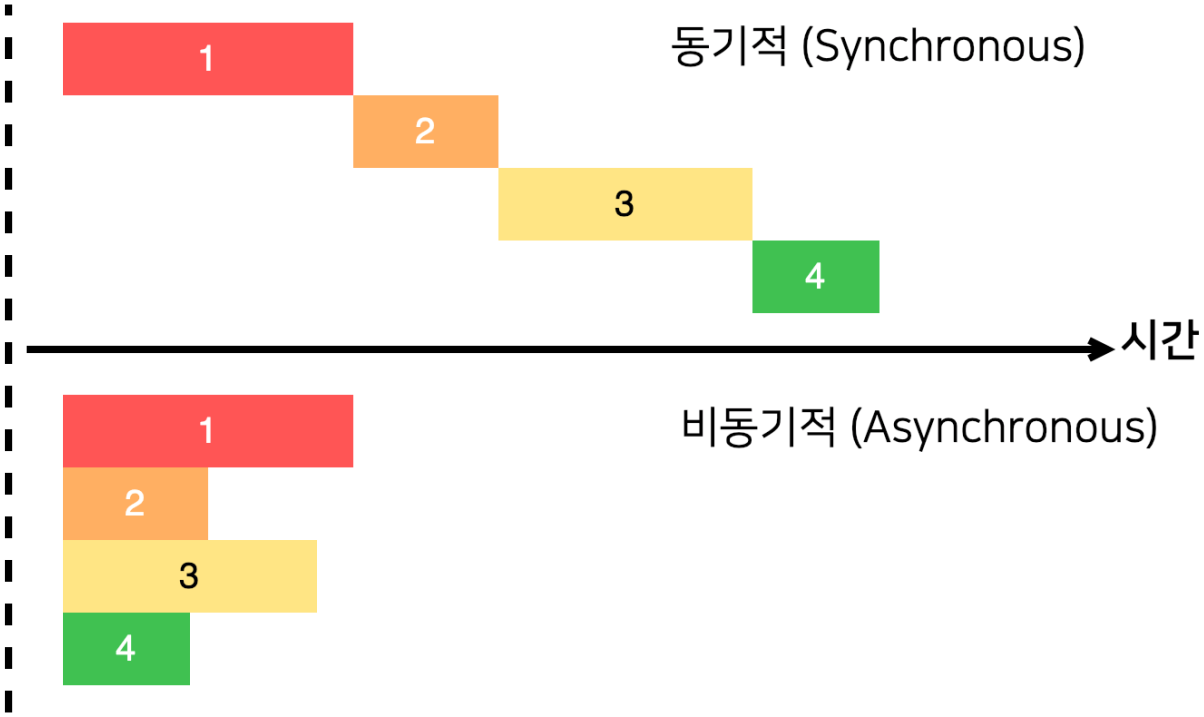
Chapter 3 자바스크립트에서 비동기 처리 다루기

태그

참고: https://ko.javascript.info/async

비동기 처리?



- 동기적 처리 → 작업이 끝날 때까지 다른 작업을 할 수 없다
- 비동기적 처리 → 흐름이 멈추지 않고 동시에 여러 작업을 처리할 수 있다.

```
function work(callback) {
  setTimeout(() => {
    const start = Date.now();
    for (let i = 0; i < 1000000000; i++) {}
    const end = Date.now();
    console.log(end - start + 'ms');
    callback();
  }, 0);
}

console.log('작업 시작!');
work(() => {
  console.log('작업이 끝났어요!')
});
console.log('다음 작업');
```

- `setTimeout` 함수는 첫번째 파라미터에 넣는 함수를 두번째 파라미터에 넣은 시간이 흐른 후 호출 → 우리가 정한 작업이 백그라운드에서 수행되기 때문에 기존의 코드 흐름을 막지 않고 동시에 다른 작업들을 진행 할 수 있다.
- `callback` 함수는 함수 타입의 값을 파라미터로 넘겨줘서, 파라미터로 받은 함수를 특정 작업이 끝나고 호출 → work 함수가 끝난 다음에 어떤 작업을 처리하고 싶다고 지정하기 위해 사용했다.
- 비동기적으로 처리하는 여러 tasks

- **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.

Promise

- 콜백함수를 사용하면 비동기적으로 처리해야 하는 일이 많아질수록, 코드의 깊이가 계속 깊어진다. (Callback Hell, 콜백지옥)
- 프로미스는 비동기 작업을 조금 더 편하게 처리 할 수 있도록 도입된 기능

```
let promise = new Promise(function(resolve, reject) {
  // executor (실행자, 실행 함수)
});
```

- executor는 `new Promise`가 만들어질 때 자동으로 실행되는데, 결과를 최종적으로 만들어내는 제작 코드를 포함
- executor는 자동으로 실행되는데 여기서 원하는 일이 처리됩니다. 처리가 끝나면 executor는 처리 성공 여부에 따라 `resolve`나 `reject`를 호출
- executor의 인수 `resolve`와 `reject`는 자바스크립트에서 자체 제공하는 콜백 (개발할 때 신경쓸 필요 x)
  - `resolve(value)` — 일이 성공적으로 끝난 경우 그 결과를 나타내는 `value`와 함께 호출
  - `reject(error)` — 에러 발생 시 에러 객체를 나타내는 `error`와 함께 호출
- `new Promise` 생성자가 반환하는 `promise` 객체는 다음과 같은 내부 프로퍼티를 갖는다.
  - `state` — 처음엔 `"pending"`(보류)이었다 `resolve`가 호출되면 `"fulfilled"`, `reject`가 호출되면 `"rejected"`로 변합니다.
  - `result` — 처음엔 `undefined`이었다 `resolve(value)`가 호출되면 `value`로, `reject(error)`가 호출되면 `error`로 변합니다.
  - 프라미스 객체의 `state`, `result` 프로퍼티는 내부 프로퍼티이므로 개발자가 직접 접근할 수 없습니다.

예시 코드

```
let promise = new Promise(function(resolve, reject) {
  // 프라미스가 만들어지면 executor 함수는 자동으로 실행됩니다.

  // 1초 뒤에 일이 성공적으로 끝났다는 신호가 전달되면서 result는 'done'이 됩니다.
  setTimeout(() => resolve("done"), 1000);
});
```

1. executor는 `new Promise`에 의해 자동으로 그리고 즉각적으로 호출된다.
2. executor는 인자로 `resolve`와 `reject` 함수를 받습니다. 이 함수들은 자바스크립트 엔진이 미리 정의한 함수이므로 개발자가 따로 만들 필요가 없습니다. 다만, `resolve`나 `reject` 중 하나는 반드시 호출해야 한다.
3. executor '처리'가 시작 된 지 1초 후, `resolve("done")`이 호출되고 결과가 만들어집니다. 이때 `promise` 객체의 상태는 다음과 같이 변한다.

```
let promise = new Promise(function(resolve, reject) {
  // 1초 뒤에 에러와 함께 실행이 종료되었다는 신호를 보냅니다.
  setTimeout(() => reject(new Error("에러 발생!")), 1000);
});
```

1. 1초 후 `reject(...)`가 호출되면 `promise`의 상태가 `"rejected"`로 변한다.

프라미스는 성공 또는 실패만 한다.

- executor에 의해 처리가 끝난 일은 결과 혹은 에러만 가질 수 있다.
- `resolve`나 `reject`는 인수를 하나만 받고(혹은 아무것도 받지 않음) 그 이외의 인수는 무시한다는 특성도 있습니다.
- executor는 대개 무언가를 비동기적으로 수행하고, 약간의 시간이 지난 후에 `resolve`, `reject`를 호출하는데, 꼭 이렇게 할 필요는 없습니다. 아래와 같이 `resolve`나 `reject`를 즉시 호출할 수도 있다. (즉시 이행 상태)

async, await

```
async function f() {
// return Promise.resolve(1); 명시적으로 Promise를 반환하는 것도 가능하다
  return 1;
}

f().then(alert); // 1
```

- function 앞에 `async`를 붙이면 해당 함수는 항상 프라미스를 반환
  - `async`가 붙은 함수는 반드시 프라미스를 반환하고, 프라미스가 아닌 것은 프라미스로 감싸 반환

```
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);
  }
}

process();
```

- `async` 함수에서 에러를 발생 시킬때에는 `throw` 를 사용하고, 에러를 잡아낼 때에는 try/catch 문을 사용

```
async function f() {

  let promise = new Promise((resolve, reject) => {
    setTimeout(() => resolve("완료!"), 1000)
  });

  let result = await promise; // 프라미스가 이행될 때까지 기다림 (*)

  alert(result); // "완료!"
}

f();
```

- await는 async 함수 안에서만 동작
  - 일반 함수엔 `await`을 사용할 수 없다!
- 자바스크립트는 `await` 키워드를 만나면 프라미스가 처리(settled)될 때까지 기다린다. 그리고 그 이후에 결과가 반환
- `await`는 `promise.then`보다 좀 더 세련되게 프라미스의 `result` 값을 얻을 수 있도록 해준다.

```
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);
}

process();
```

- 동시에 작업을 시작하고 싶다면, 다음과 같이 `Promise.all` 을 사용
- `Promise.all`를 사용 할 때에는, 등록한 프로미스 중 하나라도 실패하면, 모든게 실패 한 것으로 간주

```
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 간주