async / await

- ES8에 도입된 문법으로 Promise로직을 더 쉽고 간결하게 사용할 수 있게 해줍니다.
- async/ await가 Promise를 대체하기 위한 기능은 아니며 코드 작성 부분을 유지보수 하기 편하게 보이는 문법만 다르게 해줄 뿐입니다.

기본 사용법

function 키워드 앞에 async만 붙여주면 되고, 비동기로 처리되는 부분 앞에 await만 붙여주면 됩니다.

```
// 프로미스 객체 반환 함수
function delay(ms) {
  return new Promise(resolve => {
    setTimeout(() => {
      console.log(`${ms} 밀리초가 지났습니다.`);
      resolve()
    }, ms);
  });
}
delay(1000);
```

```
// 기존 Promise.then() 형식
function main() {
  delay(1000)
      .then(() => {
        return delay(2000);
      })
      .then(() => {
        return Promise.resolve('끝');
      })
      .then(result => {
        console.log(result);
      });
}
```

```
// 메인 함수 호출
main();
```

Promise는 then 메서드를 연속적으로 사용하여 비동기 처리 한다.

```
// async/await 방식
async function main() {
  await delay(1000);
  await delay(2000);
  const result = await Promise.resolve('끝');
  console.log(result);
}

// 메인 함수 호출
main();
```

async/ await는 비동기적 접근방식을 동기적으로 작성할 수 있게 해주어 코드가 간결해지며 가독성을 높여 유지 보수를 용이하게 해준다.

async 키워드

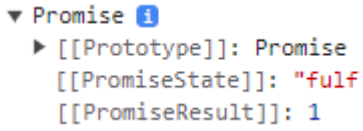
```
// 함수 선언식
async function func1() {
    const res = await fetch(url); // 요청을 기다림
    const data = await res.json(); // 응답을 JSON으로 파싱
}
func1();

// 함수 표현식
const func2 = async () => {
    const res = await fetch(url); // 요청을 기다림
    const data = await res.json(); // 응답을 JSON으로 파싱
}
func2();
```

async function에서 어떤 값을 리턴하든 무조건 프로미스 객체로 감싸져 반환 된다

```
async function func1() {
  return 1;
}

const data = func1();
console.log(data); // 프로미스 객체가 반환된다
```



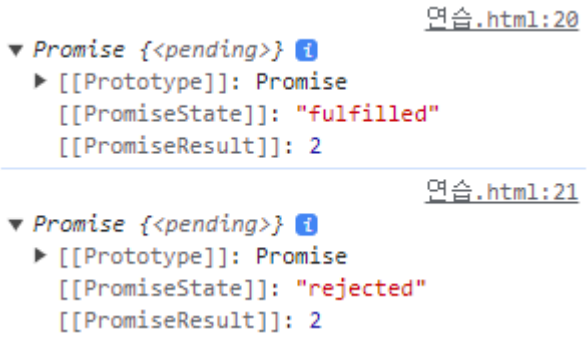
Promise 상태를 반환하기

프로미스 정적 메서드를 통해 프로미스 상태를 다르게 지정하여 반환이 가능하다.

```
async function resolveP() {
  return Promise.resolve(2);
}

async function rejectP() {
  return Promise.reject(2);
}

console.log(resolveP())
console.log(rejectP())
```



async 함수 내부에서 예외 throw 를 해도 실패 상태의 프로미스 객체가 반환되게 된다.

```
async function errorFunc() {
    throw new Error("프로미스 reject 발생시킴");
}
console.log(errorFunc())
```

async 함수의 리턴값은 프로미스 객체이기 때문에 async 함수 자체에 then 핸들러를 붙일수도 있다.

```
async function func1() {
  return 1;
}

func1()
    .then(data => console.log(data));
```

await 키워드

await 키워드는 promise.then() 보다 좀 더 세련되게 비동기 처리의 결과 값을 얻을 수 있도록 해주는 문법이라고 보면 된다

await는 Promise 처리가 끝날 때까지 기다림

비동기 처리가 완료될 때 까지 코드 실행을 일시 중지하고 wait 한다는 뜻이다.

```
async function getData() {
  const response = await fetch('https://jsonplaceholder.typic
  const data = await response.json();
  console.log(data);
```

```
}
getData()
```

1. async 함수 내에서 fetch() 비동기 함수를 호출하고, await로 fetch() 함수가 완료될 때까지 기다리게 된다.
2. fetch() 함수가 완료되면, response.json() 함수를 호출하여 await로 처리하고 데이터를 가져오는 동안 코드 실행이 중지된다.
3. 데이터가 성공적으로 가져와지면 최종 결과 값을 반환한다.

에러 처리

- async/ await의 장점은 비동기 코드를 마치 동기 코드처럼 읽히게 해준다는 것이다.
- 따라서 일반적으로 에러를 처리하기 위해 사용하는 try/ catch문을 사용하면 된다.

```
// async/await 방식
async function func() {

    try {
        const res = await fetch(url); // 요청을 기다림
        const data = await res.json(); // 응답을 JSON으로 파싱
        // data 처리
        console.log(data);
    } catch (err) {
        // 에러 처리
        console.error(err);
    }

}
func();
```

적절하지 않은 async/ await 사용

```
//1초 후에 apple이라는 문자열을 반환하는 Promise객체
function getApple(){
  return new Promise( (resolve, reject) => {
    setTimeout(() => resolve("apple"), 1000);
  })
}
//1초 후에 banana라는 문자열을 반환하는 Promise객체
function getBanana(){
  return new Promise( (resolve, reject) => {
    setTimeout(() => resolve("banana"), 1000);
  })
}

async function getFruites(){
  let a = await getApple();
  let b = await getBanana();
  console.log(`${a} and ${b}`);
}

getFruites();
```

1. 위 코드를 비동기 처리 시 각각 문자열을 반환하는 getApple()과 getBanana() 비동기 함수는 서로 연관이 없다.(둘 중 아무거나 먼저 가져와도 된다)
2. 위 코드는 await 키워드를 두 번 붙임으로써 두 과일을 가져오는데 총 2초가 걸리게 된다.

console.time()을 통해 코드의 실행시간을 확인할 수 있다

```
async function getFruites(){
  console.time();
  let a = await getApple(); // 1초 소요
  let b = await getBanana(); // 1초 소요
  console.log(`${a} and ${b}`);
  console.timeEnd();
}

getFruites();
```

3. getApple() 와 getBanana() 비동기 로직이 만일 순서를 지켜야하는 로직이라면 위와 같이 구성하여야 하는 것이 옳다.
4. 위 코드는 서로 연관 없기 때문에 반드시 순차적으로 실행 시킬 필요가 없다

적절한 async/await 사용

비동기 처리 요청과 값을 await 하는 로직을 분리시키면 된다.

```
async function getFruites(){

  let getApplePromise = getApple(); // async함수를 미리 논블록킹으
  let getBananaPromise = getBanana(); // async함수를 미리 논블록킹

  // 이렇게 하면 각각 백단에서 독립적으로 거의 동시에
    // 실행되게 된다.
  console.log(getApplePromise)
  console.log(getBananaPromise)

  let a = await getApplePromise; // 위에서 받은 프로미스객체 결과 반
  let b = await getBananaPromise; // 위에서 받은 프로미스객체 결과
  // 본래라면 1초+1초 를 기다려야 하는데, 위에서 1초기다리는 함수를
  // 바로 연속으로 비동기로 불려왔기 때문에, 대충 1.01초만 기다리면 처리된
  console.log(`${a} and ${b}`);
})
```

Promise.all 메소드 사용

```
async function getFruites(){
  console.time();

  // 구조 분해로 각 프로미스 리턴값들을 변수에 담는다.
  let [ a, b ] = await Promise.all([getApple(), getBanana()])
  console.log(`${a} and ${b}`);

  console.timeEnd();
}
```

```
getFruites();
```

top-level await

Top-level await 란 async 함수나 모듈 외부에서도 await 키워드를 사용할 수 있게 해주는 편의 기능이다(ES2022에 추가)

```
// Top Level에선 async funciton 정의없이 곧바로
// await 키워드 사용이 가능하다
const res = await fetch(url); // 요청을 기다림
const data = await res.json(); // 응답을 JSON으로 파싱
console.log(data);
```