**3장. 자바스크립트에서 비동기 처리 다루기**

\* 해당 작업이 진행되는 동안 다른 작업도 하고 싶다면 함수를 비동기 형태로 전환해주어야 함. 그렇게 하기 위해서는 **setTimeout**이라는 함수를 사용해야 함

function work() {

setTimeout(() => {

const start = Date.now();

for (let i = 0; i < 1000000000; i++) {}

const end = Date.now();

console.log(end - start + 'ms');

}, 0);

}

console.log('작업 시작!'); //1.

work(); //3. //for 루프가 돌아가는 동안 다음 작업도 실행됨

console.log('다음 작업'); //2.

**\* setTimeout:** 우리가 정한 작업이 백그라운드에서 수행되기 때문에 기존의 코드 흐름을 막지 않고 동시에 다른 작업들을 진행할 수 있음

\* 만약 work 함수가 끝난 다음에 어떤 작업을 처리하고 싶다면 **콜백 함수를 파라미터로 전달**해주면 됨

**\* 콜백 함수:** 함수 타입의 값을 파라미터로 넘겨줘서, 파라미터로 받은 함수를 특정 작업이 끝나고 호출을 해주는 것

function work(callback) {

setTimeout(() => {

const start = Date.now();

for (let i = 0; i < 1000000000; i++) {}

const end = Date.now();

console.log(end - start + 'ms'); //3.

callback();

}, 0);

}

console.log('작업 시작!'); //1.

work(() => {

console.log('작업이 끝났어요!') //4.

});

console.log('다음 작업'); //2.

**\* 다음과 같은 작업들은 주로 비동기적으로 처리하게 됨**

**- Ajax Web API 요청:** 만약 서버쪽에서 데이터를 받와아야 할 때는, 요청을 하고 서버에서 응답을 할 때 까지 대기를 해야 되기 때문에 작업을 비동기적으로 처리합니다.

**- 파일 읽기:** 주로 서버 쪽에서 파일을 읽어야 하는 상황에는 비동기적으로 처리합니다.

**- 암호화/복호화:** 암호화/복호화를 할 때에도 바로 처리가 되지 않고, 시간이 어느정도 걸리는 경우가 있기 때문에 비동기적으로 처리합니다.

**- 작업 예약:** 단순히 어떤 작업을 몇초 후에 스케쥴링 해야 하는 상황에는, setTimeout 을 사용하여 비동기적으로 처리합니다.

\* 그 외 비동기 작업: callback 함수 외에도 Promise, 그리고 async/await라는 문법을 사용하여 처리할 수 있음

**<<01. Promise>>**

**\* Promise:** 비동기 작업을 더 편하게 처리할 수 있도록 ES6에 도입된 기능

- 성공할 수도 있고, 실패할 수도 있음.

**\* Promise 만들기:**

- 성공할 때에는 resolve 호출

- 실패하면 reject를 호출

- 성공

const myPromise = new Promise((resolve, reject) => {

setTimeout(() => {

resolve(1);

}, 1000);

});

myPromise.then(n => {

console.log(n);

});

- 실패: reject를 사용하고 .catch를 통하여 실패했을시 수행할 작업을 설정할 수 있음

const myPromise = new Promise((resolve, reject) => {

setTimeout(() => {

reject(new Error());

}, 1000);

});

myPromise

.then(n => {

console.log(n);

})

.catch(error => {

console.log(error);

});

function increaseAndPrint(n) {

return new Promise((resolve, reject) => {

setTimeout(() => {

const value = n + 1;

if (value === 5) {

const error = new Error();

error.name = 'ValueIsFiveError';

reject(error);

return;

}

console.log(value); //1. //1

resolve(value);

}, 1000);

});

}

increaseAndPrint(0).then((n) => {

console.log('result: ', n); //2. //result: 1

})

function increaseAndPrint(n) {

return new Promise((resolve, reject) => {

setTimeout(() => {

const value = n + 1;

if (value === 5) {

const error = new Error();

error.name = 'ValueIsFiveError';

reject(error);

return;

}

console.log(value);

resolve(value);

}, 1000);

});

}

increaseAndPrint(0)

.then(increaseAndPrint)

.then(increaseAndPrint)

.then(increaseAndPrint)

.then(increaseAndPrint)

.then(increaseAndPrint)

.catch(e => {

console.error(e);

});

- 에러를 잡을 때 몇번째에서 발생했는지 알아내기도 어렵고 특정 조건에 따라 분기를 나누는 작업도 어렵고, 특정 값을 공유해가면서 작업을 처리하기도 까다롭습니다.

- 다음 섹션에서 배울 async/await 을 사용하면, 이러한 문제점을 깔끔하게 해결 할 수 있습니다.

**<<02.async/await>>**

**\* async/await:** ES8에 해당하는 문법으로서, Promise를 더욱 쉽게 사용할 수 있게 함

- 함수를 선언할 때 함수의 앞부분에 async 키워드를 붙이고 promise의 앞 부분에 await를 넣어주면 해당 프로미스가 끝날 때까지 기다렸다가 다음 작업을 수행할 수 있음

function sleep(ms) {

return new Promise(resolve => setTimeout(resolve, ms));

}

async function process() { //async

console.log('안녕하세요!');

await sleep(1000); // 1초쉬고 //await

console.log('반갑습니다!');

}

process();

function sleep(ms) {

return new Promise(resolve => setTimeout(resolve, ms));

}

async function process() {

console.log('안녕하세요!');

await sleep(1000); // 1초쉬고

console.log('반갑습니다!');

}

process().then(() => {

console.log('작업이 끝났어요!');

});

- async함수에서 에러를 발생시킬 때 throw를 사용하고, 에러를 잡아낼 때에는 try/catch문을 사용함

function sleep(ms) {

return new Promise(resolve => setTimeout(resolve, ms));

}

async function makeError() {

await sleep(1000);

const error = new Error();

throw error;

}

async function process() {

try {

await makeError();

} catch (e) {

console.error(e); //Error

}

}

process();

- getDog 1초, getRabbit 0.5초, getTurtle 3초 => process 함수가 실행되는 총 시간 4.5초

- getDog -> getRabbit -> getTurtle 순서대로 실행이 되고 있는데요, 하나가 끝나야 다음 작업이 시작함

function sleep(ms) {

return new Promise(resolve => setTimeout(resolve, ms));

}

const getDog = async () => {

await sleep(1000);

return '멍멍이';

};

const getRabbit = async () => {

await sleep(500);

return '토끼';

};

const getTurtle = async () => {

await sleep(3000);

return '거북이';

};

async function process() {

const dog = await getDog();

console.log(dog);

const rabbit = await getRabbit();

console.log(rabbit);

const turtle = await getTurtle();

console.log(turtle);

}

process();

- 동시에 작업을 시작하고 싶다면 Promise.all 사용

function sleep(ms) {

return new Promise(resolve => setTimeout(resolve, ms));

}

const getDog = async () => {

await sleep(1000);

return '멍멍이';

};

const getRabbit = async () => {

await sleep(500);

return '토끼';

};

const getTurtle = async () => {

await sleep(3000);

return '거북이';

};

async function process() {

const results = await Promise.all([getDog(), getRabbit(), getTurtle()]);

console.log(results); //[“멍멍이”, “토끼”, “거북이”]

}

process();

- 배열 비구조화 할당 문법을 사용한다면 각 결과값을 따로 추출해서 조회할 수 있음

function sleep(ms) {

return new Promise(resolve => setTimeout(resolve, ms));

}

const getDog = async () => {

await sleep(1000);

return '멍멍이';

};

const getRabbit = async () => {

await sleep(500);

return '토끼';

};

const getTurtle = async () => {

await sleep(3000);

return '거북이';

};

async function process() {

const [dog, rabbit, turtle] = await Promise.all([

getDog(),

getRabbit(),

getTurtle()

]);

console.log(dog);

console.log(rabbit);

console.log(turtle);

}

process();

\* Promise.all를 사용할 때, 등록한 프로미스 중 하나라도 실패하면 모든 게 실패한 것으로 간주됨

**\* Promise.race:** Promise.all과 달리, 여러 개의 프로미스를 등록해서 실행했ㅇ르 때 가장 빨리 끝난 것 하나만의 결과값을 가져옴

function sleep(ms) {

return new Promise(resolve => setTimeout(resolve, ms));

}

const getDog = async () => {

await sleep(1000);

return '멍멍이';

};

const getRabbit = async () => {

await sleep(500);

return '토끼';

};

const getTurtle = async () => {

await sleep(3000);

return '거북이';

};

async function process() {

const first = await Promise.race([

getDog(),

getRabbit(),

getTurtle()

]);

console.log(first);

}

process();

-Promise.race 의 경우엔 가장 다른 Promise 가 먼저 성공하기 전에 가장 먼저 끝난 Promise 가 실패하면 이를 실패로 간주함. 따라서, 현재 위의 코드에서 getRabbit 에서 에러를 발생시킨다면 에러를 잡아낼 수 있지만, getTurtle 이나 getDog 에서 발생한 에러는 무시됨