KB금융그룹 | 국민의 평생 금융파트너

정리하는 단원
노드 내부에서는
동기, 비동기 작업을
어떤 구조로 관리하는가?

2025년 상반기 K-디지털 트레이닝

코드 말고 그림을 통해서 이해햏보자

노드와 비동기 처리

[KB] IT's Your Life

KB 국민은행

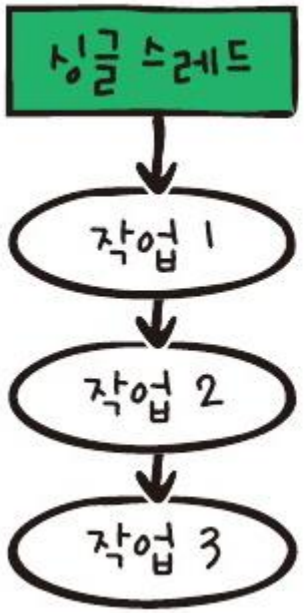
1 동기 처리와 비동기 처리는 어떻게 다를까?

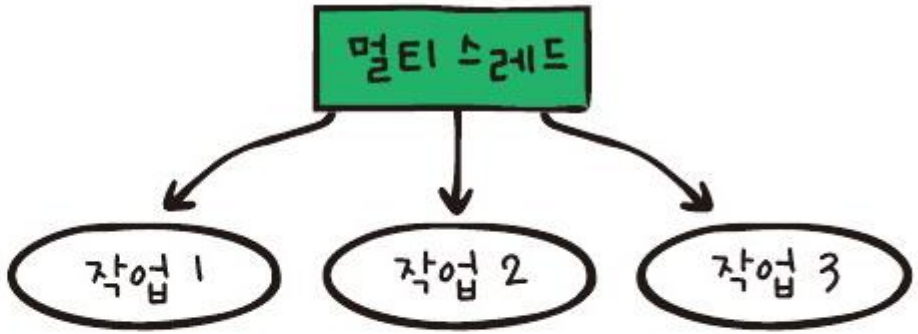
동기 처리란

스레드 thread

- 작업을 처리하기 위해 자원을 사용하는 단위
- 하나의 작업이 실행되는 최소 단위

싱글 스레드와 멀티 스레드





하나의 메인 쓰레드와

프로그래머가 필요로 생성한
다수의 워커 쓰레드

1 동기 처리와 비동기 처리는 어떻게 다를까?

동기 처리란

- 자바스크립트는 싱글 스레드 언어

chapter05/sec01/sync.js

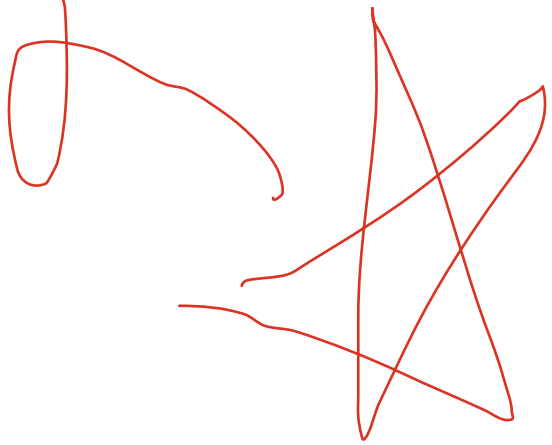
```
console.log('첫 번째 작업');
console.log('두 번째 작업');
console.log('세 번째 작업');
```

```
첫 번째 작업
두 번째 작업
세 번째 작업
```

1 동기 처리와 비동기 처리는 어떻게 다를까?

비동기 처리

- 함수 호출을 하면 작업을 의뢰하고 바로 리턴
  - 호출한 함수가 끝나도 실제 작업은 시작하지 않았음
- 더 이상 실행할 코드가 없을 때 비동기 함수가 의뢰한 작업 실행
- 의뢰한 작업이 끝났음을 통지하기 위해 콜백 함수 호출
  - 작업을 의뢰할 때 콜백 함수를 같이 등록해 둠



chapter05/sec01/async-1.js

```
console.log('첫 번째 작업');
setTimeout(() => {
  console.log('두 번째 작업');
}, 3000);

console.log('세 번째 작업');
```

call back

```
첫 번째 작업
세 번째 작업
두 번째 작업
```

chapter05/sec01/async-2.js

```
console.log('첫 번째 작업');
setTimeout(() => {
  console.log('두 번째 작업');
}, 0);

console.log('세 번째 작업');
```

앞서 설명에서 봤듯
노드는
동기부터 모두 다 처리하고 나서
그때부터 의뢰된 '비동기 작업을 처리한다

```
첫 번째 작업
세 번째 작업
두 번째 작업
```

chapter05/sec01/async-3.js

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

fs.readdir('./', (err, files) => {
  if (err) {
    return console.error(err);
  }
  console.log(files);
});
console.log('Code is done.');
```

Code is done.
['sync.js', 'async-1.js', 'async-2.js', 'async-3.js']

블로킹 I/O, blocking I/O

운영체제 관점에서 봤을 때는 blocking작업 == i/o작업

- I/O를 수행할 때 우리의 코드 실행은 멈추됨
- 동기 함수는 블록킹 I/O로 실행
- 노드의 함수명이 ~Sync()로 끝남
- I/O의 결과 데이터가 리턴됨, 에러 발생시 예외가 발생

동기함수
블로킹 함수

결국
블로킹 IO == 동기적 작업 == 동기함수 ==심플하지만 성능 별로
논 블로킹 IO == 비동기적 작업 == 비동기 함수==복잡 성능좋아

- 단점
  - I/O 작업은 시간이 많이 걸림
  - 그 시간동안 아무것도 할 수 없음

정확히는 블록킹이 동기 작업을 의미하는 것은 아니다
블로킹이면서 비동기 작업인 것도 있다.

반대로
논블로킹이면서 동기작업인 것도 있다

chapter05/sec02/blocking-1.js

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

const data = fs.readFileSync('example.txt'); // 블록킹 I/O
console.log(data); // 파일 읽기가 끝날 때까지 대기
console.log('코드 끝'); // 파일을 읽고 내용을 표시할 때까지 대기
```

chapter05/blocking-2.js

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

const server = http.createServer((req, res) => {
  if (req.url === '/home') {
    res.end('HOME');
  } else if (req.url === '/about') {
    res.end('ABOUT');
  } else {
    res.end('NOT FOUND');
  }
});

server.listen(3000, () => {
  console.log('http://localhost:3000 서버 실행 중');
});
```

chapter05/sec03/blocking-3.js

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

const server = http.createServer((req, res) => {
  if (req.url === '/home') {
    res.end('HOME');
  } else if (req.url === '/about') {
    for (let i = 0; i < 100; i++) {
      for (let j = 0; j < 100; j++) {
        console.log(`${i} ${j}`);
      }
    }
    res.end('ABOUT');
  } else {
    res.end('NOT FOUND');
  }
});

server.listen(3000, () => {
  console.log('http://localhost:3000 서버 실행 중');
});
```

블로 기다리는거 성능에 에바

논블로킹 I/O, non-blocking I/O

- 작업을 비동기로 처리

논 블럭해야 작업이 중 첩 가능 각기 진행함

chapter05/sec02/non-blocking.js

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

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

console.log('코드 끝');
```

코드 끝
This File is Example io작업완료되고 종료를 알리는 콜백함수 호출됨

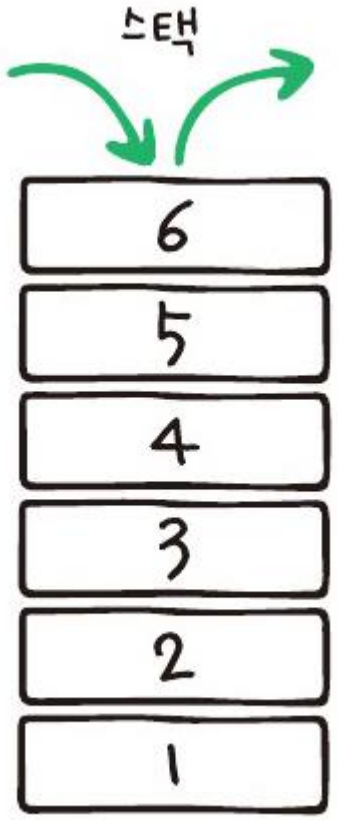
기본 처리 방법 살펴보기

- 콜 스택 call stack
  - 동기 함수 호출의 정보가 쌓이는 자료 구조

이벤트 루프가

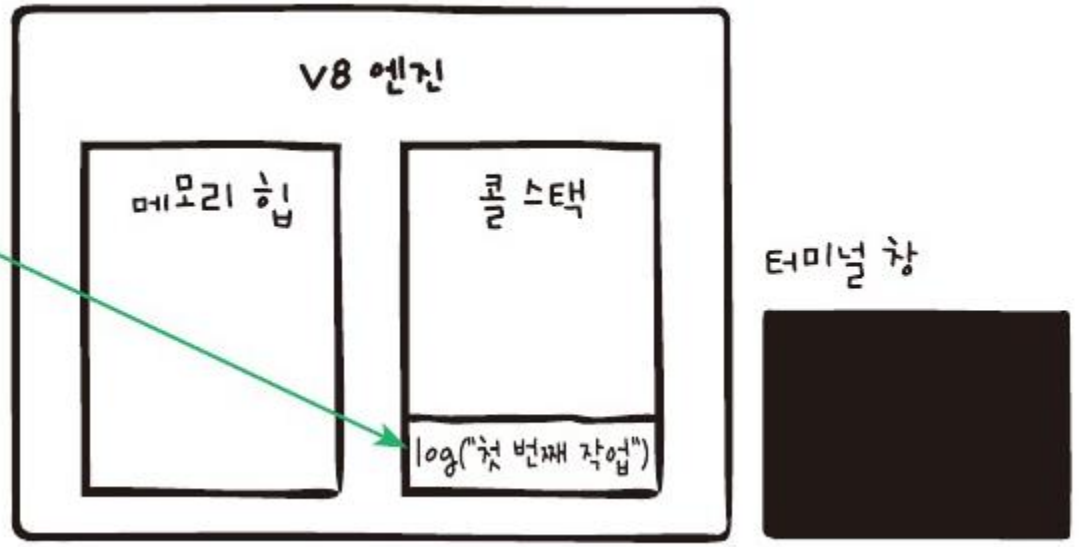
계속해서
콜스택 영역과 libuv여역을 모니터링하며

관리하고 실행함 작업들을

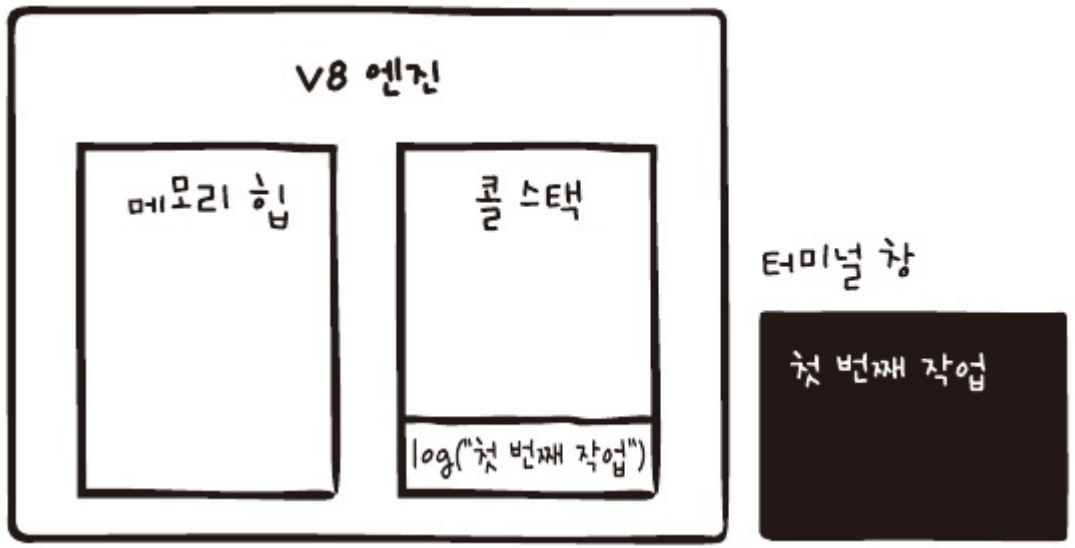


스택의 자료구조

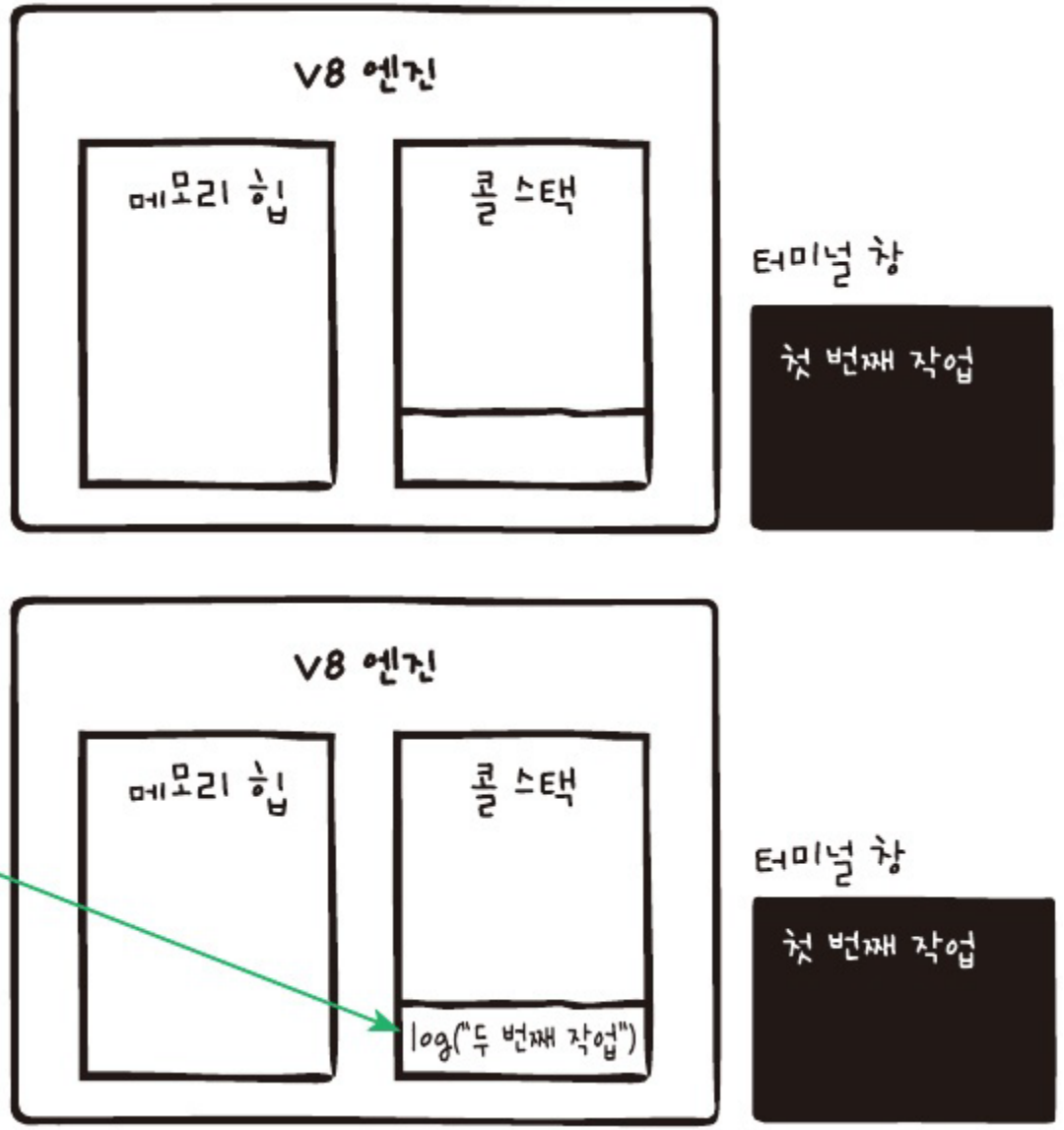
```
1   console.log("첫 번째 작업");
2   console.log("두 번째 작업");
3   console.log("세 번째 작업");
```




```
1   console.log("첫 번째 작업");
2   console.log("두 번째 작업");
3   console.log("세 번째 작업");
```



```
1   console.log("첫 번째 작업");
2   console.log("두 번째 작업");
3   console.log("세 번째 작업");
```

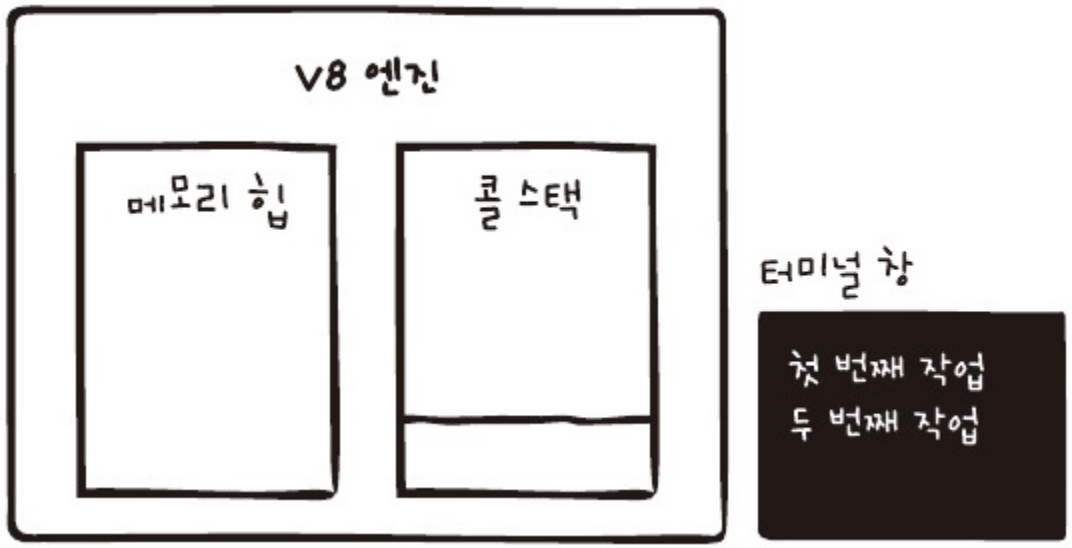



```
1   console.log("첫 번째 작업");
2   console.log("두 번째 작업");
3   console.log("세 번째 작업");
```

콜스택에 있는 작업을 실행하는 주체는 single main thread가 함. 그런 으미에서 nodejs가 싱글 쓰레드라는 것

3 이벤트 루프

```
1   console.log("첫 번째 작업");
2   console.log("두 번째 작업");
3   console.log("세 번째 작업");
```

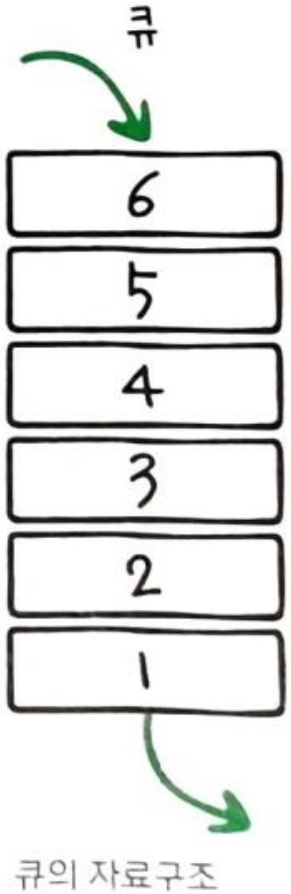


동기 작업들은 콜스택에 쌓이며
순서대로 관리대고 동작함

노드에서 비동기 작업은 ㅣ ibuv라는 것을 활용함.
libuv는 node api와 콜백 큐 영역으로 나뉘다.

이벤트 루프로 비동기 처리하기

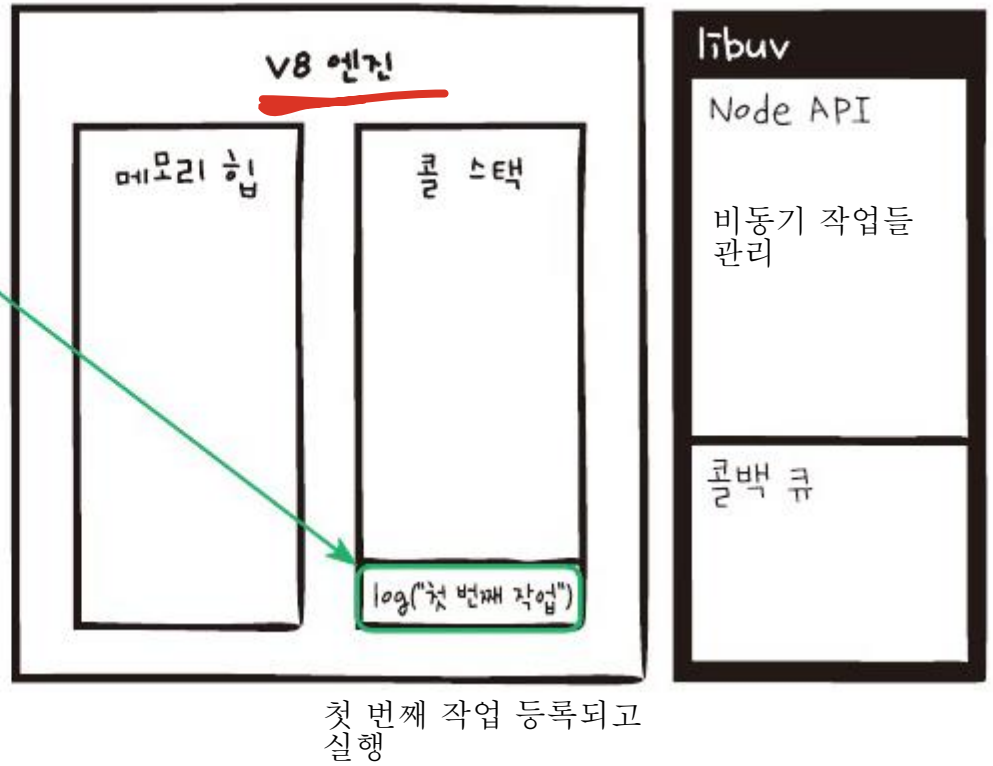
- I/O 완료 후 실행해야 할 콜백은 libuv의 큐를 이용해 관리 → 콜백 큐



큐의 자료구조

```
1  console.log("첫 번째 작업");
2  setTimeout(() => {
3    console.log("두 번째 작업");
4  }, 3000);
5  console.log("세 번째 작업");
```

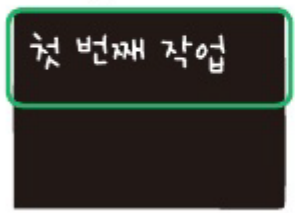
터미널 창

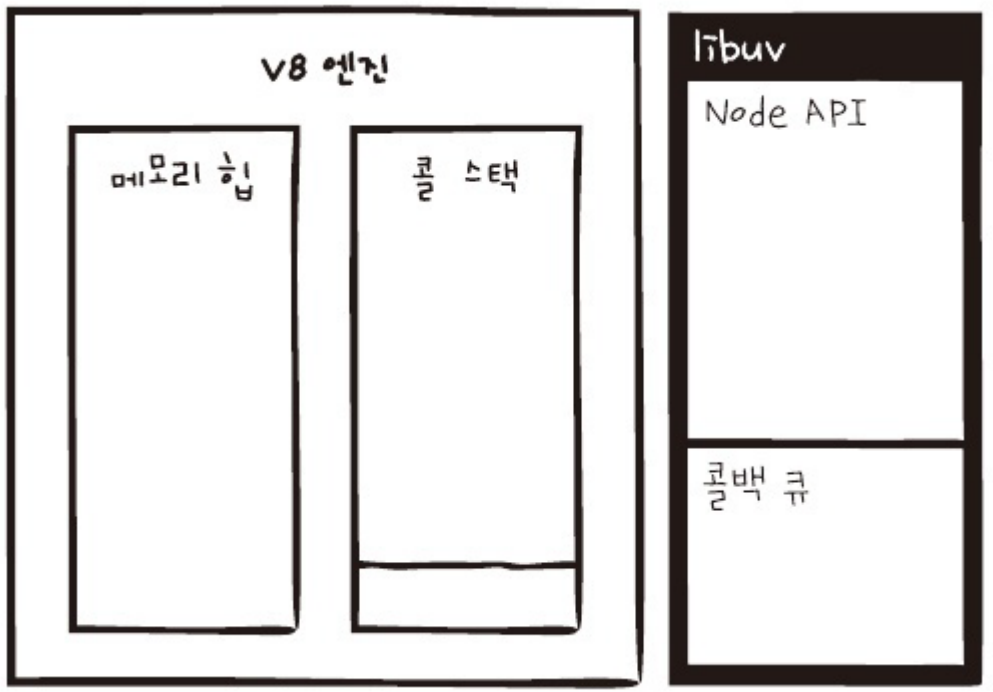


첫 번째 작업 등록되고 실행

```
1   console.log("첫 번째 작업");
2   setTimeout(() => {
3     console.log("두 번째 작업");
4   }, 3000);
5   console.log("세 번째 작업");
```

터미널 창

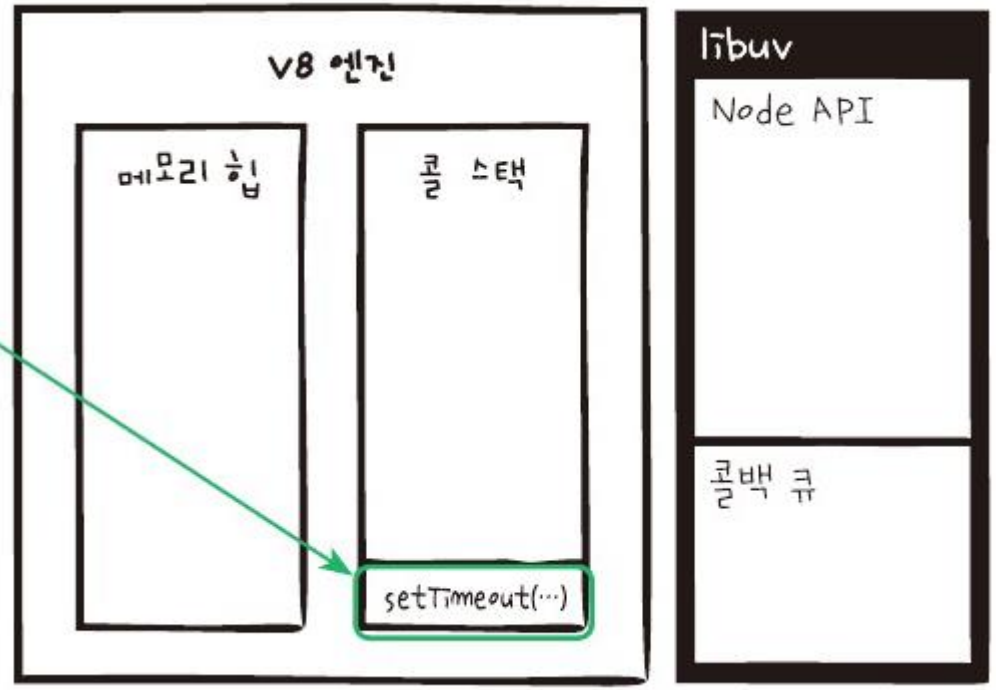




```
1  console.log("첫 번째 작업");
2  setTimeout(() => {
3    console.log("두 번째 작업");
4  }, 3000);
5  console.log("세 번째 작업");
```

터미널 창

```
첫 번째 작업
```



2번째 작업 콜스택에
등록되고

하지만 해당 함수가
node api에 있는 내장 함수 이므로

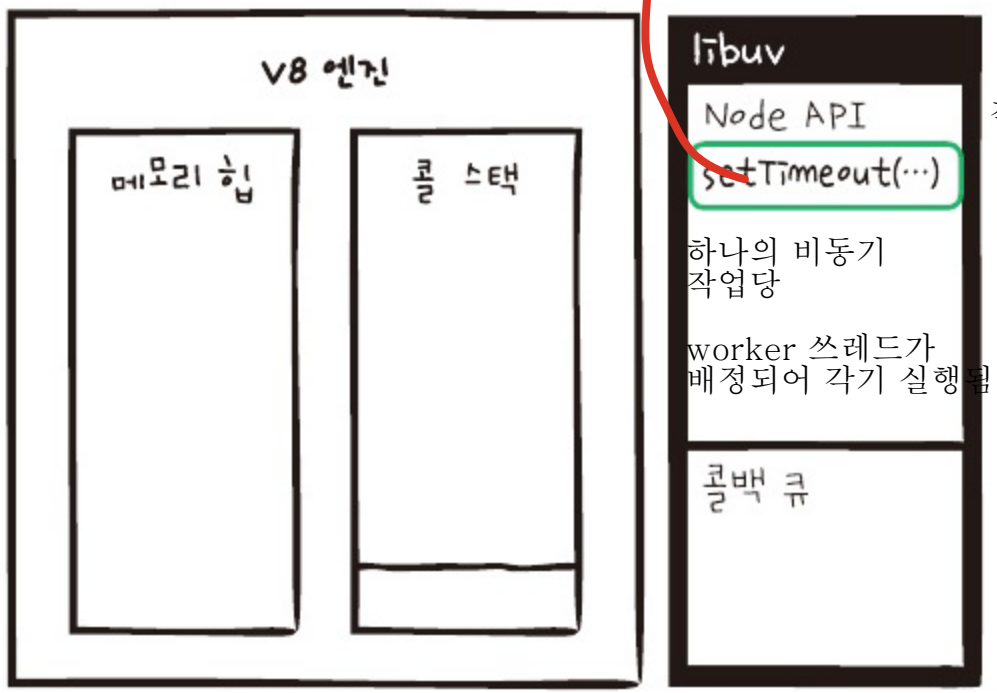
3 이벤트 루프

node api영역에서의 비동기 작업들ㄹ은
독립적으로 병렬적으로
각기 워커 쓰레드들에게 배정받아
실행된다

```
1   console.log("첫 번째 작업");
2   setTimeout(() => {
3     console.log("두 번째 작업");
4   }, 3000);
5   console.log("세 번째 작업");
```

터미널 창

```
첫 번째 작업
```



작업스레드에서 실행함

libuv의 nodeapi
영역으로 이동되고
time 시작

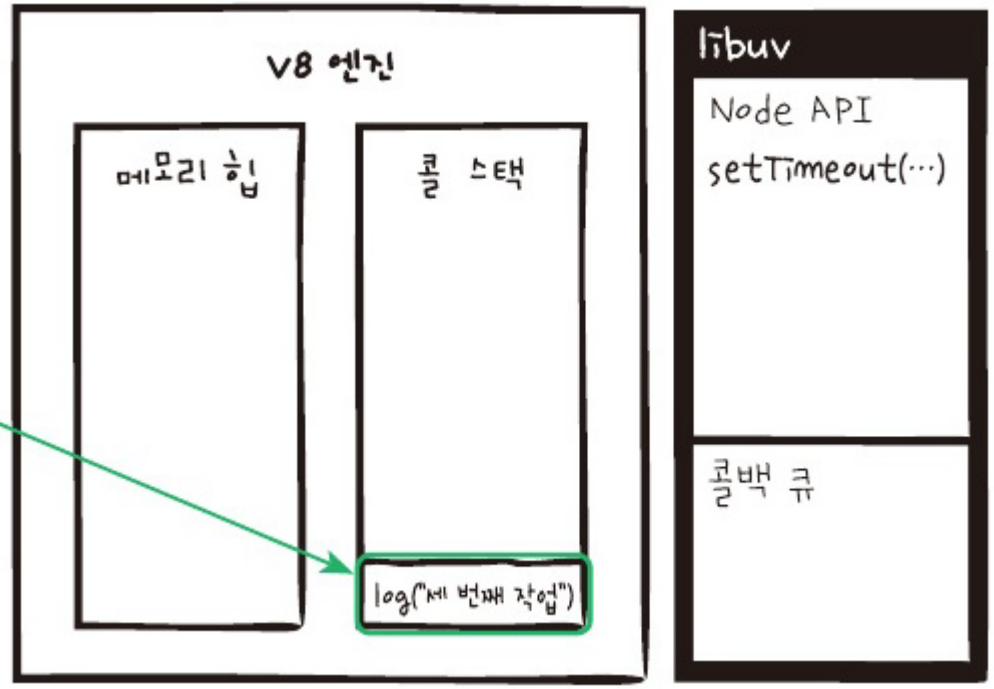
node api에서는 작업쓰레드들이 독립적으로 관리 실행하는데
정책에 따라

작업의 성격, 걸리는 시간에 따라 우선순위가 달라진다.

```
1   console.log("첫 번째 작업");
2   setTimeout(() => {
3     console.log("두 번째 작업");
4   }, 3000);
5   console.log("세 번째 작업");
```

터미널 창

```
첫 번째 작업
```

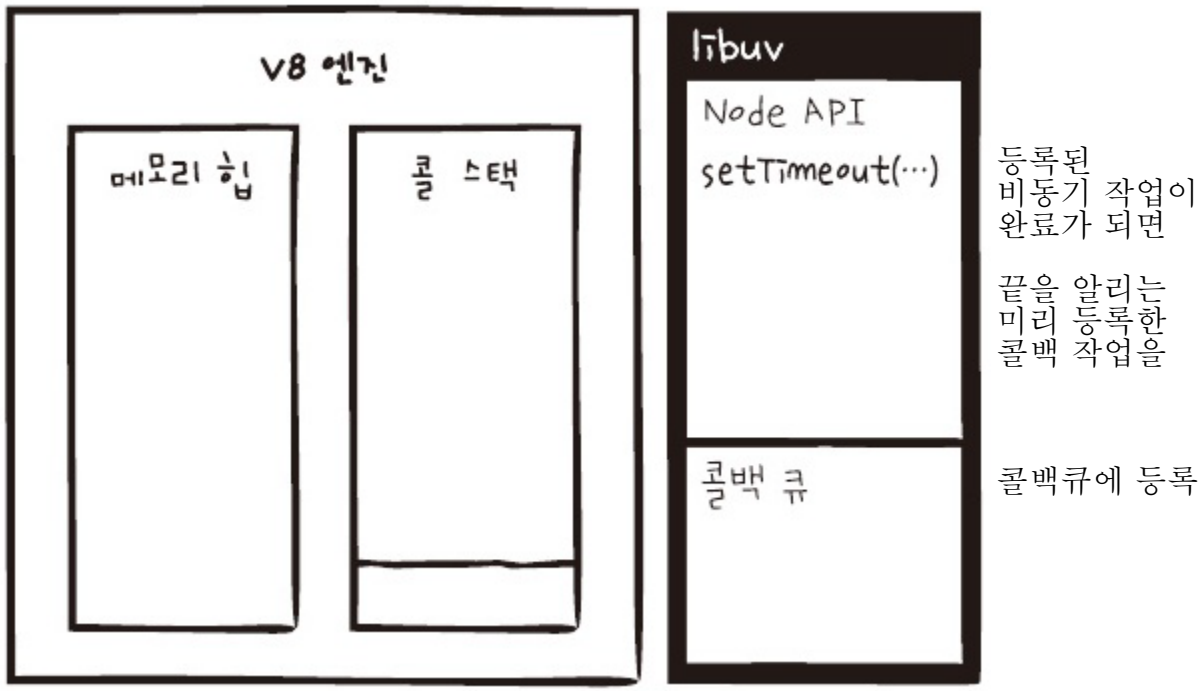


3번째 작업
등록후 바로 실행

```
1   console.log("첫 번째 작업");
2   setTimeout(() => {
3     console.log("두 번째 작업");
4   }, 3000);
5   console.log("세 번째 작업");
```

터미널 창

첫 번째 작업
세 번째 작업



등록된
비동기 작업이
완료가 되면

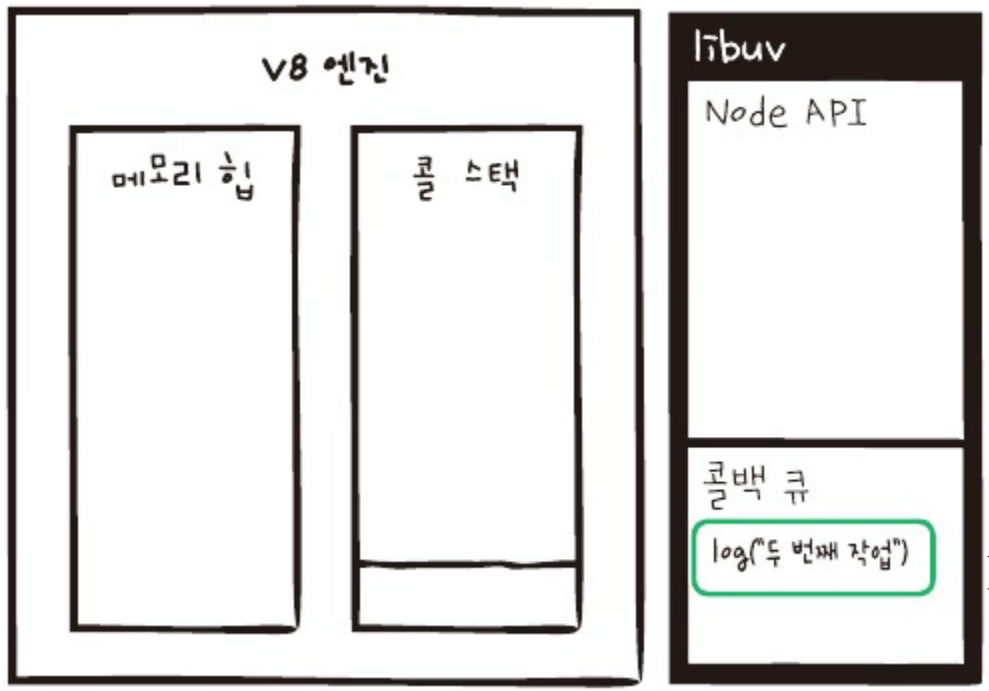
끝을 알리는
미리 등록한
콜백 작업을

콜백큐에 등록

```
1   console.log("첫 번째 작업");
2   setTimeout(() => {
3     console.log("두 번째 작업");
4   }, 3000);
5   console.log("세 번째 작업");
```

터미널 창

```
첫 번째 작업
세 번째 작업
```



콜백 큐에 작업 등록되

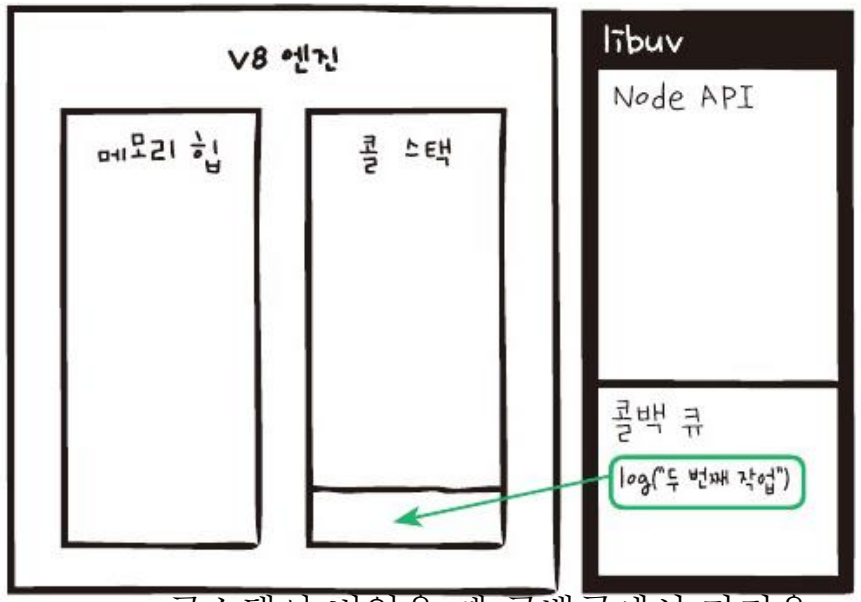
3 이벤트 루프

```
1   console.log("첫 번째 작업");
2   setTimeout(() => {
3     console.log("두 번째 작업");
4   }, 3000);
5   console.log("세 번째 작업");
```

터미널 창

```
첫 번째 작업
세 번째 작업
```

이벤트 루프가 계속 모니터링하고 관리. 작업들을



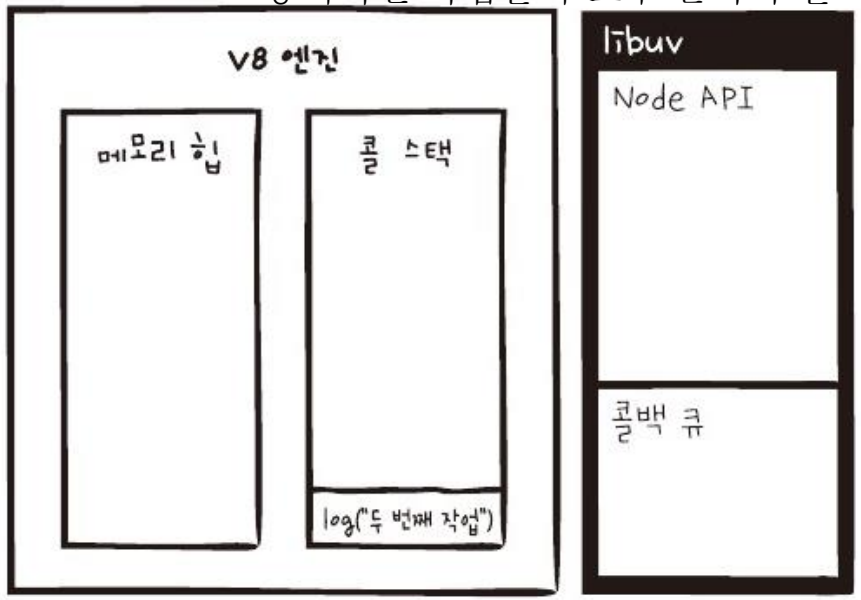
콜백큐에서
콜스택으로
콜백 작업이 옮겨
등록됨

콜스택이 비었을 때 콜백큐에서 가져옴
==>동기적인 작업들이 모두 끝나야 콜백큐를 조회한다는뜻

```
1   console.log("첫 번째 작업");
2   setTimeout(() => {
3     console.log("두 번째 작업");
4   }, 3000);
5   console.log("세 번째 작업");
```

터미널 창

```
첫 번째 작업
세 번째 작업
```



콜스택에 작업이 등록되고
실행됨

libuv영역에서 일어나는
비동기적 작업들을 진행하는 것은
worker thread들이다. 동시에 병렬적으로 진행된다.

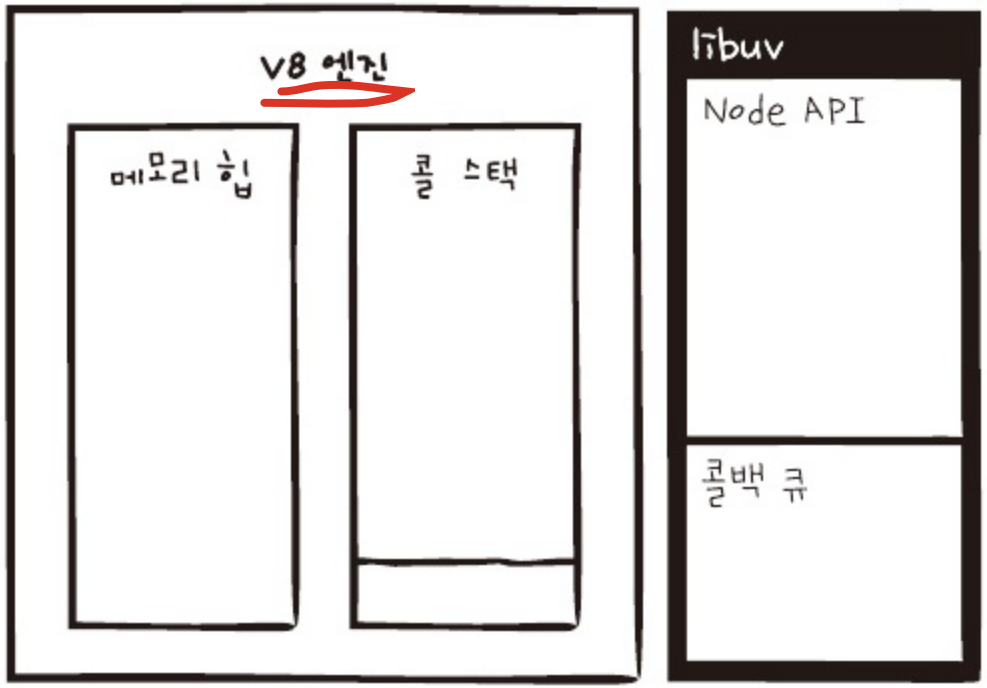
비동기적 작업들과 그 작업들이 끝나고 실행해야하는 콜백 작업과 분리해서 생각해야 한다. 햇갈리지 말자.

끝을 알리는 콜백 작업들ㅇ는 순서대로 큐에 쌓여서 콜 스택에 옮겨져 메인 싱글 스레드가 실행ㅇ한ㄷ.

```
1   console.log("첫 번째 작업");
2   setTimeout(() => {
3     console.log("두 번째 작업");
4   }, 3000);
5   console.log("세 번째 작업");
```

터미널 창

```
첫 번째 작업
세 번째 작업
두 번째 작업
```



콜 스택 libuv의 node api callback que가 다 비어있을 때
종료됨

앞서 말한 이벤트 루프가 하는 일들은 nodejs말고 브라우저에서도 비슷하게 작동한다.
참고로 chrome도 v8엔진을 사용한다

콜백 함수

문제점

- 콜백 함수에서 또다른 비동기 함수 호출을 하게됨
- 콜백 함수가 또 지정됨, 그 콜백 함수에서 또다른 비동기 함수 호출을 하게됨
- 이런 패턴이 반복되면 → 콜백 지옥!!

프라미스 promise

- 비동기 함수에서 프라미스 객체를 리턴
- 비동기 작업이 성공적으로 끝나면 then 함수를 실행
- 오류가 발생하여 실패했을 때는 catch 함수를 실행

- promise를 리턴하는 모듈 함수 가져오기

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

fs.비동기함수()
  .then((result)=> { /* 성공시 실행할 코드 */})
  .catch((err)=>{ /* 실패시 실행할 코드 */})
```

chapter05/sec04/promise.js

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

fs.readdir('./')
  .then((result) => console.log(result))
  .catch((err) => console.error(err));
```

async/await

- ECMA 2017(ES8)부터 도입된 비동기 처리 방법
- 비동기 처리를 하는 함수 앞에 async 키워드 설정
- 비동기 함수 호출시 await를 앞에 붙임
  - 비동기 함수는 반드시 Promise 객체를 리턴해야 함
- 예외 처리는 try-catch 블록으로 처리

→ 비동기 처리지만 동기 처럼 코드를 작성할 수 있음

chapter05/sec04/await.js

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

async function readDirAsyn() {
  try {
    const files = await fs.readdir('./');  // Promise 객체를 리턴하는 비동기 함수
    console.log(files);
  } catch (err) {
    console.error(err);
  }
}

readDirAsyn();
```

노드js의 ch5부터는 서버 구축 관련 내용. 하지만 우리는 java로 할 예정이므로. 여기서 nodjs 마침