async/await (8월 8일)

배정	
상태	8월
속성	@2021년 8월 8일 오후 3:56
언어	

async/await는 프로미스를 기반으로 동작한다. async/await를 사용하면 프로미스의 then/catch/finally 후속 처리 메서드에 콜백함수를 전달해서 비동기 처리 결과를 후속처리할 필요없이 동기처리처럼 프로미스를 사용 할 수있다. 다시말해, 프로미스 후속 처리 메서드 없이 마치 동기 처리처럼 프로미스가 처리 결과를 반환 하도록 구현할 수 있다.

async 함수

await 키워드는 반드시 async 함수 내부에서 사용해야한다. async 함수가 명시적으로 프로미스를 반환하지않더라도 async 함수는 암묵적으로 반환값을 resolve하는 프로미스를 반환한다.

```
// async 함수 선언문
async function foo(n) { return n; }
foo(1).then(v => console.log(v)); // 1

// async 함수 표현식
const bar = async function (n) { return n; };
bar(2).then(v => console.log(v)); // 2

// async 화살표 함수
const baz = async n => n;
baz(3).then(v => console.log(v)); // 3

// async 메서드
const obj = {
  async foo(n) { return n; }
};
obj.foo(4).then(v => console.log(v)); // 4

// async 클래스 메서드
class MyClass {
  async bar(n) { return n; }
}
const myClass = new MyClass();
myClass.bar(5).then(v => console.log(v)); // 5
```

await 키워드

await 키워드는 프로미스가 settled상태(비동기 처리가 수행된 상태) 가 될때까지 대기하다가 settled 상태가 되면 프로미스가 resolve한 처리결과를 반환한다. 처리결과는 res변수에 할당된다. await 키워드는 반드시 프로미스 앞에서 사용해야한다.

```
// awiat 키워드는 다음 실행을 일시 중지시켰다가 프로미스가 settled 상태가 되면 다시 재개한다.

async function foo() {
  const a = await new Promise(resolve => setTimeout(() => resolve(1), 3000));
  const b = await new Promise(resolve => setTimeout(() => resolve(2), 2000));
  const c = await new Promise(resolve => setTimeout(() => resolve(3), 1000));

  console.log([a, b, c]); // [1, 2, 3]
}

foo(); // 약 6초 소요된다.
```

3개 모두 다 한다고 한다면 차라리 promise.all을 이용하여 병렬화를 한다음 비동기적 실행을 하는것이 나을 수도있다. 일일히 await을 사용 한다면 시간이 연장이 된다.

이처럼 모든 프로미스에 await키워드를 사용하는 것은 주의해야한다.

```
async function foo() {
  const res = await Promise.all([
    new Promise(resolve => setTimeout(() => resolve(1), 3000)),
    new Promise(resolve => setTimeout(() => resolve(2), 2000)),
    new Promise(resolve => setTimeout(() => resolve(3), 1000))
  ]);

  console.log(res); // [1, 2, 3]
}

foo(); // 약 3초 소요된다.
```

bar 함수는 앞선 비동기 처리의 결과를 가지고 다음 비동기 처리를 수행해야한다. 따라서 비동기 처리 순서가 보장되어야하므로 모든 프로미스에 awiat 키워드를 써서 순차저으로 처리할 수밖에없다.

```
async function bar(n) {
  const a = await new Promise(resolve => setTimeout(() => resolve(n), 3000));
```

```
  // 두 번째 비동기 처리를 수행하려면 첫 번째 비동기 처리 결과가 필요하다.
  const b = await new Promise(resolve => setTimeout(() => resolve(a + 1), 2000));
  // 세 번째 비동기 처리를 수행하려면 두 번째 비동기 처리 결과가 필요하다.
  const c = await new Promise(resolve => setTimeout(() => resolve(b + 1), 1000));

  console.log([a, b, c]); // [1, 2, 3]
}

bar(1); // 약 6초 소요된다.
```

에러처리

비동기처리를 위한 콜백 패턴의 단점중 가장 심각한것은 에러처리가 곤란하다는 것이다. 에러는 호출자 방향으로 전파된다. async/await 에서 에러 처리는 try...catch 문을 사용 할 수 있다. 콜백 함수를 인수로 전달받은 비동기와는 달리 프로미스를 반환하는 비동기 함수는 명시적으로 호출할 수 있기 때문에 호출자가 명확하다.

async 함수 내에서 catch 문을 사용해서 에러처리를 하지 않으면 async 함수는 발생한 에러를 reject하는 프로미스를 반환한다.

```
// async 함수에서 try & catch를 사용한 경우
const fetch = require('node-fetch');

const foo = async () => {
  try {
    const wrongUrl = 'https://wrong.url';

    const response = await fetch(wrongUrl);
    const data = await response.json();
    console.log(data);
  } catch (err) {
    console.error(err); // TypeError: Failed to fetch
  }
};

foo();
----------------------------------------------------------------------------------------
// async 함수에서  try & catch를 사용 하지않는 경우
const fetch = require('node-fetch');

const foo = async () => {
  const wrongUrl = 'https://wrong.url';

  const response = await fetch(wrongUrl);
  const data = await response.json();
  return data;
};
```

```
foo()
  .then(console.log)
  .catch(console.error); // TypeError: Failed to fetch

// then과 catch를 밖에다 빼둠.
```