45. 프로미스

45.4. 프로미스의 에러 처리

프로미스는 콜백 패턴의 에러 처리가 곤란하다는 문제점을 해결

비동기 처리에서 발생한 에러는 then 메서드의 두 번째 콜백 함수로 처리 가능

```
const wrongUrl = 'https://jsonplaceholder.typicode.com/xxx/1';
promiseGet(wrongUrl).then(res => console.log(res), err => console.error(err));
```

비동기 처리에서 발생한 에러는 프로미스의 후속 처리 메서드 catch를 사용해 처리할 수 있음

```
const wrongUrl = 'https://jsonplaceholder.typicode.com/xxx/1';
promiseGet(wrongUrl).then(res => console.log(res)).catch(err => console.error(err));
```

catch 메서드를 호출하면 내부적으로 then을 호출. 위 예제는 내부적으로 다음과 같이 처리

```
const wrongUrl = 'https://jsonplaceholder.typicode.com/xxx/1';
promiseGet(wrongUrl)
.then(res => console.log(res));
.then(undefined, err => console.error(err));
```

then 메서드의 두 번째 콜백 함수는 첫 번째 콜백 함수에서 발생한 에러를 캐치하지 못하고, 코드가 복잡해져서 가독성이 좋지 않다

```
promiseGet('https://jsonplaceholder.typicode.com/todos/1').then(
res => console.xxx(res),
err => console.error(err))
```

catch 메서드를 모든 then 메서드를 호출한 이후에 호출하면, 비동기 처리에서 발생한 에러 뿐만 아니라, then 메서드 내부에서 발생한 에러까지 모두 캐치 가능

또한 두 번째 콜백 함수를 전달하는 것 보다, catch 메서드를 사용하는 것이 가독성이 좋고 명확함.

45.5 프로미스 체이닝

비동기 처리를 위한 콜백 패턴은 콜백 헬이 발생하는 문제가 존재

프로미스는 후속 처리 메서드를 통해 콜백 헬 해결

```
const url = 'https://jsonplacehodler.typicode.com';

promiseGet(`${url}/posts/1`);
.then({userId} => promiseGet(`${url}/users/${userId}`))
.then(userInfo => console.log(userInfo))
.catch(err => console.error(err))
```

후속 처리 메서드를 연속적으로 호출하는 것을 프로미스 체이닝이라고 한다.

프로미스 체이닝 과정

Aa 후속 처리 메서드	콜백 함수의 인수	후속 처리 메서드의 반환값
then	promiseGet 함수가 반환한 프로미스가 resolve한 값(id가 1인 post)	콜백 함수가 반환한 프로미스
then	첫 번째 then 메서드가 반환한 프로미스가 resolve한 값(post의 userId로 취득한 user 정보)	콜백 함수가 반환한 값(undefined)을 resolve한 프로미스
catch	promiseGet 함수 또는 앞선 후속 처리 메서드가 반환한 프로미스가 reject한 값	콜백 함수가 반환한 값(undefined)을 resolve한 프로미스

이처럼 후속 처리 메서드는 콜백 함수가 반환한 프로미스를 반환. 콜백 함수가 프로미스가 아닌 값을 반환하더라도 그 값을 암묵적으로 resolve 또는 reject하여 프로미스를 생성해 반환

프로미스는 프로미스 체이닝을 통해서 후속 처리를 하므로, 비동기 처리를 위한 콜백 패턴에서 사용하던 콜백 헬이 발생하지 않음. 다만 프로미스도 콜백 패턴을 사용하므로, 콜백 함수를 사용하지 않는 것은 아님.

콜백 패턴은 가독성이 좋지 않음. 이 문제는 ES8에서 도입된 async/await를 통해서 프로미스의 후속 처리 메서드 없이 마치 동기 처리처럼 프로미스가 처리 결과를 반환하도록 구현 가능.

```
const url = 'https://jsonplaceholder.typicode.com';

(async () => {
  const { userId } = await promiseGet(`${url}/posts/1`);

  const userInfo = await promiseGet(`${url}/users/${userId}`);
})();
```

45.6 프로미스이 정적 메서드

Promise는 5가지 정적 메서드를 제공

45.6.1 Promise.resolve / Promise.reject

Promise.resolve와 Promise.reject는 이미 존재하는 값을 래핑하여 프로미스를 생성하기 위해 사용

Promise.resolve 메서드는 인수로 전달 받은 값을 resolve하는 프로미스 생성

```
const resolvedPromise = Promise.resolve([1,2,3]);
resolvedPromise.then(console.log); // [1,2,3]
```

위 예제는 다음과 동일하게 동작

```
const resolvedPromise = new Promise(resolve => resolve([1,2,3]));
resolvedPromise.then(console.log)
```

Promise.reject 메서드는 인수로 전달 받은 값을 reject하는 프로미스 생성

```
const rejectedPromise = Promise.reject(new Error('error'));
rejectedPromise.catch(console.log)
```

45.6.2 Promise.all

여러 개의 비동기 처리를 모두 병렬 처리할 때 사용

```
const requestDetail1 = () => new Promise(resolve => setTimeout(() => resolve(1), 3000));
const requestDetail2 = () => new Promise(resolve => setTimeout(() => resolve(2), 2000));
const requestDetail3 = () => new Promise(resolve => setTimeout(() => resolve(3), 1000));
```

```
const res = [];

requestDetail1().then(data => { res.push(data); return requestData2(); })
                .then(data => { res.push(data); return requestData3(); })
                .then(data => { res.push(data); console.log(res) }) // [1, 2, 3] 약 6초 소요
                .catch(console.error);
```

위 예제는 첫 번째 비동기 처리에 3초, 두 번째 비동기 처리에 2초, 세 번째 비동기 처리에 1초가 소요되어 총 6초가 소요된다.

위 예제의 경우, 세 개의 비동기 처리는 서로 의존하지 않고 개별적으로 수행된다. 따라서 순차적으로 처리할 필요가 없다.

이럴때 Promise.all을 사용할 수 있다.

```
const requestDetail1 = () => new Promise(resolve => setTimeout(() => resolve(1), 3000));
const requestDetail2 = () => new Promise(resolve => setTimeout(() => resolve(2), 2000));
const requestDetail3 = () => new Promise(resolve => setTimeout(() => resolve(3), 1000));

Promise.all([requestDetail1(), requestDetail2(), requestDetail3()])
      .then(console.log) // [1 ,2, 3] 약 3초 소요
      .catch(console.error);
```

Promise.all 메서드는 프로미스를 요소로 갖는 배열 등의 이터러블을 인수로 전달받는다.

전달받은 모든 프로미스가 fulfilled 상태가 되면, 모든 처리 결과를 배여렝 저장해 새로운 프로미스를 반환.

Promise.all 메서드가 종료하는 데 걸리는 시간은, 가장 늦게 fulfilled 상태가 되는 프로미스의 처리 기산보다 조금 더 길다. 위 예제의 경우, 3초보다 조금 더 소요된다.

만약 인수로 전달받은 배열의 프로미스가 하나라도 rejected 상태가 되면, 나머지 프로미스가 fulfilled 상태가 되는 것을 기다리지 않고 즉시 종료된다.

```
const requestDetail1 = () =>
  new Promise((_, reject) =>
    setTimeout(() => reject(new Error("Error 1")), 3000)
  );
const requestDetail2 = () =>
  new Promise((_, reject) =>
    setTimeout(() => reject(new Error("Error 2")), 2000)
  );
const requestDetail3 = () =>
  new Promise((_, reject) =>
    setTimeout(() => reject(new Error("Error 3")), 1000)
  );

Promise.all([requestDetail1(), requestDetail2(), requestDetail3()])
  .then(console.log)
  .catch(console.error); // Error 3
```

위 예제의 경우, 세 번째 프로미스가 가장 먼저 reject되므로, 세 번째 프로미스가 reject한 에러가 catch 메서드로 전달.

만약 Prmoise.all 메서드의 인수로 전달받은 이터러블 요소가 프로미스가 아닌 경우, Promise.resolve 메서드를 통해서 프로미스로 래핑한다.

```
Promise.all([
  1, // -> Promise.resolve(1)
  2, // -> Promise.resolve(2)
  3, // -> Promise.resolve(3)
])
.then(console.log) // [1, 2, 3]
.then(console.log);
```

다음은 깃 허브 아이디로 깃 허브 사용자 이름을 취득하는 3개의 비동기 처리를 모두 병렬로 처리하는 예제이다.

```
const promiseGet = (url) => {
  return new Promise((resolve, reject) => {
    const xhr = new XMLHttpRequest();
    xhr.open("GET", url), xhr.send();
    xhr.onload = () => {
      if (xhr.status === 200) {
        // 성공적으로 응답을 전달받으면 resolve하는 함수를 호출
        resolve(JSON.parse(xhr.response));
      } else {
        resolve(new Error(xhr.status));
      }
    };
  });
};

const githubIds = ["jeresig", "ahejilsberg", "ungmo2"];

Promise.all(
  githubIds.map((id) => promiseGet(`https://api.github.com/users/${id}`))
) // Promise [userInfo, userInfo, userInfo]
  .then((users) => users.map((user) => user.name))
  .then(console.log)
  .catch(console.error);
```

45.6.3 Promise.race

Promise.race 역시도 Promise.all과 동일하게 프로미스를 요소로 갖는 배열 등의 이터러블을 인수로 전달받음.

Promise.all 메서드처럼 모든 프로미스가 fulfilled 상태가 되는 것을 기다리는 것이 아니라, 가장 먼저 fulfilled 상태가 된 프로미스의 처리 결과를 resolve하는 새로운 프로미스를 반환

```
Promise.race([
  new Promise((resolve) => setTimeout(() => resolve(1), 3000)),
  new Promise((resolve) => setTimeout(() => resolve(2), 2000)),
  new Promise((resolve) => setTimeout(() => resolve(3), 1000)),
])
  .then(console.log) // 3
  .catch(console.log);
```

프로미스가 rejected 상태가 되면 Promise.all 메서드와 동일하게 처리. 즉, Promise.race 메서드에 전달된 프로미스가 하나라도 rejected 상태가 되면, 에러를 reject하는 새로운 프로미스를 즉시 반환

```
Promise.race([
  new Promise((_, reject) =>
    setTimeout(() => reject(new Error("Error 3")), 3000)
  ),
  new Promise((_, reject) =>
    setTimeout(() => reject(new Error("Error 2")), 2000)
  ),
  new Promise((_, reject) =>
    setTimeout(() => reject(new Error("Error 1")), 1000)
  ),
])
  .then(console.log)
  .catch(console.log); // Error 1
```

45.6.4 Promise.allSettled

promise.allSettled 메서드는 프로미스를 요소로 갖는 배열 등의 이터러블을 인수로 전달받는다. 그리고 전달받은 프로미스가 모두 settled 상태가 되면, 처리 결과를 배열로 반환한다.

```
Promise.allSettled([
  new Promise((resolve) => setTimeout(() => resolve(1), 2000)),
  new Promise((_, reject) =>
    setTimeout(() => reject(new Error("Error!")), 1000)
  ),
]).then(console.log);
```

```
// [
//   { status: 'fulfilled', value: 1 },
//   {
//     status: 'rejected',
//     reason: Error: Error!
//         at Timeout._onTimeout (c:\Users\kyh\Desktop\coding\Algo\pratctice2 copy.js:25:29)
//         at listOnTimeout (internal/timers.js:557:17)
//         at processTimers (internal/timers.js:500:7)
//   }
// ]
```

메서드가 반환한 배열에는 fulfilled 또는 rejected 상태와 상관 없이, Promise.allSettled 메서드가 인수로 전달받은 모든 프로미스들의 처리 결과가 담겨 있음.

- 프로미스가 fulfilled 상태인 경우, 비동기 처리 상태를 나타내는 status 프로퍼티와 처리 결과를 나타내는 value 프로퍼티를 갖는다
- 프로미스가 rejected 상태인 경우, 비동기 처리 상태를 나타내는 status 프로퍼티와 에러를 나타내는 reason 프로퍼티를 갖는다

45.8 Fetch

fetch 함수는 XMLHttpRequest 객체와 마찬가지로 클라이언트 사이드 Web API이다. fetch 함수는 XMLHttpRequest 객체보다 (1) 사용법이 간단하고, (2) 프로미스를 지원하기 때문에, (3) 비동기 처리를 위한 콜백 패턴의 단점에서 사용이 자유롭다.

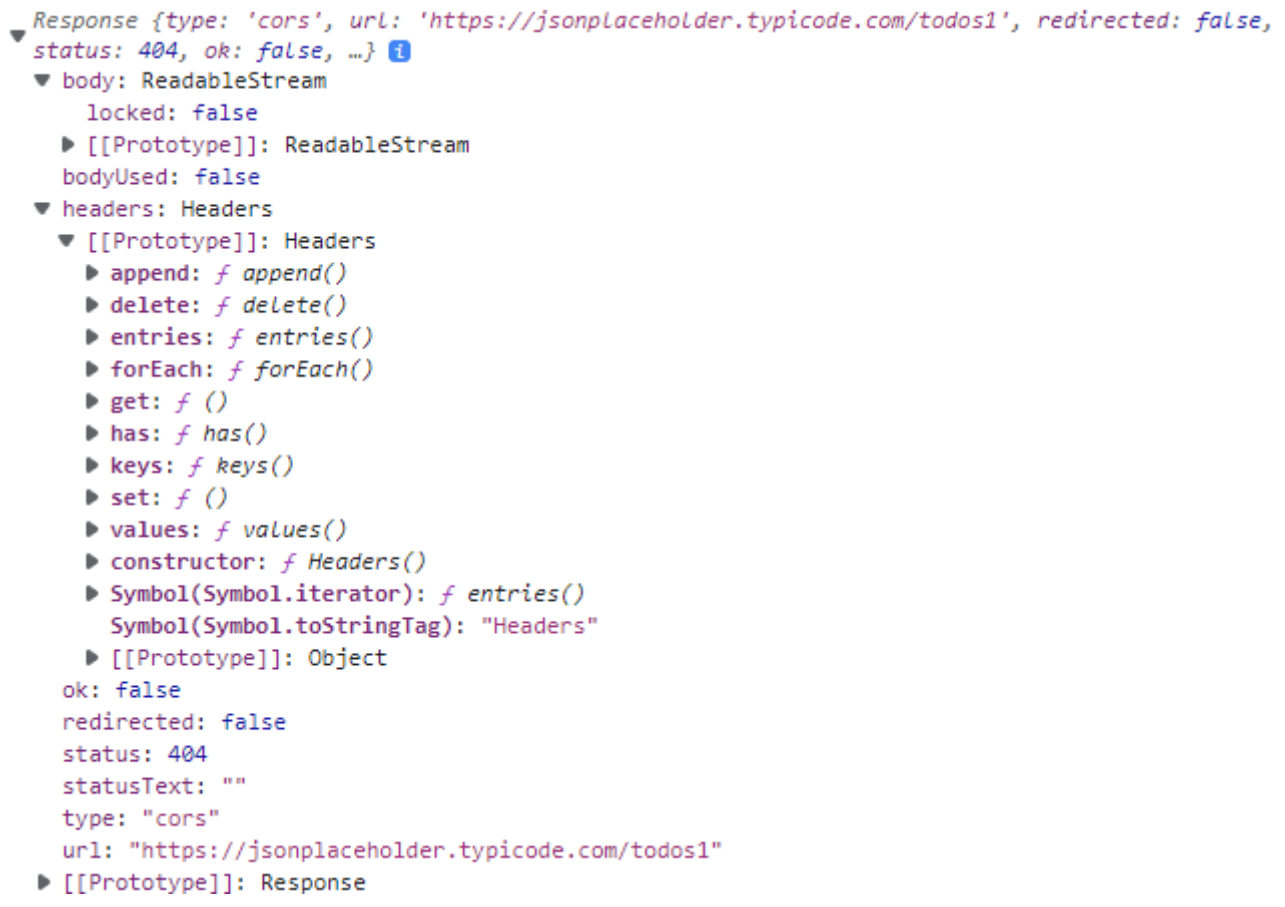
fetch 함수는 최근에 추가된 Web API로서, 인터넷 익스플로러를 제외한 대부분의 모던 브라우저에서 제공한다.

```
const promise = fetch(url, [, options]);
```

fetch 함수는 HTTP 응답을 나타내는 Response 객체를 래핑한 Promise 객체를 반환한다.

```
fetch('https://jsonplaceholder.typicode.com/todos1')
.then(response => console.log(response));
```

Response 객체는 HTTP 응답을 나타내는 다양한 프로퍼티를 제공한다.



Response.prototype에 있는 Response 객체에 포함되어 있는 HTTP 응답 몸체를 위한 다양한 메서드를 제공한다.

예를 들어, fetch 함수가 반환한 프로미스가 래핑하고 있는 MIME 타입이 application/json인 HTTP 응답 몸체를 취득하려면, Response.prototype.json 메서드를 사용한다. Response.prototype.json 메서드는 Response 객체에서 HTTP 응답 몸체를 취득하여 역직렬화 한다.


```
fetch('https://jsonplaceholder.typicode.com/todos/1')
.then(response => response.json())
.then(json => console.log(json)) // { userId: 1, id: 1, title: 'delectus aut autem", completed: false }
```

fetch 함수를 통해 HTTP 요청을 전송해보자. 첫 번째 인수로는 URL을 두 번째 인수로는 HTTP 요청 메서드, HTTP 요청 헤더, 페이로드 등을 설정한 객체를 전달한다.

```
const request = {
  get(url) {
    return fetch(url);
  },
  post(url, payload) {
    return fetch(url, {
      method: "POST",
      headers: { "content-Type": "application/json" },
      body: JSON.stringify(payload),
    });
  },
  patch(url, payload) {
    return fetch(url, {
      method: "PATCH",
      headers: { "Content-Type": "application/json" },
      body: JSON.stringify(payload),
    });
  },
  delete(url) {
    return fetch(url, { method: "DELETE" });
  },
};
```

1. GET 요청

```
request
  .get("https://jsonplaceholder.typicode.com/todos/1")
  .then((response) => response.json())
  .then((todos) => console.log(todos))
  .catch(err => console.log(err));
  // { userId: 1, id: 1, title: ~ }
```

2. POST 요청

```
request
  .post("https://jsonplaceholder.typicode.com/todos", {
    userId: 1,
    title: "JavaScript",
    completed: false,
  })
  .then((response) => response.json())
  .then((todos) => console.log(todos))
  .catch((err) => console.error(err));
// { userId: 1, title: Javascript, completed: false, id: 201 }
```

3. PATCH 요청

```
request.patch('https://jsonplaceholder.typicode.todos/1', {
  completed: true
}).then(resp
```

4. DELETE 요청

```
request
  .delete("https://jsonplaceholder.typicode.com/todos")
  .then((response) => response.json())
  .then((todos) => console.log(todos))
  .catch((err) => console.error(err));
```