1장

1장

1.1 노드의 정의

1.2 노드의 특성

1.3 노드의 역할

1.4 개발 환경 설정하기

1.5 호출 스택, 이벤트 루프

1.6 ES 2015+ 문법

1.7 FrontEnd 자바스크립트

1.1 노드의 정의

1. 노드의 정의

공식 홈페이지의 설명

- Node.js 는 크롬 V8 자바스크립트 엔진으로 빌드된 자바스크립트 런타임
- 비동기 이벤트 주도의 확장성 있는 네트워크 애플리케이션을 만들 수 있는 자바스크립트 런타임
- 주목해야 할 용어는 바로 '자바스크립트 런타임'

  * 자바스크립트 런타임 : 자바스크립트를 실행해 주는 실행 환경

노드는 서버가 아닌가요? 서버라는 말이 없네요.

- 서버의 역할도 수행할 수 있는 자바스크립트 런타임
- 노드로 자바스크립트로 작성된 서버를 실행할 수 있음.
- 서버 실행을 위해 필요한 http/https/http2 모듈을 제공

2. 런타임

Node js : 자바스크립트 런타임

- 런타임: 특정 언어로 만든 프로그램들을 실행할 수 있게 해주는 가상 머신(크롬의 V8 엔진 사용)의 상태
- 즉, 자바스크립트로 만든 프로그램들을 실행할 수 있게 해 줌
- 다른 런타임으로는 웹 브라우저가 있음 [각각 자바스크립트 해석 엔진이 내장]
- 노드 이전에도 자바스크립트 런타임을 만들기 위한 많은 시도들이 있었으나, 엔진 속도 문제로 실패

3. 내부 구조

» 2008년 V8 엔진 출시, 2009년 노드 프로젝트 시작

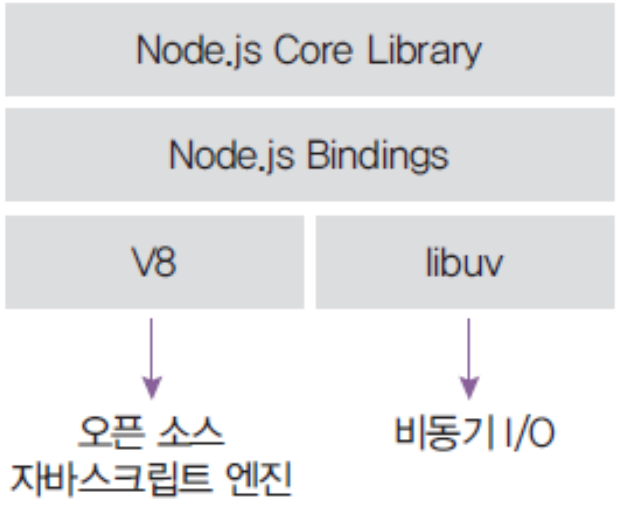
» V8 엔진이란?

- V8 엔진이란 구글이 만든 자바스크립트 기반의 웹 어셈블리 엔진
- 자바스크립트 코드 실행 전에 최적화된 기계어로 컴파일하는 엔진
- 자바스크립트 코드를 인터프리터 방식으로 해석하지 않고 즉시 기계어로 컴파일

» 노드는 V8과 libuv를 내부적으로 포함

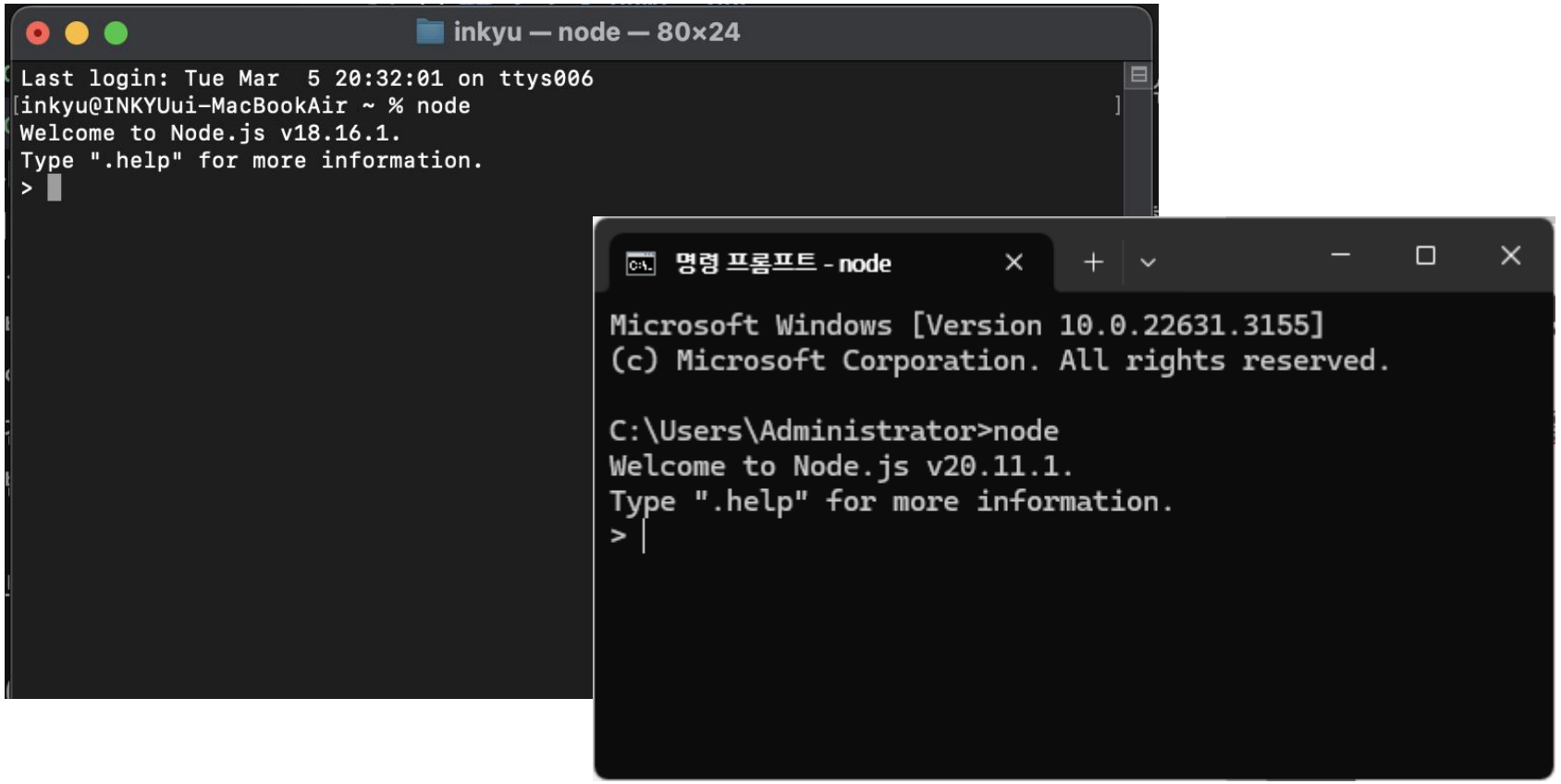
- V8 엔진: 오픈 소스 자바스크립트 엔진] -> 속도 문제 개선
- libuv: 노드의 특성인 이벤트 기반, 논블로킹 I/O 모델을 구현한 라이브러리

그림 1-3 노드의 내부 구조



3. 내부 구조

» Node JS를 이용하면, 브라우저 기반이 아닌 Node.js만으로 자바스크립트 문법 실행이 가능하다.



4. 서버

서비스를 제공하는 컴퓨터, 서버

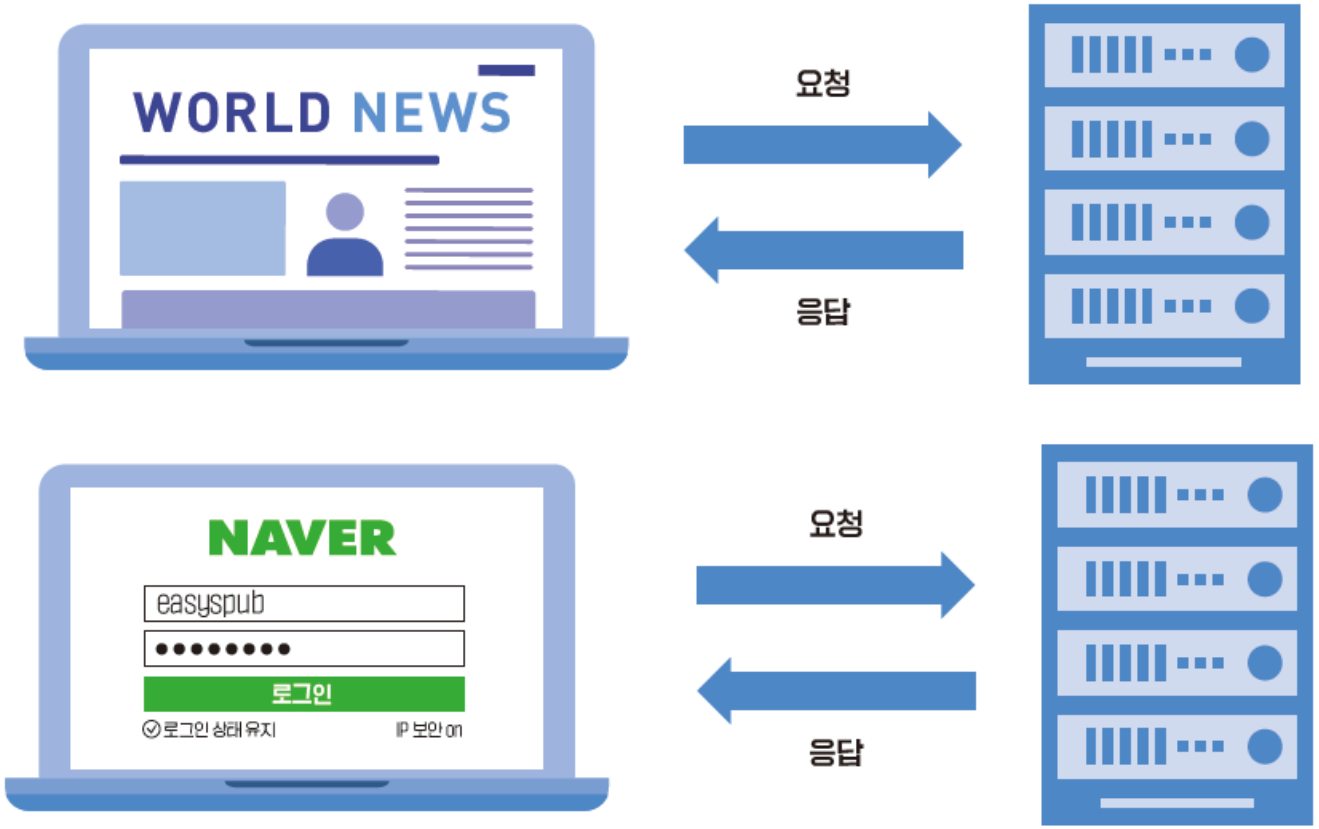
- 테니스, 탁구, 배구 등 서브를 하는 쪽, 또는 그 사람
- 음식물을 제공하기 위해 사용하는 도구
- 네트워크에서 다른 컴퓨터나 소프트웨어와 같은 클라이언트에게 서비스를 제공하는 컴퓨터

- 서버는 '제공한다'는 뜻

4. 서버

서비스를 제공하는 컴퓨터, 서버

- 서버의 역할은 요청이 들어왔을 때 해당 요청에 대한 서비스를 제공하는 것

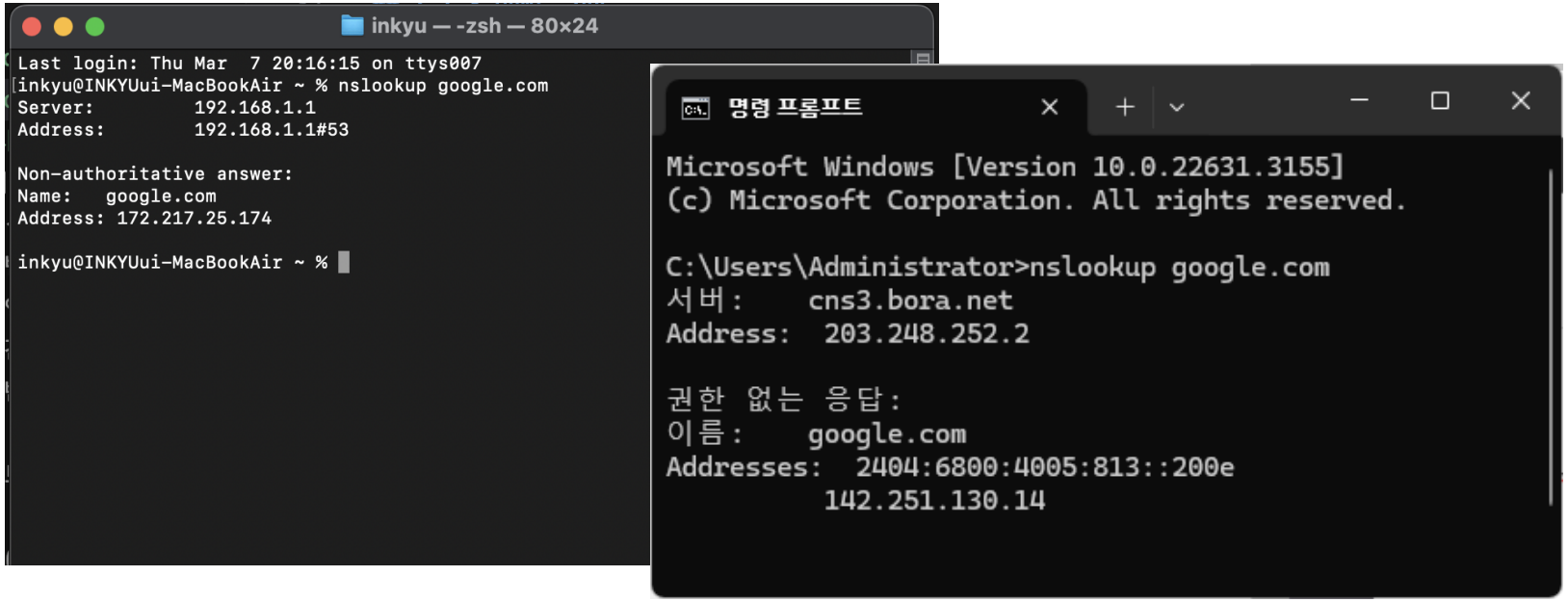


4. 서버

아이피

- 인터넷에 연결되어 있는 모든 장치를 식별할수 있도록 부여되는 고유주소
- 아이피 주소는 32비트의 숫자로 구성

- 명령프롬프트 창 또는 터미널 열기
- 구글의 아이피를 알아보기 위해 'nslookup google.com' 입력



4. 서버

아이피

- 아이피 주소를 브라우저 입력창에 입력하면 사이트로 접속
- 일반적으로 아이피와 매핑된 도메인 주소를 이용해 쉽게 접근

- 친구의 집에 놀러갈 때 보통 누구네 집으로 기억하고 찾아가는 것이지 집 주소를 기억하고 찾아가지 않는 것과 같은 원리
- ‘google.com’으로 기억하고 찾아가는 것이지 ‘172.217.25.174’으로 기억하고 찾아가는 것이 아니라는 뜻

4. 서버

고정 아이피와 유동 아이피

- 고정 아이피
  - 말 그대로 컴퓨터에 고정적으로 부여된 아이피
  - 한번 부여되면 아이피를 반납하기 전까지 다른 장비에 부여할 수 없는 고유 아이피로 보안성이 우수하여 보안이 필요한 업체나 기관에서 사용

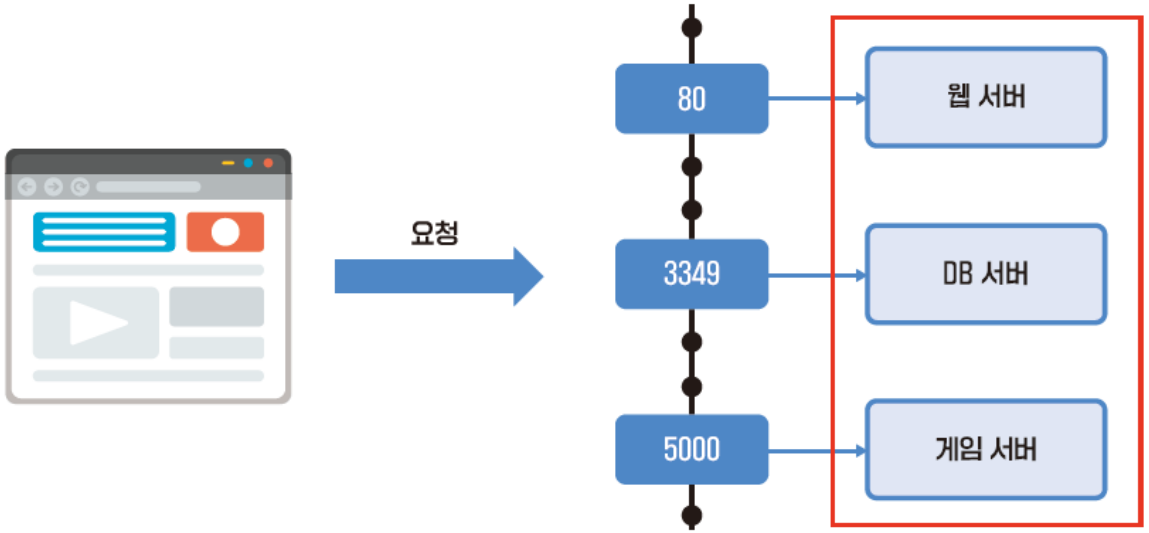
- 유동 아이피
  - 사용자들이 인터넷에 접속하는 매 순간마다 사용하고 있지 않는 아이피 주소를 임시로 발급해 주는 아이피
  - 일반적으로 가정용으로 사용하는 아이피 대부분은 유동 아이피

4. 서버

포트

- 내가 원하는 서버 애플리케이션이 응답할 수 있게 하려면 어떻게 해야 할까?

- 사전적 의미로 '항구'라는 뜻
- 항구의 역할은 선박이 안전하게 정박할 수 있도록 하는 것
- 비슷한 의미로 네트워크에서의 포트 또한 클라이언트 요청의 정확한 도착 지점 설정
- 각각의 서버는 필수적으로 포트 번호를 가짐



1.2 노드의 특성

1. 이벤트 기반

이벤트

- 이벤트란 사전적 의미로 '사건'으로, 사건은 본질적으로 발생하는 것
- 이벤트 기반(event-driven) 시스템은 요청이 발생하는 이벤트를 처리하는 시스템
- 이벤트가 발생하면 이벤트에 대한 처리를 위해 이벤트와 연결된 이벤트 리스너가 존재
- 가독성 향상을 위해 이벤트 리스너에 콜백 함수 등록

1. 이벤트 기반

- 이벤트가 발생할 때 미리 지정해둔 작업을 수행하는 방식
  - 이벤트의 예: 클릭, 네트워크 요청, 타이머 등
  - 이벤트 리스너: 이벤트를 등록하는 함수
  - 콜백 함수: 이벤트가 발생했을 때 실행될 함수

그림 10-1. 이벤트 리스너에서 직접 처리하는 경우

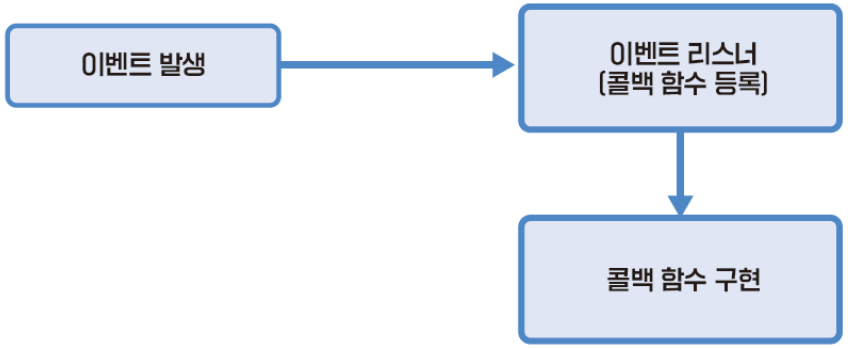


그림 10-2. 이벤트 리스너에 콜백 함수를 등록하여 이벤트 발생 시
이벤트 리스너를 통해 콜백 함수가 호출되도록 하는 경우

2. 논블로킹 I/O

논 블로킹

- Node.js 동작 방식의 가장 큰 특징
- 오래 걸리는 함수를 백그라운드로 보내서 다음 코드가 먼저 실행되게 하고, 나중에 오래 걸리는 함수를 실행
- 논 블로킹 방식 하에서 일부 코드는 백그라운드에서 병렬로 실행됨
- 일부 코드: I/O 작업(파일 시스템 접근, 네트워크 요청), 압축, 암호화 등
- 나머지 코드는 블로킹 방식으로 실행됨
- 그러므로, I/O 작업이 많을 때 노드 활용성이 극대화

그림 1-9 블로킹과 논블로킹

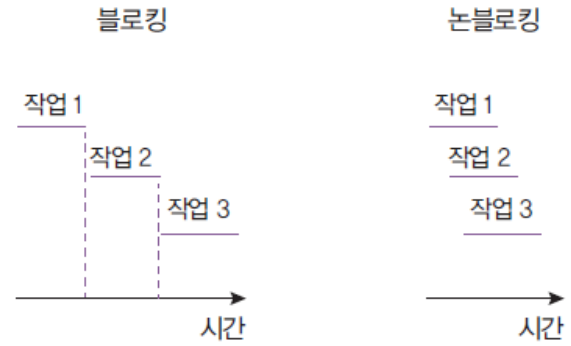
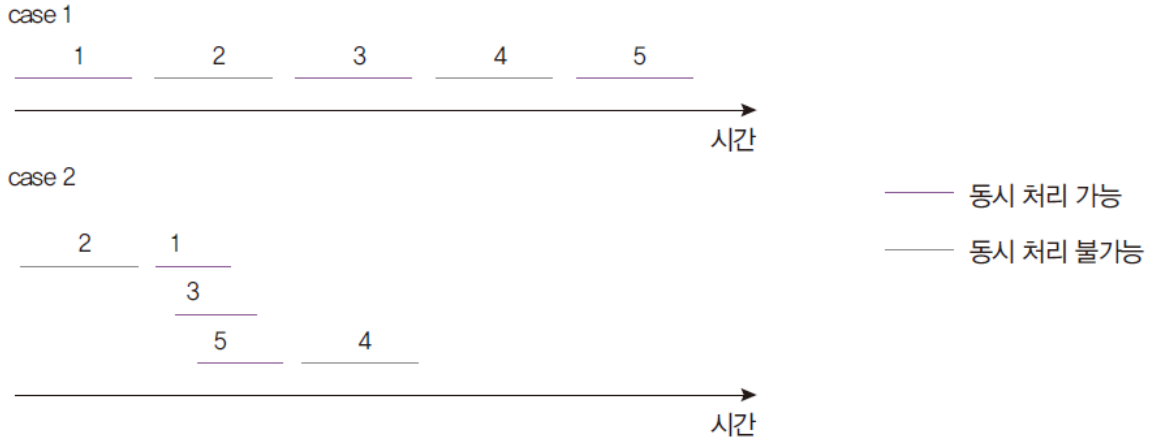


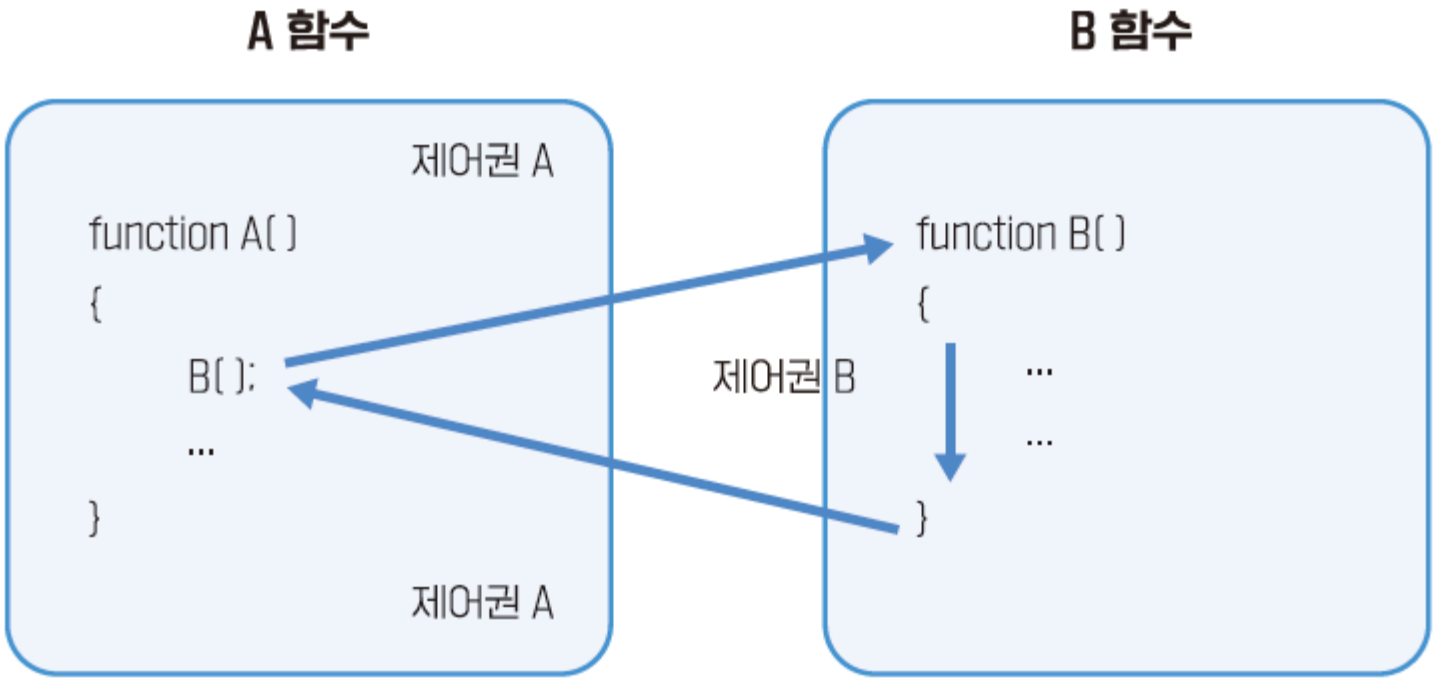
그림 1-10 동시 처리로 얻는 시간적 이득



2. 논블로킹 I/O

블로킹

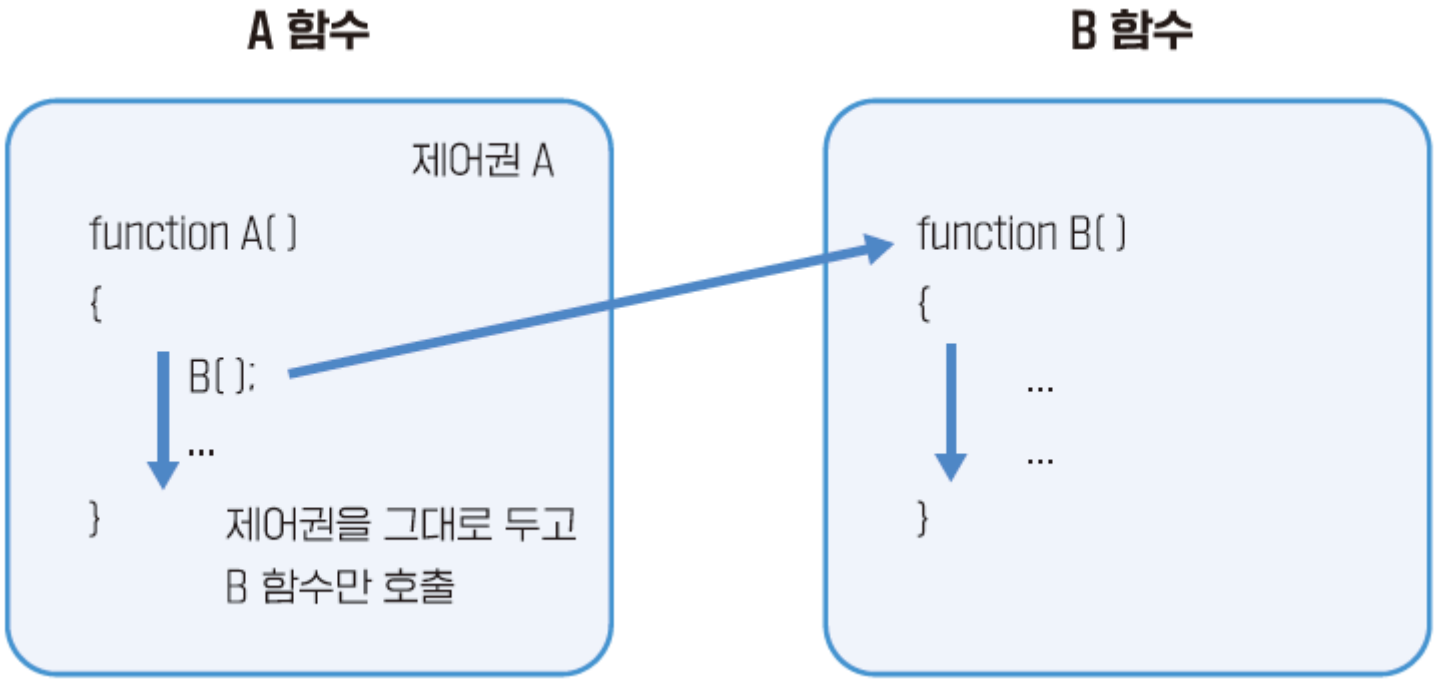
- A와 B 두 개의 함수가 있다고 가정
- 제어권은 코드를 실행할 권한을 의미



2. 논블로킹 I/O

논블로킹

- A와 B 두 개의 함수가 있다고 가정
- 제어권은 코드를 실행할 권한을 의미



3. 프로세스 vs 스레드

- 프로세스와 스레드
  - 프로세스: 운영체제에서 할당하는 작업의 단위, 프로세스 간 자원 공유X
  - 스레드: 프로세스 내에서 실행되는 작업의 단위, 부모 프로세스 자원 공유

- 노드 프로세스는 멀티 스레드이지만 직접 다룰 수 있는 스레드는 하나이기 때문에 싱글 스레드라고 표현

- 노드는 주로 멀티 스레드 대신 멀티 프로세스 활용

- 노드는 14버전부터 멀티 스레드 사용 가능

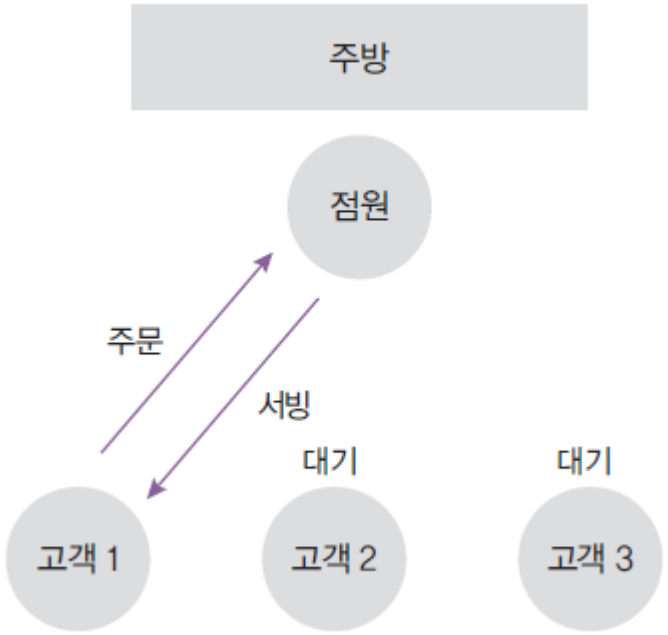
그림 1-13 스레드와 프로세스

4. 싱글 스레드

- 싱글 스레드라 주어진 일을 하나밖에 처리하지 못함
  - 블로킹이 발생하는 경우 나머지 작업은 모두 대기해야 함 -> 비효율 발생

- 주방에 비유(점원: 스레드, 주문: 요청, 서빙: 응답)

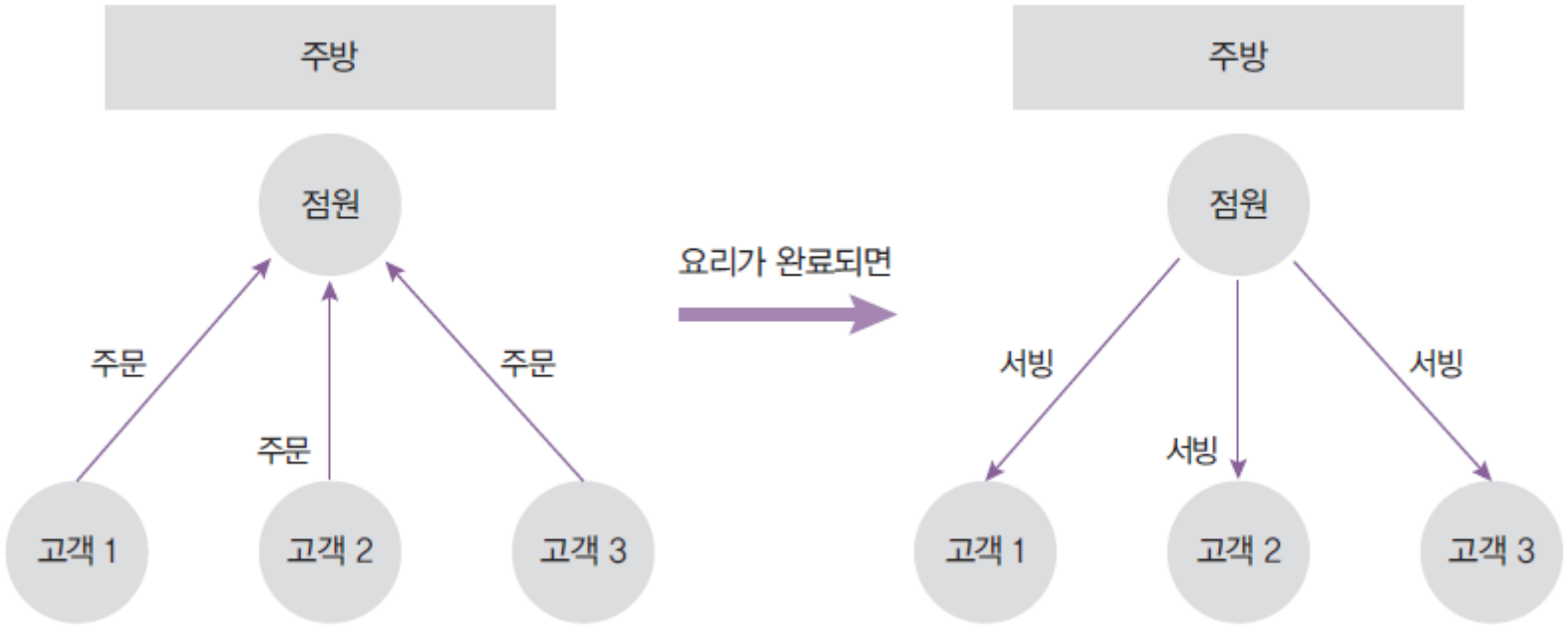
그림 1-10 싱글 스레드, 블로킹 모델



4. 싱글 스레드

- 대신 논 블로킹 모델을 채택하여 일부 코드(I/O)를 백그라운드(다른 프로세스)에서 실행 가능
  - 요청을 먼저 받고, 완료될 때 응답함
  - I/O 관련 코드가 아닌 경우 싱글 스레드, 블로킹 모델과 같아짐

그림 1-11 싱글 스레드, 논블로킹 모델



5. 멀티 스레드 모델과의 비교

- 싱글 스레드 모델은 에러를 처리하지 못하는 경우 멈춤
  - 프로그래밍 난이도 쉽고, CPU, 메모리 자원 적게 사용

- 멀티 스레드 모델은 에러 발생 시 새로운 스레드를 생성하여 극복
  - 단, 새로운 스레드 생성이나 놀고 있는 스레드 처리에 비용 발생
  - 프로그래밍 난이도 어려움
  - 스레드 수만큼 자원을 많이 사용함

- 점원: 스레드, 주문: 요청, 서빙: 응답

그림 1-12 멀티 스레드, 블로킹 모델

6. 멀티 스레드의 활용

노드 14버전

- 멀티 스레드를 사용할 수 있도록 worker_threads 모듈 도입
- CPU를 많이 사용하는 작업인 경우에 활용 가능
- 멀티 프로세싱만 가능했던 아쉬움을 달래줌.

표 1-1 멀티 스레딩과 멀티 프로세싱 비교

멀티 스레딩	멀티 프로세싱
하나의 프로세스 안에서 여러 개의 스레드 사용	여러 개의 프로세스 사용
CPU 작업이 많을 때 사용	I/O 요청이 많을 때 사용
프로그래밍이 어려움	프로그래밍이 비교적 쉬움

1.3 노드의 역할

1. 서버로서의 노드

웹 서버 개발에 최적화되어 있는 Node.js

- 자바스크립트의 V8을 분리하여 독립적으로 실행할 수 있도록 만든 것
- 크롬에서 사용되는 V8 엔진을 웹 서버 등 다른 환경에서도 활용할 수 있게 가능
  - 비동기 이벤트 주도 자바스크립트 런타임으로써 Node.js는 확장성 있는 네트워크 애플리케이션을 만들 수 있도록 설계되어 있다.

- Node.js의 본질은 독립적인 자바스크립트 실행기로, 플랫폼에 상관없이 자바스크립트 코드 기반이면 어떤 것이든 실행시킬 수 있는 것
- Node.js는 웹 서버 개발에 최적화되어 있다는 의미
- 개발에 최적화되어 있는 Node.js로 서버를 만드는 것

1. 서버로서의 노드

노드 서버의 장단점

표 1-1 노드의 장단점

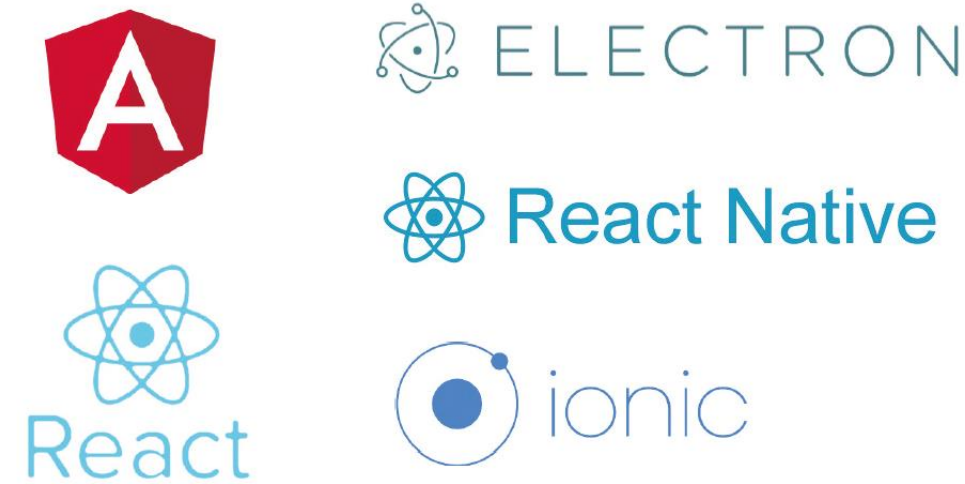
장점	단점
멀티 스레드 방식에 비해 컴퓨터 자원을 적게 사용함	싱글 스레드라서 CPU 코어를 하나만 사용함
I/O 작업이 많은 서버로 적합	CPU 작업이 많은 서버로는 부적합
멀티 스레드 방식보다 쉬움	하나뿐인 스레드가 멈추지 않도록 관리해야 함
웹 서버가 내장되어 있음	서버 규모가 커졌을 때 서버를 관리하기 어려움
자바스크립트를 사용함	어중간한 성능
JSON 형식과 호환하기 쉬움	

- CPU 작업을 위해 AWS Lambda나 Google Cloud Functions같은 별도 서비스 사용
- 페이팔, 넷플릭스, 나사, 월마트, 링크드인, 우버 등에서 메인 또는 서브 서버로 사용

2. 서버 외의 노드

- 자바스크립트 런타임이기 때문에 용도가 서버에만 한정되지 않음
- 웹, 모바일, 데스크탑 애플리케이션에도 사용
  - 웹 프레임워크: Angular, React, Vue, Meteor 등
  - 모바일 앱 프레임워크: React Native
  - 데스크탑 개발 도구: Electron(Atom, Slack, VSCode, Discord 등 제작)
- 위 프레임워크가 노드 기반으로 동작함

그림 1-16 노드 기반의 개발 도구



1.4 개발 환경 설정하기

1. 노드 설치하기

- 노드 설치
  - https://nodejs.org 접속
  - LTS 버전 설치
  - LTS는 안정된 버전, Current는 최신 버전(실험적)

1. 노드 설치하기

- 리눅스(우분투 20 LTS 기준)
  - 터미널에 아래 코드 입력

콘솔

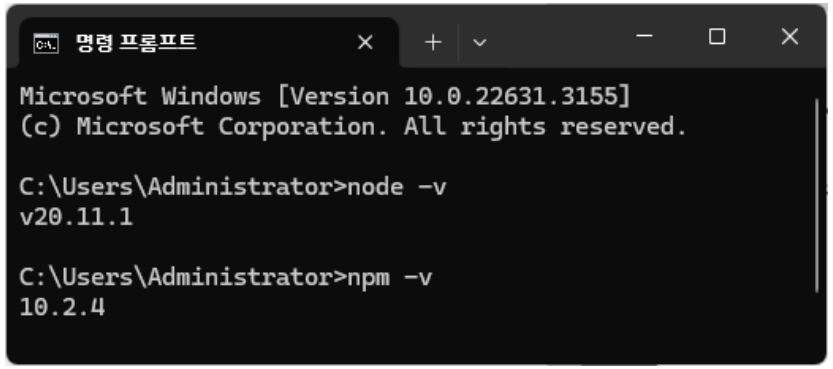
```
$ sudo apt-get update
$ sudo apt-get install -y build-essential
$ sudo apt-get install -y curl
$ curl -sL https://deb.nodesource.com/setup_18.x ¦ sudo -E bash --
$ sudo apt-get install -y nodejs
```

1. 노드 설치하기

- 설치 완료 후 윈도, 맥, 리눅스 모두 명령 프롬프트나 터미널 실행 후 다음 명령어 입력

콘솔

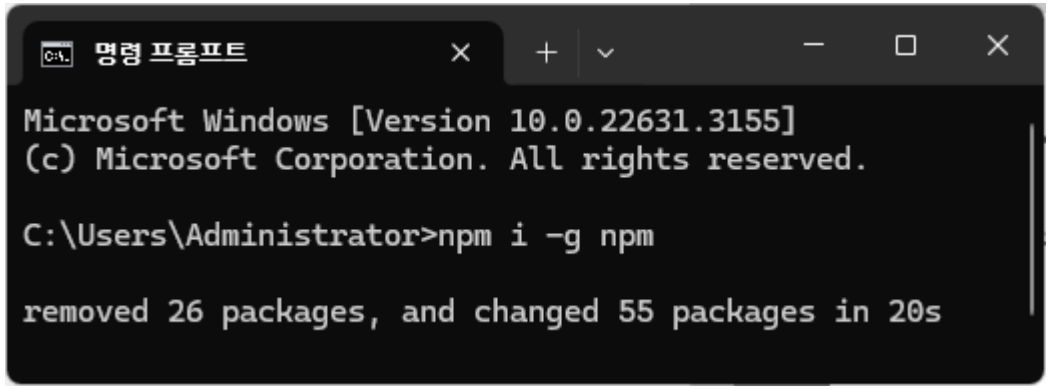
```
$ node -v
18.7.0
$ npm -v
8.15.0
```



- 버전은 다를 수 있지만 버전이 뜨면 설치 성공
  - npm 버전을 업데이트 하려면 다음 명령어 입력
  - 맥과 리눅스는 명령어 앞에 sudo 필요

콘솔

```
$ npm install -g npm
```



2. VS Code 설치하기

- VS Code: 마이크로소프트에서 제공하는 오픈 소스 코드 에디터
  - 자바스크립트, 노드에 대한 지원이 탁월함
  - 윈도, 맥, 리눅스(GUI) 모두 https://code.visualstudio.com 접속

1.5 호출 스택, 이벤트 루프

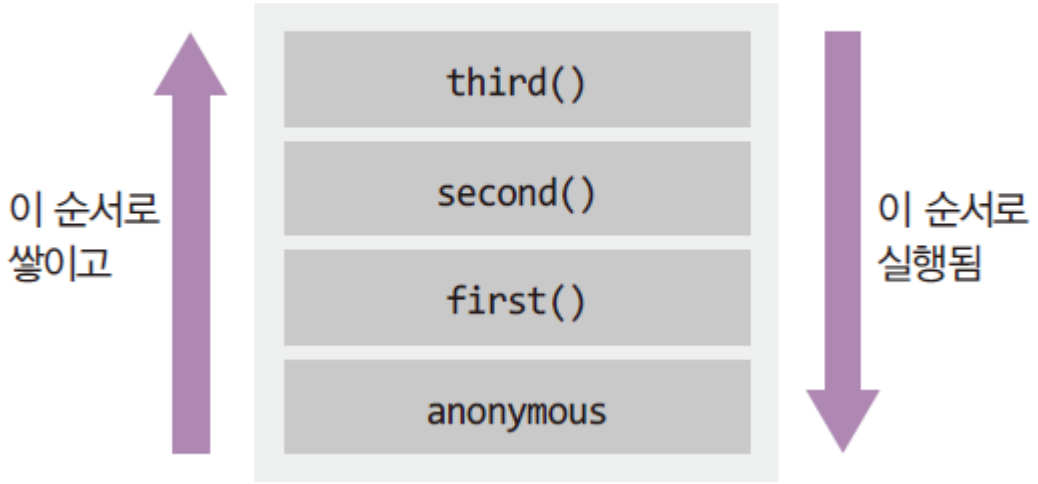
1. 호출 스택

```
let first = () => {
    second();
    console.log('첫번째');
}
let second = () => {
    third();
    console.log('두번째');
}
let third = () => {
    console.log('세번째');
}
first();
```

- 위 코드의 순서 예측해보기
  - 세 번째 -> 두 번째 -> 첫 번째
- 쉽게 파악하는 방법: 호출 스택 그리기

1. 호출 스택

그림 1-5 호출 스택



- 호출 스택(함수의 호출, 자료구조의 스택)
  - Anonymous은 가상의 전역 컨텍스트(항상 있다고 생각하는게 좋음)
  - 함수 호출 순서대로 쌓이고, 역순으로 실행됨
  - 함수 실행이 완료되면 스택에서 빠짐
  - LIFO 구조라서 스택이라고 불림

1. 호출 스택

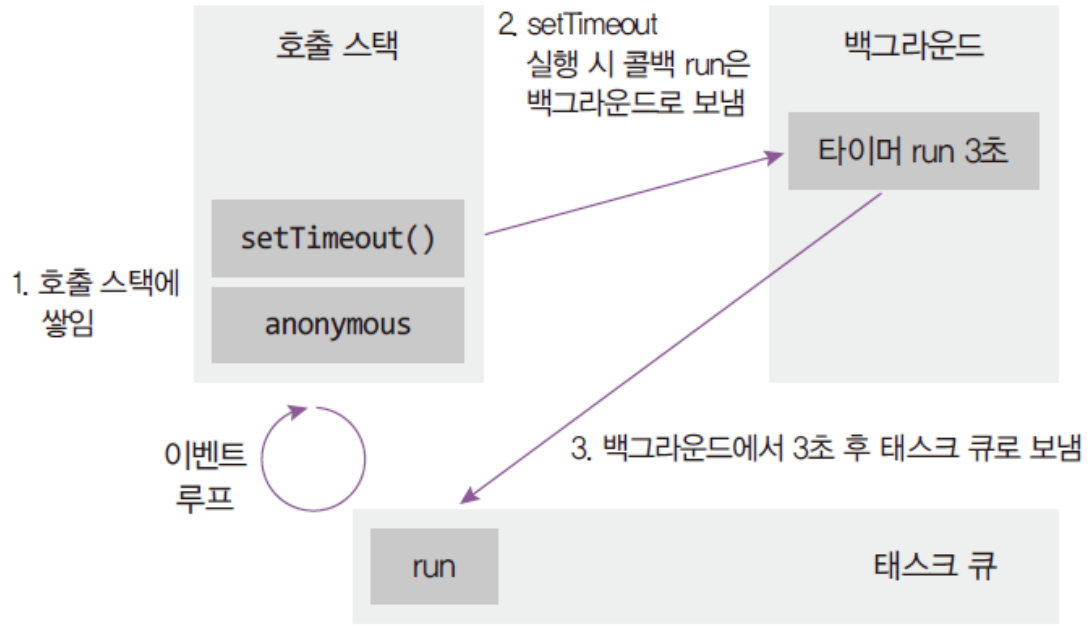
```
let run = () => console.log('3초 후 실행');
console.log('시작');
setTimeout(run, 3000);
console.log('끝');
```

» 위 코드의 순서 예측해보기

- 시작 -> 끝 -> 3초 후 실행
- 호출 스택만으로는 설명이 안 됨(run은 호출 안 했는데?)
- 호출 스택 + 이벤트 루프로 설명할 수 있음

2. 이벤트 루프

그림 1-6 이벤트 루프 1

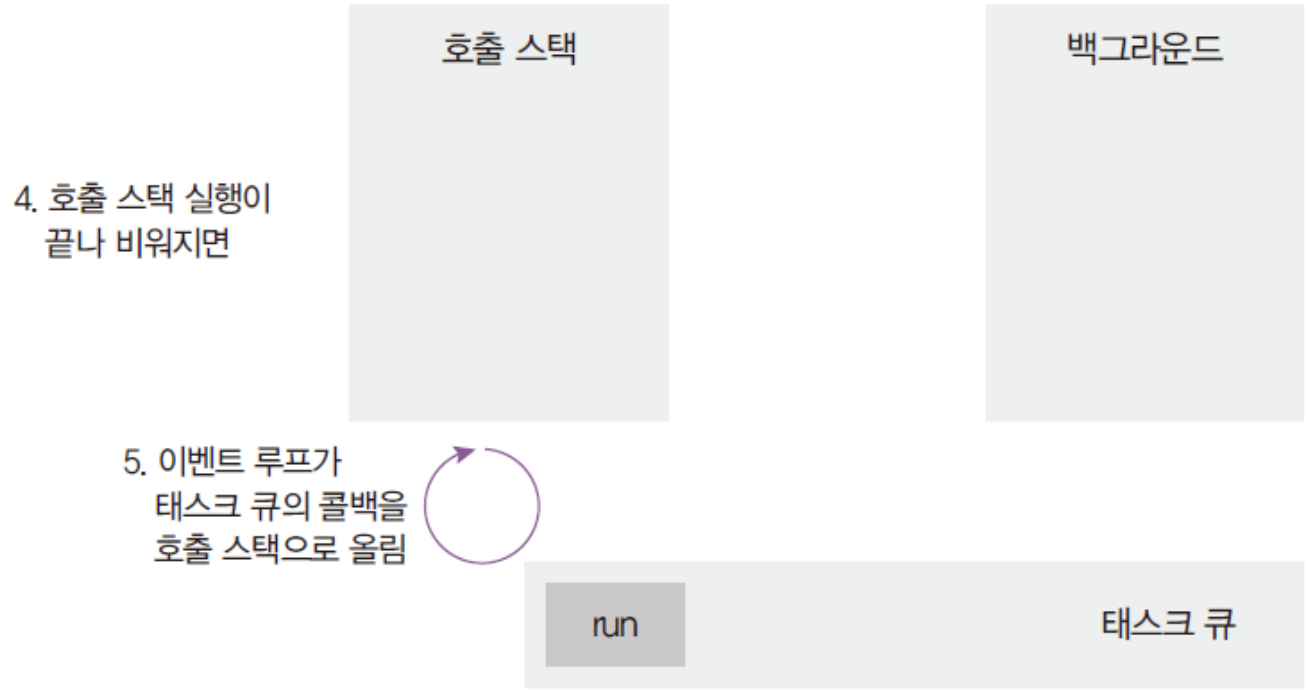


이벤트루프 구조

- 이벤트 루프: 이벤트 발생(setTimeout 등) 시 호출할 콜백함수(run)들을 관리하고, 호출할 순서를 결정하는 역할
- 태스크 큐: 이벤트 발생 후 호출되어야 할 콜백함수들이 순서대로 기다리는 공간
- 백그라운드: 타이머나 I/O 작업 콜백, 이벤트 리스너들이 대기하는 공간. 여러 작업이 동시에 실행될 수 있음

2. 이벤트 루프

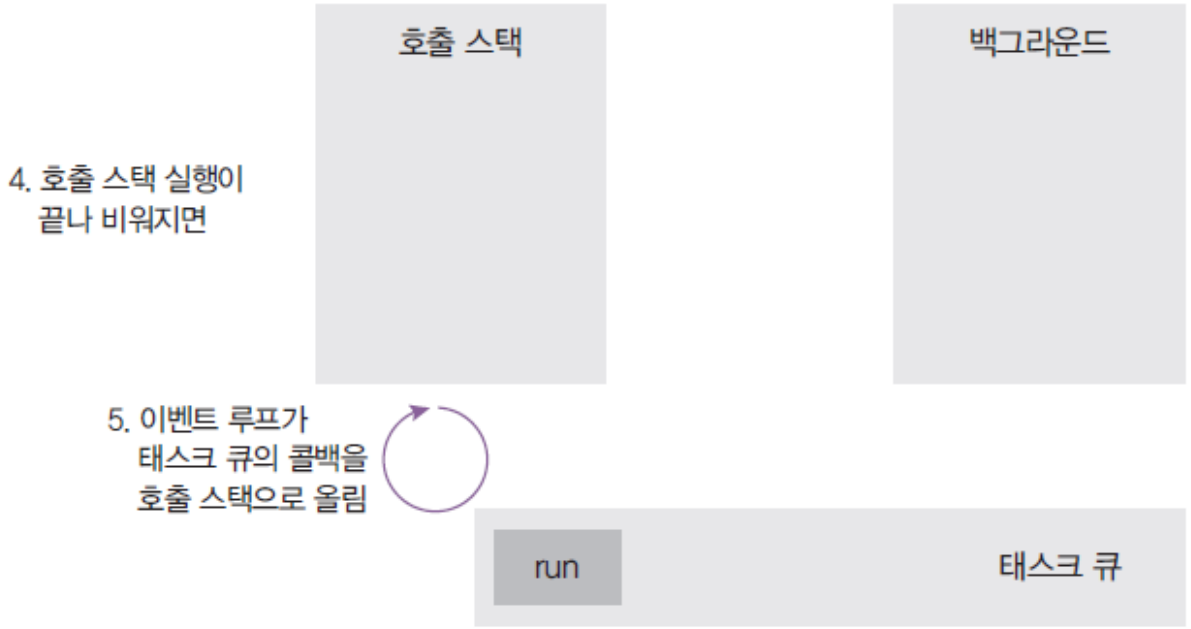
그림 1-7 이벤트 루프 2



- 예제 코드에서 setTimeout이 호출될 때 콜백 함수 run은 백그라운드로
  - 백그라운드에서 3초를 보냄
  - 3초가 다 지난 후 백그라운드에서 태스크 큐로 보내짐

2. 이벤트 루프

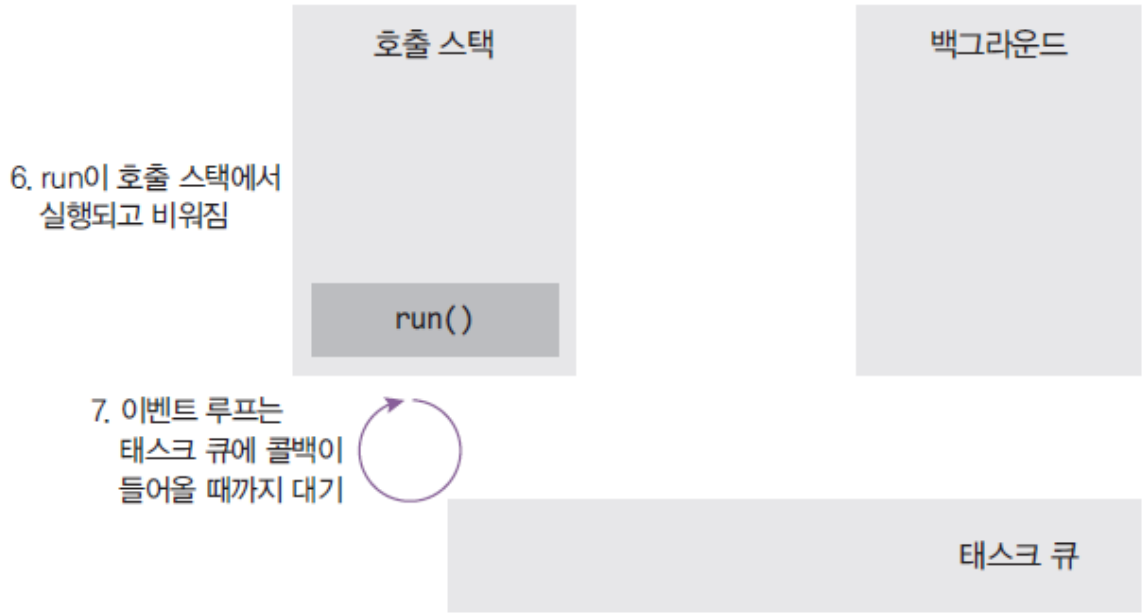
그림 1-7 이벤트 루프 2



- setTimeout과 anonymous가 실행 완료된 후 호출 스택이 완전히 비워지면,
- 이벤트 루프가 태스크 큐의 콜백을 호출 스택으로 올림
  - 호출 스택이 비워져야만 올림
  - 호출 스택에 함수가 많이 차 있으면 그것들을 처리하느라 3초가 지난 후에도 run 함수가 태스크 큐에서 대기
    -> 타이머가 정확하지 않을 수 있는 이유

2. 이벤트 루프

그림 1-8 이벤트 루프 3



- run이 호출 스택에서 실행되고, 완료 후 호출 스택에서 나감
  - 이벤트 루프는 태스크 큐에 다음 함수가 들어올 때까지 계속 대기
  - 태스크 큐는 실제로 여러 개고, 태스크 큐들과 함수들 간의 순서를 이벤트 루프가 결정함

1.6 ES2015+

1. const, let

- ES2015 이전에는 var로 변수를 선언
  - ES2015부터는 const와 let이 대체
  - 가장 큰 차이점: 블록 스코프(var은 함수 스코프)

```
if (true) {
    var num1 = 3;
}
console.log(num1); // 3

if (true) {
    const num2 = 3;
}
console.log(num2);// ReferenceError : num2 is not defined
```

- 기존: 함수 스코프(function() {}이 스코프의 기준점)
  - 다른 언어와는 달리 if나 for, while은 영향을 미치지 못함
  - const와 let은 함수 및 블록({})에도 별도의 스코프를 가짐

1. const, let

```
const num1 = 0;
num1 = 1; // TypeError: Assignment to constant variable.

let num2 = 0;
num2 = 1; // 1

const c; // SyntaxError: Missing initializer in const
declaration
```

» const는 상수

- 상수에 할당한 값은 다른 값으로 변경 불가
- 변경하고자 할 때는 let으로 변수 선언
- 상수 선언 시부터 초기화가 필요함
- 초기화를 하지 않고 선언하면 에러

2. 템플릿 문자열

- 문자열을 합칠 때 + 기호때문에 지저분함
  - ES2015부터는 ' (백틱) 사용 가능
  - 백틱 문자열 안에 ${변수} 처럼 사용

```
var num1 = 1;
var num2 = 2;
var result = num1 + num2;
var string1 = num1 + " 더하기 " + num2 + " 는 '" + result + "'";
console.log(string1); // 1 더하기 2 는 '3'
```

```
const num3 = 1;
const num4 = 2;
const result2 = num3 + num4;
const string2 = '${num3} 더하기 ${num4} 는 '${result2}'';
console.log(string2); // 1 더하기 2 는 '3'
```

3. 객체 리터럴

- ES5 시절의 객체 표현 방법
  - 속성 표현 방식에 주목

```
var sayHello = function() {
    console.log('안녕하세요');
}
var subject  = 'Node';
var person = {
    sayBye: function() {
        console.log('안녕히 가세요');
    },
    sayNode: sayNode,
};
person[subject + 'Info'] = 'Node.js는 자바스크립트 런타임';
person.sayHello(); // 안녕하세요
person.sayBye(); // 안녕히 가세요
console.log(person.subjectInfo); // Node.js는 자바스크립트 런타임
```

3. 객체 리터럴

- 훨씬 간결한 문법으로 객체 리터럴 표현 가능
  - 객체의 메서드에 :function을 붙이지 않아도 됨
  - { sayNode: sayNode }와 같은 것을 { sayNode }로 축약 가능
  - [변수 + 값] 등으로 동적 속성명을 객체 속성 명으로 사용 가능

```
var sayHello = () => console.log('안녕하세요');

const newPerson = {
    sayBye() {
        console.log('안녕히 가세요');
    },
    sayHello,
    [subject+Info]: 'Node.js는 자바스크립트 런타임',
}
newPerson.sayHello(); // 안녕하세요
newPerson.sayBye(); // 안녕히 가세요
console.log(newPerson.subjectInfo); // Node.js는 자바스크립트 런타임
```

4. 화살표 함수

- add1, add2, add3, add4는 같은 기능을 하는 함수
  - add2: add1을 화살표 함수로 나타낼 수 있음
  - add3: 함수의 본문이 return만 있는 경우 return 생략
  - add4: return이 생략된 함수의 본문을 소괄호로 감싸줄 수 있음
  - not1과 not2도 같은 기능을 함(매개변수 하나일 때 괄호 생략)

```
function add1(x, y) {
    return x + y;
}

const add2 = (x, y) => {
    return x + y;
}

const add3 = (x, y) => x + y;
const add4 = (x, y) => (x + y);
```

```
function not1(x) {
    return !x;
}

const not2 = x => !x;
```

4. 화살표 함수

```
var relationship1 = {
    name: '짱구',
    friends: ['철수', '유리', '맹구'],
    printFriends: function () {
        var that = this;
        this.friends.forEach(function (friend) {
            console.log(that.name+'의 친구', friend);
        });
    }
};
relationship1.printFriends();
```

- 화살표 함수가 기존 function() {}을 대체하는 건 아님(this가 달라짐)
  - printFriends 메서드의 this 값에 주목
  - forEach의 function의 this와 printFriends의 this는 다름
  - that이라는 중간 변수를 이용해서 printFriends의 this를 전달

4. 화살표 함수

```
const relationship2 = {
    name: '짱구',
    friends: ['철수', '유리', '맹구'],
    printFriends() {
        this.friends.forEach(friend => {
            console.log(this.name+'의 친구', friend);
        });
    }
};
relationship2.printFriends();
```

- forEach의 인자로 화살표 함수가 들어간 것에 주목
  - forEach의 화살표함수의 this와 printFriends의 this가 같아짐
  - 화살표 함수는 자신을 포함하는 함수의 this를 물려받음
  - 물려받고 싶지 않을 때: function() {}을 사용

4. 화살표 함수

```
const relationship2 = {
    name: '짱구',
    friends: ['철수', '유리', '맹구'],
    printFriends: () => {
        this.friends.forEach(friend => {
            console.log(this.name+'의 친구', friend);
        });
    }
};
relationship2.printFriends();
```

5. 구조분해 할당

```
var person = {
    status: {
        name: '짱구',
        age: 5,
    },
    getAge () {
        this.status.count++;
        return this.status.count;
    }
};
var getAge = person.getAge;
var name = person.status.name;
var age = person.status.age;
```

- var getAge, name, age에 주목
  - person부터 시작해서 속성을 찾아 들어가야 함

5. 구조분해 할당

```
const person = {
    status: {
        name: '짱구',
        age: 5,
    },
    getAge () {
        this.status.count++;
        return this.status.count;
    }
};

const { getAge, status: { name, age } } = person;
```

- const { 변수 } = 객체;로 객체 안의 속성을 변수명으로 사용 가능
  - 단, getAge()를 실행했을 때 결과가 person.getAge()와는 달라지므로 주의
- name, age처럼 속성 안의 속성도 변수명으로 사용 가능

5. 구조분해 할당

- 배열도 구조분해 할당 가능

```
var array = ['node.js', {}, 10, true];
var node = array[0];
var obj = array[1];
var bool = array[3];
```

```
const array = ['node.js', {}, 10, true];
const [node, obj, , bool] = array;
```

- const [변수] = 배열; 형식
  - 각 배열 인덱스와 변수가 대응됨
  - node는 array[0], obj = array[1], bool = array[3]

6. 클래스

프로토타입 문법을 깔끔하게 작성할 수 있는 Class 문법 도입

- Constructor(생성자), Extends(상속) 등을 깔끔하게 처리할 수 있음
- 코드가 그룹화되어 가독성이 향상됨.

```
var Human = function(type) {
    this.type = type || 'human';
}

Human.isHuman = function(human) {
    return human instanceof Human;
}

Human.prototype.breathe = function() {
    alert('h-a-a-a-m');
}

var Choi = function(type, firstName, lastName) {
    Human.apply(this, arguments);
    this.firstName = firstName;
    this.lastName = lastName;
}
```

```
Choi.prototype = Object.create(Human.prototype);

Choi.prototype.constructor = Choi;

Choi.prototype.sayName = function() {
    alert(this.firstName + ' ' + this.lastName);
}

var choiInKyu = new Choi('human', 'In Kyu', 'Choi');

Human.isHuman(choiInKyu); // true
```

6. 클래스

- 전반적으로 코드 구성이 깔끔해짐
  - Class 내부에 관련된 코드들이 묶임
  - Super로 부모 Class 호출
  - Static 키워드로 클래스 메서드 생성

```
class Human {
    constructor(type = 'human') {
        this.type = type;
    }

    static isHuman(human) {
        return human instanceof Human;
    }

    breathe() {
        alert('h-a-a-a-m');
    }
}
```

```
class Choi extends Human {
    constructor(type, firstName, lastName) {
        super(type);
        this.firstName = firstName;
        this.lastName = lastName;
    }

    sayName() {
        super.breathe();
        alert('${this.firstName} ${this.lastName}');
    }
}

const choiInKyu = new Zero('human', 'In Kyu', 'Choi');

Human.isHuman(choiInKyu); // true
```

7. 프로미스

- 콜백 헬이라고 불리는 지저분한 자바스크립트 코드의 해결책
  - 프로미스: 내용이 실행은 되었지만 결과를 아직 반환하지 않은 객체
  - Then을 붙이면 결과를 반환함
  - 실행이 완료되지 않았으면 완료된 후에 Then 내부 함수가 실행됨
  - Resolve(성공리턴값) -> then으로 연결
  - Reject(실패리턴값) -> catch로 연결
  - Finally 부분은 무조건 실행됨

```
const condition = true; // true면 resolve, false면 reject

const promise = new Promise((resolve, reject) => {
    if (condition) {
        resolve('성공');
    } else {
        reject('실패');
    }
});

promise
    .then((message) => {
        console.log(message); // 성공(resolve)한 경우 실행
    })
    .catch((error) => {
        console.error(error); // 실패(reject)한 경우 실행
    })
    .finally(() => { // 끝나고 무조건 실행
        console.log('무조건');
    });
```

7. 프로미스

프로미스의 then 연달아 사용 가능(프로미스 체이닝)

- then 안에서 return한 값이 다음 then으로 넘어감
- return 값이 프로미스면 resolve 후 넘어감
- 에러가 난 경우 바로 catch로 이동
- 에러는 catch에서 한 번에 처리

```
promise
.then((message) => {
    return new Promise((resolve, reject) => {
        resolve(message);
    });
})
.then((message2) => {
    console.log(message2);
    return new Promise((resolve, reject) => {
        resolve(message2);
    });
})
.then((message3) => {
    console.log(message3);
})
.catch((error) => {
    console.error(error);
});
```

7. 프로미스

» 콜백 패턴(3중첩)을 프로미스로 바꾸는 예제

```
function findAndSaveUser(Users) {
    Users.findOne({}, (err, user) => { // 첫 번째 콜백
        if (err) {
            return console.error(err);
        }
        user.name = 'choi';
        user.save((err) => { // 두 번째 콜백
            if (err) {
                return console.error(err);
            }
            Users.findOne({ gender: 'm' }, (err, user) => { // 세 번째 콜백
                // 생략
            });
        });
    });
}
```

7. 프로미스

- findOne, save 메서드가 프로미스를 지원한다고 가정
  - 지원하지 않는 경우 프로미스 사용법은 3장에 나옴

```
function findAndSaveUser(Users) {
    Users.findOne({})
    .then((user) => {
        user.name = 'choi';
        return user.save();
    })
    .then((user) => {
        return Users.findOne({ gender: 'm' });
    })
    .then((user) => {
        // 생략
    })
    .catch(err => {
        console.error(err);
    });
}
```

7. 프로미스

- Promise.resolve(성공리턴값): 바로 resolve하는 프로미스
- Promise.reject(실패리턴값): 바로 reject하는 프로미스

```
const promise1 = Promise.resolve('성공1');
const promise2 = Promise.resolve('성공2');
Promise.all([promise1, promise2])
    .then((result) => {
        console.log(result); // ['성공1', '성공2']
    })
    .catch((error) => {
        console.error(error);
    });
```

- Promise.all(배열): 여러 개의 프로미스를 동시에 실행
  - 하나라도 실패하면 catch로 감
  - allSettled로 실패한 것만 추려낼 수 있음

8. async/await

- 이전 챕터의 프로미스 패턴 코드
  - Async/await으로 한 번 더 축약 가능

```
function findAndSaveUser(Users) {
    Users.findOne({})
        .then((user) => {
            user.name = 'choi';
            return user.save();
        })
        .then((user) => {
            return Users.findOne({ gender : 'm' });
        })
        .then((user) => {
            // 생략
        })
        .catch(err => {
            console.error(err);
        })
}
```

8. async/await

- async function의 도입
  - 변수 = await 프로미스;인 경우 프로미스가 resolve된 값이 변수에 저장
  - 변수 await 값;인 경우 그 값이 변수에 저장

```
async function findAndSaveUser(Users) {
    let user = await Users.findOne({});
    user.name = 'choi';
    user = await user.save();
    user = await Users.findOne({ gender : 'm' });
    // 생략
}
```

8. async/await

- 에러 처리를 위해 try catch로 감싸주어야 함
  - 각각의 프로미스 에러 처리를 위해서는 각각을 try catch로 감싸주어야 함

```
async function findAndSaveUser(Users) {
    try {
        let user = await Users.findOne({});
        user.name = 'choi';
        user = await user.save();
        user = await Users.findOne({ gender : 'm' });
    // 생략
    } catch (error) {
        console.error(error);
    }
}
```

8. async/await

- 화살표 함수도 async/await 가능

```
const findAndSaveUser = async (Users) => {
    try {
        let user = await Users.findOne({});
        user.name = 'choi';
        user = await user.save();
        user = await Users.findOne({ gender : 'm' });
        // 생략
    } catch (error) {
        console.error(error);
    }
}
```

8. async/await

- Async 함수는 항상 promise를 반환(return)
  - Then이나 await을 붙일 수 있음.

```
async function findAndSaveUser(Users) {
    // 생략
}

findAndSaveUser().then(() => { /* 생략 */ });
// 또는
async function other() {
    const result = await findAndSaveUser();
}
```

8. for await of

- 노드 10부터 지원
- for await (변수 of 프로미스배열)
  - resolve된 프로미스가 변수에 담겨 나옴
  - await을 사용하기 때문에 async 함수 안에서 해야함

```
const promise1 = Promise.resolve('성공1');
const promise2 = Promise.resolve('성공2');

(async () => {
    for await (promise of [promise1, promise2]) {
        console.log(promise);
    }
})();
```

9. Map/Set

- Map은 객체와 유사한 자료구조

```
const m = new Map();

m.set('a', 'b');
m.set(3, 'c');
const d = {};
m.set(d, 'object');

m.get('a');
m.get(3);
m.get(d);
m.get({});

m.size;
```

```
for (const [k, v] of m) {
    console.log(k, v);
}

m.forEach((v, k) => {
    console.log(k, v);
})

m.has('a');
m.delete('a');
m.clear();
```

9. Map/Set

- Set은 배열과 유사한 자료구조
  - 기존 배열의 중복을 제거할 때도 사용

```
const s = new Set();
s.add(false);
s.add(1);
s.add('1');
s.add(1);
s.add(2);

s.size;
s.has(1);

for (const e of s) {
    console.log(e);
}

s.forEach((a) => {
    console.log(a);
})

s.delete(1);
s.clear();
```

```
const arr = [1, 3, 2, 7, 2, 6, 3, 5];
const result1 = new Set(arr);
const result2 = Array.from(s);
```

10. 널 병합, 옵셔널 체이닝

- ??(널 병합, nullish coalescing) 연산자
  - || 대용으로 사용되며, falsy 값 중 null과 undefined만 따로 구분함

```
const a = 0;
const b = a || 3;
console.log(b);

const c = 0;
const d = c ?? 3;
console.log(d);

const e = null;
const f = e ?? 3;
console.log(f);

const g = undefined;
const h = g ?? 3;
console.log(h);
```

10. 널 병합, 옵셔널 체이닝

- ?.(옵셔널 체이닝, optional chaining) 연산자
  - Null이나 undefined의 속성을 조회하는 경우 에러가 발생하는 것을 막음

```
const a = {};
a.b;
```

```
const c = null;

try {
    c.d;
} catch (e) {
    console.error(e);
}
c?.d;
```

```
const c = null;

try {
    c.f();
} catch (e) {
    console.error(e);
}
c?.f();
```

```
const c = null;

try {
    c[0];
} catch (e) {
    console.error(e);
}
c?.[0];
```

1.7 FrontEnd 자바스크립트

1. Axios

- 서버로 요청을 보내는 코드
  - 라이브러리 없이는 브라우저가 지원하는 XMLHttpRequest 객체 이용
  - HTML에 아래 스크립트를 추가하면 사용할 수 있음.

front.html

```
<script src="https://unpkg.com/axios/dist/axios.min.js"></script>
<script>
  // 여기에 예제 코드를 넣으세요.
</script>
```

1. Axios

GET 요청 보내기

- axios.get 함수의 인수로 요청을 보낼 주소를 넣으면 됨
- 프로미스 기반 코드라 async/await 사용 가능.

```
axios.get('https://koreanjson.com/users/')
.then((res) => {
    console.log(res);
    console.log(res.data);
})
.catch((err) => {
    console.error(err);
});


(async () => {
    try {
        const res = await axios.get('https://koreanjson.com/users/');
        console.log(res);
        console.log(res.data);
    } catch (err) {
        console.error(err);
    }
})();
```

1. Axios

- POST 요청을 하는 코드(데이터를 담아 서버로 보내는 경우)
  - 전체적인 구조는 비슷하나 두 번째 인수로 데이터를 넣어 보냄

```
(async () => {
    try {
        const res = await axios.post('https://koreanjson.com/users/', {
            "name": "이주희",
            "username": "jh1122",
            "email": "lee.jh@gmail.com",
            "phone": "010-1234-5678",
            "website": "https://leejuhee.com",
            "province": "",
            "city": "서울특별시",
            "district": "성북구",
            "street": "성북로 123",
            "zipcode": "02879",
        });
        console.log(res);
        console.log(res.data);
    } catch (err) {
        console.error(err);
    }
})();
```

2. FormData

- HTML form 태그에 담긴 데이터를 Axios 요청으로 보내고 싶은 경우
  - FormData 객체 이용

- FormData 메서드
  - Append로 데이터를 하나씩 추가
  - Has로 데이터 존재 여부 확인
  - Get으로 데이터 조회
  - getAll로 데이터 모두 조회
  - delete로 데이터 삭제
  - set으로 데이터 수정

```
const formData = new FormData();
formData.append('name', '이주희');
formData.append('item', 'orange');
formData.append('item', 'melon');

formData.has('item');
formData.has('money');

formData.get('item');
formData.getAll('item');

formData.append('hobby', ['coding', 'travel']);
formData.get('hobby');
formData.delete('hobby');
formData.get('hobby');
formData.set('item', 'apple');
formData.getAll('item');
```

2. FormData

FormData POST 요청으로 보내기

- Axios의 data 자리에 formData를 넣어서 보내면 됨

```
(async () => {
    try {
            const formData = new FormData();
            formData.append('name', '이주희');
            formData.append('username', 'jh1122');
            //...중간 생략
            formData.append('zipcode', '02879');

            const res = await axios.post('https://koreanjson.com/users/', formData);
            console.log(res);
            console.log(res.data);
        } catch (err) {
            console.error(err);
        }
})();
```

3. encodeURIComponent, decodeURIComponent

- 가끔 주소창에 한글 입력하면 서버가 처리하지 못하는 경우 발생
  - encodeURIComponent로 한글 감싸줘서 처리

```
(async () => {
    const q = '한글';
    try {
        const res = await axios.get('http://~~~~/api/search?q=${encodeURIComponent(q)}');
        console.log(res);
        console.log(res.data);
    } catch (err) {
        console.error(err);
    }
})();
```

3. encodeURIComponent, decodeURIComponent

- 노드를 encodeURIComponent하면 %EB%85%B8%EB%93%9C가 됨
  - decodeURIComponent로 서버에서 한글 해석

```
const keyword = '노드';
encodeURIComponent(keyword) // %EB%85%B8%EB%93%9C
decodeURIComponent('%EB%85%B8%EB%93%9C') // 노드
```

4. data attribute와 dataset

- HTML 태그에 데이터를 저장하는 방법
  - 서버의 데이터를 프런트엔드로 내려줄 때 사용
  - 태그 속성으로 data-속성명
  - 자바스크립트에서 태그.dataset.속성명으로 접근 가능
    - data-user-job -> dataset.userJob
    - data-id -> dataset.id
  - 반대로 자바스크립트 dataset에 값을 넣으면 data-속성이 생김
    - dataset.monthSalary = 10000 를 하면 data-month-salary="10000"이 추가된다.

```
<ul>
      <li data-id="1" data-level="easy">HTML</li>
      <li data-id="2" data-level="easy">CSS</li>
      <li data-id="3" data-level="normal">JavaScript</li>
      <li data-id="4" data-level="normal">Python</li>
      <li data-id="5" data-level="hard">SQL</li>
</ul>
```

```
const list = document.querySelectorAll('li');

list.forEach((li) => {
    console.log(li.dataset.id, li.dataset.level);

    if (li.dataset.level === 'hard') {
        li.dataset.actualLevel = 'easy';
    }
})
```