**동기와 비동기**

사실은 여러분은 지금까지 동기와 비동기를 이해하고 해결 하려 모든 내용들을 봐도 과언이 아닐 것입니다. 이 내용을 한번에 이해하는 것은 매우 어렵기에 지속적인 공부와 질문으로 공부를 지속 해주시기를 바라겠습니다.

**setTimeOut , setInterval**

setTimeOut함수는 첫번 째 인자로 콜백을 받고, 두번 째 인자로 숫자 타입의 밀리세컨드를 받는다. 이 함수는 실행되고 2번째 인자로 받은 밀리 세컨드가 경과되면 첫번째로 받은 콜백함수를 실행 시키는 타이머 함수이다.

*//익명함수를 전달하는 경우 애로우 펑션*

setTimeout(()=>{

console.log("1초가 경과되었습니다.")

},1000);

function sayHello(){

for(let i = 0; i < 10; i++){

console.log("hello");

}

}

*//기명 함수를 전달하는 경우*

setTimeout(sayHello,2000);

setInterval함수는 2번째로 받은 밀리 세컨드가 경과 될때 마다 첫번째 전달 인자로 받은 콜백을 반복적으로 실행하는 함수 이다.

setInterval(()=>{

console.log("이 함수는 주기적으로 반복합니다.")

},3000);

**자바스크립트의 비동기적 실행**

이제는 setTimeOut함수를 오래 걸리는 것 부터 해보자 .

setTimeout(()=>{

console.log("3초 뒤 실행");

},3000);

setTimeout(()=>{

console.log("2초 뒤 실행");

},2000);

setTimeout(()=>{

console.log("1초 뒤 실행")

},1000);

이 코드의 움직임이 어떻게 될 까 ? 코드만 본다면 3초 뒤 실행 문구가 종료 되면 2초 뒤에 2초뒤 실행 종료 후 1초 뒤 1초뒤 실행 순으로 진행 될 것 같다. 그러나 실행을 해보면 다음과 같은 결과를 볼수 있다.

*//1초 뒤 실행*

*//2초 뒤 실행*

*//3초 뒤 실행*

분명 코드 순서는 3,2,1 순 인데 어째서 반대로 실행 된 것 인가? 아니면 노드 혹은 내 컴퓨터가 이상한 것인가 ? 그러나 모든 것은 정상이다.

다음과 같은 코드도 동일하게 이상하게 움직인다.

let message = "nothing";

setTimeout(()=>{

message = "1 derived Message"

},1000);

console.log(message)

console.log는 setTimeOut뒤에 작성되어 있습니다. 그리고 setTimeOut함수의 콜백 함수는 1초뒤에 작동 하도록 되어 있습니다. 그러나 message의 결과는 기존 할당과 같은 nothing입니다. 이와 같은 결과는 왜 그럴까요 ?

**동기와 비동기.**

그것은 자바스크립트는 비동기적으로 동작 하기 때문입니다. 동기적이라는 말은 어떤 태스크(일)이 끝나면 다음 태스크(일)을 순서대로 하는 형태를 말합니다. 제가 일하는 카페의 업무 형태를 크게 보면 다음과 같습니다.

1. 손님이 주문한다.
2. 주문 내역서를 보고 커피를 제조한다.
3. 커피의 제조를 끝내면 손님에게 커피를 전달 합니다.

이 3개의 태스크로 이루어진 루틴은 너무나도 평범하게 보입니다. 그러나 이 동기적으로 이루어진 위의 루틴은 다른 손님이 와도 해당 루틴을 다 끝내기 전까지 다른 업무를 하지 못하는 치명적인 단점이 있습니다. 사실 모든 태스크에는 더욱 세부적인 업무들과 각기 다른 대기 시간이 있습니다. 예를 들어 손님이 결제할 때 카드리더기가 본사와 통신하는 4초간의 시간이 있고 고객님에게 명세서를 건네줍니다. 또 자동 커피 그라인더가 있기에 이 그라인더에 버튼을 누르면 약 15초간 자동으로 커피가 갈려 나옵니다. 그리고 커피 머신의 추출 버튼을 누르면 24초 동안 커피 머신은 자동으로 추출하여 나옵니다. 동기적으로 이 업무를 하게 되면 저는 약 44초간의 아무것도 하지 않고 시간을 허비 하며 기다릴 수 밖에 없습니다. 여기에 반하여 비동기적으로 업무를 한다는 말은 해당 업무가 끝날 때까지 기다리는 것이 아니라, 해당 업무를 실행 시켜놓고 다른 업무를 보러 가는 것을 뜻하는 겁니다. 그러나 기계가 본사와 통신을 끝내야 손님에게 영수증을 주며, 커피 머신의 추출이 끝나야 손님에게 커피를 줄 수 있습니다. 이는 어떻게 해결 할 수 있을까요 ? 그것은 바로 각 기계가 해당 업무가 끝나면 저에게 알람을 통해 해당 업무가 끝났다는 것을 알려주기만 하면 저는 그 일이 언제 끝났는지 알 수 있기에 이를 통해 해결 할 수 있습니다.

자바스크립트는 브라우저에서 작동하고, 많은 외부와 데이터를 주고 받는 통신을 처리해야 하기 때문에 비동기적으로 작동해야 합니다. 만약 자바스크립트가 동기적으로 작동한다면 로그인을 처리하거나 게시물을 받거나, 여러 사진을 다운로드 받을 때 해당 업무가 끝날 때 까지 아무것도 못하게 됩니다. 즉 User는 서버와 본인의 컴퓨터가 통신하는 동안 아무것도 못하고 기다려야 하는 것을 의미합니다. 그러나 여전히 우리는 비동기적으로 작동하는 환경에서 업무를 동기적으로 처리하도록 해야하는 문제가 남아 있습니다. 자바스크립트는 비동기코드를 동기적으로 사용 할 수 있도록 여러 방법을 제공합니다. Es6이전에는 방법이 단 한가지였고 코드를 엉망으로 만들고 어렵게 만들어 놨습니다. 그러나 es6부터 비동기적 프로그램을 만들 수 있도록 편리한 기능을 제공합니다.

**콜백을 통한 해결 방법**

콜백을 이용하여 코드의 흐름을 제어하는 방법은 꽤나 어렵고 코드를 난해하게 만든다. 이 콜백은 많은 이들을 자바스크립트에서 좌절 하게 만들었다. 콜백을 이용한다는 것은 함수형 프로그래밍을 통해 함수의 기능을 나누는 의미도 있지만 코드의 흐름에서 함수를 호출 할 위치를 정할 수 있게 한다.

아까 setTimeOut을 동기적으로 실행 해보도록 하겠다.

setTimeout(()=>{

console.log("3초 뒤 실행");

setTimeout(()=>{

console.log("2초 뒤 실행");

setTimeout(()=>{

console.log("1초 뒤 실행");

},1000)

},2000)

},3000)

이렇게 콜백을 이용하면 해당 함수의 실행 순서를 제어 할 수 있다 . 순서대로 3초 가 실행되는 경우 2초뒤 실행이라는 함수가 실행된다. 그리고 2초뒤 실행 함수가 실행 할 때 1초뒤 실행이라는 함수를 실행 했기 때문에 코드의 흐름을 제어 할 수 있다.

**콜백을 이용하여 제어하는 모습**

function loginRequest(id,pw){

console.log(one);

function authentication(one){

console.log(one);

function getUserInfo(one){

console.log(one)

function getUserInfoArticle(one){

console.log(one);

}

getUserInfoArticle();

}

getUserInfo();

}

authentication();

}

*//함수를 전역에서 사용 못하게 해야하는 경우도 많기에, 함수가 이렇게 옆으로 길어진다.*

*//이런 걸 콜백 지옥이라고 한다.*

**이벤트 기반**

이벤트 기반이란 이벤트가 발생할 때 미리 지정해둔 작업을 수행하는 방식을 의미합니다. 이벤트로는 클릭이나 네트워크요청 등이 있을 수 있습니다.

이제 자바스크립트의 비동기적 프로그래밍이 어떻게 동작하는지 한번 알아봅시다.

이벤트 기반 모델에서는 이벤트 루프라는 개념이 등장합니다. 여러 이벤트가 동시에 발생했을 때 어떤 순서로 콜백 함수를 호출할지를 이벤트 루프가 판단합니다. 노드는 자바스크립트 코드에서 맨 위부터 한 줄씩 실행합니다. 함수 호출 부분을 발견했다면 호출 한 함수를 호출 스택에 넣습니다.

function first(){

second();

console.log("first");

}

function second(){

third();

console.log("second");

}

function third(){

console.log("third");

}

first();

*//이 순서라면 호출 스택에 first,second,third <-top 순으로 쌓이게 된다.*

*//실행은 반대로 된다.*

이번에는 setTimeOut함수를 이용하여 보겠다.

function run(){

console.log("3초 후 실행")

}

console.log('시작')

setTimeout(run,3000);

console.log("끝")

*//결과는 시작, 끝, 3초 후 실행이다.*

콘솔의 결과는 예측이 쉽지만 호출 스택으로는 설명하기 힘듭니다. setTimeOut함수의 콜백의 run이 언제 호출 스택에 언제 들어가는 지 알기 어렵기 때문입니다. 이를 파악하기 위해서는 이벤트 루프,태스크 큐, 백그라운드를 알아야 합니다.

이벤트 루프 : 이벤트 발생 시 호출할 콜백 함수들을 관리하고, 호출된 콜백함수의 실행 순서를 결정하는 역할을 담당합니다. 노드가 종료될 때 까지 이벤트 처리를 위한 작업을 반복하므로 루프라고 한다.

태스크 큐 이벤트 발생 후 호출되어야 할 콜백 함수들이 기다리는 공간입니다. 콜백들이 이벤트 루프가 정한 순서대로 줄을 서있기에 콜백 큐라고도 불립니다.

백그라운드 : 타이머나 io작업 콜백 또는 이벤트 리스너들이 대기 하는 곳이다.

function run(){

console.log("3초 후 실행")

}

console.log('시작')

setTimeout(run,3000);

console.log("끝")

*//결과는 시작, 끝, 3초 후 실행이다.*

*//setTimeOut이 실행되면 호출 스택에 들어 갑니다.*

*//이후 setTimeOut이 실행되면 run콜백을 백그라운드로 보내고 호출 스택에서 빠집니다.*

*//백그라운드에서는 3초를 센 후 run함수를 태스크 큐로 보냅니다.*

*//이제 이벤트 루프는 호출 스택이 비어있으면 태스크 큐에서 함수를 하나씩 가져와 호출 스택에 넣고 실행 합니다.*

*//호출 스택으로 올려진 run은 실행되고 실행 완료 후 호출 스택에서 비워집니다.*

*//이떄 만약 호출 스택에 함수들이 너무 많이 차 있으면 3초가 지난 후에도 run함수가 실행되지 않을 수 도 있습니다.*

*//이벤트 루프는 호출 스택이 비어 있을때만 태스크 큐에 있는 run함수를 호출 스택으로 가져오니까요.*

**프라미스**

기존의 콜백은 비동기 처리를 온전히 해낼 수 있지만 이런 콜백이 사용되는 경우가 많아 질 수 록 단점이 드러났다. 그 단점은 비동기 처리를 순차적으로 실행할 필요가 있는 경우에 비동기 처리를 중첩시켜 표현 하므로 에러, 예외처리가 어렵다는 것과 중첩으로 인한 복잡도가 증가하는 것이다. 크게 이 두가지 단점을 해결하기 위해 프로미스는 유저들이 라이브러리로 만들어 사용했고, 이것을 es6에서는 언어적 차원에서 지원하게 되었다. 프라미스는 어떤 함수에서 반환하는 객체로 이 객체의 생성자에는 콜백이 들어간다. 이 콜백의 첫번째, 두번째 메서드는 각각 resolve와 reject함수가 들어있는데, resolve함수는 이 프라미스객체가 올바른 값을 반환할때 쓰이며, reject함수는 만약 결과값이 에러 상태 값인 경우 사용한다.

프로미스는 콜백에 코드를 더욱 아름 답게 만들어준다. 나 또한 비동기/동기처리에서 프라미스를 자주 사용한다.

**프로미스 객체의 생성**

프라미스의 생성 방법은 new Promise(()=>{}) 이다.

다음 함수를 보자.

function loginRequest(){

const findUser = Math.floor(Math.random() \* 10);

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

setTimeout(()=>{

*//통신중 표현*

if(findUser % 2){

const user = {

id : "dummy",

pw : "dummy",

}

resolve(user);

}else{

reject(false);

}

},2000);

})

}

이 함수는 간단하게 findUser에 값에 따라 만약 찾았다는 의미로 짝수라면 user객체를 프라미스resolve에 담아 주고 아니라면 reject에 false를 담아주는 녀석이다. 위의 예제를 보면 resolve함수와 reject함수에 각각 성공과 실패 시 전달하는 값이 다르다 . 이 함수를 이제 프라미스 체인을 통해 동기적으로 진행 해보겠다.

loginRequest().then((result)=>{

console.log(result);

}).catch((err)=>{

console.log(err);

})

이 프라미스 체인은 함수가 반환하는 프라미스 객체가 resolve거나 reject에 따라서 .then()메서드 혹은 .cathch의 콜백의 인자로 resolve,reject에 전달인자로 들어간 덴과 캐치의 콜백 함수의 매개변수로 들어간다.

function loginRequest(){

const findUser = Math.floor(Math.random() \* 10);

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

setTimeout(()=>{

*//통신중 표현*

if(findUser % 2 ){

const user = {

id : "dummy",

pw : "dummy",

}

resolve(user);

}else{

reject("로그인 오류가 발생하였습니다.");

}

},2000);

})

}

function findArticle(user){

console.log(user);

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

const findArticle = Math.floor(Math.random() \* 10);

setTimeout(()=>{

*//해당 유저의 게시물을 찾는 중*

console.log("해당 유저의 게시물을 찾는중")

if(findArticle % 6){

user.articleList = [{

title : "dummy",

article : "dummy"

}]

resolve(user);

}else{

reject("게시물을 찾다 오류가 발생하였습니다.");

}

},1500);

})

}

function findMail(addedArticleUser){

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

const findMail = Math.floor(Math.random() \* 10);

setTimeout(()=>{

console.log("해당 유저의 받은 쪽지를 찾는 중")

if(findArticle % 4){

addedArticleUser.mail = {

recieve : [{

to : "dummyId",

from : "dummyId",

}]

}

resolve(addedArticleUser);

}else{

reject("게시물을 찾다 오류가 발생하였습니다.");

}

},1500);

})

}

loginRequest().then(

findArticle

).then(

findMail

).then((userInfo)=>{

console.log("로그인이 완료되었습니다.");

console.log(userInfo);

}).catch((err)=>{

console.log(err)

})

이렇게 기명함수로 작성하기도 하지만 then내부에서 이렇게 작성하는 것도 가능하다.

loginRequest().then((user)=>{

console.log(user);

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

const findArticle = Math.floor(Math.random() \* 10);

setTimeout(()=>{

*//해당 유저의 게시물을 찾는 중*

console.log("해당 유저의 게시물을 찾는중")

if(findArticle % 6){

user.articleList = [{

title : "dummy",

article : "dummy"

}]

resolve(user);

}else{

reject("게시물을 찾다 오류가 발생하였습니다.");

}

},1500);

})

}).then((user)=>{

console.log(user)

}).catch((err)=>{

console.log(err)

})

콜백을 사용하기 전에 전처리 작업들이 필요하다면 다음과 같이 작성 하면 된다.

loginRequest().then((user)=>{

console.log("기타 등등 작업을 하는 중");

console.log("작업 중");

return findArticle(user);

}).then((addedArticleUser)=>{

console.log(addedArticleUser);

console.log("메일 작업 전에 서버에 로그 보내고 이것 저것 하는 중");

return findMail(addedArticleUser)

}).then((userObj)=>{

console.log(userObj);

console.log("사용자 컴퓨터로 보내는 중");

}).catch((err)=>{

console.log(err)

})

위의 예제는 유저의 정보를 가져와야 해당 유저 정보를 바탕으로 데이터베이스에서 나머지 정보를 가져와 붙일 수 있다. 그러나 user정보만 가져온다면 findArticle과 findMail은 순서 상관없이 할 수 있다. 이처럼 순서가 없는 프로그램을 병렬적으로 만들 수 있는 메서드가 있다. 바로 Promise.all이다.

간단한 예제

const param = true;

const promise1 = new Promise(function(resolve,reject){

if(param){

resolve("바보");

}

else{

reject("아닌데");

}

});

const promise2 = new Promise(function(resolve,reject){

if(param){

resolve("바보2");

}

else{

reject("아닌데2");

}

});

Promise.all([findMail,findArticle]).then(function(values){

console.log("1,2,3 모두완료",values);

}).catch((err)=>{

console.log(err);

})

이 프라미스 올은 배열 내부에 있는 프라미스들이 리졸브던, 리젝트던 모두 반환 될때 까지 결과를 주지 않습니다. 또한 이 프라미스는 해당 배열에 있는 한 개의 프라미스라도 리젝트를 발생시키면 모든 프라미스를 reject한 것으로 보고 동작을 중지 합니다.

받은 쪽지와 게시물 함수를 프라미스.all로 처리한 경우

loginRequest().then((user)=>{

console.log("기타 등등 작업을 하는 중");

console.log("작업 중");

Promise.all([findArticle(user),findMail(user)]).then((result)=>{

console.log(result);

}).catch((err)=>{

console.log(`에러 발생 : ${err}`);

})

})

만약 오류가 나더라도 간단하게 처리하고 다음 절차로 넘어 가고 싶으면 다음과 같은 기법도 가능 하다.

var p1 = new Promise((resolve, reject) => {

setTimeout(() => resolve('p1\_지연\_이행'), 1000);

});

var p2 = new Promise((resolve, reject) => {

reject(new Error('p2\_즉시\_거부'));

});

Promise.all([

p1.catch(error => { return error }),

p2.catch(error => { return error }),

]).then(values => {

console.log(values[0]) *// "p1\_지연\_이행"*

console.log(values[1]) *// "Error: p2\_즉시\_거부"*

})

지금까지 프라미스 객체에 대한 내용을 학습했다. Promise를 쓰면 콜백보다 훨씬 간결하고 복잡성을 줄일 수 있을 것 같다. 그러나 promise객체를 사용할 시 저렇게나 깔끔하게 작성되는 경우는 보기 드물고, 덴 내부에서도 무수히 많은 프라미스 객체가 생겨나서 콜백보다는 덜 하지만 옆으로 쭉쭉 흘러가는 형태의 코드가 발생 할 수 도 있다.

loginRequest().then((user)=>{

console.log("기타 등등 작업을 하는 중");

console.log("작업 중");

Promise.all([findArticle(user),findMail(user)]).then((result)=>{

result[0].then((logInfo)=>{

messageLog(logInfo).then(()=>{

authentication().then(()=>{

})

}).then(()=>{

})

})

}).catch((err)=>{

console.log(`에러 발생 : ${err}`);

})

})

**Async & await**

Async/await은 절자척 언어에서 작성하는 코드와 같이 사용법도 간단하고 이해하기도 쉽습니다. Function 키워드 앞에 async 만 붙여주면 되고 비동기로 처리되는 부분에 await만 붙여주면 됩니다. 다만 await부분이 프라미스를 반환해야 합니다.

function findArticle(user){

console.log(user);

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

const findArticle = Math.floor(Math.random() \* 10);

setTimeout(()=>{

*//해당 유저의 게시물을 찾는 중*

console.log("해당 유저의 게시물을 찾는중")

if(findArticle % 2){

user.articleList = [{

title : "dummy",

article : "dummy"

}]

resolve(user);

}else{

reject("게시물을 찾다 오류가 발생하였습니다.");

}

},1500);

})

}

function findMail(addedArticleUser){

console.log(addedArticleUser)

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

const findMail = Math.floor(Math.random() \* 10);

setTimeout(()=>{

console.log("해당 유저의 받은 쪽지를 찾는 중")

if(findMail % 5){

addedArticleUser.mail = {

recieve : [{

to : "dummyId",

from : "dummyId",

}]

}

resolve(addedArticleUser);

}else{

reject("쪽지를 찾다 오류가 발생하였습니다.");

}

},3000);

})

}

function getUserInfo(){

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

setTimeout(()=>{

resolve({

id : "dummy",

pw : "dummy"

});

},2000);

})

}

async function loginRequest(){

const findUser = Math.floor(Math.random() \* 10);

*// console.log(userInfo)*

const userInfo = await getUserInfo();

const addedArticle = await findArticle(userInfo);

const addedMail = await findMail(addedArticle);

console.log(addedMail);

}

loginRequest().catch((err)=>{

console.log(err);

})

위의 복잡했던 내용이 이렇게 쉽게 정리가 됩니다. 다만 아쉽게도 프라미스객체나 async/await은 현재 브라우저에서 지원하지 않습니다. 그렇기에 node환경에서만 사용할 수 있고, 이를 바닐라 js로 사용하기 위해서는 트랜스 컴파일러라고 불리우는 바벨이라는 외부 라이브러리를 통해 es7문법을 es5로 변환하여야 합니다.