필기 노트

2021년 1월 12일 화요일 오후 7:35

1. 호출 스택

third()
second()
first()
anonymous

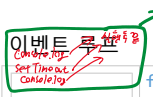
```
function first(){
  second();
  console.log("첫 번째")
}
function second(){
  third()
  console.log("두 번째")
}
function third(){
  console.log("세 번째")
}
first()
```

anomymous는 가상의 전역 컨텍스트

자바스크립트는 함수를 스택(LIFO)방식으로 호출 후 실행

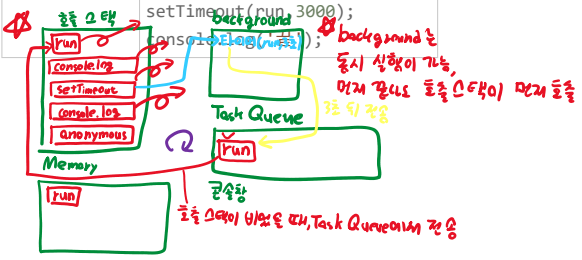
Last In First Out

2. 이벤트 루프



anonymous

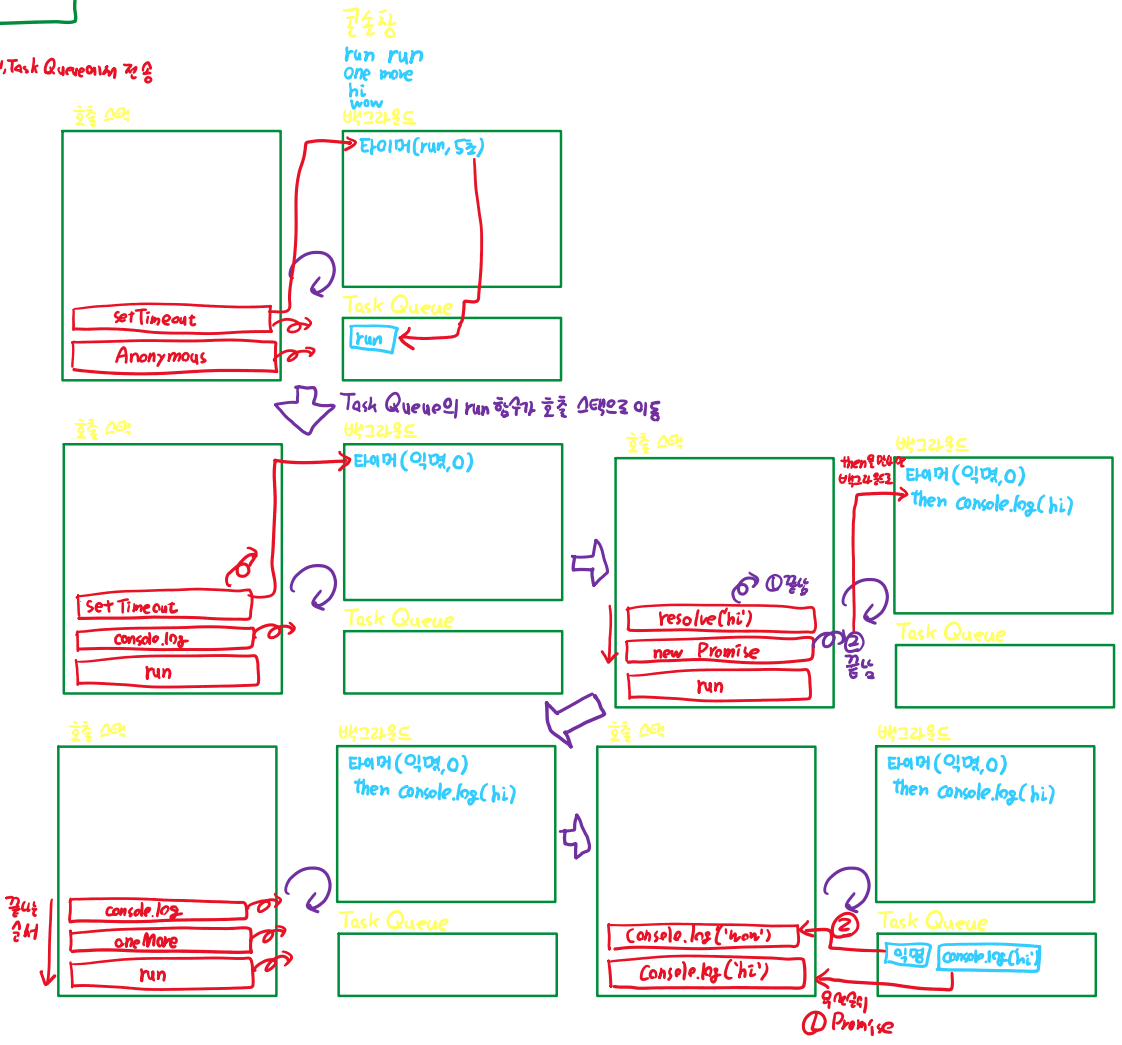
```
function run(){
  console.log("3초 후 실행");
}
console.log("시작");
setTimeout(run, 3000);
console.log("끝");
```



- 복잡한 예제 분석

```
function oneMore(){
  console.log('one more');
}
function run(){
  console.log('run run');
  setTimeout(() => {
    console.log('wow')
  },0)
  new Promise((resolve) => {
    resolve('hi');
  })
    .then(console.log);
  oneMore();
}
setTimeout(run, 5000);
```

Promise then/catch
Process.nextTick 재귀기능



3. ES2015+ 문법

- var, let, const

var : 문법에서 사라져도 지장 X, 레거시 코드 분석 시에는 특성을 알아야 함

var은 함수 스코프, let 및 const는 블록 스코프

var은 블록 스코프를 무시하므로 블록 밖에서도 접근 가능

let, const는 브록 스코프이므로 블록 밖에서 접근 불가

d	var	let, const
if, while, for	무시	접근 불가
function	접근 불가	접근 불가

const는 변수 자체에 =을 한번만 붙일 수 있다

const b = {name:'zerocho'};

b.name = 'hello; 는 가능

- 템플릿 문자열

var result = '이 과자는' + won + '원 입니다';

-> const result = `이 과자는 ${won}원 입니다.`;

함수 호출 시에도 a(); 대신 a``;도 가능함

- 객체 리터럴

옛 방식	새 방식

var sayNode = function(){
  console.log('Node');
};
var es = 'ES';
var oldObject = {
  sayJS : function(){
  console.log('Js');
  },
  sayNode : sayNode,
};
oldObject[es + 6] = 'Fantastic';
oldObject.sayNode(); //Node
oldObject.sayJS(); //Js
console.log(oldObject.ES6);
	let sayNode = function(){
	  console.log('Node');
	};
	const newObject = {
	  sayJS(){
	  console.log('JS');
	  },
	  sayNode,
	  [es + 6] : 'Fantastic',
	};
	newObject.sayNode(); //Node
	newObject.sayJS``; // JS
	console.log(newObject.ES6);

- 화살표 함수

```
function add1(x, y){
  return x + y;
}
// add1을 화살표 함수로 나타냄
const add2 = (x,y) => {
  return x + y;
}
// return만 있는 경우 생략 가능
const add3 = (x,y) => x + y;
// return이 생략된 경우 본문을 소괄호로 감싸줄 수 있음
const add4 = (x,y) => (x + y);
// 아래 두 함수는 동일한 기능
function not1(x){
  return !x;
}
const not2 = x => !x;
```

let relationship1 = {
  name:'zero',
  friends: ['nero','hero','xero'],
  logFriends: function(){
  var that = this;
  this.friends.forEach(function(friend){
  console.log(that.name, friend);
  })
  }
}
relationship1.logFriends();
	const relationship2 = {
	  name: 'zero',
	  friends: ['nero','hero','xero'],
	  logFriends(){
	  this.friends.forEach(friend => {
	  console.log(this.name, friend);
	  })
	  }
	}
	relationship2.logFriends();

예시

```
this
button.addEventListener('click',function(){
        console.log(this.textContent);
}
btn2.addEventListener('click',()=>{
        console.log(this.textContent);
}
```

```
}
btn2.addEventListener('click',()=>{
    console.log(this.textContent);
}
btn2.addEventListener('click',(e)=>{
    console.log(e.target.textContent);
}
```

결론 : this를 쓸거면 function, 안쓸거면 아예 화살표 함수로 통일

- 구조분해 문법

```
// 과거의 사용 방법
const example = {a: 123, b:{c:135, d:146}}
const a = example.a;
const b = example.b.d;
console.log(a)
console.log(b)
// 구조분해 문법 사용 시
const {c, b:{d}} = example
console.log(c)
console.log(d)


// 과거의 사용 방법
const arr = [1,2,3,4,5]
const x = arr[0];
const y = arr[1];
const z = arr[4];
// 구조분해 문법 사용 시
const [x,y,,,z] = arr;
```

- 클래스

프로토타입 문법을 깔끔하게 작성할 수 있는 Class 문법 도입

과거 방식

```
// 예전 방식
let Human = function(type){
    this.type = type || 'human';
};
Human.isHuman = function(human){
    return human instanceof Human;
}
// 프로토타입 메서드
Human.prototype.breathe = function(){
    alert('h-a-a-a-m');
};
let Zero = function(type, firstName, lastName){
    Human.apply(this, arguments);
    this.firstName = firstName;
    this.lastName = lastName;
}
Zero.prototype = Object.create(Human.prototype);
Zero.prototype.constructor = Zero;
Zero.prototype.sayName = function(){
    alert(this.firstname + ' ' + this.lastName);
};
let oldZero = new Zero('human', 'Zero','Cho');
Human.isHuman(oldZero);
```

현재 방식

```
// 현재 방식
class Human{
    constructor(type = 'human'){
        this.type = type;
    }
    static isHuman(human){
        return human instanceof Human;
    }
    breathe(){
        alert('h-a-a-a-m');
    }
}
class Zero extends Human{
    constructor(type, firstName, lastName){
        super(type);
        this.firstName = firstName;
        this.lastName = lastName;
    }
    sayName(){
        super.breathe();
        alert(`${this.firstName} ${this.lastName}`)
    }
}
const newZero = new Zero('human','Zero','Cho');
Human.isHuman(newZero);
```

- 프로미스

내용이 실행은 되었지만 결과를 아직 반환하지 않은 객체

Then을 붙이면 결과를 반환함

resolve(성공 리턴값) -> then으로 연결 reject(실패 리턴값) -> catch로 연결 Finally는 무조건 실행

사용 예시

```
const promise = new Promise((resolve, reject) => {
  if (condition){
    resolve('성공');
  }
  else{
    reject('실패');
  }
})
promise
  .then((message) => {
    console.log(message);
  })
  .catch((error)=>{
    console.error(error);
  })
  .finally(()=>{
```

```
    console.log('무조건')
})
```

콜백 사용	Promise 사용

// 콜백 사용 시
function findAndSaveUser(Users){
  Users.findOne({}, (err, user)=>{ // 첫 번째 콜백
  if(err)
  return console.error(err);
  user.name = 'zero';
  user.save((err) => {
  if(err) // 두 번째 콜백
  return console.error(err);
  Users.findOne({gender:'m'}, (err, user)=>{ // 세 번째 콜백
  // 생략
  });
  });
  });
}
↳ 싫다 어렵다
	// 프로미스 사용 시
	function findAndSaveUser(Users){
	  Users.findOne({})
	  .then((user) => {
	  user.name = 'zero';
	  return user.save();
	  })
	  .then((user)=>{
	  return Users.findOne({gender : 'm'});
	  })
	  .then((user) => {
	  // 생략
	  })
	  .catch(err => {
	  console.error(err);
	  });
	}

```
// Promise.all(배열)은 여러 개의 프로미스를 동시에 실행, 하나라도 실패하면 catch로 감
// allSettled로 실패한 것만 추려낼 수도 있음
const promise1 = Promise.resolve('성공1');
const promise2 = Promise.resolve('성공2');
const promise3 = Promise.reject('실패1');
Promise.all([promise1, promise2])
  .then((result) => {
    console.log(result);    // ['성공1', '성공2']
  })
  .catch((error)=>{
    console.error(error);
  })
```

- Async / await

  then으로 여러 개 사용하는 것을 줄이는 방법

Promise 사용	async / await 사용

// 기존 방법
function findAndSaveUser(Users){
  Users.findOne({})
  .then((user)=>{
  user.name = 'zero';
  return user.save();
  })
  .then((user)=>{
  return Users.findOne({gender:'m'});
  })
  .then((user)=>{
  // 생략
  })
  .catch(err => {
  console.error(error);
  });
}
	// await과 async 사용
	// 실행 순서가 오른쪽에서 왼쪽으로(<-)
	// try/catch로 예외처리
	async function findAndSaveUsers(Users){
	  try{
	  let user = await Users.findOne({});
	  user.name = 'zero';
	  user = await user.save();
	  user = await Users.findOne({gender:'m'});
	  // 생략
	  }
	  catch(error){
	  console.error(error);
	  }
	}

  async에서 return한 것들은 then으로 받아야 한다

  async도 결국에 promise여서 promise의 성질을 가져간다(then이나 await 사용)

  화살표 함수로도 사용 가능

  for await of

```
// for await of
const promise1 = Promise.resolve('성공1');
const promise2 = Promise.resolve('성공2');
(async ()=>{
  for await (promise of [promise1, promise2]){
    console.log(promise);
  }
})();
```

- 프론트엔드 자바스크립트

  Ajax : 서버로 요청을 보내는 코드, Ajax 요청 시 Axios 라이브러리를 사용

  아래는 get 요청 방법

```
<!-- Ajax axios를 사용하깅 위한 코드-->
<script src="https://unpkg.com/axios/dist/axios.min.js"></script>
<script>
  axios.get("https://www.zerocho.com/api/get")
    .then((result) => {
      console.log("result");
```

```
            console.log(result.data);   //{}
        })
        .catch((error) => {
            console.error(error);
        });
    (async () => {
        try{
            const result = await axios.get("https://www.zerocho.com/api/get");
            console.log(result);
            console.log(result.data);   //{}
        }
        catch(error){
            console.error(error);
        }
    })
</script>
```

아래는 Post 요청 방법

```
<script>
    (async () => {
        try{
            const result = await axios.post('https://www.zerocho.com/api/post/json',{
                name: 'zerocho',
                birth: 1994,
            });
            console.log(result);
            console.log(result.data);   //{}
        }catch(error){
            console.error(error);
        }
    })();
</script>
```

- FormData 메서드

  html form 태그에 담긴 데이터를 Ajax 요청으로 보내고 싶은 경우

  Append : 데이터 하나씩 추가

  Has : 데이터 존재 여부 확인 — URI의 공개된 데이터 검색

  Get : 데이터 조회

  getAll : 데이터 모두 조회

  delete : 데이터 삭제

  set : 데이터 수정

```
const formData = new FormData();
formData.append('name','zerocho');
formData.append('item','orange');
formData.append('item','melon');
formData.has('item');   //true
formData.has('money');  //false
formData.get('item');   //orange
formData.getAll('item');    //['orange','melon']
formData.append('test',['hi','zero']);
formData.get('test');   //hi, zero
formData.delete('test');
formData.get('test');   //null
formData.set('item','apple');
formData.getAll('item');    //['apple']
```

  가끔 주소창에 한글을 치면 이해를 못하는 경우가 생김 -> encodeURIComponent로 한글 감싸줘서 처리

  이를 decodeURIComponent로 서버에서 한글 해석

- data attribute와 dataset

  HTML 태그에 데이터 저장법

  서버의 데이터를 Front-end로 내려줄 때 사용

  태그 속성으로 data-속성명, 자바스크립트에서 태그.dataset.속성명으로 접근 가능

  이 방법은 공개된 데이터만 접근 가능, 누구나 접근할 수 있기 때문에

  html to javascript

  - data-user-job -> dataset.userJob
  - data-id -> dataset.id

  javascript to html

  - dataset.monthSalary = 10000 -> data-month-salary = "10000"