day55-async-await

태그	
날짜	@2022년 12월 16일

async와 await은 자바스크립트의 비동기 처리 패턴 중 가장 최근에 나온 문법이다.

기존의 비동기 처리 방식인 콜백 함수와 프로미스의 단점을 보완하고 개발자가 읽기 좋은 코드를 작성할 수 있게 도와준다.

개발자에게 읽기 좋은 코드란?

```
var user = {
  id: 1,
  name: 'Josh'
};
if (user.id === 1) {
  console.log(user.name); // Josh
}
```

이 코드는 user라는 변수에 객체를 할당한 뒤 조건문으로 사용자의 아이디를 확인하고 콘솔에 사용자의 name을 찍는 간단한 코드다.

우리는 이렇게 위에서 부터 아래로 한 줄 한 줄 차근히 읽으면서 사고하는 것이 편하다.

읽기 좋은 코드와 async & await가 무슨 상관일까.

조금 전에 읽고 이해한 방식대로 코드를 구성하는 것이 async, await 문법의 목적이다.

```
var user = fetchUser('domain.com/users/1');
if (user.id === 1) {
  console.log(user.name);
}
```

`fetchUser()`
라는 메서드를 호출하면 앞에서 봤던 코드처럼 사용자 객체를 반환한다고 해보겠습니다. 그

리고 여기서 `fetchUser()`
메서드가 서버에서 사용자 정보를 가져오는 HTTP 통신 코드라고 가정한다면 위 코드는 async & await 문법이 적용된 형태라고 보셔도 됩니다.

---

async & await 적용된 코드와 그렇지 않은 코드

자 저희가 조금 전에 본 코드가 대체 어떤 의미인지 한번 알아보겠습니다. 먼저 아까 살펴봤던 `logName()` 함수 코드를 다시 보겠습니다.

```
function logName() {
  var user = fetchUser('domain.com/users/1');
  if (user.id === 1) {
    console.log(user.name);
  }
}
```

여기서 `fetchUser()`
라고 하는 코드는 서버에서 데이터를 받아오는 HTTP 통신 코드라고 가정했습니다. 일반적으로 자바스크립트의 비동기 처리 코드는 아래와 같이 콜백을 사용해야지 코드의 실행 순서를 보장받을 수 있죠.

```
function logName() {
  // 아래의 user 변수는 위의 코드와 비교하기 위해 일부러 남겨놓았습니다.
  var user = fetchUser('domain.com/users/1', function(user) {
    if (user.id === 1) {
      console.log(user.name);
    }
  });
}
```

이미 위와 같이 콜백으로 비동기 처리 코드를 작성하는 게 익숙하신 분들이라면 문제가 없겠지만, 이 사고방식에 익숙하지 않은 분들은 고개가 갸우뚱할 겁니다.

그래서 저희가 처음 프로그래밍을 배웠던 그때 그 사고로 돌아가는 것이죠. 아래와 같이 간단하게 생각하자구요.

```
// 비동기 처리를 콜백으로 안해도 된다면..
function logName() {
  var user = fetchUser('domain.com/users/1');
  if (user.id === 1) {
    console.log(user.name);
```

```
  }
}
```

서버에서 사용자 데이터를 불러와서 변수에 담고, 사용자 아이디가 1이면 사용자 이름을 출력한다.

이렇게 하려면 async await만 붙이시면 됩니다 :)

```
// async & await 적용 후
async function logName() {
  var user = await fetchUser('domain.com/users/1');
  if (user.id === 1) {
    console.log(user.name);
  }
}
```

---

async & await 기본 문법

```
async function 함수명() {
  await 비동기_처리_메서드_명();
}
```

먼저 함수의 앞에 `async` 라는 예약어를 붙입니다. 그러고 나서 함수의 내부 로직 중 HTTP 통신을 하는 비동기 처리 코드 앞에 `await`를 붙입니다. 여기서 주의하셔야 할 점은 비동기 처리 메서드가 꼭 프로미스 객체를 반환해야 `await`가 의도한 대로 동작합니다.

일반적으로 `await`의 대상이 되는 비동기 처리 코드는 Axios 등 프로미스를 반환하는 API 호출 함수입니다.

---

async & await 간단한 예제

```
function fetchItems() {
  return new Promise(function(resolve, reject) {
    var items = [1,2,3];
    resolve(items)
  });
}

async function logItems() {
  var resultItems = await fetchItems();
```

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

먼저 `fetchItems()` 함수는 프로미스 객체를 반환하는 함수입니다. 프로미스는 "자바스크립트 비동기 처리를 위한 객체"라고 배웠었죠. `fetchItems()` 함수를 실행하면 프로미스가 이행(Resolved)되며 결과 값은 `items` 배열이 됩니다.

그리고 이제 `logItems()` 함수를 보겠습니다. `logItems()` 함수를 실행하면 `fetchItems()` 함수의 결과 값인 `items` 배열이 `resultItems` 변수에 담깁니다. 따라서, 콘솔에는 `[1,2,3]`이 출력되죠.

`await`를 사용하지 않았다면 데이터를 받아온 시점에 콘솔을 출력할 수 있게 콜백 함수나 `.then()`등을 사용해야 했을 겁니다. 하지만 async await 문법 덕택에 비동기에 대한 사고를 하지 않아도 되는 것이죠.

※참고: 만약 위 코드가 왜 비동기 처리 코드인지 잘 이해가 안 가신다면 `fetchItems()`를 아래의 함수들로 바꿔서 실행해보셔도 괜찮습니다 :)

```
// HTTP 통신 동작을 모방한 코드
function fetchItems() {
  return new Promise(function(resolve, reject) {
    setTimeout(function() {
      var items = [1,2,3];
      resolve(items)
    }, 3000);
  });
}

// jQuery ajax 코드
function fetchItems() {
  return new Promise(function(resolve, reject) {
    $.ajax('domain.com/items', function(response) {
      resolve(response);
    });
  });
}
```

---

async & await 실용 예제

async & await 문법이 가장 빛을 발하는 순간은 여러 개의 비동기 처리 코드를 다룰 때입니다. 아래와 같이 각각 *사용자*와 *할 일 목록*을 받아오는 HTTP 통신 코드가 있다고 하겠습니다.

```
function fetchUser() {
  var url = 'https://jsonplaceholder.typicode.com/users/1'
  return fetch(url).then(function(response) {
    return response.json();
  });
}

function fetchTodo() {
  var url = 'https://jsonplaceholder.typicode.com/todos/1';
  return fetch(url).then(function(response) {
    return response.json();
  });
}
```

위 함수들을 실행하면 각각 사용자 정보와 할 일 정보가 담긴 프로미스 객체가 반환됩니다.

자 이제 이 두 함수를 이용하여 할 일 제목을 출력해보겠습니다. 살펴볼 예제 코드의 로직은 아래와 같습니다.

1. `fetchUser()`를 이용하여 사용자 정보 호출
2. 받아온 사용자 아이디가 `1`이면 할 일 정보 호출
3. 받아온 할 일 정보의 제목을 콘솔에 출력

그럼 코드를 보겠습니다.

```
async function logTodoTitle() {
  var user = await fetchUser();
  if (user.id === 1) {
    var todo = await fetchTodo();
    console.log(todo.title); // delectus aut autem
  }
}
```

`logTodoTitle()`를 실행하면 콘솔에 *delectus aut autem*가 출력될 것입니다. 위 비동기 처리 코드를 만약 콜백이나 프로미스로 했다면 훨씬 더 코드가 길어졌을 것이고 인덴팅 뿐만 아니라 가독성도 좋지 않았을 겁니다. 이처럼 async await 문법을 이용하면 기존의 비동기 처리 코드 방식으로 사고하지 않아도 되는 장점이 생깁니다.

※참고: 위 함수에서 사용한 `fetch()` API는 크롬과 같은 최신 브라우저에서만 동작합니다. 브라우저 지원 여부는 다음 링크로 확인해보세요. fetch API 브라우저 지원표

async & await 예외 처리

async & await에서 예외를 처리하는 방법은 바로 try catch입니다. 프로미스에서 에러 처리를 위해 `.catch()`를 사용했던 것처럼 async에서는 `catch {}` 를 사용하시면 됩니다.

조금 전 코드에 바로 `try catch` 문법을 적용해보겠습니다.

```
async function logTodoTitle() {
  try {
    var user = await fetchUser();
    if (user.id === 1) {
      var todo = await fetchTodo();
      console.log(todo.title); // delectus aut autem
    }
  } catch (error) {
    console.log(error);
  }
}
```

위의 코드를 실행하다가 발생한 네트워크 통신 오류뿐만 아니라 간단한 타입 오류 등의 일반적인 오류까지도 `catch`
로 잡아낼 수 있습니다. 발견된 에러는 `error`
 객체에 담기기 때문에 에러의 유형에 맞게 에러 코드를 처리해주시면 됩니다.