Node.js 기초 문법

* export를 통해서 메소드를 만들어 놓고, 나중에 다른 js파일에서 이를 불러와 쓸 수 있다.

// test.js

exports.double = function(r) {

return r \* r; };

exports.plus = function(r) {

return r + r; };

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

// main.js

var caculator = require('./test.js'); //여기서 test.js의 메소드 불러옴

console.log('Double value is '+ caculator.double(3));

console.log('Plus value is ' + caculator.plus(3));;

=========================================================================

var util = require('util');

var data = util.format('%d, %s, %j', 6, 'chapter', {cotent: 'module'});

console.log(data);

util.log('message');

=========================================================================

File System

크게 비동기, 동기 방식으로 나뉘어짐.

|  |  |  |
| --- | --- | --- |
|  | 읽기 | 쓰기 |
| 비동기 | fs.readFile(filename, [options], callback)  filename의 파일을 [options]의 방식으로 읽은 후 callback으로 전달된 함수를 호출합니다. | fs.writeFile(filename, data, [options], callback)  filename의 파일에 [options]의 방식으로 data 내용을 쓴 후 callback 함수를 호출합니다. |
| 동기 | fs.readFileSync(filename, [options])  filename의 파일을 [options]의 방식으로 읽은 후 문자열을 반환합니다. | fs.writeFileSync(filename, data, [options])  filename의 파일에 [options]의 방식으로 data 내용을 씁니다. |

[option] 에는 인코딩 방식이 오며,

callback에는 function이 들어가고 I/O 작업 완료시 수행되는 function을 의미함.

예제

|  |  |
| --- | --- |
| //main.js  var fs = require('fs');  // 비동기적 읽기  fs.readFile('text.txt', 'utf8', function(err, data) {  console.log('비동기적 읽기 ' + data);  });  -----------------------------------------------------------  // 동기적 읽기  var text = fs.readFileSync('text.txt', 'utf8');  console.log('동기적 읽기 ' + text); | // main.js  var fs = require('fs');  var data = 'fs.writeFile test';  fs.writeFile('text1.txt', data, 'utf8', function(err){  console.log('비동기적 파일 쓰기 완료');  });  -------------------------------------------------------------  fs.writeFileSync('text2.txt', data, 'utf8');  console.log('동기적 파일 쓰기 완료'); |

실행결과 ‘동기’의 결과가 먼저 나온다.

예외 처리

동기 : try / catch 사용

비동기 : if / else 로 사용 => callback함수의 매개변수로 err가 전달 되기 때문에 try/catch를 사용할 필요가 없다.

=========================================================================

이벤트 모듈

event.EventEmitter라는 인스턴스를 사용,

node js에 내장되어있는 EventEmitter클래스 객체를 생성해서 사용함.

관련 메소드 들로는

이벤트 생성

emitter.addListener(event,listener)

emitter.on(evnet,listener)

한 번만 연결

emitter.once(event, listener) : 이벤트를 한 번만 연결한 후 제거.

이벤트 발생

emitter.emit(eventName[, ...args]) : 이벤트를 발생시킵니다.

이벤트 제거

emitter.removeListener(event, listener) : 특정 이벤트의 특정 이벤트 핸들러를 제거   
이 메소드를 이용해 리스너를 삭제하면 리스너 배열의 인덱스가 갱신됨.

emitter.removeAllListeners([event]) : 모든 이벤트 핸들러를 제거..

이벤트 허용 개수 설정

emitter.setMaxListeners(n) : n으로 한 이벤트에 최대허용 개수를 정해줍니다. node.js는 기본값으로 한 이벤트에 10개의 이벤트 핸들러를 작성할 수 있는데, 11개 이상을 사용하고 싶다면 n값을 넘겨주면 됩니다. n값으로 0을 넘겨 주면 연결 개수 제한이 사라집니다.

var EventEmitter = require('events'); // EventEmitter 클래스를 받아와서

var custom\_event = new EventEmitter(); // 새 객체를 하나 만듬.

custom\_event.on('call', function() { // 이벤트를 생성(핸들러를 연결)함.call이 이벤트 이름이다.

console.log('이벤트 콜');

});

custom\_event.emit('call'); //이벤트 발생시킴

custom\_event.removeAllListeners(); //이벤트 제거

custom\_event.emit('call'); // 리스너를 제거하였기에 아무것도 발생 X

=========================================================================

상속

function Foo() { }

Foo.prototype = {

bar: function() { console.log('Foo\_bar 실행'); } //bar 메소드를 가지는 Foo 객체

};

function Bar() { }

Bar.prototype = Object.create(Foo.prototype); //Foo prototype객체 원형을 상속받은 Bar 객체

Bar.prototype.baz = function() { console.log('Bar\_baz 실행'); }; Bar 객체에 baz 메소드 추가.

Foo.prototype.bar();

Bar.prototype.bar();

Bar.prototype.baz();

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

또다른 방법

var util = require('util');

function Bar() { }

util.inherits(Bar, Foo); //Bar는 Foo를 상속받음.

Bar.prototype.baz = function() {

console.log('Bar\_baz 실행');

};

Foo.prototype.bar();

Bar.prototype.bar();

Bar.prototype.baz();

=========================================================================

프로토 타입

만약 Say라는 프

* evan.say로 접근 시도
* evan객체에 say 프로퍼티가 없네? \_\_proto\_\_를 통해 원본 객체로 올라가보자!
* User.prototype객체의 say 프로퍼티에 접근함.



이런식으로 계속 생성해 나갈 수 있다.

이러한 것을 prototype chain이라고 한다.

=========================================================================

출처 : <https://edu.goorm.io/learn/lecture/557/%ED%95%9C-%EB%88%88%EC%97%90-%EB%81%9D%EB%82%B4%EB%8A%94-node-js/lesson/174361/file-system-%EB%AA%A8%EB%93%88>

그 외 필요한 것들은

<https://nodejs.org/api/fs.html> 여기서 찾으면 됩니다.

aws에서

dynamo db에 요청할 때

{

"TableName": "테이블 이름 지정한 거",

"FilterExpression": "user\_id = :id and user\_pw = :pw",

"ExpressionAttributeValues": {

":id": " ~ ",

":pw": " ~ "

}

}

이렇게 요청해서 하나만 검색 해 올 수 있음.