Node.js 교과서 개정2판

길벗

3장

3.1 REPL 사용하기

1. REPL

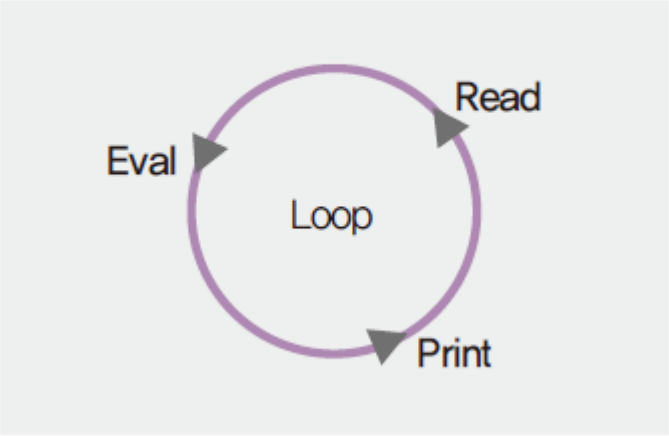
» 자바스크립트는 스크립트 언어라서 즉석에서 코드를 실행할 수 있음

- 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 결괏값이 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
>
```

3.2 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 결괏값이 출력됨

helloWorld.js

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

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

helloWorld();
```

콘솔

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

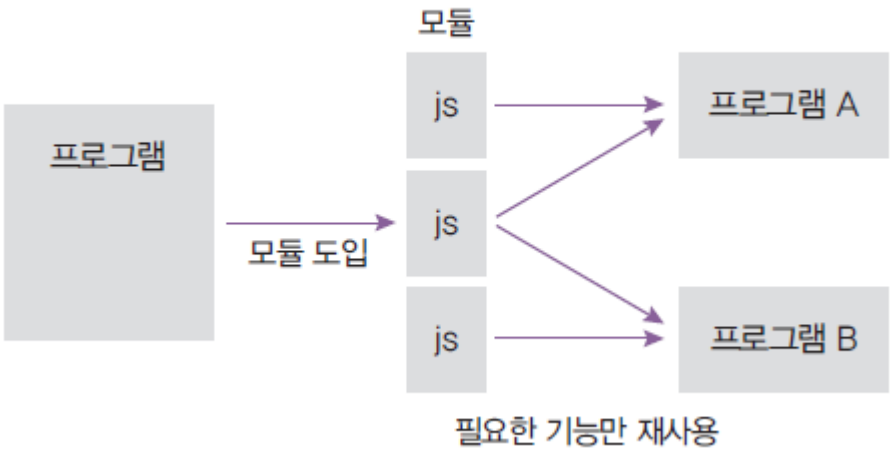
3.3 모듈로 만들기

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

var.js

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

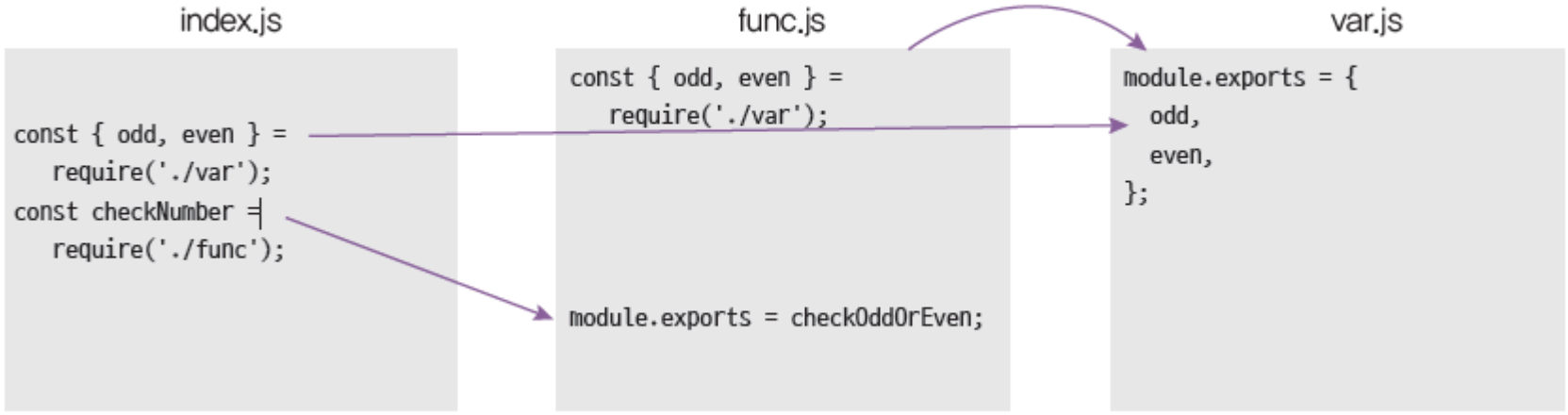
```
const { odd, even } = require('./var');
const checkNumber = require('./func');

function checkStringOddOrEven(str) {
  if (str.length % 2) { // 홀수면
    return odd;
  }
  return even;
}

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. ES2015 모듈

» 자바스크립트 자체 모듈 시스템 문법이 생김

- 아직 노드에서의 지원은 완벽하지 않음. mjs 확장자를 사용해야 함.
- 크게는 require 대신 import, module.exports 대신 export default를 쓰는 것으로 바뀜

func.mjs

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

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

export default checkOddOrEven;
```

3.4 노드 내장 객체 알아보기

1. global

노드의 전역 객체

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

콘솔

```
$ node
> global
{
  global: [Circular *1],
  clearInterval: [Function: clearInterval],
  clearTimeout: [Function: clearTimeout],
  ...
}
> global.console
{
  log: [Function: bound consoleCall],
  warn: [Function: bound consoleCall],
  dir: [Function: bound consoleCall],
  ...
}
```

2. global 속성 공유

» global 속성에 값을 대입하면 다른 파일에서도 사용 가능

globalA.js

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

globalB.js

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

global.message = '안녕하세요';
console.log(A());
```

콘솔

```
$ node globalB
안녕하세요
```

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

console.js

```
const string = 'abc';
const number = 1;
const boolean = true;
const obj = {
  outside: {
    inside: {
      key: 'value',
    },
  },
};
console.time('전체 시간');
console.log('평범한 로그입니다 쉼표로 구분해 여러 값을 찍을 수 있습니다');
console.log(string, number, boolean);
console.error('에러 메시지는 console.error에 담아주세요');

console.table([{ name: '제로', birth: 1994 }, { name: 'hero', birth: 1988}]);

console.dir(obj, { colors: false, depth: 2 });
console.dir(obj, { colors: true, depth: 1 });

console.time('시간 측정');
for (let i = 0; i < 100000; i++) {}
console.timeEnd('시간 측정');

function b() {
  console.trace('에러 위치 추적');
}
function a() {
  b();
}
a();

console.timeEnd('전체 시간');
```

4. console 예제 실행하기

node console로 실행

```
평범한 로그입니다 쉼표로 구분해 여러 값을 찍을 수 있습니다
abc 1 true
에러 메시지는 console.error에 담아주세요
┌─────────┬──────────┬──────────┐
│ (index) │   name   │  birth   │
├─────────┼──────────┼──────────┤
│    0    │ '제로'   │   1994   │
│    1    │ 'hero'   │   1988   │
└─────────┴──────────┴──────────┘
{ outside: { inside: { key: 'value' } } }
{ outside: { inside: [Object] } }
시간 측정: 1.017ms
Trace: 에러 위치 추적
    at b (C:\Users\zerocho\console.js:26:11)
    at a (C:\Users\zerocho\console.js:29:3)
    at Object.<anonymous> (C:\Users\zerocho\console.js:31:1)
전체 시간: 5.382ms
```

5. 타이머 메서드

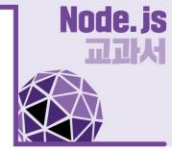
» set 메서드에 clear 메서드가 대응됨

- 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.

6. 타이머 예제



» 다음 예제의 콘솔 출력을 맞춰보자

- setTimeout(콜백, 0)보다 setImmediate 권장

timer.js

```
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);

setTimeout(() => {
  clearTimeout(timeout2);
  clearInterval(interval);
}, 2500);
const immediate = setImmediate(() => {
  console.log('즉시 실행');
});

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

clearImmediate(immediate2);
```

6. 타이머 예제 결과

콘솔

```
$ node timer
즉시 실행
1초마다 실행
1.5초 후 실행
1초마다 실행
```

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~~	

7. __filename, __dirname

- __filename: 현재 파일 경로
- __dirname: 현재 폴더(디렉터리) 경로

filename.js

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

콘솔

```
$ node filename.js
C:\Users\zerocho\filename.js
C:\Users\zerocho
```

8. module, exports

» module.exports 외에도 exports로 모듈을 만들 수 있음

- 모듈 예제의 var.js를 다음과 같이 바꾼 후 실행
- 동일하게 동작함
- 동일한 이유는 module.exports와 exports가 참조 관계이기 때문
- exports에 객체의 속성이 아닌 다른 값을 대입하면 참조 관계가 깨짐

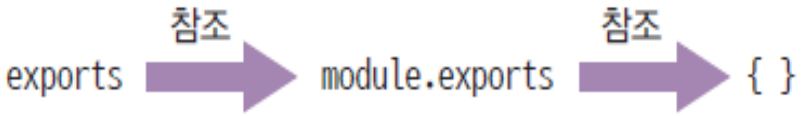
var.js

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

콘솔

```
$ node index
짝수입니다
홀수입니다
```

그림 3-5 exports와 module.exports의 관계



9. 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

this.js

```
console.log(this);
console.log(this === module.exports);
console.log(this === exports)

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

콘솔

```
$ node this
{}
true
true
function false true
```

10. require의 특성

» 몇 가지 알아둘 만한 속성이 있음

- require가 제일 위에 올 필요는 없음
- require.cache에 한 번 require한 모듈에 대한 캐싱 정보가 들어있음.
- require.main은 노드 실행 시 첫 모듈을 가리킴.

require.js

```
console.log('require가 가장 위에 오지 않아도 됩니다.');

module.exports = '저를 찾아보세요.';

require('./var');

console.log('require.cache입니다.');
console.log(require.cache);
console.log('require.main입니다.');
console.log(require.main === module);
console.log(require.main.filename);
```

11. 순환참조

Node.js 교과서

» 두 개의 모듈이 서로를 require하는 상황을 조심해야 함

- Dep1이 dep2를 require하고, dep2가 dep1을 require 함.
- Dep1의 module.exports가 함수가 아니라 빈 객체가 됨(무한 반복을 막기 위해 의도됨)
- 순환참조하는 상황이 나오지 않도록 하는 게 좋음

dep1.js

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

dep2.js

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

dep-run.js

```
const dep1 = require('./dep1');
const dep2 = require('./dep2');

dep1();
dep2();
```

콘솔

```
$ node dep-run
require dep1 {}
require dep2 [Function (anonymous)]
dep2 [Function (anonymous)]
dep1 {}
```

12. process

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

콘솔

```
$ node
> process.version
v14.0.0 // 설치된 노드의 버전입니다.
> process.arch
x64 // 프로세서 아키텍처 정보입니다. arm, ia32 등의 값일 수도 있습니다.
> process.platform
win32 // 운영체제 플랫폼 정보입니다. linux나 darwin, freebsd 등의 값일 수도 있습니다.
> process.pid
14736 // 현재 프로세스의 아이디입니다. 프로세스를 여러 개 가질 때 구분할 수 있습니다.
> process.uptime()
199.36 // 프로세스가 시작된 후 흐른 시간입니다. 단위는 초입니다.
> process.execPath
C:\\Program Files\\nodejs\\node.exe // 노드의 경로입니다.
> process.cwd()
C:\\Users\\zerocho // 현재 프로세스가 실행되는 위치입니다.
> process.cpuUsage()
{ user: 390000, system: 203000 } // 현재 cpu 사용량입니다.
```

13. process.env

- 시스템 환경 변수들이 들어있는 객체
  - 비밀키(데이터베이스 비밀번호, 서드파티 앱 키 등)를 보관하는 용도로도 쓰임
  - 환경 변수는 process.env로 접근 가능

```
const secretId = process.env.SECRET_ID;
const secretCode = process.env.SECRET_CODE;
```

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

```
NODE_OPTIONS=--max-old-space-size=8192
UV_THREADPOOL_SIZE=8
```

14. process.nextTick(콜백)

» 이벤트 루프가 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

nextTick.js

```
setImmediate(() => {
  console.log('immediate');
});
process.nextTick(() => {
  console.log('nextTick');
});
setTimeout(() => {
  console.log('timeout');
}, 0);
Promise.resolve().then(() => console.log('promise'));
```

콘솔

```
$ node nextTick
nextTick
promise
timeout
immediate
```

15. process.exit(코드)

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

exit.js

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

콘솔

```
$ node exit
1
2
3
4
종료!
```

3.5 노드 내장 모듈 사용하기

1. os

운영체제의 정보를 담고 있음

- 모듈은 require로 가져옴(내장 모듈이라 경로 대신 이름만 적어줘도 됨)

os.js

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

console.log('운영체제 정보----------------------------------');
console.log('os.arch():', os.arch());
console.log('os.platform():', os.platform());
console.log('os.type():', os.type());
console.log('os.uptime():', os.uptime());
console.log('os.hostname():', os.hostname());
console.log('os.release():', os.release());

console.log('경로----------------------------------------');
console.log('os.homedir():', os.homedir());
console.log('os.tmpdir():', os.tmpdir());

console.log('cpu 정보----------------------------------');
console.log('os.cpus():', os.cpus());
console.log('os.cpus().length:', os.cpus().length);
console.log('메모리 정보----------------------------------');
console.log('os.freemem():', os.freemem());
console.log('os.totalmem():', os.totalmem());
```

콘솔

```
$ node os
운영체제 정보----------------------------------
os.arch(): x64
os.platform(): win32
os.type(): Windows_NT
os.uptime(): 53354
os.hostname(): DESKTOP-RRANDNC
os.release(): 10.0.18362
경로----------------------------------
os.homedir(): C:\Users\zerocho
os.tmpdir(): C:\Users\zerocho\AppData\Local\Temp
cpu 정보----------------------------------
os.cpus(): [ { model: 'Intel(R) Core(TM) i5-9400F CPU @ 2.90GHz',
    speed: 2904,
    times: { user: 970250, nice: 0, sys: 1471906, idle: 9117578, irq: 359109 } },
    // 다른 코어가 있다면 나머지 코어의 정보가 나옴
 ]
os.cpus().length: 6
메모리 정보----------------------------------
os.freemem(): 23378612224
os.totalmem(): 34281246720
```

2. os 모듈 메서드

- os.arch(): process.arch와 동일합니다.
- os.platform(): process.platform과 동일합니다.
- os.type(): 운영체제의 종류를 보여줍니다.
- os.uptime(): 운영체제 부팅 이후 흐른 시간(초)을 보여줍니다. process.uptime()은 노드의 실행 시간이었습니다.
- os.hostname(): 컴퓨터의 이름을 보여줍니다.
- os.release(): 운영체제의 버전을 보여줍니다.
- os.homedir(): 홈 디렉터리 경로를 보여줍니다.
- os.tmpdir(): 임시 파일 저장 경로를 보여줍니다.
- os.cpus(): 컴퓨터의 코어 정보를 보여줍니다.
- os.freemem(): 사용 가능한 메모리(RAM)를 보여줍니다.
- os.totalmem(): 전체 메모리 용량을 보여줍니다.

3. path

- 폴더와 파일의 경로를 쉽게 조작하도록 도와주는 모듈
  - 운영체제별로 경로 구분자가 다름(Windows: '₩', POSIX: '/')

path.js

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

const string = __filename;

console.log('path.sep:', path.sep);
console.log('path.delimiter:', path.delimiter);
console.log('------------------------------');
console.log('path.dirname():', path.dirname(string));
console.log('path.extname():', path.extname(string));
console.log('path.basename():', path.basename(string));
console.log('path.basename - extname:', path.basename(string, path.extname(string)));
console.log('------------------------------');
console.log('path.parse()', path.parse(string));
console.log('path.format():', path.format({
  dir: 'C:\\users\\zerocho',
  name: 'path',
  ext: '.js',
}));
console.log('path.normalize():', path.normalize('C://users\\\\zerocho\\\\path.js'));
console.log('------------------------------');
console.log('path.isAbsolute(C:\\):', path.isAbsolute('C:\\'));
console.log('path.isAbsolute(./home):', path.isAbsolute('./home'));
console.log('------------------------------');
console.log('path.relative():', path.relative('C:\\users\\zerocho\\path.js', 'C:\\'));
console.log('path.join():', path.join(__dirname, '..', '..', '/users', '.',
➥ '/zerocho'));
console.log('path.resolve():', path.resolve(__dirname, '..', 'users', '.',
➥ '/zerocho'));
```

콘솔

```
$ node path
path.sep: \
path.delimiter: ;
------------------------------
path.dirname(): C:\Users\zerocho
path.extname(): .js
path.basename(): path.js
path.basename - extname: path
------------------------------
path.parse() {
  root: 'C:\\',
  dir: 'C:\\Users\\zerocho',
  base: 'path.js',
  ext: '.js',
  name: 'path'
}
path.format(): C:\users\zerocho\path.js
path.normalize(): C:\users\zerocho\path.js
------------------------------
path.isAbsolute(C:\\): true
path.isAbsolute(./home): false
------------------------------
path.relative(): ..\..\..
path.join(): C:\Users\zerocho
path.resolve(): C:\zerocho
```

4. path 모듈 메서드

- path.sep: 경로의 구분자입니다. Windows는 ₩, POSIX는 /입니다.
- path.delimiter: 환경 변수의 구분자입니다. process.env.PATH를 입력하면 여러 개의 경로가 이 구분자로 구분되어 있습니다. Windows는 세미콜론(;)이고 POSIX는 콜론(:)입니다.
- path.dirname(경로): 파일이 위치한 폴더 경로를 보여줍니다.
- path.extname(경로): 파일의 확장자를 보여줍니다.
- path.basename(경로, 확장자): 파일의 이름(확장자 포함)을 보여줍니다. 파일의 이름만 표시하고 싶다면 basename의 두 번째 인자로 파일의 확장자를 넣어주면 됩니다.
- path.parse(경로): 파일 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 변환해줍니다.
- path.isAbsolute(경로): 파일의 경로가 절대경로인지 상대경로인지 true나 false로 알려줍니다.
- path.relative(기준경로, 비교경로): 경로를 두 개 넣으면 첫 번째 경로에서 두 번째 경로로 가는 방법을 알려줍니다.
- path.join(경로, .. .): 여러 인자를 넣으면 하나의 경로로 합쳐줍니다. 상대경로인 ..(부모 디렉터리)과 .(현 위치)도 알아서 처리해줍니다.
- path.resolve(경로, .. .): path.join()과 비슷하지만 차이가 있습니다. 차이점은 다음에 나오는 Note에서 설명합니다.

5. 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(..)
  -
```
path.join('/a', '/b', 'c'); /* 결과: /a/b/c/ */
path.resolve('/a', '/b', 'c'); /* 결과: /b/c */
```
가 실행되는 위치가 기준

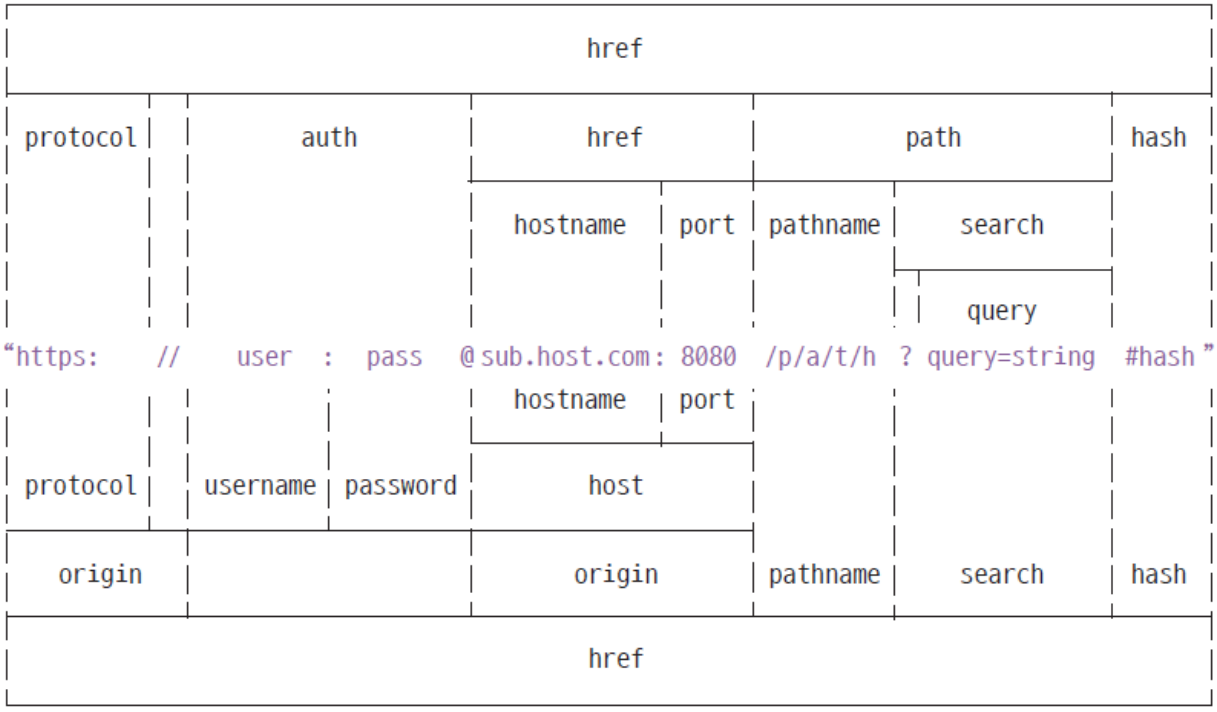
- ₩₩와 ₩ 차이: ₩는 윈도 경로 구분자, ₩₩는 자바스크립트 문자열 안에서 사용(₩가 특수문자라 ₩₩로 이스케이프 해준 것)

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

6. url 모듈

- 인터넷 주소를 쉽게 조작하도록 도와주는 모듈
  - url 처리에 크게 두 가지 방식이 있음(기존 노드 방식 vs WHATWG 방식
  - 아래 그림에서 가운데 주소를 기준으로 위쪽은 기존 노드 방식, 아래쪽은 WHATWG 방식

그림 3-7 WHATWG와 노드의 주소 체계



7. url 모듈 예제

url.js

```
const url = require('url');                                          ------1

const { URL } = url;
const myURL = new URL('http://www.gilbut.co.kr/book/bookList.aspx?sercate1=001001000#
anchor');
console.log('new URL():', myURL);
console.log('url.format():', url.format(myURL));
console.log('------------------------------');
const parsedUrl = url.parse('http://www.gilbut.co.kr/book/bookList.aspx?sercate1=001001
000#anchor');
console.log('url.parse():', parsedUrl);
console.log('url.format():', url.format(parsedUrl));
```

1 url 모듈 안에 URL 생성자가 있습니다. 이 생성자에 주소를 넣어 객체로 만들면 주소가 부분별로 정리됩니다. 이 방식이 WHATWG의 url입니다. WHATWG에만 있는 username, password, origin, searchParams 속성이 존재합니다.

8. url 모듈 예제 결과

콘솔

```
$ node url
new URL(): URL {
  href: 'http://www.gilbut.co.kr/book/bookList.aspx?sercate1=001001000#anchor',
  origin: 'http://www.gilbut.co.kr',
  protocol: 'http:',
  username: '',
  password: '',
  host: 'www.gilbut.co.kr',
  hostname: 'www.gilbut.co.kr',
  port: '',
  pathname: '/book/bookList.aspx',
  search: '?sercate1=001001000',
  searchParams: URLSearchParams { 'sercate1' => '001001000' },
  hash: '#anchor' }
url.format(): http://www.gilbut.co.kr/book/bookList.aspx?sercate1=001001000#anchor
------------------------------
url.parse(): Url {
  protocol: 'http:',
  slashes: true,
  auth: null,
  host: 'www.gilbut.co.kr',
  port: null,
  hostname: 'www.gilbut.co.kr',
  hash: '#anchor',
  search: '?sercate1=001001000',
  query: 'sercate1=001001000',
  pathname: '/book/bookList.aspx',
  path: '/book/bookList.aspx?sercate1=001001000',
  href: 'http://www.gilbut.co.kr/book/bookList.aspx?sercate1=001001000#anchor' }
url.format(): http://www.gilbut.co.kr/book/bookList.aspx?sercate1=001001000#anchor
```

9. url 모듈 메서드

- 기존 노드 방식 메서드
  - url.parse(주소): 주소를 분해합니다. WHATWG 방식과 비교하면 username과 password대신 auth 속성이 있고, searchParams 대신 query가 있습니다.
  - url.format(객체): WHATWG 방식의 url과 기존 노드의 url 모두 사용할 수 있습니다. 분해되었던 url 객체를 다시 원래 상태로 조립합니다.

10. searchParams

- WHATWG 방식에서 쿼리스트링(search) 부분 처리를 도와주는 객체
  - ?page=3&limit=10&category=nodejs&category=javascript 부분

searchParams.js

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

const myURL = new URL('http://www.gilbut.co.kr/?page=3&limit=10&category=nodejs&categor
y=javascript');
console.log('searchParams:', myURL.searchParams);
console.log('searchParams.getAll():', myURL.searchParams.getAll('category'));
console.log('searchParams.get():', myURL.searchParams.get('limit'));
console.log('searchParams.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();
```

11. searchParams 예제 결과

- getAll(키): 키에 해당하는 모든 값들을 가져옵니다. category 키에는 두 가지 값, 즉 nodejs와 javascript의 값이 들어 있습니다.
- get(키): 키에 해당하는 첫 번째 값만 가져옵니다.
- has(키): 해당 키가 있는지 없는지를 검사합니다.
- keys(): searchParams의 모든 키를 반복기(iterator, ES2015 문법) 객체로 가져옵니다.
- values(): searchParams의 모든 값을 반복기 객체로 가져옵니다.
- append(키, 값): 해당 키를 추가합니다. 같은 키의 값이 있다면 유지하고 하나 더 추가합니다.
- set(키, 값): append와 비슷하지만 같은 키의 값들을 모두 지우고 새로 추가합니다.
- delete(키): 해당 키를 제거합니다.
- toString(): 조작한 searchParams 객체를 다시 문자열로 만듭니다. 이 문자열을 search에 대입하면 주소 객체에

콘솔

```
$ node searchParams
searchParams: URLSearchParams {
  'page' => '3',
  'limit' => '10',
  'category' => 'nodejs',
  'category' => 'javascript' }
searchParams.getAll(): [ 'nodejs', 'javascript' ]
searchParams.get(): 10
searchParams.has(): true
searchParams.keys(): URLSearchParams Iterator { 'page', 'limit', 'category', 'category'
➥ }
searchParams.values(): URLSearchParams Iterator { '3', '10', 'nodejs', 'javascript' }
[ 'es3', 'es5' ]
[ 'es6' ]
[]
searchParams.toString(): page=3&limit=10&category=nodejs&category=javascript
```

12. querystring

» 기존 노드 방식에서는 url querystring을 querystring 모듈로 처리

- querystring.parse(쿼리): url의 query 부분을 자바스크립트 객체로 분해해줍니다.
- querystring.stringify(객체): 분해된 query 객체를 문자열로 다시 조립해줍니다.

querystring.js

```
const url = require('url');
const querystring = require('querystring');

const parsedUrl = url.parse('http://www.gilbut.co.kr/?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));
```

콘솔

```
$ node querystring
querystring.parse(): [Object: null prototype] {
  page: '3',
  limit: '10',
  category: [ 'nodejs', 'javascript' ]
}
querystring.stringify(): page=3&limit=10&category=nodejs&category=javascript
```

13. 단방향 암호화(crypto)

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

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

그림 3-8 해시 함수

14. Hash 사용하기(sha512)

- createHash(알고리즘): 사용할 해시 알고리즘을 넣어줍니다.
  - md5, sha1, sha256, sha512 등이 가능하지만, md5와 sha1은 이미 취약점이 발견되었습니다.
  - 현재는 sha512 정도로 충분하지만, 나중에 sha512마저도 취약해지면 더 강화된 알고리즘으로 바꿔야 합니다.
- update(문자열): 변환할 문자열을 넣어줍니다.
- digest(인코딩): 인코딩할 알고리즘을 넣어줍니다.
  - base64, hex, latin1이 주로 사용되는데, 그중 base64가 결과 문자열이 가장 짧아 애용됩니다. 결과물로

querystring.js

```
const url = require('url');
const querystring = require('querystring');

const parsedUrl = url.parse('http://www.gilbut.co.kr/?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));
```

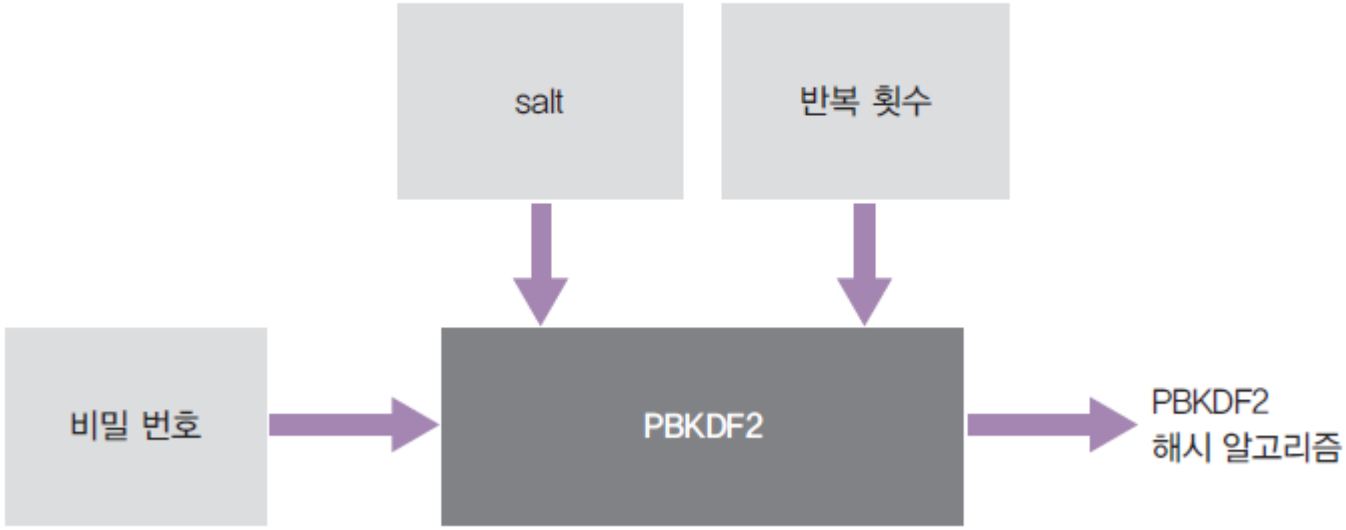
콘솔

```
$ node querystring
querystring.parse(): [Object: null prototype] {
  page: '3',
  limit: '10',
  category: [ 'nodejs', 'javascript' ]
}
querystring.stringify(): page=3&limit=10&category=nodejs&category=javascript
```

14. pbkdf2

- 컴퓨터의 발달로 기존 암호화 알고리즘이 위협받고 있음
  - sha512가 취약해지면 sha3으로 넘어가야함
  - 현재는 pbkdf2나, bcrypt, scrypt 알고리즘으로 비밀번호를 암호화
  - Node는 pbkdf2와 scrypt 지원

그림 3-9 pbkdf2



15. 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

pbkdf2.js

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

crypto.randomBytes(64, (err, buf) => {
  const salt = buf.toString('base64');
  console.log('salt:', salt);
  crypto.pbkdf2('비밀번호', salt, 100000, 64, 'sha512', (err, key) => {
    console.log('password:', key.toString('base64'));
  });
});
```

콘솔

```
$ node pbkdf2
salt: OnesIj8wznyKgHva1fmulYAgjf/OGLmJnwfy8pIABchHZF/Wn2AM2Cn/9170Y1AdehmJ0E5CzLZULps+da
F6rA==
password: b4/FpSrZulVY28trzNXsl4vVfhOKBPxyVAvwnUCWvF1nnXS1zsU1Paq2p68VwUfhB0LDD44hJOf+tL
e3HMLVmQ==
```

16. 양방향 암호화

» 대칭형 암호화(암호문 복호화 가능)

- Key가 사용됨
- 암호화할 때와 복호화 할 때 같은 Key를 사용해야 함

cipher.js

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

const algorithm = 'aes-256-cbc';
const key = 'abcdefghijklmnopqrstuvwxyz123456';
const iv = '1234567890123456';
const cipher = crypto.createCipheriv(algorithm, key, iv);
let result = cipher.update('암호화할 문장', 'utf8', 'base64');
result += cipher.final('base64');
console.log('암호화:', result);

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

17. 양방향 암호화 메서드

- crypto.createCipheriv(알고리즘, 키, iv): 암호화 알고리즘과 키, 초기화벡터를 넣어줍니다.
  - 암호화 알고리즘은 aes-256-cbc를 사용했습니다. 다른 알고리즘을 사용해도 됩니다.
  - 사용 가능한 알고리즘 목록은 crypto.getCiphers()를 하면 볼 수 있습니다.
  - 키는 32바이트, 초기화벡터(iv)는 16바이트로 고정입니다.
- cipher.update(문자열, 인코딩, 출력 인코딩): 암호화할 대상과 대상의 인코딩, 출력 결과물의 인코딩을 넣어줍니다.
  - 보통 문자열은 utf8 인코딩을, 암호는 base64를 많이 사용합니다.
- cipher.final(출력 인코딩): 출력 결과물의 인코딩을 넣어주면 암호화가 완료됩니다.
- crypto.createDecipheriv(알고리즘, 키, iv): 복호화할 때 사용합니다. 암호화할 때 사용했던 알고리즘과 키, iv를 그대로 넣어주어야 합니다.
- decipher.update(문자열, 인코딩, 출력 인코딩): 암호화된 문장, 그 문장의 인코딩, 복호화할 인코딩을 넣어줍니다.
  - createCipher의 update()에서 utf8, base64 순으로 넣었다면 createDecipher의 update()에서는 base64, utf8 순으로 넣으면 됩니다.
- decipher.final(출력 인코딩): 복호화 결과물의 인코딩을 넣어줍니다.

18. util

각종 편의 기능을 모아둔 모듈

- deprecated와 promisify가 자주 쓰임

util.js

```
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.error(error);
  });
```

콘솔

```
$ node util
3
(node:7264) DeprecationWarning: dontUseMe 함수는 deprecated되었으니 더 이상 사용하지 마세요!
(Use `node --trace-deprecation ...` to show where the warning was created)
60b4RQbrx1j130x4r95fpZac9lmcHyitqwAm8gKsHQKF8tcNhvcTfW031XaQqHlRKzaVkcENmIV25fDVs3SB
7g==
```

19. util의 메서드

- util.deprecate: 함수가 deprecated 처리되었음을 알려줍니다.
  - 첫 번째 인자로 넣은 함수를 사용했을 때 경고 메시지가 출력됩니다.
  - 두 번째 인자로 경고 메시지 내용을 넣으면 됩니다. 함수가 조만간 사라지거나 변경될 때 알려줄 수 있어 유용합니다.
- util.promisify: 콜백 패턴을 프로미스 패턴으로 바꿔줍니다.
  - 바꿀 함수를 인자로 제공하면 됩니다. 이렇게 바꾸어두면 async/await 패턴까지 사용할 수 있어 좋습니다.
  - 3.5.5.1절의 randomBytes와 비교해보세요. 프로미스를 콜백으로 바꾸는 util.callbackify도 있지만 자주 사용되지는 않습니다.

Note ≡ deprecated이란?

deprecated는 프로그래밍 용어로, '중요도가 떨어져 더 이상 사용되지 않고 앞으로는 사라지게 될' 것이라는 뜻입니다. 새로운 기능이 나와서 기존 기능보다 더 좋을 때, 기존 기능을 deprecated 처리하곤 합니다. 이전 사용자를 위해 기능을 제거하지는 않지만 곧 없앨 예정이므로 더 이상 사용하지 말라는 의미입니다.

20. worker_threads

Node.js 교과서

- 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

worker_threads.js

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

if (isMainThread) { // 부모일 때
  const worker = new Worker(__filename);
  worker.on('message', message => console.log('from worker', message));
  worker.on('exit', () => console.log('worker exit'));
  worker.postMessage('ping');
} else { // 워커일 때
  parentPort.on('message', (value) => {
    console.log('from parent', value);
    parentPort.postMessage('pong');
    parentPort.close();
  });
}
```

콘솔

```
$ node worker_threads
from parent ping
from worker pong
worker exit
```

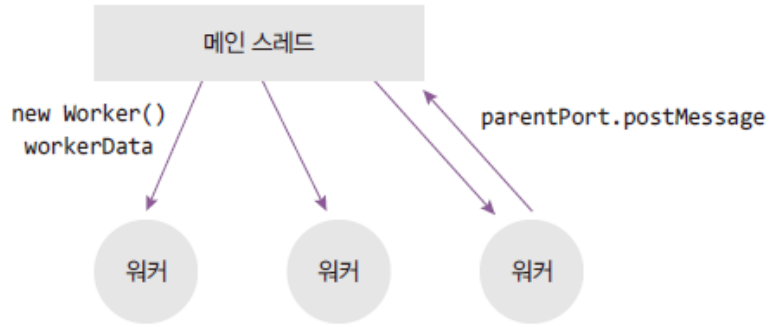
21. 여러 워커스레드 사용하기

» new Worker 호출하는 수만큼 워커 스레드가 생성됨

worker_data.js

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

if (isMainThread) { // 부모일 때
  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('from worker', message));
    worker.on('exit', () => {
      threads.delete(worker);
      if (threads.size === 0) {
        console.log('job done');
      }
    });
  }
} else { // 워커일 때
  const data = workerData;
  parentPort.postMessage(data.start + 100);
}
```

그림 3-10 메인 스레드와 워커의 통신



콘솔

```
$ node worker_data
from worker 101
from worker 102
job done
```

22. 소수 찾기 예제

» 워커 스레드를 사용하지 않을 때(싱글 스레드일 때)

prime.js

```
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.length);
```

콘솔

```
$ node prime
prime: 8.745s
664579
```

23. 소수 찾기 예제

워커 스레드를 사용할 때

prime-worker.js

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

const min = 2;
let primes = [];

function findPrimes(start, range) {
  let isPrime = true;
  let 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;
  }
}

if (isMainThread) {
  const max = 10000000;
  const threadCount = 8;
  const threads = new Set();
  const range = Math.ceil((max - min) / threadCount);
  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: range + ((max - min
➥ + 1) % threadCount) } }));
  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.length);
      }
    });
    worker.on('message', (msg) => {
      primes = primes.concat(msg);
    });
  }
} else {
  findPrimes(workerData.start, workerData.range);
  parentPort.postMessage(primes);
}
```

콘솔

```
$ node prime-worker
prime: 1.752s
664579
```

23. child_process

» 노드에서 다른 프로그램을 실행하고 싶거나 명령어를 수행하고 싶을 때 사용

- 현재 노드 프로세스 외에 새로운 프로세스를 띄워서 명령을 수행함.
- 명령 프롬프트의 명령어와 .js 을 노드를 통해 실행(리눅스라면 ls를 대신 적을 것)

exec.js

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

var 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
```

24. child_process

파이썬 프로그램 실행하기

- 파이썬3이 설치되어 있어야 함.

test.py

```
print('hello python')
```

spawn.js

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

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

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

콘솔

```
$ node spawn
hello python
```

25. 기타 모듈들

- **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.

3.6 파일 시스템 접근하기

1. fs

파일 시스템에 접근하는 모듈

- 파일/폴더 생성, 삭제, 읽기, 쓰기 가능
- 웹 브라우저에서는 제한적이었으나 노드는 권한을 가지고 있음
- 파일 읽기 예제(결과의 버퍼는 뒤에서 설명함)

readme.txt

```
저를 읽어주세요.
```

readFile.js

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

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

콘솔

```
$ node readFile
<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.
```

2. fs 프로미스

» 콜백 방식 대신 프로미스 방식으로 사용 가능

- require('fs').promises
- 사용하기 훨씬 더 편해서 프로미스 방식을 추천함

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);
  });
```

3. fs로 파일 만들기

파일을 만드는 예제

writeFile.js

```
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);
  });
```

콘솔

```
$ node writeFile
글이 입력됩니다.
```

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 ㄱ
- 우측 코드 콘솔 예측해보기

readme2.txt

```
저를 여러 번 읽어보세요.
```

async.js

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

console.log('시작');
fs.readFile('./readme2.txt', (err, data) => {
  if (err) {
    throw err;
  }
  console.log('1번', data.toString());
});
fs.readFile('./readme2.txt', (err, data) => {
  if (err) {
    throw err;
  }
  console.log('2번', data.toString());
});
fs.readFile('./readme2.txt', (err, data) => {
  if (err) {
    throw err;
  }
  console.log('3번', data.toString());
});
console.log('끝');
```

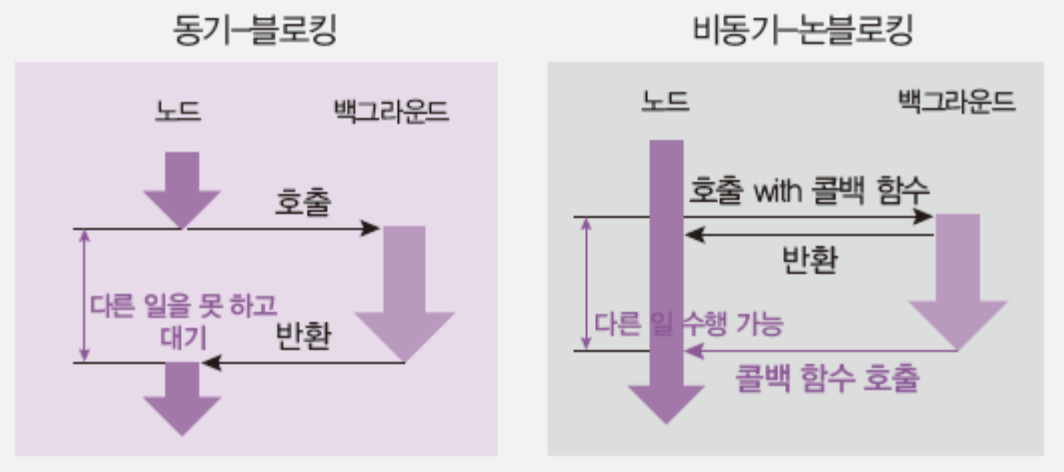
콘솔

```
$ node async
시작
끝
2번 저를 여러 번 읽어보세요.
3번 저를 여러 번 읽어보세요.
1번 저를 여러 번 읽어보세요.
```

5. 동기 메서드와 비동기 메서드

- 이전 예제를 여러 번 실행해보기
  - 매 번 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

sync.js

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

console.log('시작');
let data = fs.readFileSync('./readme2.txt');
console.log('1번', data.toString());
data = fs.readFileSync('./readme2.txt');
console.log('2번', data.toString());
data = fs.readFileSync('./readme2.txt');
console.log('3번', data.toString());
console.log('끝');
```

콘솔

```
$ node sync
시작
1번 저를 여러 번 읽어보세요.
2번 저를 여러 번 읽어보세요.
3번 저를 여러 번 읽어보세요.
끝
```

7. 비동기 메서드로 순서 유지하기

콜백 형식 유지

- 코드가 우측으로 너무 들어가는 현상 발생(콜백 헬)

asyncOrder.js

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

console.log('시작');
fs.readFile('./readme2.txt', (err, data) => {
  if (err) {
    throw err;
  }
  console.log('1번', data.toString());
  fs.readFile('./readme2.txt', (err, data) => {
    if (err) {
      throw err;
    }
    console.log('2번', data.toString());
    fs.readFile('./readme2.txt', (err, data) => {
      if (err) {
        throw err;
      }
      console.log('3번', data.toString());
      console.log('끝');
    });
  });
});
```

콘솔

```
$ node asyncOrder
시작
끝
1번 저를 여러 번 읽어보세요.
2번 저를 여러 번 읽어보세요.
3번 저를 여러 번 읽어보세요.
```

8. 비동기 메서드로 순서 유지하기

프로미스로 극복

asyncOrderPromise.js

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

console.log('시작');
fs.readFile('./readme2.txt')
  .then((data) => {
    console.log('1번', data.toString());
    return fs.readFile('./readme2.txt');
  })
  .then((data) => {
    console.log('2번', data.toString());
    return fs.readFile('./readme2.txt');
  })
  .then((data) => {
    console.log('3번', data.toString());
    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

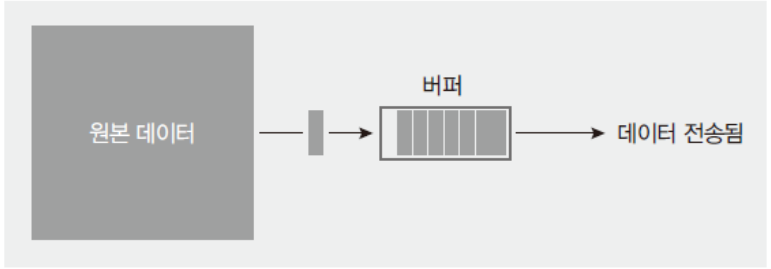
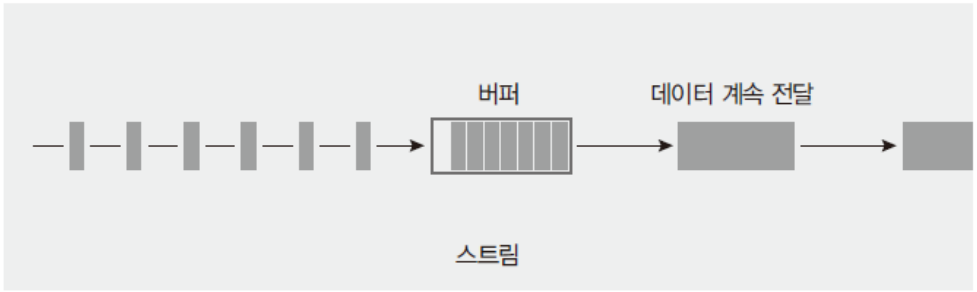


그림 3-13 스트림



10. 버퍼 사용하기

노드에서는 Buffer 객체 사용

buffer.js

```
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);
```

콘솔

```
$ node buffer
from(): <Buffer ec a0 80 eb a5 bc 20 eb b2 84 ed 8d bc eb a1 9c 20 eb b0 94 ea bf 94 eb
b3 b4 ec 84 b8 ec 9a 94>
length: 32
toString(): 저를 버퍼로 바꿔보세요
concat(): 띄엄 띄엄 띄어쓰기
alloc(): <Buffer 00 00 00 00 00>
```

11. Buffer의 메서드

» 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.

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

readme3.txt

```
저는 조금씩 조금씩 나눠서 전달됩니다. 나눠진 조각을 chunk라고 부릅니다.
```

createReadStream.js

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

const readStream = fs.createReadStream('./readme3.txt', { highWaterMark: 16 });
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);
});
```

콘솔

```
$ node createReadStream
data : <Buffer ec a0 80 eb 8a 94 20 ec a1 b0 ea b8 88 ec 94 a9> 16
data : <Buffer 20 ec a1 b0 ea b8 88 ec 94 a9 20 eb 82 98 eb 88> 16
data : <Buffer a0 ec 84 9c 20 ec a0 84 eb 8b ac eb 90 a9 eb 8b> 16
data : <Buffer 88 eb 8b a4 2e 20 eb 82 98 eb 88 a0 ec a7 84 20> 16
data : <Buffer ec a1 b0 ea b0 81 ec 9d 84 20 63 68 75 6e 6b eb> 16
data : <Buffer 9d bc ea b3 a0 20 eb b6 80 eb a6 85 eb 8b 88 eb> 16
data : <Buffer 8b a4 2e> 3
end : 저는 조금씩 조금씩 나눠서 전달됩니다. 나눠진 조각을 chunk라고 부릅니다.
```

13. 파일 쓰는 스트림 사용하기

fs.createWriteStream

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

createWriteStream.js

```
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();
```

콘솔

```
$ node createWriteStream
파일 쓰기 완료
```

14. 스트림 사이에 pipe 사용하기

» pipe로 여러 개의 스트림을 이을 수 있음

- 스트림으로 파일을 복사하는 예제

readme4.txt

```
저를 writeme3.txt로 보내주세요.
```

pipe.js

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

const readStream = fs.createReadStream('readme4.txt');
const writeStream = fs.createWriteStream('writeme3.txt');
readStream.pipe(writeStream);
```

콘솔

```
$ node pipe
```

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)

gzip.js

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

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

콘솔

```
$ node gzip
```

16. 큰 파일 만들기

» 1GB 정도의 파일을 만들어 봄.

- createWriteStream으로 만들어야 메모리 문제가 생기지 않음.

createBigFile.js

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

for (let i = 0; i <= 10000000; i++) {
  file.write('안녕하세요. 엄청나게 큰 파일을 만들어 볼 것입니다. 각오 단단히 하세요!\n');
}
file.end();
```

콘솔

```
$ node createBigFile
```

17. 메모리 체크하기

버퍼 방식과 스트림 방식 메모리 사용량을 비교해보기

buffer-memory.js

```
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);
```

콘솔

```
$ node buffer-memory
before:  18137088
buffer:  1019133952
```

stream-memory.js

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

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

const readStream = fs.createReadStream('./big.txt');
const writeStream = fs.createWriteStream('./big3.txt');
 readStream.pipe(writeStream);
 readStream.on('end', () => {
   console.log('stream: ', process.memoryUsage().rss);
 });
```

콘솔

```
$ node stream-memory
before:  18087936
stream:  62472192
```

18. 기타 fs 메서드

파일 및 폴더 생성

fsCreate.js

```
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);
    fs.rename('./folder/file.js', './folder/newfile.js');
  })
  .then(() => {
    console.log('이름 바꾸기 성공');
  })
  .catch((err) => {
    console.error(err);
  });
```

콘솔

```
$ node fsCreate
폴더 없음
폴더 만들기 성공
빈 파일 만들기 성공 3
이름 바꾸기 성공
$ node fsCreate
이미 폴더 있음
```

19. access, mkdir, open, rename

파일 및 폴더 생성, 삭제

- fs.access(경로, 옵션, 콜백): 폴더나 파일에 접근할 수 있는지를 체크합니다. 두 번째 인자로 상수들을 넣었습니다. F_OK는 파일 존재 여부, R_OK는 읽기 권한 여부, W_OK는 쓰기 권한 여부를 체크합니다. 파일/폴더나 권한이 없다면 에러가 발생하는데, 파일/폴더가 없을 때의 에러 코드는 ENOENT입니다.
- fs.mkdir(경로, 콜백): 폴더를 만드는 메서드입니다. 이미 폴더가 있다면 에러가 발생하므로 먼저 access() 메서드를 호출해서 확인하는 것이 중요합니다.
- fs.open(경로, 옵션, 콜백): 파일의 아이디(fd 변수)를 가져오는 메서드입니다. 파일이 없다면 파일을 생성한 뒤 그 아이디를 가져옵니다. 가져온 아이디를 사용해 fs.read()나 fs.write()로 읽거나 쓸 수 있습니다. 두 번째 인자로 어떤 동작을 할 것인지 설정할 수 있습니다. 쓰려면 w, 읽으려면 r, 기존 파일에 추가하려면 a입니다. 예제에서는 w로 설정했으므로 파일이 없을 때 새로 만들 수 있었습니다. r이었다면 에러가 발생하였을 것입니다.
- fs.rename(기존 경로, 새 경로, 콜백): 파일의 이름을 바꾸는 메서드입니다. 기존 파일 위치와 새로운 파일 위치를 적어주면 됩니다. 반드시 같은 폴더를 지정할 필요는 없으므로 잘라내기 같은 기능을 할 수도 있습니다.

20. 폴더 내용 확인 및 삭제

- fs.readdir(경로, 콜백): 폴더 안의 내용물을 확인할 수 있습니다. 배열 안에 내부 파일과 폴더명이 나옵니다.
- fs.unlink(경로, 콜백): 파일을 지울 수 있습니다. 파일이 없다면 에러가 발생하므로 먼저 파일이 있는지를 꼭 확인해야 합니다.
- fs.rmdir(경로, 콜백): 폴더를 지울 수 있습니다. 폴더 안에 파일이 있다면 에러가 발생하므로 먼저 내부 파일을 모두 지우고 호출해야 합니다.

fsDelete.js

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

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

콘솔

```
$ node fsDelete
폴더 내용 확인 [ 'newfile.js' ]
파일 삭제 성공
폴더 삭제 성공
```

21. 기타 fs 메서드

파일을 복사하는 방법

copyFile.js

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

fs.copyFile('readme4.txt', 'writeme4.txt')
  .then(() => {
    console.log('복사 완료');
  })
  .catch((error) => {
    console.error(error);
  });
```

콘솔

```
$ node copyFile
복사 완료
```

22. 기타 fs 메서드

» 파일을 감시하는 방법(변경 사항 발생 시 이벤트 호출)

watch.js

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

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

콘솔

```
$ node watch
// 내용물 수정 후
change target.txt
change target.txt
// 파일명 변경 또는 파일 삭제 후
rename target.txt
```

23. 스레드풀 알아보기

Node.js 교과서

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

threadpool.js

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

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

crypto.pbkdf2(pass, salt, 1000000, 128, 'sha512', () => {
  console.log('1:', Date.now() - start);
});

crypto.pbkdf2(pass, salt, 1000000, 128, 'sha512', () => {
  console.log('2:', Date.now() - start);
});

crypto.pbkdf2(pass, salt, 1000000, 128, 'sha512', () => {
  console.log('3:', Date.now() - start);
});

crypto.pbkdf2(pass, salt, 1000000, 128, 'sha512', () => {
  console.log('4:', Date.now() - start);
});

crypto.pbkdf2(pass, salt, 1000000, 128, 'sha512', () => {
  console.log('5:', Date.now() - start);
});

crypto.pbkdf2(pass, salt, 1000000, 128, 'sha512', () => {
  console.log('6:', Date.now() - start);
});

crypto.pbkdf2(pass, salt, 1000000, 128, 'sha512', () => {
  console.log('7:', Date.now() - start);
});

crypto.pbkdf2(pass, salt, 1000000, 128, 'sha512', () => {
  console.log('8:', Date.now() - start);
});
```

콘솔

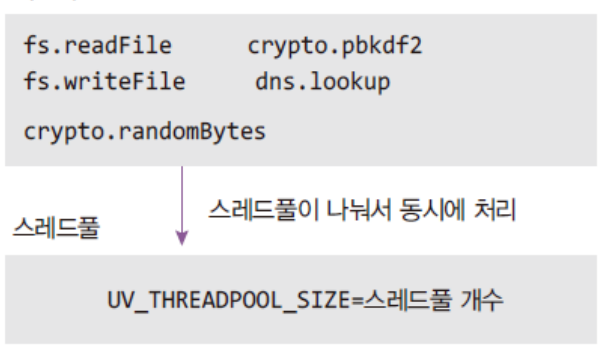
```
$ node threadpool
4: 1548
2: 1583
1: 1590
3: 1695
6: 3326
5: 3463
7: 3659
8: 3682
```

24. UV_THREAD_SIZE

» 스레드풀을 직접 컨트롤할 수는 없지만 개수 조절은 가능

- 윈도우라면 터미널에 SET UV_THREADPOOL_SIZE=개수
- 맥, 리눅스라면 UV_THREADPOOL_SIZE=개수
- 이전 예제를 스레드풀 개수를 바꾼 뒤 재실행해보기

그림 3-14 스레드풀 개수만큼 작업을 동시에 처리합니다.



3.7 이벤트 이해하기

1. 이벤트 만들고 호출하기

events 모듈로 커스텀 이벤트를 만들 수 있음

- 스트림에 쓰였던 on('data'), on('end') 등과 비교

- on(이벤트명, 콜백): 이벤트 이름과 이벤트 발생 시의 콜백을 연결해줍니다. 이렇게 연결하는 동작을 이벤트 리스닝이라고 부릅니다. event2처럼 이벤트 하나에 이벤트 여러 개를 달아줄 수도 있습니다.
- addListener(이벤트명, 콜백): on과 기능이 같습니다.
- emit(이벤트명): 이벤트를 호출하는 메서드입니다. 이벤트 이름을 인자로 넣어주면 미리 등록해뒀던 이벤트 콜백이 실행됩니다.
- once(이벤트명, 콜백): 한 번만 실행되는 이벤트입니다. myEvent.emit('event3')을 두 번 연속 호출했지만 콜백이 한 번만 실행됩니다.
- removeAllListeners(이벤트명): 이벤트에 연결된 모든 이벤트 리스너를 제거합니다.event4가 호출되기 전에 리스너를 제거했으므로 event4의 콜백은 호출되지 않습니다.
- removeListener(이벤트명, 리스너): 이벤트에 연결된 리스너를 하나씩 제거합니다. 역시event5의 콜백도 호출되지 않습니다.
- off(이벤트명, 콜백): 노드 10 버전에서 추가된 메서드로, removeListener와 기능이 같습니다.
- listenerCount(이벤트명): 현재 리스너가 몇 개 연결되어 있는지 확인합니다.

2. 커스텀 이벤트 예제

event.js

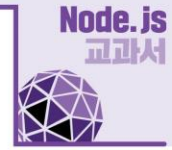
```
const EventEmitter = require('events');

const myEvent = new EventEmitter();
myEvent.addListener('event1', () => {
  console.log('이벤트 1');
});
myEvent.on('event2', () => {
  console.log('이벤트 2');
});
myEvent.on('event2', () => {
  console.log('이벤트 2 추가');
});
myEvent.once('event3', () => {
  console.log('이벤트 3');
}); // 한 번만 실행됨

myEvent.emit('event1'); // 이벤트 호출
myEvent.emit('event2'); // 이벤트 호출

myEvent.emit('event3'); // 이벤트 호출
myEvent.emit('event3'); // 실행 안 됨

myEvent.on('event4', () => {
  console.log('이벤트 4');
});
myEvent.removeAllListeners('event4');
myEvent.emit('event4'); // 실행 안 됨

const listener = () => {
  console.log('이벤트 5');
};
myEvent.on('event5', listener);
myEvent.removeListener('event5', listener);
myEvent.emit('event5'); // 실행 안 됨

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

콘솔

```
$ node event
이벤트 1
이벤트 2
이벤트 2 추가
이벤트 3
2
```

3.8 예외 처리하기

1. 예외 처리



- 예외(Exception): 처리하지 못한 에러
  - 노드 스레드를 멈춤
  - 노드는 기본적으로 싱글 스레드라 스레드가 멈춘다는 것은 프로세스가 멈추는 것
  - 에러 처리는 필수

2. try catch문

» 기본적으로 try catch문으로 예외를 처리

- 에러가 발생할 만한 곳을 try catch로 감쌈

error1.js

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

콘솔

```
$ node error1
시작
Error: 서버를 고장내주마!
    at Timeout.setInterval [as _onTimeout] (C:\Users\zerocho\error1.js:4:11)
    at ontimeout (timers.js:469:11)
    at tryOnTimeout (timers.js:304:5)
    at Timer.listOnTimeout (timers.js:264:5)
시작
Error: 서버를 고장내주마!
    at Timeout.setInterval [as _onTimeout] (C:\Users\zerocho\error1.js:4:11)
    at ontimeout (timers.js:469:11)
    at tryOnTimeout (timers.js:304:5)
    at Timer.listOnTimeout (timers.js:264:5)
// 계속 반복
```

3. 노드 비동기 메서드의 에러

» 노드 비동기 메서드의 에러는 따로 처리하지 않아도 됨

- 콜백 함수에서 에러 객체 제공

error2.js

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

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

콘솔

```
$ node error2
{ Error: ENOENT: no such file or directory, unlink 'C:\Users\zerocho\abcdefg.js'
  errno: -4058,
  code: 'ENOENT',
  syscall: 'unlink',
  path: 'C:\\Users\\zerocho\\abcdefg.js' }
{ Error: ENOENT: no such file or directory, unlink 'C:\Users\zerocho\abcdefg.js'
  errno: -4058,
  code: 'ENOENT',
  syscall: 'unlink',
  path: 'C:\\Users\\zerocho\\abcdefg.js' }
// 계속 반복
```

4. 프로미스의 에러

Node.js 교과서

» 프로미스의 에러는 따로 처리하지 않아도 됨

- 버전이 올라가면 동작이 바뀔 수 있음

error3.js

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

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

콘솔

```
$ node error3
(node: 35384) UnhandledPromiseRejectionWarning: Error: ENOENT: no such file or
directory, unlink 'C:\Users\zerocho\abcdefg.js'
(Use `node --trace-warnings ...` to show where the warning was created)
(node:35384) UnhandledPromiseRejectionWarning: Unhandled promise rejection. This error
originated either by throwing inside of an async function without a catch block, or by
rejecting a promise which was not handled with .catch(). To terminate the node process
on unhandled promise rejection, use the CLI flag `--unhandled-rejections=strict` (see
https://nodejs.org/api/cli.html#cli_unhandled_rejections_mode). (rejection id: 1)
(node:35384) [DEP0018] DeprecationWarning: Unhandled promise rejections are
deprecated. In the future, promise rejections that are not handled will terminate the
Node.js process with a non-zero exit code. // 계속 반복
```

5. 예측 불가능한 에러 처리하기

Node.js 교과서

» 최후의 수단으로 사용

- 콜백 함수의 동작이 보장되지 않음
- 따라서 복구 작업용으로 쓰는 것은 부적합
- 에러 내용 기록 용으로만 쓰는 게 좋음

error4.js

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

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

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

콘솔

```
$ node error4
예기치 못한 에러 Error: 서버를 고장내주마!
...
실행됩니다
예기치 못한 에러 Error: 서버를 고장내주마!
예기치 못한 에러 Error: 서버를 고장내주마!
// 계속 반복
```

6. 프로세스 종료하기

윈도

콘솔

```
$ netstat -ano | findstr 포트
$ taskkill /pid 프로세스아이디 /f
```

맥/리눅스

콘솔

```
$ lsof -i tcp:포트
$ kill -9 프로세스아이디
```