2장

2장

2.1 REPL 사용하기

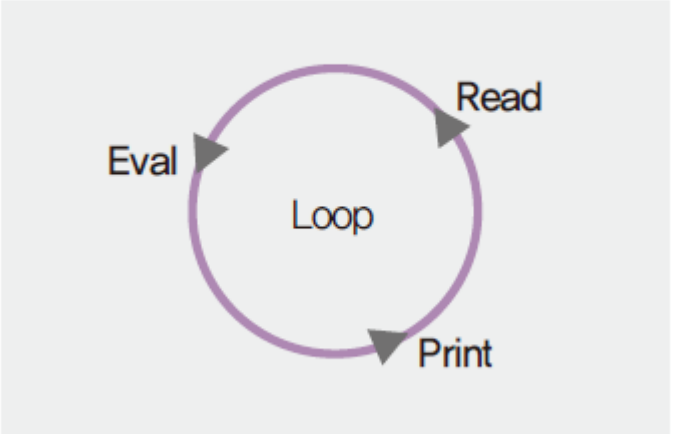
1. REPL

- 자바스크립트는 스크립트 언어라서 즉석에서 코드를 실행할 수 있음
  - REPL이라는 콘솔 제공
  - R(Read), E(Evaluate), P(Print), L(Loop)
  - 윈도에서는 명령 프롬프트, 맥이나 리눅스에서는 터미널에 node 입력

콘솔

```
$ node
>
```

그림 3-1 REPL



1. REPL

- 프롬프트가 > 모양으로 바뀌면, 자바스크립트 코드 입력
- 입력한 값의 결과값이 바로 출력됨.
  - 간단한 코드를 테스트하는 용도로 적합
  - 긴 코드를 입력하기에는 부적합

콘솔

```
> const str = 'Hello world, hello node';
undefined
> console.log(str);
Hello world, hello node
undefined
>
```

2.2 JS 파일 실행하기

1. JS 파일을 만들어 실행하기

- 자바스크립트 파일을 만들어 통째로 코드를 실행하는 방법
  - 아무 폴더(디렉터리)에서 helloWorld.js를 만들어보자
  - node [자바스크립트 파일 경로]로 실행
  - 실행 결과값이 출력됨

helloWorld.js

```
function helloWorld() {
  console.log('Hello World');
  helloNode();
}

function helloNode() {
  console.log('Hello Node');
}

helloWorld();
```

콘솔

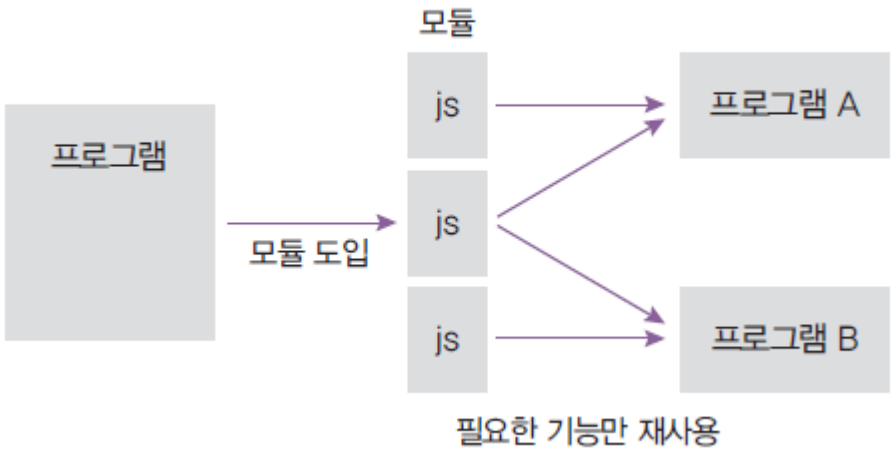
```
$ node helloWorld
Hello World
Hello Node
```

2.3 모듈로 만들기

1. 모듈

- 노드는 자바스크립트 코드를 모듈로 만들 수 있음
  - 모듈: 특정한 기능을 하는 함수나 변수들의 집합
  - 모듈로 만들면 여러 프로그램에서 재사용 가능

그림 3-2 모듈과 프로그램



2. 모듈 만들어보기

- 같은 폴더 내에 var.js, func.js, index.js 만들기
  - 파일 끝에 module.exports로 모듈로 만들 값을 지정
  - 다른 파일에서 require(파일 경로)로 그 모듈의 내용 가져올 수 있음

```
const odd = '홀수입니다';
const even = '짝수입니다';

module.exports = {
  odd,
  even,
};
```

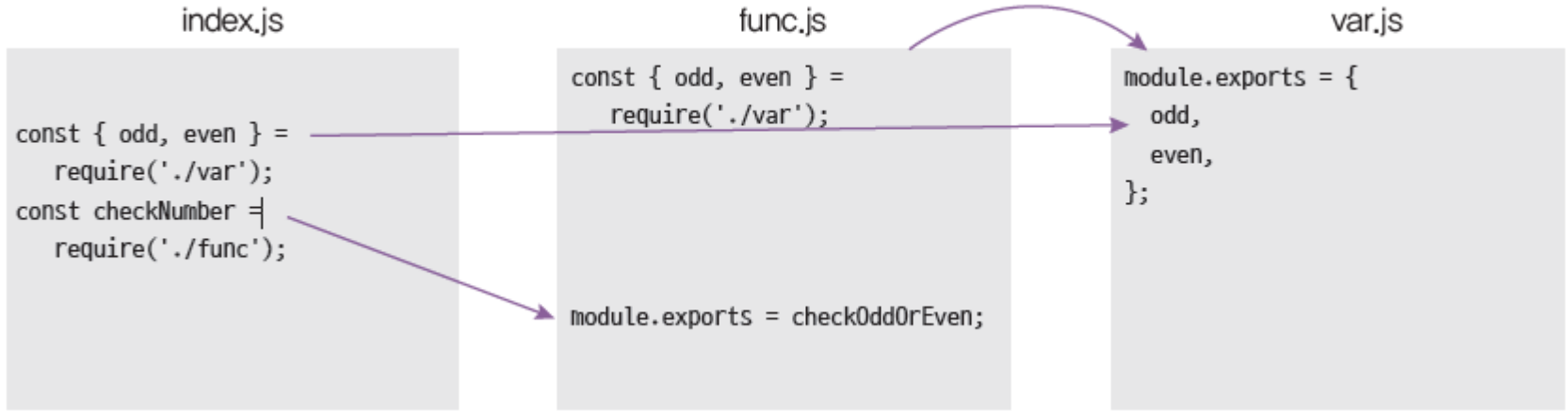
```
const { odd, even } = require('./var');

function checkOddOrEven(num) {
  if (num % 2) {
    return odd;
  }
  return even;
}

module.exports = checkOddOrEven;
```

```
const { odd, even } = require('./var');
const checkNumber = require('./func');

function checkStringOddOrEven(str) {
  return checkNumber(str.length);
}

console.log(checkNumber(10));
console.log(checkStringOddOrEven('hello'));
```

3. 파일 간의 모듈 관계

- node index로 실행
  - const { odd, even } 부분은 module.exports를 구조분해 할당한 것임(2장 참고)

그림 3-3 require와 module.exports



콘솔

```
$ node index
짝수입니다
홀수입니다
```

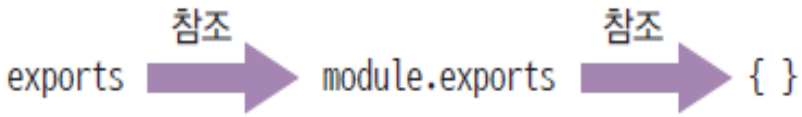
4. module, exports

- module.exports 외에도 exports로 모듈을 만들 수 있음
  - 모듈 예제의 var.js를 다음과 같이 바꾼 후 실행

```
exports.odd = '홀수입니다';
exports.even = '짝수입니다';
```

  - 동일하게 동작함
  - 동일한 이유는 module.exports와 exports가 참조 관계이기 때문

그림 3-5 exports와 module.exports의 관계



  - exports에 객체의 속성이 아닌 다른 값을 대입하면 참조 관계가 깨짐

5. this

- 노드에서 this를 사용할 때 주의점이 있음
  - 최상위 스코프의 this는 module.exports를 가리킴
  - 그 외에는 브라우저의 자바스크립트와 동일
  - 함수 선언문 내부의 this는 global(전역) 객체를 가리킴

```
console.log(this);
console.log(this == {});
console.log(this === module.exports); console.log(this === exports);
console.log(this === global);
```

```
function whatIsThis() {
console.log('function', this === exports, this === global);
}
whatIsThis();
```

6. require의 특성

- 몇 가지 알아둘 속성이 있음
  - require가 제일 위에 올 필요는 없음
  - require.cache에 한 번 require한 모듈에 대한 캐시 정보가 들어있음.
  - require.main은 노드 실행 시 첫 모듈을 가리킴.

```
module.exports = '데이터'

require('./var');
console.log('require.cache')
console.log(require.cache); // 모듈 캐싱 : 한 번 require한 모듈은 다시 require하지 않음
console.log('--------------------');
console.log('require.main')
console.log(require.main === module); // 노드 실행 시 첫 모듈
console.log('--------------------');
console.log('require.main.filename')
console.log(require.main.filename); // 노드 실행 시 첫 모듈의 파일명
```

7. 순환 참조 주의

- 모듈 A가 B를 require하고, B가 다시 A를 require 하는 경우
  - 무한 반복을 막기 위해 순환 참조 대상이 빈 객체가 됨

```
const dep2 = require('./dep2');
console.log('require dep2', dep2);

module.exports = () => {
console.log('dep2', dep2)
};
```

```
const dep1 = require('./dep1');
console.log('require dep1', dep1);

module.exports = () => {
console.log('dep1', dep1)
};
```

```
const dep1 = require('./dep1');

dep1();
```

```
require dep1 {}
require dep2 [Function (anonymous)]
dep2 [Function (anonymous)]
(node:26808) Warning: Accessing non-existent property
'Symbol(nodejs.util.inspect.custom)' of module exports
inside circular dependency
(Use 'node --trace-warnings ...' to show where the warning
was created)
(node:26808) Warning: Accessing non-existent property
'constructor' of module exports inside circular dependency
(node:26808) Warning: Accessing non-existent property
'Symbol(Symbol.toStringTag)' of module exports inside
circular dependency
```

8. ES 모듈

- 자바스크립트 자체 모듈 시스템 문법이 생김
  - mjs 확장자를 사용하거나 package.json에 type: “module”을 추가해야 함.
  - 크게는 require 대신 import, module.exports 대신 export default, exports 대신 export를 쓰는 것으로 바뀜

```
export const odd = 'MJS 홀수입니다';
export const even = 'MJS 짝수입니다';
```

```
import { odd, even } from './var.mjs';

function checkOddOrEven(num) {
  if (num % 2) return odd;
  return even;
}

export default checkOddOrEven;
```

```
import { odd, even } from './var.mjs';
import checkNumber from './func.mjs';

function checkStringOddOrEven(str) {
  return checkNumber(str.length);
}

console.log(checkNumber(10));
console.log(checkStringOddOrEven('hello'));
```

9. CommonJS와 ES 모듈의 차이

차이점	CommonJS 모듈	ECMAScript 모듈
문법	module.exports = 변수1 require('./파일명'); exports.변수2 = 값2; const 변수2 = require('./파일명'); const 변수3 = 값3; const 변수4 = 값4; exports.변수3 = 변수3; exports.변수4 = 변수4; const { 변수3, 변수4 } = require('./파일명');	export default 변수1; import './파일명.mjs' export const 변수2 = 값2; import { 변수2 } from './파일명.mjs' const 변수3 = 값3; const 변수4 = 값4; export { 변수3, 변수4 }; import { 변수3, 변수4 } from './파일명.mjs'
확장자	js, cjs	js (package.json에 type: "module" 필요)
확장자 생략 다이나믹 임포트 index 생략 서로 간 호출	가능	불가능
__filename, __dirname, require, module.exports, exports	가능	불가능
top level await	불가능	가능

10. 다이내믹 임포트, top level await

- 코드 중간에 동적으로 불러올 수 있음
  - Commonjs에서는 require()
  - ES 모듈에서는 import()
- mjs 파일에서 최상위 스코프에선 async 없이 await할 수 있음

```
const isExecute = true;

if (isExecute) {
   const sayHello = require('./func');
   sayHello();
}
console.log('성공');
```

```
const isExecute = true;

if (isExecute) {
   const { sayHello } = import('./func.mjs');
   sayHello();
}
console.log('성공');
```

```
const isExecute = true;

if (isExecute) {
   const { sayHello } = await import('./func.mjs');
   sayHello();
}
console.log('성공');
```

11. __filename, __dirname

- __filename: 현재 파일 경로
- __dirname: 현재 폴더(디렉터리) 경로
- ES 모듈에서는 쓸 수 없어서 import.meta.url을 써야 함

```
console.log(__filename);
console.log(__dirname);
```

```
console.log(import.meta.url);
```

2.4 노드 내장 객체 알아보기

1. global

- 노드의 전역 객체
  - 브라우저의 window같은 역할
  - 모든 파일에서 접근 가능
  - window처럼 생략도 가능(console, require도 global의 속성)

```
console.log(global);
console.log(global.console);
```

2. global 속성 공유

- global 속성에 값을 대입하면 다른 파일에서도 사용 가능

```
module.exports = () => global.msg;
```

```
const A = require('./globalA');
global.msg = '안녕하세요';

console.log(A());
```

3. console 객체

- 브라우저의 console 객체와 매우 유사
  - console.time, console.timeEnd: 시간 로깅
  - console.error: 에러 로깅
  - console.log: 평범한 로그
  - console.dir: 객체 로깅
  - console.trace: 호출스택 로깅

4. 타이머 메서드

- set 메서드에 clear 메서드가 대응됨
  - set 메서드의 리턴 값(아이디)을 clear 메서드에 넣어 취소

  - **setTimeout(콜백 함수, 밀리초)**: 주어진 밀리초(1000분의 1초) 이후에 콜백 함수를 실행합니다.
  - **setInterval(콜백 함수, 밀리초)**: 주어진 밀리초마다 콜백 함수를 반복 실행합니다.
  - **setImmediate(콜백 함수)**: 콜백 함수를 즉시 실행합니다.

  - **clearTimeout(아이디)**: setTimeout을 취소합니다.
  - **clearInterval(아이디)**: setInterval을 취소합니다.
  - **clearImmediate(아이디)**: setImmediate를 취소합니다.

5. 타이머 예제

- setTimeout(콜백, 0)보다 setImmediate 권장

```
const timeout = setTimeout(() => { console.log('1.5초 후 실행'); }, 1500);
const interval = setInterval(() => { console.log('1초마다 실행'); }, 1000);
const timeout2 = setTimeout(() => { console.log('실행되지 않습니다'); }, 3000);

const immediate = setImmediate(() => { console.log('즉시 실행'); });
const immediate2 = setImmediate(() => { console.log('실행되지 않습니다'); });

setTimeout(() => {
  clearTimeout(timeout2);
  clearInterval(interval);
}, 2500);

clearImmediate(immediate2);
```

그림 3-4 실행 순서

초	실행	콘솔
0	immediate ~~immediate2~~	즉시 실행
1	interval	1초마다 실행
1.5	timeout	1.5초마다 실행
2	interval	1초마다 실행
2.5	~~timeout2~~ ~~interval~~	

6. process

- 현재 실행중인 노드 프로세스에 대한 정보를 담고 있음
  - 컴퓨터마다 출력값이 PPT와 다를 수 있음

  - process.version : 설치된 노드의 버전
  - process.arch : 프로세서 아키텍처 정보
  - process.platform : 운영체제 플랫폼 정보
  - process.pid : 현재 프로세스의 PID
  - process.uptime() : 프로세스가 시작된 후 흐른 시간
  - process.execPath : 노드의 경로
  - process.cwd() : 현재 프로세스의 실행 위치
  - process.cpuUsage() : 현재 CPU 사용량
  - process.exit(); // 노드 프로세스 종료

```
Welcome to Node.js v18.16.1.
Type ".help" for more information.
> process.version
'v18.16.1'
> process.arch
'arm64'
> process.platform
'darwin'
> process.pid
39005
> process.uptime()
37.038349875
> process.execPath
'/usr/local/bin/node'
> process.cwd()
'/Users/inkyu/Documents/nodejs'
> process.cpuUsage()
{ user: 501740, system: 74660 }
```

7. process.env

- 시스템 환경 변수들이 들어있는 객체
  - 비밀키(데이터베이스 비밀번호, 서드파티 앱 키 등)를 보관하는 용도로도 쓰임
  - 환경 변수는 process.env로 접근 가능
  - dotenv 모듈을 설치하여 사용 [npm install -g dotenv]

```
SECRET_ID = '외부에서 설정한 ID'
SECRET_KEY = '외부에서 설정한 KEY'
```

```
require('dotenv').config();

const serectId = process.env.SECRET_ID;
const serectKey = process.env.SECRET_KEY;

console.log(serectId);
console.log(serectKey);
```

  - 일부 환경 변수는 노드 실행 시 영향을 미침
  - 예시) NODE_OPTIONS(노드 실행 옵션 지정), UV_THREADPOOL_SIZE(스레드풀 개수 설정)
    - max-old-space-size는 노드가 사용할 수 있는 메모리를 지정하는 옵션

```
NODE_OPTIONS = --max-old-space-size=8192
UV_THREADPOOL_SIZE = 128
```

8. process.nextTick(콜백)

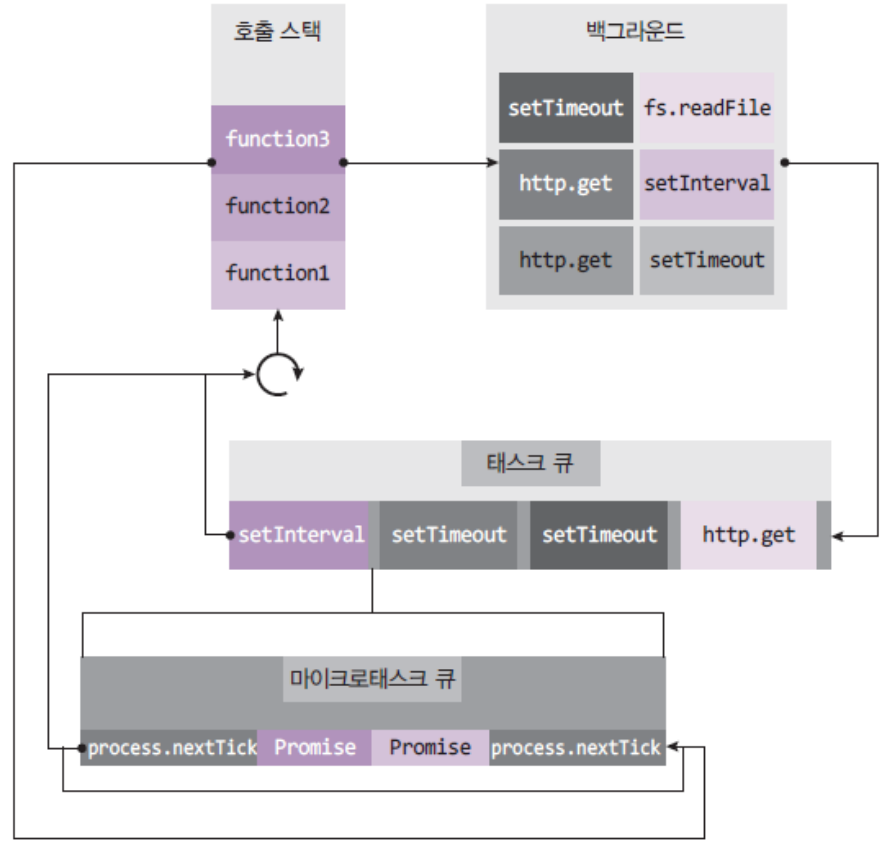
- 이벤트 루프는 다른 콜백 함수들보다 nextTick의 콜백 함수를 우선적으로 처리함
  - 너무 남용하면 다른 콜백 함수들 실행이 늦어짐
  - 비슷한 경우로 promise가 있음(nextTick처럼 우선순위가 높음)
  - 아래 예제에서 setImmediate, setTimeout보다 promise와 nextTick이 먼저 실행됨

```
setImmediate(()=> {
   console.log('immediate');
})

process.nextTick(()=> {
   console.log('nextTick');
})

setTimeout(()=> {
   console.log('timeout');
}, 0)

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

console.log('no callback func');
```

9. 마이크로태스크

- Promise, nextTick 등
  - setImmediate, setTimeout보다 promise와 nextTick이 먼저 실행됨

그림 3-6 태스크와 마이크로태스크



10. process.exit(코드)

- 현재의 프로세스를 멈춤
  - 코드가 없거나 0이면 정상 종료
  - 이외의 코드는 비정상 종료를 의미함

```
let i = 1;

setInterval(() => {
    if (i === 5) {
        console.log('종료!');
        process.exit(0);
    }
    console.log(i);
    i += 1;
}, 1000);
```

2.5 노드 내장 모듈 사용하기

1. os

- 운영체제의 정보를 담고 있음
  - 모듈은 require로 가져옴(내장 모듈이라 경로 대신 이름만 적어줘도 됨)

```
const os = require('os');

console.log('운영체제 정보----------------------');
console.log('프로세서 아키텍처: ', os.arch());
console.log('운영체제 플랫폼: ', os.platform());
console.log('운영체제 종류: ', os.type());
console.log('운영체제 부팅시간(초): ', os.uptime());
console.log('컴퓨터 이름: ', os.hostname());
console.log('운영체제 버전: ', os.release());

console.log('경로----------------------------');
console.log('홈 디렉터리 경로: ', os.homedir());
console.log('임시 파일 저장 경로: ', os.tmpdir());

console.log('cpu 정보-------------------------');
console.log('컴퓨터 코어 정보: ', os.cpus());
console.log('컴퓨터 코어 개수: ', os.cpus().length);

console.log('메모리 정보-------------------------');
console.log('사용 가능한 메모리 용량: ', os.freemem());
console.log('전체 메모리 용량: ', os.totalmem());
```

2. path

- 폴더와 파일의 경로를 쉽게 조작하도록 도와주는 모듈
  - 운영체제별로 경로 구분자가 다름
  - 윈도우즈 타입 : C:₩User₩user POSIX 타입 : /home/user

3. 알아둬야할 path 관련 정보

- join과 resolve의 차이: resolve는 /를 절대경로로 처리, join은 상대경로로 처리
  - 상대 경로: 현재 파일 기준. 같은 경로면 점 하나(.), 한 단계 상위 경로면 점 두 개(..)
  - 절대 경로는 루트 폴더나 노드 프로세스가 실행되는 위치가 기준

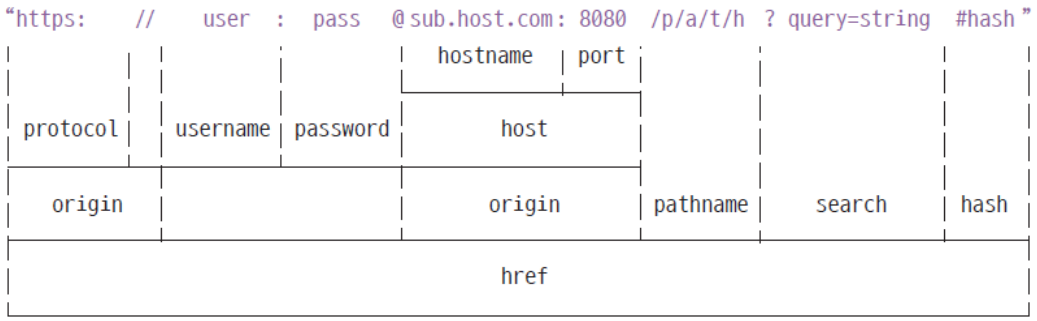
```
path.join('/a', '/b', 'c'); /* 결과: /a/b/c/ */
path.resolve('/a', '/b', 'c'); /* 결과: /b/c */
```

- ₩₩와 ₩ 차이: ₩는 윈도 경로 구분자, ₩₩는 자바스크립트 문자열 안에서 사용(₩가 특수문자라 ₩₩로 이스케이프 해준 것)

- 윈도에서 POSIX path를 쓰고 싶다면: path.posix 객체 사용
  - POSIX에서 윈도 path를 쓰고 싶다면: path.win32 객체 사용

4. url 모듈

- 인터넷 주소를 쉽게 조작하도록 도와주는 모듈
  - url 처리에 크게 두 가지 방식이 있음(기존 노드 방식 vs WHATWG 방식)
  - 요즘은 WHATWG 방식만 사용함
  - WHATWG 방식 주소 체계는 다음과 같음




```
const url = require('url'); // 생략 가능

const { URL } = url;
const myURL = new URL('https://~~~~~?..=..&..=..#..');

console.log('url 객체 : ', myURL);
console.log('url search 부분 : ', myURL.search);
console.log('url search params 객체 : ', myURL.searchParams);
console.log('url 조립 : ', url.format(myURL));
```

5. searchParams

- WHATWG 방식에서 쿼리스트링(search) 부분 처리를 도와주는 객체

```
console.log('searchParams:', myURL.searchParams);
console.log('searchParams.getAll():', myURL.searchParams.getAll('query'));
console.log('searchParams.get():', myURL.searchParams.get('query'));
console.log('searchParams.has():', myURL.searchParams.has('query'));

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

myURL.searchParams.append('query', 'node');
// myURL.searchParams.set('query', 'node');
// myURL.searchParams.delete('query');
console.log(myURL.searchParams.getAll('query'));

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

6. dns

- DNS를 다룰 때 사용하는 모듈
  - 도메인을 통해 IP나 DNS 레코드를 얻고자 할 때 사용
  - A(ipv4 주소)
  - AAAA(ipv6 주소)
  - NS(네임서버)
  - SOA(도메인 정보)
  - CNAME(별칭, www가 붙은 주소는 주로 별칭)
  - MX(메일 서버)

```
const dns = require('dns').promises;

const domain = 'naver.com';

async function findIpAddressInfo() {
  const ip = await dns.lookup(domain);
  console.log('IP', ip);

  const ipv4 = await dns.resolve(domain, 'A ' );
  console.log(ipv4, 'A [IPv4 주소]');

  const ns = await dns.resolve(domain, 'NS ' );
  console.log('NS [네임서버]', ns);

  const soa = await dns.resolve(domain, 'SOA ' );
  console.log('SOA [도메인정보]', soa);

  const cname = await dns.resolve('www.' + domain, 'CNAME ' );
  console.log('CNAME [별칭]', cname);

  const mx = await dns.resolve(domain, 'MX ' );
  console.log('MX [메일서버]', mx);

  const any = await dns.resolve(domain, 'ANY ' );
  console.log('ANY [모든 정보]', any);
}

findIpAddressInfo();
```

7. 단방향 암호화(crypto)

- 암호화는 가능하지만 복호화는 불가능
  - 암호화: 평문을 암호로 만듦
  - 복호화: 암호를 평문으로 해독

그림 3-8 해시 함수

- 단방향 암호화의 대표 주자는 해시 기법
  - 문자열을 고정된 길이의 다른 문자열로 바꾸는 방식
  - abcdefgh 문자열 -> qvew

8. Hash 사용하기(sha512)

- createHash(알고리즘): 사용할 해시 알고리즘을 넣어줍니다.
  - md5, sha1, sha256, sha512 등이 가능 (md5와 sha1은 취약점이 발견)
  - 현재는 sha512 정도로 충분
- update(문자열): 변환할 문자열을 넣어줍니다.
- digest(인코딩): 인코딩할 알고리즘을 넣어줍니다.
  - base64, hex, latin1이 주로 사용되는데, 그중 base64가 결과 문자열이 가장 짧아 자주 사용

```
const crypto = require('crypto');

const pass = '비밀번호';

console.log('base64:', crypto.createHash('sha512').update(pass).digest('base64'));
console.log('hex:', crypto.createHash('sha512').update(pass).digest('hex'));
```

9. pbkdf2

- 컴퓨터의 발달로 기존 암호화 알고리즘이 위협받고 있음
  - sha512가 취약해지면 sha3으로 넘어가야함
  - 현재는 pbkdf2나, bcrypt, scrypt 알고리즘으로 비밀번호를 암호화

그림 3-9 pbkdf2

10. pbkdf2 예제

- 컴퓨터의 발달로 기존 암호화 알고리즘이 위협받고 있음
  - crypto.randomBytes로 64바이트 문자열 생성 -> salt 역할
  - pbkdf2 인수로 순서대로 비밀번호, salt, 반복 횟수, 출력 바이트, 알고리즘
  - 반복 횟수를 조정해 암호화하는 데 1초 정도 걸리게 맞추는 것이 권장됨

```
const crypto = require('crypto');

const pass = '비밀번호';

crypto.randomBytes(64, (err, buf) => {
  // buf : 64바이트 길이의 랜덤한 문자열
  const salt = buf.toString('base64’);
  console.log('salt: ', salt);
  crypto.pbkdf2(pass, salt, 100000, 64, 'sha512', (err, key) => {
    console.log('password:', key.toString('base64'));
  });
});
```

11. 양방향 암호화

- 대칭형 암호화(암호문 복호화 가능)
  - Key가 사용됨
  - 암호화할 때와 복호화 할 때 같은 Key를 사용해야 함

```
const crypto = require('crypto');

const algorithm = 'aes-256-cbc';
const key = 'abcdefghijklmnopqrstuvwxyz123456';
const iv = '1234567890123456';

const message = "비밀 메시지";

// 암호화하기
const cipher = crypto.createCipheriv(algorithm, key, iv);
let result = cipher.update(message, 'utf8', 'base64');
result += cipher.final('base64');
console.log('암호문:', result);

// 복호화하기
const decipher = crypto.createDecipheriv(algorithm, key, iv);
let result2 = decipher.update(result, 'base64', 'utf8');
result2 += decipher.final('utf8');
console.log('평문:', result2);
```

12. util

- 각종 편의 기능을 모아둔 모듈
  - deprecated와 promisify가 자주 쓰임

```
const util = require('util');
const crypto = require('crypto');

const dontUseMe = util.deprecate((x, y) => {
  console.log(x + y);
}, 'dontUseMe 함수는 deprecated되었으니 더 이상 사용하지 마세요!');

dontUseMe(1, 2);

const randomBytesPromise = util.promisify(crypto.randomBytes);
randomBytesPromise(64)
  .then((buf) => {
    console.log(buf.toString('base64'));
  })
  .catch((error) => {
    console.log(error);
});
```

13. worker_threads

- 노드에서 멀티 스레드 방식으로 작업할 수 있음.
  - isMainThread: 현재 코드가 메인 스레드에서 실행되는지, 워커 스레드에서 실행되는지 구분
  - 메인 스레드에서는 new Worker를 통해 현재 파일(__filename)을 워커 스레드에서 실행시킴
  - worker.postMessage로 부모에서 워커로 데이터를 보냄
  - parentPort.on('message')로 부모로부터 데이터를 받고, postMessage로 데이터를 보냄

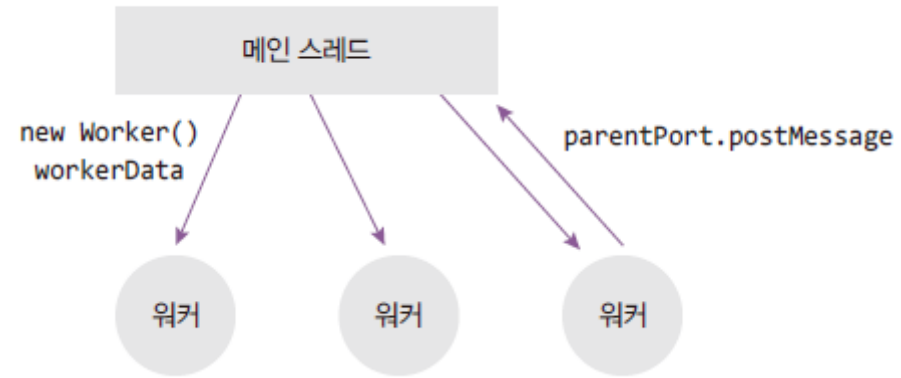
```
const { Worker, isMainThread, parentPort } = require('worker_threads');

if (isMainThread) {
   const worker = new Worker(__filename);
   worker.on('message', (msg) => console.log('부모가 받은 데이터 :', msg));
   worker.on('exit', (code) => console.log('exit', code));
   worker.postMessage('부모가 워커에 전송한 데이터');
} else {
   parentPort.on('message', (msg) => {
      console.log('워커가 받은 데이터 :' ,msg);
      parentPort.postMessage('워커가 부모에게 전송한 데이터');
      parentPort.close();
   });
}
```

14. 여러 워커스레드 사용하기

- new Worker 호출하는 수만큼 워커 스레드가 생성됨

```
const { Worker, isMainThread, parentPort, workerData } = require('worker_threads');

if (isMainThread) {
  const threads = new Set();
  threads.add(new Worker(__filename, {
    workerData: { start: 1 },
  }));
  threads.add(new Worker(__filename, {
    workerData: { start: 2 },
  }));
  for (let worker of threads) {
    worker.on('message', message => console.log(message+ '번 워커로부터'));
    worker.on('exit', () => {
      threads.delete(worker);
      if (threads.size === 0) {
        console.log('job done');
      }
    });
  }
} else {
  const data = workerData;
  parentPort.postMessage(data.start);
}
```

그림 3-10 메인 스레드와 워커의 통신



15. 소수 찾기 예제

- 워커 스레드를 사용하지 않을 때(싱글 스레드일 때) 2 ~ 10,000,000 사이의 소수를 구하는데 걸리는 시간을 측정

```
const min = 2;
const max = 10000000;
const primes = [];

function generatePrimes(start, range) {
  let isPrime = true;
  const end = start + range;
  for (let i = start; i < end; i++) {
    for (let j = min; j < Math.sqrt(end); j++) {
      if (i !== j && i % j === 0) {
        isPrime = false;
        break;
      }
    }
    if (isPrime) {
      primes.push(i);
    }
    isPrime = true;
  }
}

console.time('prime');
generatePrimes(min, max);
console.timeEnd('prime');
console.log(primes);
```

15. 소수 찾기 예제

- 워커 스레드를 사용할 때

```
const { Worker, isMainThread, parentPort, workerData } = require('worker_threads');

const min = 2;
let primes = [];

function generatePrimes(start, range) {
  ...
}

if (isMainThread) { // 메인 스레드
  const max = 10000000;
  const threadCount = 8; // 8개의 스레드를 사용한다.
  const threads = new Set();
  const range = Math.ceil((max - min) / threadCount); // 1250000
  let start = min;
  console.time('prime');
  for (let i = 0; i < threadCount - 1; i++) {
    const wStart = start;
    threads.add(new Worker(__filename, { workerData: { start: wStart, range } }));
    start += range;
  }
  threads.add(new Worker(__filename, { workerData: { start, range: max - start } }));
  for (let worker of threads) {
    worker.on('error', (err) => { throw err });
    worker.on('exit', () => {
      threads.delete(worker);
        if (threads.size === 0) {
          console.timeEnd('prime');
          console.log(primes);
        }
    });
    worker.on('message', (msg) => { primes = primes.concat(msg) });
  }
} else { // 워커 스레드
  generatePrimes(workerData.start, workerData.range);
  parentPort.postMessage(primes);
}
```

16. child_process

- 노드에서 다른 프로그램을 실행하고 싶거나 명령어를 수행하고 싶을 때 사용
  - 현재 노드 프로세스 외에 새로운 프로세스를 띄워서 명령을 수행함.
  - 명령 프롬프트의 명령어인 dir을 노드를 통해 실행 (맥북은 ls를 대신 적을 것)

```
const exec = require('child_process').exec;

const process = exec('ls');

process.stdout.on('data', function(data) {
   console.log(data.toString());
}); // 실행 결과

process.stderr.on('data', function(data) {
   console.error(data.toString());
}); // 실행 에러
```

17. child_process

- 파이썬 프로그램 실행하기
  - 맥북의 경우, python을 python3로!

```
const { spawn } = require('child_process');
var process = spawn('python', ['test.py']);

process.stdout.on('data', (data) => {
   console.log(data.toString());
});

process.stderr.on('data', (data) => {
   console.error(data.toString());
});
```

2.6 파일 시스템 접근하기

1. fs

- 파일 시스템에 접근하는 모듈
  - 파일/폴더 생성, 삭제, 읽기, 쓰기 가능
  - 웹 브라우저에서는 제한적이었으나 노드는 권한을 가지고 있음

```
const fs = require('fs');

fs.readFile('readme.txt', 'utf8', (err, data) => {
  if (err) {
    throw err;
  }
  console.log(data);
});

fs.readFile('readme.txt', (err, data) => {
  if (err) {
    throw err;
  }
  console.log(data);
  console.log(data.toString());
});
```

2. fs 프로미스

- 콜백 방식 대신 프로미스 방식으로 사용 가능
  - require('fs').promises
  - 사용하기 훨씬 더 편해서 프로미스 방식을 추천함

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

fs.readFile('./readme.txt', 'utf8')
  .then((data) => {
    console.log(data);
  })
  .catch((err) => {
    console.error(err);
  });

fs.readFile('./readme.txt')
  .then((data) => {
    console.log(data);
    console.log(data.toString());
  })
  .catch((err) => {
    console.error(err);
  });
```

3. fs로 파일 만들기

- 파일을 만드는 예제

```
const fs = require('fs');

fs.writeFile('./writeme.txt', '글이 입력됩니다', (err) => {
   if (err) {
      throw err;
   }
   fs.readFile('./writeme.txt', 'utf-8', (err, data) => {
      if (err) {
         throw err;
      }
      console.log(data);
   });
});
```

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

fs.writeFile('./writeme.txt', '글이 입력됩니다.')
   .then(() => {
      fs.readFile('./writeme.txt')
         .then((data) => {
            console.log(data.toString());
         });
      })
   .catch((err) => {
      console.error(err);
   });
```

4. 동기 메서드와 비동기 메서드

- 노드는 대부분의 내장 모듈 메서드를 비동기 방식으로 처리
  - 비동기는 코드의 순서와 실행 순서가 일치하지 않는 것을 의미
  - 일부는 동기 방식으로 사용 가능

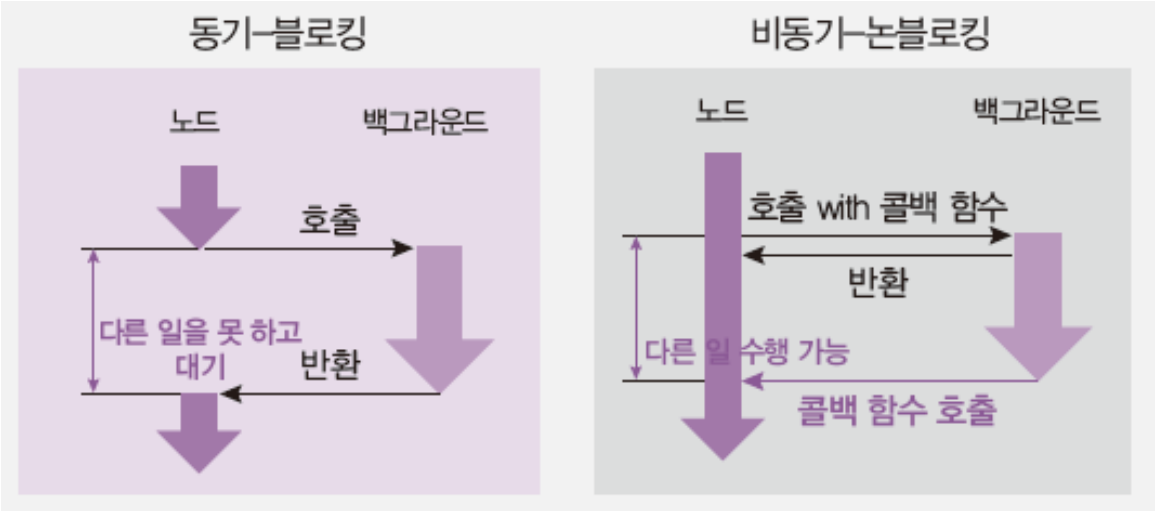
```
const fs = require('fs');

console.log('시작');

fs.readFile('./readme2.txt', 'utf-8', (err, data) => {
  if (err) throw err;
  console.log('1번', data);
});
fs.readFile('./readme2.txt', 'utf-8', (err, data) => {
  if (err) throw err;
  console.log('2번', data);
});
fs.readFile('./readme2.txt', 'utf-8', (err, data) => {
  if (err) throw err;
  console.log('3번', data);
});

console.log('끝');
```

5. 동기 메서드와 비동기 메서드

- 매번 순서가 다르게 실행됨
  - 순서에 맞게 실행하려면?

- 동기와 비동기: 백그라운드 작업 완료 확인 여부
- 블로킹과 논 블로킹: 함수가 바로 return 되는지 여부

- 노드에서는 대부분 동기-블로킹 방식과 비동기-논 블로킹 방식임.



6. 동기 메서드 사용하기

- readFile() -> readFileSync()

```
const fs = require('fs');

console.log('시작');

let data = fs.readFileSync('./readme2.txt', 'utf-8')
console.log('1번', data);

data = fs.readFileSync('./readme2.txt', 'utf-8')
console.log('2번', data);

data = fs.readFileSync('./readme2.txt', 'utf-8')
console.log('3번', data);

console.log('끝');
```

7. 비동기 메서드로 순서 유지하기

- 콜백 형식 유지
  - 코드가 우측으로 너무 들어가는 현상 발생 (콜백 지옥)

```
const fs = require('fs');

console.log('시작');

fs.readFile('./readme2.txt', 'utf-8', (err, data) => {
    if (err) throw err;
    console.log('1번', data);
    fs.readFile('./readme2.txt', 'utf-8', (err, data) => { // 콜백 안에 또 콜백
        if (err) throw err;
        console.log('2번', data);
        fs.readFile('./readme2.txt', 'utf-8', (err, data) => { // 콜백 안에 또 콜백
            if (err) throw err;
            console.log('3번', data);
            console.log('끝');
        })
    })
})
```

8. 비동기 메서드로 순서 유지하기

프로미스로 극복

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

console.log('시작');

fs.readFile('./readme2.txt', 'utf-8')
  .then((data) => {
    console.log('1번', data);
    return fs.readFile('./readme2.txt', 'utf-8');
  })
  .then((data) => {
    console.log('2번', data);
    return fs.readFile('./readme2.txt', 'utf-8');
  })
  .then((data) => {
    console.log('3번', data);
    console.log('끝');
  })
  .catch((err) => {
    console.error(err);
});
```

9. 버퍼와 스트림 이해하기

- 버퍼: 일정한 크기로 모아두는 데이터
  - 일정한 크기가 되면 한 번에 처리
  - 버퍼링: 버퍼에 데이터가 찰 때까지 모으는 작업
- 스트림: 데이터의 흐름
  - 일정한 크기로 나눠서 여러 번에 걸쳐서 처리
  - 버퍼(또는 청크)의 크기를 작게 만들어서 주기적으로 데이터를 전달
  - 스트리밍: 일정한 크기의 데이터를 지속적으로 전달하는 작업

그림 3-12 버퍼

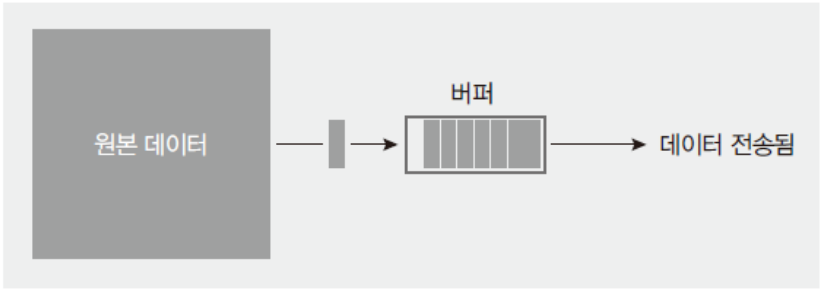
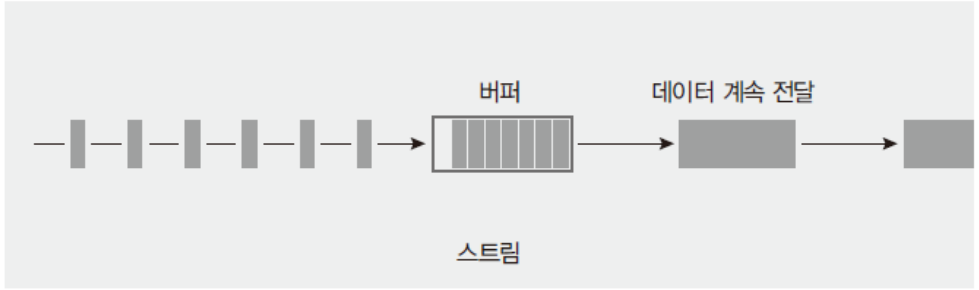


그림 3-13 스트림



10. 버퍼 사용하기

- 노드에서는 Buffer 객체 사용
  - from(문자열): 문자열을 버퍼로 바꿀 수 있습니다. length 속성은 버퍼의 크기를 알려줍니다. 바이트 단위입니다.
  - toString(버퍼): 버퍼를 다시 문자열로 바꿀 수 있습니다. 이때 base64나 hex를 인자로 넣으면 해당 인코딩으로도 변환할 수 있습니다.
  - concat(배열): 배열 안에 든 버퍼들을 하나로 합칩니다.
  - alloc(바이트): 빈 버퍼를 생성합니다. 바이트를 인자로 지정해주면 해당 크기의 버퍼가 생성됩니다.

```
const buffer = Buffer.from('저를 버퍼로 바꿔보세요.')

console.log('from():', buffer)
console.log('length:', buffer.length)
console.log('toString():', buffer.toString())

const array = [Buffer.from('띄엄 '), Buffer.from('띄엄 '), Buffer.from('띄어쓰기')]
const buffer2 = Buffer.concat(array)
console.log('concat():', buffer2.toString())

const buffer3 = Buffer.alloc(5)
console.log('alloc():', buffer3)
```

12. 파일 읽는 스트림 사용하기

fs.createReadStream

- createReadStream에 인자로 파일 경로와 옵션 객체 전달
- highWaterMark 옵션은 버퍼의 크기(바이트 단위, 기본값 64KB)
- data(chunk 전달), end(전달 완료), error(에러 발생) 이벤트 리스너와 같이 사용

```
const fs = require('fs');

const readStream = fs.createReadStream('./readme.txt', { highWaterMark: 16 }); // 16바이트씩 읽음
const data = [];

readStream.on('data', (chunk) => { // 16바이트씩 읽어서 chunk에 담음
  data.push(chunk); // data 배열에 chunk를 넣음
  console.log('data: ', chunk, '길이: ', chunk.length);
});

readStream.on('end', () => {
  console.log('end:', Buffer.concat(data).toString());
});

readStream.on('error', (err) => {
  console.log('error:', err);
});
```

13. 파일 쓰는 스트림 사용하기

- fs.createWriteStream
  - createReadStream에 인자로 파일 경로 전달
  - write로 chunk 입력, end로 스트림 종료
  - 스트림 종료 시 finish 이벤트 발생

```
const fs = require('fs');

const writeStream = fs.createWriteStream('./writeme2.txt');

writeStream.on('finish', () => {
   console.log('파일 쓰기 완료');
});

writeStream.write('이 글을 씁니다.\n');
writeStream.write('한 번 더 씁니다.');
writeStream.end();
```

14. 스트림 사이에 pipe 사용하기

- pipe로 여러 개의 스트림을 이을 수 있음
  - 스트림으로 파일을 복사하는 예제

```
const fs = require('fs');

const readStream = fs.createReadStream('readme.txt', { highWaterMark: 16 });
const writeStream = fs.createWriteStream('writeme3.txt');

readStream.pipe(writeStream);
```

15. 여러 개의 스트림 연결하기

- pipe로 여러 개의 스트림을 이을 수 있음
  - 파일을 압축한 후 복사하는 예제
  - 압축에는 zlib 내장 모듈 사용(createGzip으로 .gz 파일 생성)

```
const fs = require('fs');
const zlib = require('zlib'); // 파일 압축 모듈

const readStream = fs.createReadStream('./readme.txt');
const zlibStream = zlib.createGzip();
const writeStream = fs.createWriteStream('./readme4.txt.gz');

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

16. 큰 파일 만들기

- 1GB 정도의 파일을 만들어 봄.
  - createWriteStream으로 만들어야 메모리 문제가 생기지 않음.

```
const fs = require('fs');

const file = fs.createWriteStream('./big.txt');

for (let i = 0; i <= 10_000_000; i++) {
  file.write('안녕하세요. 엄청나게 큰 파일을 만들어 볼 것입니다. 각오 단단히 하세요!\n');
}

file.end();
```

17. 메모리 체크하기

- 버퍼 방식과 스트림 방식 메모리 사용량을 비교해보기

```
const fs = require('fs');

console.log('before', process.memoryUsage().rss);

const data1 = fs.readFileSync('./big.txt');

fs.writeFileSync('./big2.txt', data1);
console.log('buffer: ', process.memoryUsage().rss);
```

```
const fs = require('fs');

console.log('before', process.memoryUsage().rss);

const readStream = fs.createReadStream('./big.txt');
const writeStream = fs.createWriteStream('./big2.txt');

readStream.pipe(writeStream);
readStream.on('end', () => {
  console.log('stream: ', process.memoryUsage().rss);
});
```

18. 기타 fs 메서드

파일 및 폴더 생성

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

fs.access('./folder', constants.F_OK | constants.W_OK | constants.R_OK)
  .then(() => {
    return Promise.reject('이미 폴더 있음');
  })
  .catch((err) => {
    if (err.code === 'ENOENT') {
      console.log('폴더 없음');
      return fs.mkdir('./folder');
    }
    return Promise.reject(err);
  })
  .then(() => {
    console.log('폴더 만들기 성공');
    return fs.open('./folder/file.js', 'w');
  })
  .then((fd) => {
    console.log('빈 파일 만들기 성공', fd);
    return fs.rename('./folder/file.js', './folder/newfile.js');
  })
  .then(() => {
    console.log('이름 바꾸기 성공');
  })
  .catch((err) => {
    console.error(err);
  });
```

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

fs.stat('./folder')
  .then((stats) => {
    console.log(stats);
  })
  .catch((err) => {
    console.error(err);
  });
```

19. 폴더 내용 확인 및 삭제

- fs.readdir(경로, 콜백): 폴더 안의 내용물을 확인 (배열 안에 내부 파일과 폴더명)
- fs.unlink(경로, 콜백): 파일을 삭제 (파일이 없다면 에러가 발생)
- fs.rmdir(경로, 콜백): 폴더를 삭제 (폴더 안에 파일이 있다면 에러가 발생)

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

fs.readdir('./folder ' )
   .then((dir) => {
      console.log('폴더 내용 확인', dir);
      if (dir[0] == 'newfile.js') {
         return fs.unlink('./folder/' + dir[0]);
      }
   })
   .then(() => {
      console.log('파일 삭제 성공 ' );
      return fs.rmdir('./folder');
   })
   .then(() => {
      console.log('폴더 삭제 성공 ' );
   })
   .catch((err) => {
      console.error(err);
   });
```

20. 기타 fs 메서드

- 파일을 복사

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

fs.copyFile('readme.txt', 'copy.txt')
    .then(() => {
        console.log('복사 완료');
    })
    .catch((error) => {
        console.error(error);
    });
```

21. 기타 fs 메서드

- 파일을 감시하는 방법 (변경 사항 발생 시 이벤트 호출)

```
const fs = require('fs');

fs.writeFile('target.txt', '', (err) => {
  if (err) {
    console.error('파일 생성 중 오류 발생:', err);
  } else {
    console.log('target.txt 파일을 변경하거나 삭제하세요.');
  }
});

fs.watch('./target.txt', (eventType, filename) => {
  console.log(eventType, filename);
});
```

22. 스레드풀 알아보기

- fs, crypto, zlib, dns, lookup 모듈의 메서드를 실행할 때는 백그라운드에서 동시에 실행됨
  - 스레드풀이 동시에 처리해줌

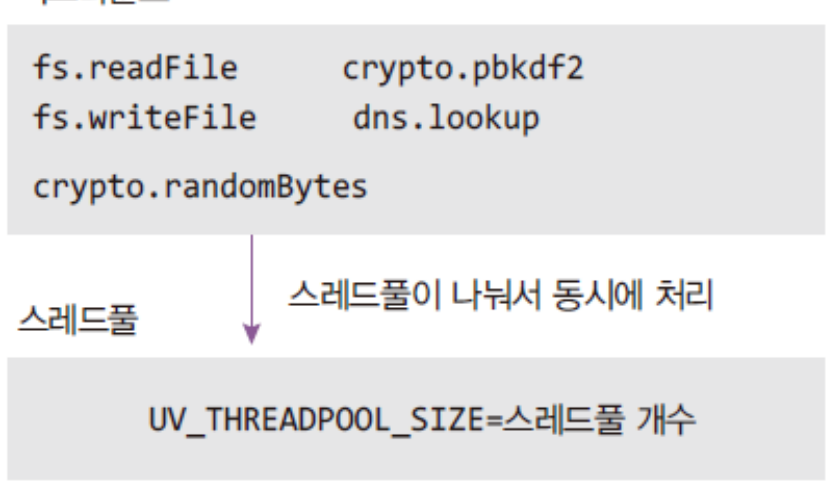
```
const crypto = require('crypto');

const pass = 'pass';
const salt = 'salt';
const start = Date.now();

for (let i = 0; i < 8; i++) {
   crypto.pbkdf2(pass, salt, 1000000, 512, 'sha512', () => {
      console.log('${i}: ', Date.now() - start);
   });
}
```

23. UV_THREAD_SIZE

- 스레드풀을 직접 컨트롤할 수는 없지만 개수 조절은 가능
  - 윈도우 : set UV_THREADPOOL_SIZE=4
  - 맥, 리눅스 : export UV_THREADPOOL_SIZE=4

  - 스레드풀 개수를 바꾼 뒤 재실행해보기

그림 3-14 스레드풀 개수만큼 작업을 동시에 처리합니다.



2.7 이벤트 이해하기

1. 이벤트 만들고 호출하기

- events 모듈로 커스텀 이벤트를 만들 수 있음
  - on(이벤트명, 콜백), addListener(이벤트명, 콜백) : 이벤트 이름과 이벤트 발생 시의 콜백을 연결
  - emit(이벤트명): 이벤트 호출
  - once(이벤트명, 콜백): 한 번만 실행되는 이벤트
  - removeAllListeners(이벤트명): 이벤트에 연결된 모든 이벤트 리스너를 제거
  - off(이벤트명, 콜백), removeListener(이벤트명, 리스너): 이벤트에 연결된 리스너를 하나씩 제거
  - listenerCount(이벤트명): 현재 리스너가 몇 개 연결되어 있는지 확인

2. 커스텀 이벤트 예제

```
const EventEmitter = require('events’);
const myEvent = new EventEmitter();

myEvent.addListener('event1', () => console.log('첫번째 이벤트'));
myEvent.emit('event1');

myEvent.on('event2', () => console.log('두번째 이벤트'));
myEvent.emit('event2');
myEvent.on('event2', () => console.log('+ 두번째 이벤트에 추가'));
myEvent.emit('event2');

console.log(myEvent.listenerCount('event2'));

myEvent.once('event3', () => console.log('세번째 이벤트는 한번만 실행됨'));
myEvent.emit('event3');
myEvent.emit('event3');

myEvent.on('event4', () => console.log('네번째 이벤트'));
myEvent.emit('event4');
myEvent.removeAllListeners('event4');
myEvent.emit('event4');

const listener5 = () => console.log('이벤트 5');
myEvent.addListener('event5', listener5);
myEvent.removeListener('event5', listener5);
myEvent.emit('event5');

const listener6 = () => console.log('이벤트 6');
myEvent.on('event6', listener6);
myEvent.off('event6', listener6);
myEvent.emit('event6');
```

2.8 예외 처리하기

1. 예외 처리

- 예외(Exception): 처리하지 못한 에러
  - 노드 프로세스/스레드를 멈춤
  - 노드는 기본적으로 싱글 스레드처럼 작동하기 때문에, 스레드가 멈춘다는 것은 프로세스가 멈추는 것
  - 에러 처리는 필수

- 기본적으로 try catch문으로 예외를 처리
  - 에러가 발생할 만한 곳을 try-catch로 감쌈

```
setInterval(() => {
  console.log('시작');
  try {
    throw new Error('서버를 고장내주마!');
  } catch (err) {
    console.error(err);
  }
}, 1000);
```

2. 노드 비동기 메서드의 에러

- 노드 비동기 메서드의 에러는 따로 처리하지 않아도 됨
  - 콜백 함수에서 에러 객체 제공

```
const fs = require('fs');

setInterval(() => {
  fs.unlink('./abcdefg.js', (err) => {
    if (err) console.error(err);
  });
}, 1000);
```

3. 노드 비동기 메서드의 에러

- 프로미스의 에러는 반드시 catch!

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

setInterval(() => {
   fs.unlink('./abcdefg.js’)
      .then(() => {
         console.log('success’);
      }).catch((err) => {
         console.error(err);
      });
}, 1000);
```

4. 예측 불가능한 에러 처리

- 최후의 수단으로 사용
  - 에러 내용 기록 용으로만 쓰는 게 좋음

```
process.on('uncaughtException', function(err) {
    console.error('예기치 못한 에러', err);
});

setInterval(function() {
    throw new Error('서버를 고장내주마!');
}, 1000);

setTimeout(function() {
    console.log('실행됩니다.');
}, 2000);
```

5. 자주 발생하는 에러들

- node : command not found → 노드를 설치하지 않았거나, 설치하였다고 해도 환경 변수가 제대로 설정되어 있지 않은 경우
- ReferenceError : **모듈** is not defined → 모듈을 require 또는 import 하지 않은 경우
- Error : Cannot find module **모듈** → 해당 모듈을 require 또는 import를 했으나, 설치하지 않은 경우
- Error [ERR_MODULE_NOT_FOUND] → 존재하지 않는 모듈을 불러오려는 경우
- Error : Can't set headers after they are sent → 요청에 대한 응답 메서드가 두 번 이상 사용된 경우
- FATAL ERROR : CALL_AND_RETRY_LAST Allocation failed-JavaScript heap out of memory → 코드를 실행할 때 메모리가 부족해서 스크립트가 정상적으로 작동하지 않는 경우
- UnhandledPromiseRejectionWarning : Unhandled promise rejection → 프로미스 사용에 catch를 빠뜨린 경우
- EADDRINUSE **포트번호** → 해당 포트 번호에 이미 다른 프로세스가 연결되어 있는 경우
- EACCES **또는** EPERM → 노드가 작업을 수행하는 데 권한이 충분하지 않은 경우
- EJSONPARSE → package.json 등의 JSON 파일에 문법 오류가 있는 경우
- ECONNREFUSED → 요청을 보냈으나 연결이 성립하지 않는 경우
- ETARGET → package.json에 기록한 패키지 버전이 존재하지 않는 경우
- ETIMEOUT → 요청을 보냈으나 응답이 시간 내에 오지 않은 경우
- ENOENT : no such file or directory → 지정한 폴더나 파일이 존재하지 않는 경우 (맥이나 리눅스에서는 대소문자도 구분한다)

6. 프로세스 종료하기

- 윈도우

```
netstat -ano | findstr 포트번호
taskkill /pid 프로세스아이디 /f
```

- 맥 / 리눅스

```
losf -i tcp:포트번호
kill -9 프로세스아이디
```