하는 프로미스는 기본적으로 에러가 발생해도 에러를 reject 하지 않고 불린 타입의 ok 상태를 false로 설정한 response 객체를 resolve 한다.
- 오프라인 등의 네트워크 장애나 cors 에러에 의해 요청이 완료되지 못한 경우에만 프로미스를 reject한다.
- 따라서 fetch 함수가 반환한 프로미스가 resolve한 불린 타입의 ok 상태를 확인해 명시적으로 에러를 처리할 필요가 있음

axios

> axios 모든 에러를 reject 하는 프로미스를 반환, 따라서 모든 에러를 catch에서 처리할수 있어 편리함. 또한 axios는 인터셉터, 요청 설정 등 fetch 보다 다양한 기능을 지원

▼ 질문 답변

- 명시적으로 promise 를 생성하는 것은 **이벤트 기반 작업**, **특정 조건에서 비동기 작업 제어**, **타임아웃 또는 취소 처리**와 같은 고급 비동기 로직에서 유용함.
- 그러나 일반적인 비동기 작업에서는 async/await 를 사용하는 것이 훨씬 간단하고 가독성이 측면에서 좋음 따라서 두 방법을 적절히 혼합해서 사용하는 것이 가장 효율적

1. 특정 조건이나 시점에 비동기 처리를 제어

- promise 객체를 직접 생성하면 resolve와 reject를 명시적으로 호출할 수 있기 때문에, 특정 조건이나 시점에 비동기 처리를 제어할 때 유용

- 후속 처리 메서드는 비동기 작업이 끝난 뒤 결과를 처리하기 위한 도구, 하지만 명시적으로 promise 를 생성하는 건 **비동기 작업의 완료를 개발자가 원하는 시점에서 제어**해야 할 때 필요함

```
function test(ms) {
  return new Promise((resolve) => {
    setTimeout(() => {
      ;//setTimeout으로 지정한 시간후에 비동기 완료
      console.log('작업완료')
      resolve();
    }, ms);
  });
}

// 사용
test(1000).then(() => console.log("작업 완료"));
```

2. 일정시간후에 에러처리

```
function test(url, timeout) {
  return new Promise((resolve, reject) => {
    const timer = setTimeout(() => {
      reject(new Error("timed out"));
    }, timeout);

    fetch(url)
      .then((response) => {
        clearTimeout(timer);//timeOut을 꼭 취소해야한다.
        resolve(response);
      })
      .catch((error) => {
        clearTimeout(timer);
        reject(error);
      });
  });
}

// 사용
```

```
test("https://api.example.com", 3000)
  .then((response) => console.log(response))
  .catch((error) => console.error("에러 발생", error));
```

▼ -

fetch 함수와 promise를 명시적으로 생성하는것의 차이

1. `fetch` 함수와 Promise 체이닝

`fetch` 함수는 기본적으로 Promise를 반환합니다. 따라서 별도로 `new Promise()`를 생성할 필요 없이 바로 체이닝(`.then`, `.catch`)을 사용하여 비동기 처리가 가능

예제

```
fetch('https://jsonplaceholder.typicode.com/posts/1')
    .then((response) => {
        if (!response.ok) {
            throw new Error('HTTP 오류!');
        }
        return response.json(); // 응답 데이터를 JSON으로 변환
    })
    .then((data) => {
        console.log('데이터:', data); // 성공적으로 데이터를 처리
    })
    .catch((error) => {
        console.error('에러 발생:', error.message); // 에러 처
    });
```

2. 명시적으로 `new Promise`를 생성하는 방식

`fetch`는 이미 Promise를 반환하기 때문에 `new Promise()`**로 감싸는 것은 불필요한 경우가 많습니다.** 하지만 명시적으로 Promise를 생성하면 더 세부적인 비동기 작업 흐름을 정의하거나, 조건에 따라 `resolve`나 `reject`를 호출할 수 있습니다.

예제

```
const fetchWithPromise = () => {
    return new Promise((resolve, reject) => {
        fetch('https://jsonplaceholder.typicode.com/posts/
            .then((response) => {
                if (!response.ok) {
                    reject(new Error('HTTP 오류!')); // 명시
                } else {
                    response.json().then(resolve); // 명시적
                }
            })
            .catch((error) => {
                reject(error); // 네트워크 에러 처리
            });
    });
};

// 사용 예
fetchWithPromise()
    .then((data) => {
        console.log('데이터:', data);
    })
    .catch((error) => {
        console.error('에러 발생:', error.message);
    });
```

3. 두방식의 차이

구분	`fetch` 체이닝 방식	`new Promise` 생성 방식
코드 간결성	간결하고 직관적 (`fetch` 자체가 Promise 반환).	비교적 복잡하고 코드가 길어짐.
추가 제어 필요성	기본적인 비동기 작업 처리에 적합.	세부적인 `resolve` / `reject` 호출이 필요할 때 유용.
성능	성능적으로 더 효율적 (`fetch` 자체를 활용).	불필요한 래핑으로 성능에 약간의 오버헤드 발생.
사용 용도	간단한 비동기 로직 처리.	복잡한 작업 흐름 제어나 커스터마이징 필요 시.

4. 권장되는 방법

`fetch` 체이닝을 사용하는 것이 더 권장됩니다.

- **이유:**
  1. `fetch` 자체가 Promise를 반환하므로, 추가적으로 `new Promise`를 생성할 필요가 없습니다.
  2. 코드가 간결하고 읽기 쉬워 유지보수가 용이합니다.
  3. 성능 측면에서도 더 효율적입니다.

명시적으로 `new Promise`를 사용하는 경우

- 특정 상황에서 `resolve` / `reject` 호출을 세밀하게 제어해야 하거나, `fetch`와 다른 비동기 로직을 결합해야 할 때 사용합니다.
- 예: 파일 읽기, 여러 비동기 작업을 묶어서 처리할 때.

xmlHttpRequest?

`XMLHttpRequest`(XHR)는 **비동기 데이터 요청**을 웹 서버에 보낼 때 사용되는 자바스크립트 객체입니다. 브라우저에서 AJAX(Asynchronous JavaScript and XML) 요청을 보낼 수 있게 해주는 기본 도구 중 하나입니다.

`XMLHttpRequest`는 브라우저가 제공하는 API 중 하나이며, 오늘날 `fetch` API와 같은 최신 기술이 많이 사용되지만, 여전히 일부 구형 프로젝트나 특정 상황에서 사용됩니다.

특징

1. **비동기 통신 지원**:
   - 페이지를 새로고침하지 않고 서버와 데이터를 주고받을 수 있습니다.
   - 이를 통해 빠르고 동적인 웹 애플리케이션을 만들 수 있습니다.
2. **HTTP 요청 전송**:
   - GET, POST, PUT, DELETE 같은 HTTP 메서드를 사용하여 요청을 보낼 수 있습니다.
3. **다양한 데이터 형식 지원**:
   - XML, JSON, HTML, 텍스트 등 다양한 형식으로 데이터를 주고받을 수 있습니다.

예제

```
const xhr = new XMLHttpRequest(); // XMLHttpRequest 객체 생

xhr.open('GET', 'https://jsonplaceholder.typicode.com/post
xhr.onreadystatechange = function () {
    if (xhr.readyState === 4) { // 요청이 완료되었을 때 (ready
        if (xhr.status === 200) { // HTTP 상태 코드가 200일 때
            console.log('응답 데이터:', xhr.responseText);
        } else {
            console.error('오류 발생:', xhr.status);
        }
    }
};
xhr.send(); // 요청 전송
```

코드 설명:

1. `new XMLHttpRequest()`: XHR 객체 생성.
2. `open(method, url, async)`: 요청 초기화.
   - `method`: HTTP 메서드(GET, POST 등).
   - `url`: 요청할 서버 주소.
   - `async`: 비동기 여부(`true`가 기본).

3. `onreadystatechange`: 요청 상태가 변경될 때 호출될 콜백 함수.
    - `readyState`: 요청 상태(0~4).
        - `0`: 요청 초기화되지 않음.
        - `1`: 연결 설정 완료.
        - `2`: 요청 수신됨.
        - `3`: 요청 처리 중.
        - `4`: 요청 완료.
    - `status`: HTTP 상태 코드(예: `200`은 성공, `404`는 페이지 없음).
4. `send()`: 요청 전송.

XMLHttpRequest와 fetch의 차이점

특징	XMLHttpRequest	fetch API
문법	콜백 기반	Promise 기반
사용성	복잡하고, 추가 설정 필요	간결하고 사용이 직관적
JSON 데이터 처리	JSON 파싱 필요 (`JSON.parse()`)	내장된 `.json()` 메서드 지원
에러 처리	`onerror`를 통해 수동 처리	`.catch()`로 간단히 처리 가능
프로미스	지원하지 않음 (구형 방식)	Promise와 async/await 지원 (최신 방식)
CORS 지원	제한적	더 나은 CORS 정책 관리

결론

- `XMLHttpRequest`는 **구형 방식**으로, 현대적인 코드에서는 잘 사용되지 않습니다.
- 대신 `fetch` **API**가 더 권장되며, 비동기 작업에서는 `Promise` 기반의 `fetch` 또는 `axios` 같은 라이브러리를 사용하는 것이 일반적입니다.
- 하지만 `XMLHttpRequest`는 여전히 구형 브라우저나 기존 코드베이스에서 유지보수를 위해 사용될 수 있습니다. 이를 이해하는 것은 과거와의 호환성을 위해 중요합니다!

- 참고 : 마이크로와 매크로 태스크 https://whales.tistory.com/130
- 프로미스- javascript.info

https://ko.javascript.info/promise-basics