**Node.js**

**1. 핵심개념 이해하기**

**1.1 이벤트 기반**

이벤트 기반(event-driven)이란 이벤트가 발생할 때 미리 지정해둔 작업을 수행하는 방식을 의미합니다. 이벤트로는 클릭이나 네트워크 요청이 있을 수 있습니다.

이벤트 기반 시스템에서는 특정 이벤트가 발생할 때 무엇을 할지 미리 등록해 두어야 합니다. 이를 이벤트 리스터(event listener)에 콜백(callback) 함수를 등록한다고 표현합니다.

이벤트 기반 모델에서는 이벤트 루프(event loop)라는 개념이 등장합니다. 여러 이벤트가 동시에 발생 했을 때 어떤 순서로 콜백 함수를 호출할지를 이벤트 루프가 판단합니다.

**\* 이벤트 루프**

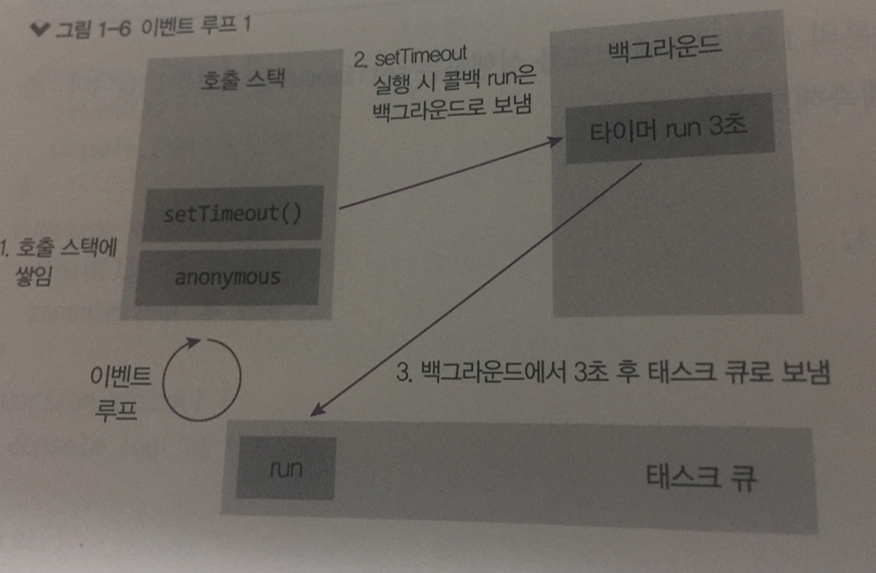
이벤트 발생 시 호출할 콜백 함수들을 관리하고, 호출할 콜백 함수의 실행 순서를 결정하는 역할을 담당합니다. 노드가 종료될 때까지 이벤트 처리를 위한 작업을 반복하므로 루프(loop)라고 부릅니다.

**\* 백그라운드**

setTimeout 같은 타이머나 이벤트리스너들이 대기하는 곳입니다. 자바스크립트가 아닌 다른 언어로 작성된 프로그램이라고 봐도 됩니다. 여러 작업이 동시에 실행될 수 있습니다.

**\* 태스크 큐**

이벤트 발생 후, 백그라운드에서는 태스크 큐로 타이머나 이벤트 리스너의 콜백 함수를 보냅니다. 정해진 순서대로 콜백들이 줄을 서 있으므로 콜백 큐라고 부릅니다. 콜백들은 보통 완료된 순서대로 줄을 서 있지만 특정한 경우에는 순서가 바뀌기도 합니다.



**1.2 논 블로킹 I/O**

이벤트 루프를 잘 활용하면 오래 걸리는 작업을 효율적으로 처리할 수 있습니다. 작업에는 두 가지 종류가 있는데, 동시에 실행될 수 있는 작업과 동시에 실행될 수 없는 작업입니다. 기본적으로 여러분이 작성한 자바스크립트 코드는 동시에 실행될 수 없습니다. 하지만 자바스크립트상에서 돌아가는 것이 아닌 I/O 작업 같은 것은 동시에 처리될 수 있습니다.

I/O는 입력(Input) / 출력(Output)을 의미합니다. 파일 시트템 접근(파일 읽기, 파일 쓰기, 폴더 만들기 등)이나 네트워크를 통한 요청 같은 작업이 I/O의 일종입니다. 이러한 작업을 할 때 노드는 논 블로킹 방식으로 처리하는 방법을 제공합니다.

**논 블로킹** - 이전 작업이 완료될 때까지 대기하지 않고 다음 작업을 수행함을 뜻합니다.

**블로킹** - 이전 작업이 끝나야만 다음 작업을 수행하는 것을 의미 합니다.

**1.3 싱글 스레드**

**2. 알아두어야 할 자바스크립트**

**2.1 ES2015+**

2015년도에 자바스크립트 문법에 매우 큰 변화가 있었습니다. 바로 ES2015(ES6)라고도 부릅니다)가 등장하였습니다. 2015년을 기점으로 매년 문법 변경 사항이 발표되고 있으며, 새로운 문법 상세에 대해서도 활발한 논의가 이루어지고 있습니다.

**2.1.1 const, let**

if (true) {

var x = 3;

}

console.log(x); // 3

if (true) {

const y = 3;

}

console.log(y); // Uncaught ReferenceError: y is not defined

x는 정상적으로 출력되는데 y는 에러가 발생합니다. var은 함수 스코프를 가지므로 if문의 블록과 관계 없이 접근할 수 있습니다. 하지만 const와 let은 블록 스코프를 가지므로 블록 밖에서는 변수에 접근할 수 없습니다.

const, let과 var은 스코프의 종류가 다릅니다. 다만 let과 const는 한 번 값을 할당하면 다른 값을 할당할 수 없습니다(상수)

**2.1.2 템플릿 문자열**

템플릿 문자열은 큰 따옴표나 작은 따옴표로 감싸는 기존 문자열과 다르게 역따옴표(`)로 감싸게 됩니다. 특이한 점은 문자열 안에 변수를 넣을 수 있습니다.

const a = 1;

const b = 2;

var c = `${a} + ${b} = ${a+b}입니다`;

**2.1.3 객체 리터럴**

**ES5 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); // Fantastic

**ES6에서는 다음과 같이 다시 쓸 수 있습니다.**

const newObject = {

SayJS() {  
 console.log('JS');  
 },

sayNode,

[es + 6] : 'Fantastic',

};

newObject.sayNode(); // Node

newObject.sayJS(); // JS

console.log(newObject.ES6); // Fantastic

{ name : name, age : age } // ES5

{ name : age } // ES6

**2.1.4 화살표 함수**

function add1(x, y) {  
 return x + y;  
}

**const add2 = (x, y) => {**

**return x + y;**

**};**

**const add3 = (x, y) => x + y;**

**const add4 = (x, y) => (x + y);**

function not1(x) {

return !x;

}

**const not2 = x => !x;**

**2.1.5 구조분해할당**

구조분해 할당을 사용하면 객체와 배열로부터 속성이나 요소를 쉽게 꺼낼 수 있습니다.

객체에 대한 구조분해 할당

const candyMachine = {

status : {

name : 'node',

count : 5,

},

getCandy() {  
 this.status.count--;

return this.status.count;

}

};

const { getCandy, status : { count } } = candyMachine;

배열에 대한 구조분해 할당

const array = ['nodeJs', {}, 10, true];

const [node, obj, , bool] = array;

**2.1.6 프로미스**

자바스크립트와 노드에서는 주로 비동기를 접합니다. 특히 이벤트 리스너를 사용할 때 콜백 함수를 자주 사용합니다. ES6 부터는 자바스크립트와 노드의 API들이 콜백 대신 프로미스(Promise)기반으로 재고성되며, 악명 높은 콜백 지옥(callback hell) 현상을 극복했다는 평가를 받고 있습니다.

프로미스를 쉽게 설명하면 **실행은 바로 하되 결과값은 나중에 받는 객체**입니다. 결과 값은 실행이 완료된 후 **then**이나 **catch** 메서드를 통해 받습니다.

const condition = true;

const promise = new Promise((resolve, reject) => {

if (condition) {

resolve('성공');

} else {

reject('실패');

}

});

promise

.then((message) => {

console.log(message); // 성공(resolve)한 경우 실행

})

.catch((error) => {

console.error(error); // 실패(reject)한 경우 실행

})

.finally(() => { // 끝나고 무조건 실행

console.log(”무조건 실행”);

});

\* 프로미스 내부에서 resolve가 호출이 되면 then이 실행이 되고, reject가 호출이 되면 catch가 실행됩니다.

\* finally부분은 성공/실패 여부와 상관없이 실행됩니다.

\* resolve, reject에 넣어준 인수는 각각 then과 catch의 매개변수에서 받을 수 있습니다.

**\* Promise.all**

프로미스 여러 개를 한 번에 실행할 수 있는 방법

const promise1 = Promise.resolve('성공1');

const promise2 = Promise.resolve('성공2');

Promise.all([promise1, promise2])

.then((result) => {

console.log(result); // ['성공1', '성공2'];

})

.catch((error) => {

console.error(error);

});

**2.1.7 async / await**

프로미스가 콜백 지옥을 해결했다지만, 여전히 코드가 장황합니다. then과 catch가 계속 반복되기 때문입니다. async / await 문법은 프로미스를 사용한 코드를 한 번 더 깔끔하게 줄입니다.

**Promise 문법**

const http = require('http');

const fs = require('fs').promises;

http.createServer((req, res) => {

fs.readFile('./server.html')

.then((data) => {

res.writeHead(200, { 'Content-Type' : 'text/html; charset=utf-8' });

res.end(data);

})

.catch((err) => {

console.error(err);

res.writeHead(500, {'Content-Type' : 'text/html; charset=utf-8' });

res.end(err.message);

});

})

.listen(8081, () => {

console.log(”8081번 포트에서 서버 대기 중 입니다!”);

});

**async / await 문법**

const http = require('http');

const fs = require('fs').promises;

http.createServer(async (req, res) => {

try {

const data = await fs.readFile('./server.html');

res.writeHead(200, { 'Content-Type' : 'text/html; charset=utf-8' });

res.end(data);

} catch (err) {

console.error(err);

res.writeHead(500, { 'Content-Type' : 'text/plain; charset=utf-8' });

res.end(err.message);

}

})

.listen(8081, () => {

console.log(”8081번 포트에서 서버 대기 중 입니다!”);

});

**2.2 프론트엔드 자바스크립트**

**2.2.1 AJAX**

AJAX(Asynchronous Javascript And XML)는 비동기적 웹 서비스를 개발할 때 사용하는 기법입니다. 이름에 XML이 들어 있지만 꼭 XML을 사용해야 하는 것은 아니며, 요즘에는 JSON을 많이 사용합니다. 쉽게 말해 페이지 이동 없이 서버에 요청을 보내고 응답을 받는 기술입니다. 웹 사이트 중에서 페이지 전환 없이 새로운 데이터를 불러오는 사이트는 대부분 AJAX 기술을 사용하고 있다고 보면 됩니다.

보통 AJAX요청은 jQuery나 axios 같은 라이브러리를 이용해서 보냅니다.

**2.2.2 encodeURIComponent , decodeURIComponent**

AJAX 요청을 보낼 때 “http://localhost:4000/search/노드”처럼 주소에 한글이 들어가는 경우가 있습니다. 서버 종류에 따라 다르지만 서버가 한글 주소를 이해하지 못하는 경우가 있는데 이럴 때는 window 객체의 메서드인 encodeURIComponent 메서드를 사용합니다.

받는 쪽에서는 decodeURIComponent를 사용하면 됩니다.

encodeURIComponent와 decodeURIComponent는 한글을 처리하기 위한 것으로 생각하면 됩니다.

**3. 노드 기능 알아보기**

**3.1 모듈 만들기**

노드는 코드를 모듈로 만들 수 있다는 점에서 브라우저의 자바스크립트와 다릅니다. 모듈이란 특정한 기능을 하는 함수나 변수의 집합입니다.

모듈로 만들면 여러 프로그램에 해당 모듈을 재사용 할 수 있습니다.

**var.js**

const odd = ‘홀수입니다.’;

const event = ‘짝수입니다.’;

module,exports = {

odd,

even,

};

**func.js**

const { odd, even } = require(‘./var’);

function checkOddOrEven(num) {

if (num % 2) { // 홀수면

return odd;

}

return even;

}

module.exports = checkOddOrEven;

**index.js**

const { odd, even } = require(‘./var’);

const checkNumber = require(‘./func’);

function checkStringOddOrEven(str) {

if (str.length % 2) { // 홀수명

return odd;

}

return even;

}

console.log(checkNumber(10));

console.log(checkStringOddOrEven(‘hello’));

**3.2 노드 내장 객체**

**3.2.1 global**

global 객체, 브라우저의 window와 같은 브라우저 객체입니다. 전역 객체이므로 모든 파일에서 접근할 수 있습니다.

window.open 메서드를 그냥 open으로 호출할 수 있는 것처럼 global도 생략할 수 있습니다.

이전 절에서 사용했던 require 함수도 global.require에서 global이 생략된 것입니다.

노드 콘솔에서 로그를 기록하는 console 객체도 global.console입니다.

**globalA.js**

module.exports = () => global.message;

**globalB,js**

const A = require(‘./globalA’);

global.message = ‘안녕하세요’;

console.log(A());

globalA 모듈의 함수는 global.message 값을 반환 합니다.

globalB.js에서는 global 객체에 속성명이 message인 값을 대입하고 globalA 모듈의 함수를 호출합니다. 콘솔 결과는 globalB에서 넣은 global.message 값을 globalA에서도 접근할 수 있습니다.

**3.2.2 console**

**console.time(레이블)**

console.timeEnd(레이블)과 대응대어 같은 레이블을 가진 time과 timeEnd 사이의 시간을 측정합니다.

**console.log(내용)**

평번함 로그를 콘솔에 표시합니다. console.log(내용, 내용, ...)처럼 여러 내용을 동시에 표시할 수 있습니다.

**console.error(에러 내용)**

에러를 콘솔에 표시합니다.

**console.table(배열)**

배열의 요소로 객체 리터럴을 넣으면 객체의 속성들이 테이블 형식으로 표현됩니다.

**console.dir(객체, 옵션)**

객체를 콘솔에 표시할 때 사용합니다.

**console.trace(레이블)**

에러가 어디서 발생했는지 추적할 수 있게 합니다

**3.2.3 타이머**

타이머 기능을 제공하는 함수인 setTimeout, setInterval, setImmediate는 노드에서 window대신 global 객체 안에 들어 있습니다.

**setTimeout(**콜백함수, 밀리초)

주어진 밀리초(1000분의 1초) 이후에 콜백 함수를 실행 합니다.

**setInterval**(콜백함수, 밀리초)

주어진 밀리초마다 콜백 함수를 반복 실행합니다.

**setImmediate**(콜백함수)

콜백 함수를 즉시 실행합니다.

**clearTimeout**(아이디) : setTimeout을 취소합니다.

**clearInterval**(아이디) : setInterval을 취소합니다.

**clearImmediate**(아이디) : setImmediate를 취소합니다.

예시)

const immediate = setImmediate(() => {

console.log(“즉시 실행”);

});

const immediate2 = setImmediate(() => {

console.log(“실행되지 않습니다.”);

});

clearImmediate(immediate2);

**3.2.4 \_\_filename, \_\_dirname**

\_\_filename – 파일명을 포함한 현재 파일 경로

\_\_dirname – 현재 파일의 경로

**3.2.5 process**

process 객체는 현재 실행되고 있는 노드 프로세스에 대한 정보를 담고 있습니다.

process.version – 설치된 노드의 버전

process.arch – 프로세스의 아키텍쳐 예) x64

process.platform – 운영체제 플랫폼 정보 (win32, linux...)

process.pid – 현재 프로세스의 아이디

process.uptime() - 프로세스가 시작된 후 흐른 시간(단위 – 초)

process.execPath – 노드의 경로

process.cwd() - 현재 프로세스가 실행되는 위치  
process.cpuUsage() - 현재 CPU 사용량

**3.2.6 process.env**

시스템의 환경 변수 – 노드에 직접적인 영향을 미침

사용자가 추가한 환경변수

const secretId = process.env.SECRET\_ID;

const secretCode = process.env.SECRET\_CODE;

**3.2.7 process.nextTick(콜백)**

이벤트 루프가 다른 콜백 함수들보다 nextTick의 콜백 함수를 우선으로 처리하도록 만듭니다,

nextTick.js

setImmediate(() => {

console.log(‘immediate’);

});

proccess.nextTick(() => { // 가장 먼저 실행됨

console.log(‘nextTick’);

});

setTimeout(() => {

console.log(‘timeout’);

}, 0);

Promise.resolve().then(() => console.log(‘promise’));

**3.2.8 process.exit(코드)**

실행 중인 노드 프로세스를 종료합니다. 서버 환경에서 이 함수를 사용하면 서버가 멈추므로 특수한 경우를 제외하고는 서버에서 잘 사용하지 않습니다. 하지만 서버 외의 독립적인 프로그램에서는 수동으로 노드를 멈추기 위해 사용합니다.

**3.3 노드 내장 모듈**

**3.3.1 os**

웹브라우저에 사용되는 자바스크립트는 운영체제의 정보를 가져올 수 없지만, 노드는 os 모듈에 정보가 담겨 있어 정보를 가져올 수 있습니다.

os.arch() : process.arch()와 동일 – 프로세스의 아키텍쳐(x64 ... )

os,platform() : process.platform과 동일 - 운영체제 플랫폼 정보 (win32, linux...)

os.type() : 운영체제의 종류를 보여줍니다.

os.uptime() : 운영체제 부팅 이후 흐른 시간(초)을 보여줍니다.

(process.uptime()은 노드의 실행 시간)

os.hostname() : 컴퓨터의 이름을 보여줍니다.

os.release() : 운영체제의 버전을 보여줍니다.

os.homedir() : 홈 디렉토리 경로를 보여줍니다.

os.tmpdir() : 임시파일 저장경로를 보여줍니다.

os.cpus() : 컴퓨터의 코어 정보를 보여줍니다.

os.freemem() : 사용 가능한 메모리(RAM)를 보여줍니다.

os.totalmem() : 전체 메모리 용량을 보여줍니다.

**3.3.2 path**

폴더와 파일의 경로를 쉽게 조작하도록 도와주는 모듈입니다. path 모듈이 필요한 이유 중 하나는 운영체제별로 경로 구분자가 다르기 때문입니다. 크게 윈도 타입과 POSIX 타입으로 구분됩니다. POSIX는 유닉스 기반의 운영체제들을 의미하며 맥과 리눅스가 속해 있습니다.

윈도 : C:\javascript\node처럼 \로 구분합니다.

POSIX : /javascript/node처럼 /로 구분합니다.

path.sep : 경로의 구분자입니다. 윈도는 \, POSIX는 /입니다

path.delimiter : 환경 변수의 구분자입니다. process.env.PATH를 입력하면 여러 개의 경로가 이 구분자로 구분되어 있습니다. 윈도는 세미콜론(;)이고, POSIX는 콜론(:)입니다.

path.dirname(경로) : 파일의 위치한 폴더 경로를 보여줍니다.

path.extname(경로) : 파일의 확장자를 보여줍니다.

path.basename(경로, 확장자) : 파일의 이름(확장자 포함)을 표시합니다. 파일의 이름만 표시하고 싶다면 basename의 두 번째 인수를 파일의 확장자를 넣으면 됩니다.

path.parse(경로) : 파일의 경로를 root, dir, base, ext, name으로 분리합니다.

path.format(객체) : path.parse()한 객체를 파일 경로로 합칩니다.

path.normalize(경로) : /나 \를 실수로 여러 번 사용했거나 혼용했을 때 정상적인 경로로 변환합니다.

path,isAbsolute(경로) : 파일의 경로가 절대경로인지 상대경로인지를 true나 false로 알립니다.

path.relative(기준경로, 비교경로) : 경로를 두 개 넣으면 첫 번째 경로에서 두 번째 경로로 가는 방법을 알립니다.

path.join(경로, ...) : 여러 인수를 넣으면 하나의 경로로 합칩니다. 상대경로인 ..(부모 디렉터리)과 .(현 위치)도 알아서 처리합니다.

**3.3.3 url**

인터넷 주소를 쉽게 조작하도록 도와주는 모듈 입니다. URL처리에는 크게 두 가지 방식이 있습니다. 노드 버전 7에서 추가된 WHATWG(웹 표준을 정하는 단체의 이름) 방식의 url과 예전부터 노드에서 사용하던 방식의 url이 있습니다.

**url.js**

const url = require('url');

const { URL } = url;

const myURL = new URL('http://yonggyo.com/book/bookList.aspx?category=001001000#anchor');

console.log('new URL():', myURL);

console.log('url.format():', url.format(myURL));

console.log('------------------------------------------------');

const parsedUrl = url.parse('http://yonggyo.com/book/bookList.aspx?category=001001000#anchor');

console.log('url.parse():', parsedUrl);

console.log('url.format():', url.format(parsedUrl));

**결과**

new URL(): URL {

href: 'http://yonggyo.com/book/bookList.aspx?category=001001000#anchor',

origin: 'http://yonggyo.com',

protocol: 'http:',

username: '',

password: '',

host: 'yonggyo.com',

hostname: 'yonggyo.com',

port: '',

pathname: '/book/bookList.aspx',

search: '?category=001001000',

searchParams: URLSearchParams { 'category' => '001001000' },

hash: '#anchor'

}

url.format(): http://yonggyo.com/book/bookList.aspx?category=001001000#anchor

------------------------------------------------

url.parse(): Url {

protocol: 'http:',

slashes: true,

auth: null,

host: 'yonggyo.com',

port: null,

hostname: 'yonggyo.com',

hash: '#anchor',

search: '?category=001001000',

query: 'category=001001000',

pathname: '/book/bookList.aspx',

path: '/book/bookList.aspx?category=001001000',

href: 'http://yonggyo.com/book/bookList.aspx?category=001001000#anchor'

}

url.format(): http://yonggyo.com/book/bookList.aspx?category=001001000#anchor

**url.parse(주소) :** 주소를 분해합니다. WHATWG 방식과 비교하면 username과 password 대신 auth 속성이 있고, searchParams 대신 query가 있습니다.

**url.format(객체) :** WHATWG 방식 url과 기존 노드의 url을 모두 사용할 수 있습니다. 분해되었던 url 객체를 다시 원상태로 조립합니다

WHATWG 방식은 search 부분을 searchParams라는 특수한 객체로 반환하므로 유용합니다. search 부분은 보통 주소를 통해 데이터를 전달할 때 사용됩니다. search는 물음표(?)로 시작하고, 그 뒤에 키=값 형식으로 데이터를 전달합니다. 여러 키가 있을 경우에는 &로 구분합니다.

**searchParams.js**

const { URL } = require('url');

const myURL = new URL('http://yonggyo.com/?page=3&limit=10&category=nodejs&category=javascript');

console.log('searchParams : ', myURL.searchParams);

console.log('searchParams.getAll() : ', myURL.searchParams.getAll('category'));

console.log('searchParams.get() : ', myURL.searchParams.get('limit'));

console.log('searhcParams.has() : ', myURL.searchParams.has('page'));

console.log('searchParams.keys() : ', myURL.searchParams.keys());

console.log('searchParams.values() : ', myURL.searchParams.values());

myURL.searchParams.append('filter', 'es3');

myURL.searchParams.append('filter', 'es5');

console.log(myURL.searchParams.getAll('filter'));

myURL.searchParams.set('filter', 'es6');

console.log(myURL.searchParams.getAll('filter'));

myURL.searchParams.delete('filter');

console.log(myURL.searchParams.getAll('filter'));

console.log('searchParams.toString():', myURL.searchParams.toString());

myURL.search = myURL.searchParams.toString();

URL 생성자를 통해 myURL이라는 주소 객체를 만들었습니다. myURL 안에는 searchParams 객체가 있습니다. 이 객체는 search 부분을 조작하는 다양한 메서드를 지원합니다.

\* getAll(키) : 키에 해당하는 모든 값들을 가져옵니다. category키에는 nodejs와 javascript라는 두 가지 값이 들어 있습니다.

\* get(키) : 키에 해당하는 첫 번째 값만 가져옵니다.

\* has(키) : 해당 키가 있는지 없는지를 검사합니다.

\* keys() : searchParams의 모든 키를 반복키(iterator)(ES2015문법) 객체로 가져옵니다.

\* values() : searchParams의 모든 값을 반복키 객체로 가져옵니다.

\* append(키, 값) : 해당 키를 추가 합니다. 같은 키의 값이 있다면 유지하고 하나 더 추가합니다.

\* set(키, 값) : append와 비슷하지만, 같은 키의 값을 모두 지우고 새로 추가합니다.

\* delete(키) : 해당 키를 제거합니다.

\* toString() : 조작한 searchParams 객체를 다시 문자열로 만듭니다. 이 문자열을 search에 대입하면 주소 객체에 반영됩니다.

query와 같은 문자열보다 searchParams가 유용한 이유는 query의 경우 다음에 배우는 querystring모듈을 한 번 더 사용해야 하기 때문입니다.

**3.3.4 querystring**

WHATWG 방식의 url 대신 기존 노드의 url을 사용할 때, search 부분을 사용하기 쉽게 사용하기 쉽게 객체로 만드는 모듈입니다.

**querystring.js**

const url = require('url');

const querystring = require('querystring');

const parsedUrl = url.parse('http://yonggyo.com/?page=3&limit=10&category=nodejs&category=javascript');

const query = querystring.parse(parsedUrl.query);

console.log('querystring.parse():', query);

console.log('querystring.stringify():', querystring.stringify(query));

**결과**

querystring.parse(): [Object: null prototype] {

page: '3',

limit: '10',

category: [ 'nodejs', 'javascript' ]

}

querystring.stringify(): page=3&limit=10&category=nodejs&category=javascript

\* **querystring.parse(쿼리)** : url의 query 부분을 자바스크립트 객체로 분해합니다.

\* **querystring.stringify(객체)** : 분해된 query 객체를 문자열로 다시 조립합니다.

**3.3.5 child\_process**

노드에서 다른 프로그램을 실행하고 싶거나 명령어를 수행하고 싶을 때 사용하는 모듈입니다. 이 모듈을 통해 다른 언어의 코드(예를 들면, 파이썬)를 실행하고 결과값을 받을 수 있습니다.

**exec.js**

const exec = require('child\_process').exec;

const process = exec('dir');

process.stdout.on('data', function(data) {

console.log(data.toString());

}); // 실행 결과

process.stderr.on('data', function(data) {

console.error(data.toString());

}); // 실행 에러

**spawn.js**

const spawn = require('child\_process').spawn;

const process = spawn('python', ['test.py']);

process.stdout.on('data', function(data) {

console.log(data.toString());

});

process.stderr.on('data', function(data) {

console.error(data.toString());

});

**3.3.6 기타 모듈들**

자세한 사항은 공식 문서를 참조

**assert** : 값을 비교하여 프로그램이 제대로 동작하는지 테스트하는 데 사용합니다.

**dns** : 도메인 이름에 대한 IP 주소를 얻어내는 데 사용합니다.

**net** : HTTP보다 로우 레벨인 TCP나 IPC 통신을 할 때 사용합니다.

**string\_decoder** : 버퍼 데이터를 문자열로 바꾸는 데 사용합니다.

**tls** : TLS와 SSL에 관련된 작업을 할 때 사용합니다.

**tty** : 터미널과 관련된 작업을 할 때 사용합니다.

**dgram** : UDP와 관련된 작업을 할 때 사용합니다.

**v8** : V8 엔진에 직접 접근할 때 사용합니다.

**vm** : 가상 머신에 직접 접근할 때 사용합니다.

**3.4 파일 시스템 접근하기**

fs 모듈은 파일 시스템에 접근하는 모듈입니다. 즉, 파일을 생성하거나 삭제하고, 일거나 쓸 수 있습니다. 폴더도 만들거나 지울 수 있습니다.

**readFile.js**

const fs = require('fs');

fs.readFile('./readme.txt', (err, data) => {

if (err) {

throw err;

}

console.log(data);

console.log(data.toString());

});

**결과**

<Buffer ec a0 80 eb a5 bc 20 ec 9d bd ec 96 b4 ec a3 bc ec 84 b8 ec 9a 94 2e>

저를 읽어주세요.

readFile의 결과물은 버퍼(Buffer)라는 형식으로 제공됩니다.

버퍼를 toString()을 사용해 사람이 읽을 수 있는 형식인 문자열로 변경합니다.

fs는 기본적으로 콜백 형식의 모듈이므로 실무에서 사용하기가 불편하므로 fs 모듈을 프로미스 형식으로 바꿔주는 방법을 사용

**readFilePromise.js**

const fs = require('fs').promises;

fs.readFile('./readme.txt')

.then((data) => {

console.log(data);

console.log(data.toString());

})

.catch((err) => {

console.error(err);

});

const fs = require('fs').promises;

fs.writeFile('./writeme.txt', '글이 입력됩니다.')

.then(() => {

return fs.readFile('./writeme.txt');

})

.then((data) => {

console.log(data.toString());

})

.catch((err) => {

console.error(err);

});

\* 동기 메서드와 비동기 메서드

비동기 예)

실행순서가 유지가 되지 않음

**async.js**

const fs = require('fs');

console.log('시작');

fs.readFile('./readme.txt', (err, data) => {

if (err) {

throw err;

}

console.log("1번", data.toString());

});

fs.readFile('./readme.txt', (err, data) => {

if (err) {

throw err;

}

console.log("2번", data.toString());

});

fs.readFile('./readme.txt', (err, data) => {

if (err) {

throw err;

}

console.log('3번', data.toString());

});

console.log("끝");

동기 예)   
실행 순서가 유지됨

다만 다른 작업이 완료 될 때 까지 메인스레드가 대기 하고 있으므로 비효율적이며 성능상의 문제가 발생함

**sync.js**

const fs = require('fs');

console.log('시작');

let data = fs.readFileSync('./readme.txt');

console.log('1번', data.toString());

data = fs.readFileSync('./readme.txt');

console.log('2번', data.toString());

data = fs.readFileSync('./readme.txt');

console.log('3번', data.toString());

console.log('끝');

비동기 순서 유지 예)

const fs = require('fs');

console.log('시작');

fs.readFile('./readme.txt', (err, data) => {

if (err) {

throw err;

}

console.log('1번', data.toString());

fs.readFile('./readme.txt', (err, data) => {

if (err) {

throw err;

}

console.log('2번', data.toString());

fs.readFile('./readme.txt', (err, data) => {

if (err) {

throw err;

}

console.log("3번", data.toString());

console.log('끝');

});

});

});

콜백 안에 콜백이 중첩되는 구조로 순서가 늘어날수록 구조가 복잡해진다

비동기 순서 유지 – 프로미스 방식 예)

const fs = require('fs').promises;

console.log('시작');

fs.readFile('./readme.txt')

.then((data) => {

console.log("1번", data.toString());

return fs.readFile('./readme.txt');

})

.then((data) => {

console.log("2번", data.toString());

return fs.readFile('./readme.txt');

})

.then((data) => {

console.log("3번", data.toString());

console.log("끝");

})

.catch((err) => {

console.error(err);

});

비동기 순서 유지 – async ~ await 예)

**asyncAwait.js**

const fs = require('fs').promises;

( async function() {

console.log('시작');

try {

const data1 = await fs.readFile('./readme.txt');

console.log("1번", data1.toString());

const data2 = await fs.readFile('./readme.txt');

console.log("2번", data2.toString());

const data3 = await fs.readFile('./readme.txt');

console.log("3번", data3.toString());

} catch (err) {

console.error(err);

}

console.log("끝");

})();

**3.5 버퍼와 스트림 이해하기**

파일을 읽거나 쓰는 방식은 버퍼를 이용하는 방식과 스트림을 이용하는 방식이 있습니다.

버퍼 – 파일을 저장하는 공간

스트림 – 일정하게 쪼개진 버퍼에 나눠서 전송하는 방식

**파일 읽기**

**createReadStream.js**

const fs = require('fs');

const readStream = fs.createReadStream('./readme.txt', { highWaterMark : 8 });

const data = [];

readStream.on('data', (chunk) => {

data.push(chunk);

console.log('data :', chunk, chunk.length);

});

readStream.on('end', () => {

console.log('end :', Buffer.concat(data).toString());

});

readStream.on('error', (err) => {

console.log('error', err);

});

**파일 쓰기**

**createWriteStream.js**

const fs = require('fs');

const writeStream = fs.createWriteStream('./writeme.txt');

writeStream.on('finish', () => {

console.log("파일 쓰기 완료");

});

writeStream.write("글쓰기1\n");

writeStream.write("글쓰기2\n");

writeStream.end();

**파일을 읽고 쓰기**

**pipe.js**

const fs = require('fs');

const readStream = fs.createReadStream('readme.txt');

const writeStream = fs.createWriteStream('writeme.txt');

readStream.pipe(writeStream);

**gzip.js**

const zlib = require('zlib');

const fs = require('fs');

const readStream = fs.createReadStream('./readme.txt');

const zlibStream = zlib.createGzip();

const writeStream = fs.createWriteStream('./writeme.txt.gz');

readStream.pipe(zlibStream).pipe(writeStream);