


비동기 프로그래밍



Reference

비동기 프로그래밍. 왜 하죠?

당신은 카페 바리스타입니다. 도와주는 사람도 없이 혼자서 주문을 받고 커피를 만들지만, 뭐 그러려니 합시다. 오늘도 문을 열고 손님을 맞이할 준비를 하네요. 언제나 조심스러운 성격인 당신은 한땀한땀 커피

 https://vlog.io/@two_jay/callback-%EA%B3%BC-%EB%B9%84%EB%8F%99%EA%B8%B0-%ED%94%84%EB%A1%9C%EA%B7%B8%EB%9E%98%EB%B0%8D



들어가기 전에

저희는 1인 카페의 사장님입니다. 1인 카페이기 때문에 혼자 주문도 받고 제조도 해야 합니다. 그럼에도 저희는 나름 명성이 자자한 카페인데요. 저희가 제조를 정~말 정성스럽게 하기 때문입니다! 얼마나 정성스럽게 하냐면 제조를 하는 동안 다른 일은 하나도 하지 않고 오직 제조에만 열중합니다. 아메리카노는 3분, 라떼는 4분, 허니브레드는 10분이나 걸리죠.

제조 중 주문을 받거나, 아메리카노를 만들면서 허니브레드를 만들거나 할 수는 없습니다. 그럼 저희만의 프라이드가 깎이게 되겠죠.

이제 손님이 들어왔습니다. 허니브레드 하나와 아메리카노를 주문 받고 우리는 허니브레드부터 만들기 시작합니다. 1.. 2.. 3.. 10분이 지나고 아메리카노를 만드려고 하는데 다른 손님이 들어오네요. 물론 저희는 아메리카노를 정성스럽게 만들어야 되기 때문에 두 번째 손님은 저희가 아메리카노를 다 만들 때까지 기다려야 됩니다. 1.. 2.. 3분이 지나고 이제 두 번째 주문을 받았는데... 맵소사 아메리카노 3잔, 라떼 2잔, 허니브레드 3개를 내놓으라고 하네요. 그리고 주문을 받자마자 세 번째 손님이 들어옵니다.

과연 저희 카페가 성공할 수 있을까요?

자바스크립트가 ππ...

자바스크립트는 위의 예시처럼 일합니다. 이걸 가리켜 싱글스레드 언어라고 합니다. 한 번에 1가지 작업밖에 못한다는 소리죠. 하지만 생각해봅시다. 이딴 언어를 과연 누가 쓸까요? 웹 페이지 하나 로딩하는데 엄청난 시간이 걸릴 겁니다. 아래 예시를 볼까요?

```
const btn = document.querySelector('button');
btn.addEventListener('click', () => {
  let myDate;

  // 여길 1000만 번이나 돌리고 있어야 됩니다. 어느 세월예요?
  for (let i = 0; i < 10000000; i++) {
    let date = new Date();
    myDate = date;
  }

  console.log(myDate);

  // 그럼 애는 언제쯤 실행될까요?
  let pElement = document.createElement('p');
  pElement.textContent = '난 언제쯤 실행될까..';
  document.body.appendChild(pElement);
});
```

이와 동일하게 웹페이지를 로딩하려면 헤더부터 천~천히 순차적으로 로딩이 되어겠죠. 그런데 요즘 웹페이지를 보면 분명히 자바스크립트로 코딩이 됐을텐데도 엄청난 속도로 로딩이 됩니다. 시대가 지나면서 오히려 페이지에 표시할 양은 많아지는데도 로딩 속도는 점점 빨라지죠. 유튜브에서는 동영상 로딩이 끝나기 전에 다른 동영상으로 갈 수도 있습니다. 이게 어떻게 되는걸까요?

여기에 대한 해답이 **비동기 프로그래밍**입니다. 위와 같이 한 번에 하나씩만 하는 것을 **동기 프로그래밍**, 동시에 하는 것을 **비동기 프로그래밍**이라고 하죠.

실제로 일어나는 일

비동기의 대표적인 예시인 setTimeout을 보겠습니다.

```
setTimeout(() => { console.log('setTimeout!!') }, 3000)
```

이렇게 하면 3초 뒤에 콘솔에 텍스트가 찍히게 되죠. 이걸 좀 더 깊게 파보겠습니다.

JS는 모든 작업을 순차적으로 **콜스택**이라는 곳에 집어넣습니다. 이름에서도 알 수 있듯이 Stack 구조를 가지기 때문에 FILO 구조를 가지게 되죠. 그 후 콜스택에서 들어온 작업을 순서대로 수행을 하게 됩니다. 그러다가 setTimeout을 보면 이렇게 받아들이죠.



이거 setTimeout이네? 브라우저야, 이거 타이머 API에 맡겨서 시간 지나면 알려 줘. 그럼 내가 콜백함수로 처리 해줄게

저렇게 맡겨놓고 콜스택의 다음 작업을 먼저 수행하는거죠. 그럼 다음 코드는 어떻게 실행될까요?

```
console.log('A')
setTimeout(() => { console.log('B') }, 2000)
console.log('C')
```

예상하신 대로 A → C → B 순서로 실행됩니다. 이런 식으로 비동기 프로그래밍이 구현됩니다.

Callback Hell

물론 콜백을 통해 실행하는 것도 완벽하지 않습니다. 이전 예제에서나 setTimeout이 1개니까 다행이지 이게 여러 번 중첩되어 있다고 생각하면... 끔찍합니다. 그리고 이런 것을 콜백

지옥 즉, **Callback Hell** 이라고 합니다.

```
비동기함수 함수1 (function 콜백() {  
  비동기함수 함수2 (function 콜백() {  
    비동기함수 함수3 (function 콜백() {  
      .....  
    }  
  }  
})  
}
```

어우 끔찍합니다. 물론 돌아가긴 하겠지만 저 안에서 에러가 났다면? 코드 읽기도 싫은데 에러까지 고쳐야 된다면 저는 개발자를 때려치울 겁니다. (물론 못때려치겠지만요)

이런 상황을 고치기 위해 드디어 **Promise** 라는 개념이 나왔습니다.

Promise

ES6 에서 처음으로 공개된 개념입니다. **비동기적 작업을 하는 함수의 리턴 타입으로 쓰이죠.** 그냥 아무 생각 없이 쓰던 `async / await` 와 같은 함수의 리턴 타입이 `Promise` 라는 소리입니다. 일단 기본적인 틀을 보겠습니다.

```
// 기본적으로 promise는 인터페이스와 같으므로 new 생성자를 사용합니다.  
// promise는 resolve(성공)와 reject(실패) 라는 콜백 함수 2개를 인자로 사용하는 함수를  
// 인자로 받습니다.  
let promise = new Promise((resolve, reject) => {});  
  
// 기본 예제  
let myFirstPromise = new Promise((resolve, reject) => {  
  setTimeout(() => {  
    resolve("Success!!");  
  }, 250);  
})  
  
// .then, .catch 구문도 이때 추가됐죠  
myFirstPromise.then(successMsg => console.log("Yeah! " + successMsg))  
// 0.25초 후에 Yeah! Success!! 가 출력될겁니다.
```

조금 더 자세히 봐봅시다. Promise는 함수 안의 작업이 성공적으로 동작했을 때 resolve 를 실행시키고, 실패했을 경우 reject 를 실행시킵니다.

```
// Promise 안의 함수였던..!  
(resolve, reject) => { setTimeout(...) } 가 성공했을 경우 resolve 리턴하는 것이죠  
  
// 물론 이 함수는 실패할 건덕지가 없기 때문에 reject는 안쓰이긴 합니다.
```

원래는 이렇게 명령을 실행하려면 미리 실행 성공, 실패시에 실행할 함수를 정의했어야 됐으나 Promise 이후부터는 그냥 Promise만 반환하면 되기 때문에 훨씬 간단해 졌습니다.

```
// 이랬던걸  
let function1 = function(isSuccess) {  
  function do_something_async(if_success, if_fail) {  
    // 많이 오래 걸리는 작업  
  
    if(isSuccess) {  
      if_success();  
    } else {  
      if_fail();  
    }  
  }  
}  
  
function if_success() {  
  // 성공성공  
}  
  
function if_fail() {  
  // 실패실패  
}  
  
// 이렇게 쓸 수 있어졌죠  
let function2 = function(isSuccess) {  
  return new Promise(function (resolve, reject) {  
    // 많이 오래 걸리는 작업  
  
    if (isSuccess) resolve();  
    else reject();  
  })  
}  
  
// resolve면 then으로, reject면 catch로 갈 수 있습니다.  
function2(true).then(() => if_success()).catch(e => if_fail())
```

```
// 물론 이정도 양의 코드만 보면 위가 더 쉬워보일 수 있지만
// 코드의 양이 많아진다면 점점 밑이 쉬워보이실 겁니다.

// 만약 위와 같은 콜백 함수가 여러 개 있다고 하면 아래와 같이 결과를 실행할 수도 있습니다.
our_work.then(() => {
  console.log('1')
  return callback_on_success1()
}).then(() => {
  console.log('2')
  return callback_on_success2()
}).then(() => {
  console.log('3')
  return callback_on_success3()
}).catch(e => {
  console.log(e)
  return reject()
})

// our_work 콜백 함수 실행 후 성공이면 1을 출력하고 callback_on_success1을 실행합니다.
// callback_on_success1 실행 후 성공이면 2를 출력하고 callback_on_success2를 실행합니다.
// 그러해 진행하다가 중간에 에러가 생기면 어디서든 바로 catch로 갑니다.
```

물론! Promise 도 문제는 있습니다. 여러 콜백함수를 진행하다가 중간에 리턴 값이 꼬인다면.. 일일이 반환값을 디버거로 뜯어봐야 합니다. 중간에 다른 값이 필요할 때도 Callback Hell과 같이 가독성이 떨어집니다.

```
function work_with_promise() {
  promise.then(data => {
    // promise 함수 성공했을 경우에 대해 작업을 하다가
    ...

    // 다른 promise를 가지고 오고 싶어졌어!!
    return another_promise(data).then(data1 => {
      // another_promise 함수 성공했을 경우에 대한 작업을 또 하다가
      ...

      // 또 다른 promise를...!!
      return new_promise(data1).then(.....)
    })
  })
}
```

물론 뭐... 가독성이 조금은 나아졌지만 아직 어질어질 합니다.

async / await

프로젝트를 하시다 보면 정말 많이 보이는 async / await 구문이 Promise 이후 나왔습니다. 위에서 async / await 함수의 리턴 값이 Promise 라고 했었죠.

한 번 어떻게 쓰이나 봅시다.

```
// 특정 유저를 가져와 입력한 id와 대조 후, 같으면 콘솔에 이름 띄우기
async function log_username(id) {
  // 서버에서 데이터를 받아오는 작업은 시간이 걸리기 때문에 비동기를 사용합니다.
  let user = await fetchUser(url)

  if (user.id === id) {
    console.log(user.username)
  }
}

// 에러 캐치는 try / catch 문법을 사용합니다.
async function log_username(id) {
  try {
    let user = await fetchUser(url)

    if (user.id === id) {
      console.log(user.username)
    }
  } catch(e) {
    console.log(e)
  }
}
```

확실히 훨씬 간단합니다. 일반 코드를 작성할 때와 다르게 하나도 없으니 가독성도 너무 좋습니다. 더해서 다수의 비동기 작업을 처리할 때는 훨씬 좋습니다.

```
async function get_resources() {
  // data1, 2, 3를 모두 받아 올 때까지 기다린 후
  let data1 = await fetch(url1).then(res => console.log(res))
  let data2 = await fetch(url2).then(res => console.log(res))
  let data3 = await fetch(url3).then(res => console.log(res))

  // 리턴됩니다.
  return { data1, data2, data3 }
}
```

끝!!