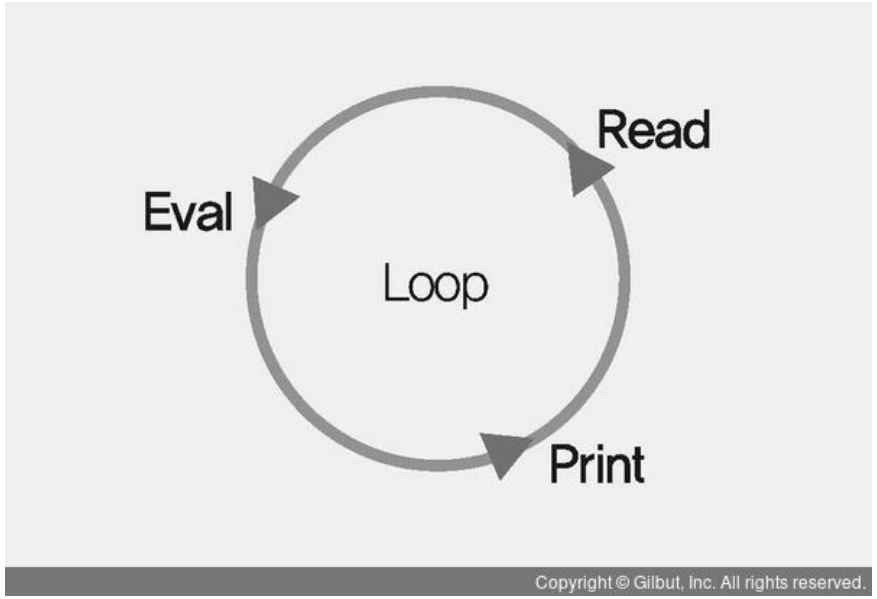
3장 노드 기능 알아보기

22.09.22 차수지

1장 요약 : https://maze-slouch-c9c.notion.site/1-acd23f34e13b4991ad00ebd90c0c1011

목차

1. REPL 사용하기



- JS === 스크립트 언어
  - 미리 컴파일 하지 않아도 즉석에서 코드 실행 가능
  - 콘솔 : 브라우저 콘솔 탭
- 노드 콘솔 **REPL**
  - Read : 입력한 코드 읽기
  - Eval : 입력한 코드 해석하기
  - Print : 결고물 반환
  - Loop : 종료할 때까지 반복

1. REPL 켜기

```
$ node
>
```

2. 간단한 문자열 출력해보기

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

- 짧은 코드 테스트 용으로는 좋지만, 여러 줄의 코드를 실행하기에는 불편함
- 긴 코드인 경우, 코드를 JS 파일로 만든 후 파일을 통째로 실행하는 것이 좋음

2. JS 파일 실행하기

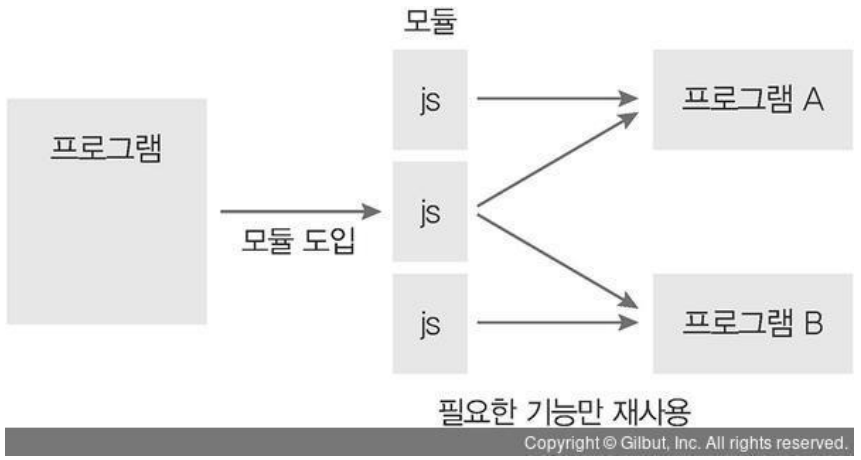
```
// helloWorld.js
const helloWorld = () => {
  console.log("Hello World");
  helloNode();
};

const helloNode = () => {
  console.log("Hello Node");
};

helloWorld();
```

콘솔에서 실행 (REPL ✕)

```
ch3>node helloWorld
Hello World
Hello Node
```

3. 모듈로 만들기

- 노드의 특징 (vs. JS)
  - 코드를 모듈로 만들 수 있다
- **모듈**
  - 특정한 기능을 하는 함수나 변수들의 집합
  - 자체로도 하나의 프로그램
  - 다른 프로그램의 부품으로도 사용 가능
  - 여러 프로그램에 재사용 가능
  - 보통 파일 하나가 모듈이 됨
  - 파일별로 코드를 모듈화 할 수 있어 관리하기 편함



Part 3 코드 

- 장점
  - 여러 파일에 걸쳐 재사용되는 함수나 변수를 모듈로 만들어두면 편리함
- 단점
  - 모듈이 많아지고 관계가 얽히면 구조를 파악하기 어려움

📌 노드에서는 대부분의 파일이 다른 파일을 모듈로 사용하고 있으므로 모듈을 만들고 사용하는 방법을 꼭 알아둬야 함

4. 노드 내장 객체 알아보기

1) global

- 전역 객체 ex) 브라우저의 window
- 모든 파일에서 접근 가능
- 생략 가능
  - window.open → open
  - global.require → require
- global 객체 내부 ∋ 많은 속성들
- 내부를 보려면 REPL 이용

2) console

- global 객체 안에 들어 있음
- 브라우저에서의 console과 거의 비슷함
- 디버깅 용
  - console.log(내용)
  - console.time(레이블)
  - console.error(에러)
  - Console.table(배열)
  - Console.dir(객체, 옵션)
  - console.trace(레이블)

3) 타이머

- global에 들어 있음
- 타이머 함수
  - setTimeout(callback, ms)
  - setInterval(callback, ms)
  - setImmediate(callback)
- 타이머 취소하기 (id 사용)
  - clearTimeout(id)
  - clearInterval(id)
  - clearImmediate(id)

4) _ _filename, _ _dirname

- 경로에 대한 정보 제공
- 파일에 _ _filename과 _ _dirname을 넣어두면, 실행 시 현재 파일명과 현재 파일 경로로 바뀜
- 경로 문제 → path 모듈 사용

5) module, exports, require

- module
  - 모듈 만들 때 사용
  - module.exports = { odd };
- exports
  - exports.odd = '홀수';
  - 참조 관계 깨지지 않게 조심
- require
  - 모듈을 불러옴
  - require.cache
    - 파일 이름이 속성명으로 들어있음
    - 한 번 require한 파일은 require.cache에 저장된 후 재사용 됨
  - require.main
    - 노드 실행 시 첫 모듈을 가리킴
  - require.main === module
    - true : 현재 파일이 첫 모듈
    - require.main.filename : 이름 확인

6) process

- 현재 실행되고 있는 노드 프로세스에 대한 정보를 담고 있음
  - process.version : 설치된 노드의 버전
  - process.arch : 프로세서 아키텍처 정보
  - process.platform : 운영체제 플랫폼 정보
  - process.pid : 현재 프로세스의 아이디
  - process.uptime() : 프로세스가 시작된 후 흐른 시간 (s)
  - process.execPath : 노드의 경로
  - process.cwd() : 현재 프로세스가 실행되는 위치
  - process.cpuUsage() : 현재 cpu 사용량

- process.env
  - 시스템 환경 변수들
  - 임의로 환경 변수 저장 가능
  - 서비스의 중요한 키를 저장하는 공간으로도 사용됨
    - const secretId = process.env.SECRET_ID;
- process.nextTick(콜백)
  - 이벤트 루프가 다른 콜백함수들보다 nextTick의 콜백 함수를 우선으로 처리하도록 만듦
  - setImmediate, setTimeout 보다 먼저 실행됨
- process.exit(코드)
  - 실행 중인 노드 프로세스 종료
  - 서버 환경에서 사용하면 서버가 멈추기 때문에, 특수한 경우를 제외하고는 서버에서 잘 사용하지 않음
  - 서버 외의 독립적인 프로그램에서 수동으로 노드 멈출 때 사용

5. 노드 내장 모듈 사용하기

1) os

- 운영체제의 정보를 가져올 수 있음
- 컴퓨터 내부 자원에 빈번하게 접근하는 경우 사용됨
  - 일반적인 웹 서비스 제작시에는 사용 빈도 낮음
  - 운영 체제별로 다른 서비스 제공할 때는 유용

- os.arch() === process.arch
- os.platform() === process.platform
- os.type : 운영체제의 종류
- os.uptime : 운영체제 부팅 이후 흐른 시간 (s)
- os.hostname() : 컴퓨터의 이름
- os.release() : 운영체제의 버전
- os.homedir() : 홈 디렉터리 경로
- os.tmpdir() : 임시 파일 저장 경로
- os.cpus() : 컴퓨터의 코어 정보
- os.freemem() : 사용 가능한 메모리(RAM)
- os.totalmem() : 전체 메모리 용량

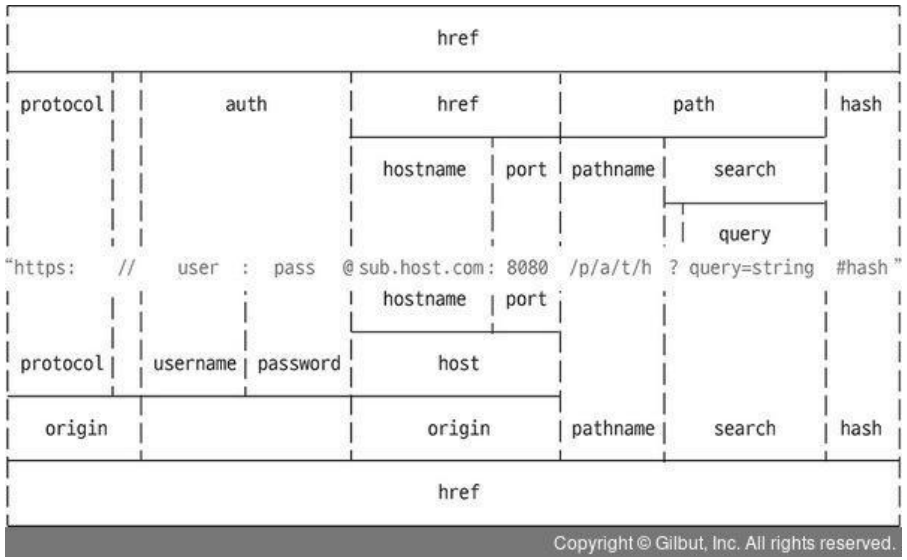
2) path

- 폴더와 파일의 경로를 쉽게 조작하도록 도와주는 모듈
- 운영체제별로 경로 구분자가 다르기 때문에 필요함

- path.sep : 경로의 구분자 (\ 또는 /)
- path.delimiter : 환경 변수의 구분자
- path.dirname(경로) : 파일이 위치한 폴더 경로
- path.extname(경로) : 파일의 확장자
- path.basename(경로, 확장자) : 파일의 이름 표시
- path.parse(경로) : 파일의 경로를 root, dir, base, ext, name으로 분리
- path.format(객체) : path.parse()한 객체를 파일 경로로 함침
- path.normalize(경로) : 경로 구분자를 여러 번 사용했거나 혼용했을 때 정상적인 경로로 변환
- path.isAbsolute(경로) : 절대경로인지 상대경로인지 true false
- path.relative(기준경로, 비교경로) : 첫번째 경로에서 두번째 경로로 가는 방법 알려줌
- path.join(경로, ...) : 여러 인수를 넣으면 하나의 경로로 합침
- Path.resolve(경로, ...) : path.join과 유사함

3) url

- 인터넷 주소를 쉽게 조작하도록 도와주는 모듈
- url 처리 방법
  1. WHATWG 방식
  2. 예전부터 노드에서 사용하던 방식



1 WHATWG 방식의 url : url 모듈 안에 URL 생성자가 있어서, 이 생성자에 주소를 넣어 객체로 만들면 주소가 부분별로 정리됨

```
1
const url = require("url");

const { URL } = url;
const myURL = new URL(
  "https://www.google.com"
);
console.log("new URL():", myURL);
console.log("url.format():", url.format(myURL));
console.log("------------------------------------
");

2
const parsedUrl = url.parse(
  "https://www.google.com"
);
console.log("url.parse():", parsedUrl);
console.log("url.format():", url.format(parsedUrl));
```

2 기존 노드 방식

- url.parse(주소) : 주소를 분해함
- url.format(객체) : 분해되었던 url 객체를 다시 원래 상태로 조립

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

const myURL = new URL("https://www.google.com");
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();
```

- getAll(키) : 키에 해당하는 모든 값들을 가져옴
- get(키) : 키에 해당하는 첫 번째 값만 가져옴
- has(키) : 해당 키가 있는지 없는지 검사
- keys() : searchParams의 모든 키를 반복기(iterator) 객체로 가져옴
- values() : searchParams의 모든 값을 반복기(iterator) 객체로 가져옴
- append(키, 값) : 해당 키를 추가함. 같은 키의 값이 있다면 유지하고 하나 더 추가함
- set(키, 값) : append와 비슷함. 같은 키의 값들을 모두 지우고 새로 추가함
- delete(키) : 해당 키 제거
- toString() : 조작한 searchParams 객체를 다시 문자열로 만듦

4) querystring

- WHATWG 방식의 url 대신 기존 노드의 url을 사용할 때, search 부분을 사용하기 쉽게 객체로 만드는 모듈
  - querystring.parse(쿼리) : url의 query 부분을 js 객체로 분해
  - querystring.stringify(객체) : 분해된 query 객체를 문자열로 다시 조립함

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

const parsedUrl = url.parse(
  "https://www.google.com);
const query = querystring.parse(parsedUrl.query);
console.log("querystring.parse():", query);
console.log("querystring.stringify():",
querystring.stringify(query));
```

5) crypto

1 단방향 암호화 (해시 함수)

- 복호화할 수 없는 암호화 방식
  - 한번 암호화하면 원래 문자열을 찾을 수 없음

  ex) 비밀번호

- 해시 기법 : 어떠한 문자열을 고정된 길이의 다른 문자열로 바꿔버리는 방식
  - createHash(알고리즘) : 사용할 해시 알고리즘
  - update(문자열) : 변환할 문자열
  - digest(인코딩) : 인코딩할 알고리즘

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

console.log(
  "base64:",
  crypto.createHash("sha512").update("비밀번
호").digest("base64")
);
console.log(
  "hex:",
  crypto.createHash("sha512").update("비밀번
호").digest("hex")
);
console.log(
  "base64:",
  crypto.createHash("sha512").update("다른 비밀번
호").digest("base64")
);
```

```
base64: dvfV6nyLRRt3NxKSlTHOkkEGgqW2HRtfu19Ou/psUXvwlebbXCboxIPmDYOFRIpqav2eUTBFuHaZri5x+usy1g==
hex: 76f7d5ea7c8b451b773712929531ce92410682a5b61d1b5fbb5f4ebbfa6c517bf095e6db5c26e8c483e60d8385448a6a6afd9e513045b87699ae2e71faeb32d6
base64: cx49cjC8ctKtMzwJGBY853itZeb6qxzXGvuUJkbWTGn5VXAFbAwXGEOxU2Qksoj+aM2GWPhc1O7mmkyohXMsQw==
```

2 양방향 암호화

- 암호화된 문자열을 복호화할 수 있음
- 키 사용
- 공식 문서
  - https://nodejs.org/api/crypto.html
- npm 패키지 crypto-js
  - https://www.npmjs.com/package/crypto-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);
```

```
암호화: iiopeG2GsYlk6ccoBoFvEH2EBDMWv1kK9bNuDjYxiN0=
복호화: 암호화할 문장
```

6) util

- 각종 편의 기능을 모아둔 모듈
- util.deprecate : 함수가 deprecated 처리되었음을 알림
  - deprecated : 중요도가 떨어져 더 이상 사용되지 않고 앞으로는 사라지게 될 것이라는 뜻 (곧 없앨 예정이니 더 이상 사용하지 마라)
- util.promisify : 콜백 패턴을 프로미스 패턴으로 바꿈
  - async/await 패턴까지 사용할 수 있어 좋음

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

const dontUseMe = util.deprecate((x, y) => {
  console.log(x + y);
}, "dontUseMe 함수는 deprecated되었으니 더 이상 사용 금지");
dontUseMe(1, 2);

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

```
(node:21732) DeprecationWarning: dontUseMe 함수는 deprecated되었으니 더 이상 사용 금지
(Use `node --trace-deprecation ...` to show where the warning was created)
lafI8/KjEMTfeVZ1QUUQF+JCl0oy+kd+8grwOQvdb/3PBzKHO+PKjrPlNypIlr2jg1w7L89JOw+KnHKm9ANLBA==
```

7) worker_threads

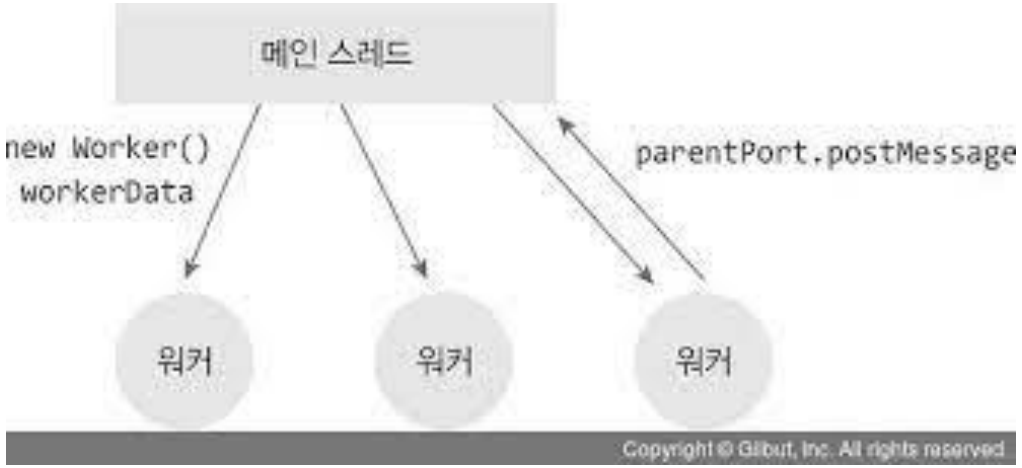
- 노드에서 멀티 스레드 방식으로 작업

📌 간단한 사용 방법

```
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();
  });
}
```

```
from parent ping
from worker pong
worker exit
```

- 여러 개의 워커 스레드에 데이터 남기기



📌 워커 스레드를 N개 사용했다고 해서 속도가 N배 빨라지는 것은 아님

📌 스레드를 생성하고 통신하는 데 상당한 비용이 발생하므로, 이 점을 고려해서 멀티 스레딩을 해야 함

```
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);
}
```

```
from worker 102
from worker 101
job done
```

8) child_process

- 다른 프로그램을 실행하고 싶거나 명령어를 수행하고 싶을 때 사용하는 모듈
- 다른 언어의 코드(ex. 파이썬)를 실행하고 결괏값을 받을 수 있음
- 현재 노드 프로세스 외에 새로운 프로세스를 띄워서 명령을 수행하고, 노드 프로세스에 결과를 알려줌

1 명령어 dir (맥은 ls) 입력 → 실행하면 현재 폴더의 파일 목록이 표시됨

```
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
```

📌 결과는 data 이벤트 리스너에 버퍼 형태로 전달됨
- stdout (표준 출력) : 성공적인 결과
- stderr (표준 에러) : 실패한 결과

- 파이썬 실행하기

```
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());
});
```

📌 exec
  - 셸을 실행해서 명령어 수행

📌 spawn
  - 새로운 프로세스를 띄우면서 명령어 실행
  - 세 번째 인수로 { shell: true }를 제공하면 exec 처럼 셸을 실행해서 명령어 수행

6. 파일 시스템 접근하기

fs 모듈

- 파일 시스템에 접근하는 모듈
- 파일 생성, 삭제, 읽기, 쓰기
- 폴더 생성, 삭제
- 자바스크립트는 대부분 파일 시스템 접근이 금지되어 있음
- 기본적으로 콜백 형식의 모듈이므로 실무에서 사용하기 불편함
  - fs 모듈을 프로미스 형식으로 바꿔주는 방법 사용

📌 파일 경로는 node 명령어를 실행하는 콘솔 기준

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

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

```
<Buffer ec 9d bd ec 96 b4>
읽어
```

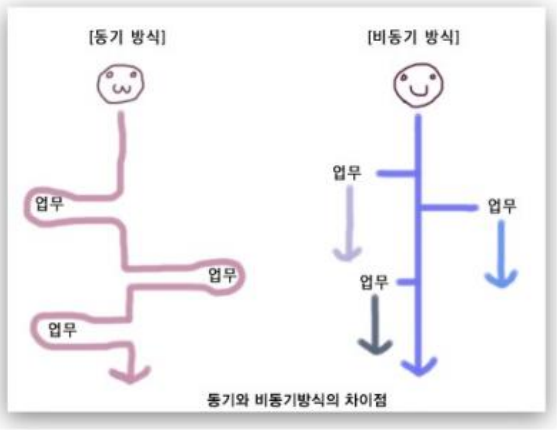
📌 readFile의 결과물은 버퍼(Buffer)의 형식으로 제공됨 (메모리의 데이터)

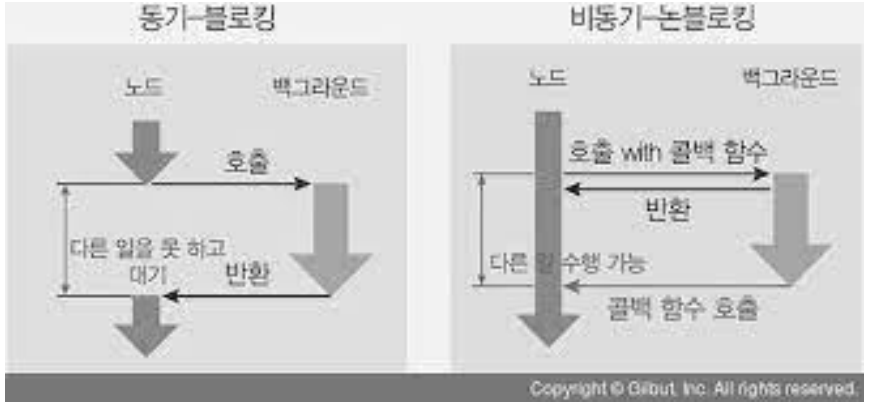
동기 메서드와 비동기 메서드

- 노드는 대부분의 메서드를 **비동기 방식**으로 처리함
- 몇몇 메서드는 동기 방식으로도 사용할 수 있음

동기 vs 비동기

- **동기(synchronous)**
  - 요청과 결과가 한 자리에서 동시에 일어남
  - 요청을 하면 시간이 얼마나 걸려도 요청한 자리에서 결과가 주어짐
- **비동기(asynchronous)**
  - 요청과 결과가 동시에 일어나지 않음
  - 요청 내용에 대해서 바로 응답을 받지 않아도 됨

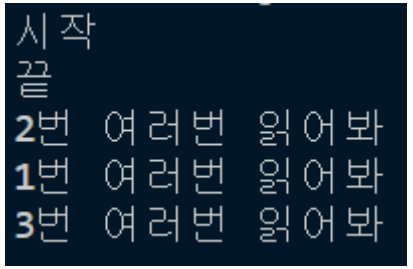




동기와 비동기 : 백그라운드 작업 완료 확인 여부

블로킹과 논블로킹 : 함수가 바로 return 되는지 여부

```
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");
```



📌 '시작'과 '끝'을 제외하고는, 반복 실행할 때마다 결과가 달라짐

📌 비동기 메서드들은 백그라운드에 해당 파일을 읽으라고만 요청하고 다음 작업으로 넘어가기 때문에,

1. console.log('시작') 찍음
2. 파일 읽기 요청 3번 보내기
3. console.log('끝') 찍음
4. 읽기가 완료되면 백그라운드가 다시 메인 스레드에 알림
5. 메인스레드가 등록된 콜백 함수 실행시킴

💡 수백 개의 I/O 요청이 들어와도 메인 스레드는 백그라운드에 요청 처리를 위임하기 때문에 얼마든지 요청을 더 받을 수 있어 좋음

비동기 방식으로 하되 순서를 유지하고 싶다면?

```
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("1번", data.toString());
      console.log("끝");
    });
  });
});
```

```
시작
1번 여러번 읽어봐
2번 여러번 읽어봐
1번 여러번 읽어봐
끝
```

콜백 지옥은 async/await으로 어느 정도 해결 가능함

버퍼와 스트림 이해하기

6. 파일 시스템 접근하기

fs 모듈

- 파일 시스템에 접근하는 모듈
- 파일 생성, 삭제, 읽기, 쓰기
- 폴더 생성, 삭제
- 자바스크립트는 대부분 파일 시스템 접근이 금지되어 있음
- 기본적으로 콜백 형식의 모듈이므로 실무에서 사용하기 불편함
  - fs 모듈을 프로미스 형식으로 바꿔주는 방법 사용

📌 파일 경로는 node 명령어를 실행하는 콘솔 기준

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

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

```
<Buffer ec 9d bd ec 96 b4>
읽어
```

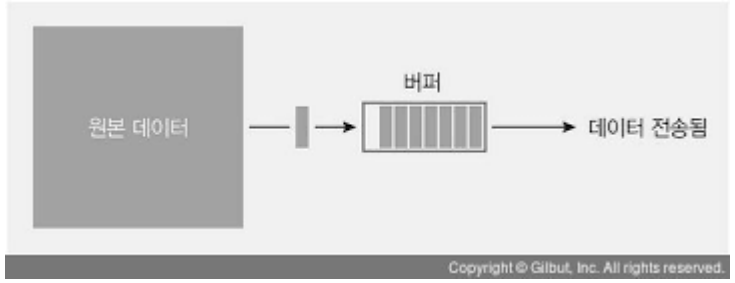
📌 readFile의 결과물은 버퍼(Buffer)의 형식으로 제공됨 (메모리의 데이터)

💡 앞에서 계속 data.**toString()** 을 썼던 이유

: data가 버퍼이기 때문!

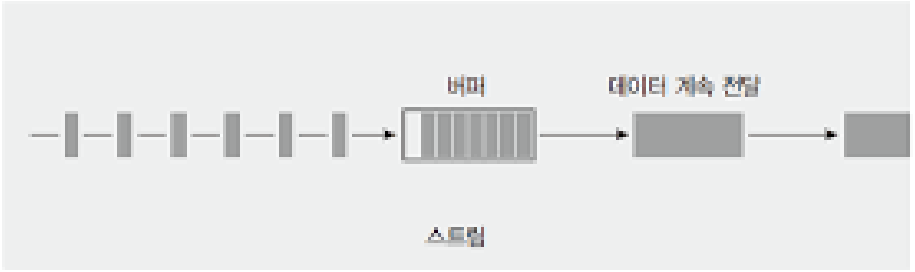
버퍼

- 버퍼링 : 영상을 재생할 수 있을 때까지 데이터를 모으는 동작
- 버퍼 : 메모리에 저장된 데이터



스트림

- 스트리밍 : 방송인의 컴퓨터에서 시청자의 컴퓨터로 영상 데이터를 조금씩 전송하는 동작
- 스트림 : 버퍼의 크기를 작게 만든 후 여러 번으로 나눠 보내는 방식을 편리하게 만든 것
- 파이핑 : 스트림끼리 연결하는 것
- 스트림을 사용하면 효과적으로 데이터를 전송할 수 있음



스레드풀

- 비동기 메서드들은 백그라운드에서 실행되고, 실행된 후에는 다시 메인 스레드의 콜백 함수나 프로미스의 then 부분이 실행됨
- 이때 fs 메서드를 여러 번 실행해도 백그라운드에서 동시에 처리됨 → **스레드풀**이 있기 때문!
- 스레드풀을 사용하는 모듈
  - fs, crypto, zlib, dns.lookup 등
- 스레드풀은 작업을 동시에 처리하므로, 어느 것이 먼저 처리될지 모름
- 기본적인 스레드풀의 개수는 4개
  - 만약 8개의 작업이 있다면, 처음 네 작업이 동시에 실행되고 그것들이 종료되면 다음 네 개의 작업이 실행됨
  - 임의로 개수 조절 가능 **UV_THREADPOO_SIZE = 숫자** @cmd

7. 이벤트 이해하기

이벤트 관리를 위한 메서드들

- on(이벤트명, 콜백) : 이벤트 이름과 이벤트 발생 시의 콜백을 연결함 (이벤트 리스닝)
- addListner(이벤트명, 콜백) === on(이벤트명, 콜백)
- emit(이벤트명) : 이벤트 호출
- once(이벤트명, 콜백) : 한 번만 실행되는 이벤트
- removeAllListeners(이벤트명) : 이벤트에 연결된 모든 이벤트 리스너 제거
- removeListner(이벤트명, 리스너) : 이벤트에 연결된 리스너를 하나씩 제거함
- off(이벤트명, 콜백) removeListner(이벤트명, 리스너)
- listenerCount(이벤트명) : 현재 리스너가 몇 개 연결되어 있는지 확인

8. 예외 처리하기

- 예외
  - 처리하지 못한 에러
  - 실행 중인 노드 프로세스를 멈추게 만듦
  - 노드는 메인 스레드가 하나뿐이므로 전체 서버가 멈춤
  - 에러 로그가 기록되더라도 작업은 계속 진행될 수 있도록 에러를 처리하는 방법이 필요함

```
setInterval(() => {
  console.log(시작);
  try {
    // 에러 강제 발생
    throw new Error("서버 고장");
  } catch (err) {
    console.error(err);
  }
}, 1000);
```

📌 에러가 발생할 것 같은 부분을 미리 try/catch로 감싸면 됨

- 노드 자체에서 잡아주는 에러

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

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

📌 fs.unlink : 존재하지 않는 파일을 지움

- 프로미스의 에러는 catch하지 않아도 알아서 처리됨

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

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

📌 그래도 프로미스를 사용할 때는 항상 catch 붙이는 것 권장

- 예측 불가능한 에러를 처리하는 방법

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

setInterval(() => {
  throw new Error("서버 고장");
}, 1000);

setTimeout(() => {
  console.log("실행됨");
}, 2000);
```

📌 process 객체에 uncaughtException 이벤트 리스너 달음

- 처리하지 못한 에러 발생 시 이벤트 리스너가 실행되고 프로세스가 유지됨
- uncaughtException은 최후의 수단으로 사용 권장
  - 노드는 uncaughtException 이벤트 발생 후 다음 동작이 제대로 동작하는 지 보장 X (즉, 복구 작업 코드를 넣어 두었더라도 그것이 동작하는지 확인할 수 없음)
  - 단순히 에러 내용을 기록하는 정도로 사용하고, 에러를 기록한 후 process.exit()로 프로세스를 종료하는 것이 좋음