# 컴퓨터 네트워크

# 2장

# 2.1 호출 스택, 이벤트 루프

# 1. 호출 스택

```
function first() {
  second();
  console.log('첫 번째');
}
function second() {
  third();
  console.log('두 번째');
}
function third() {
  console.log('세 번째');
}
first();
```

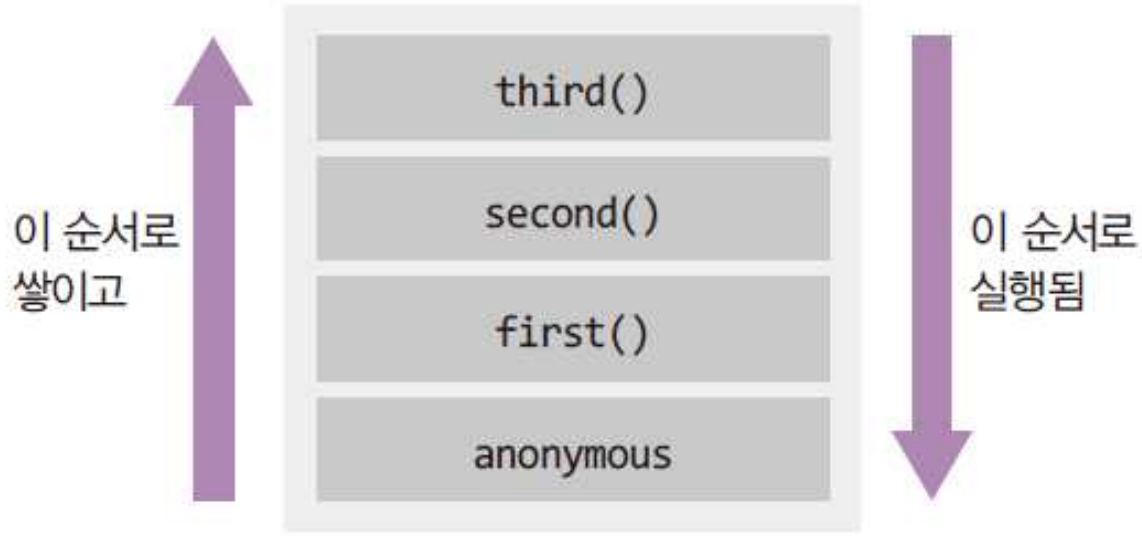
» 위 코드의 순서 예측해보기

- 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

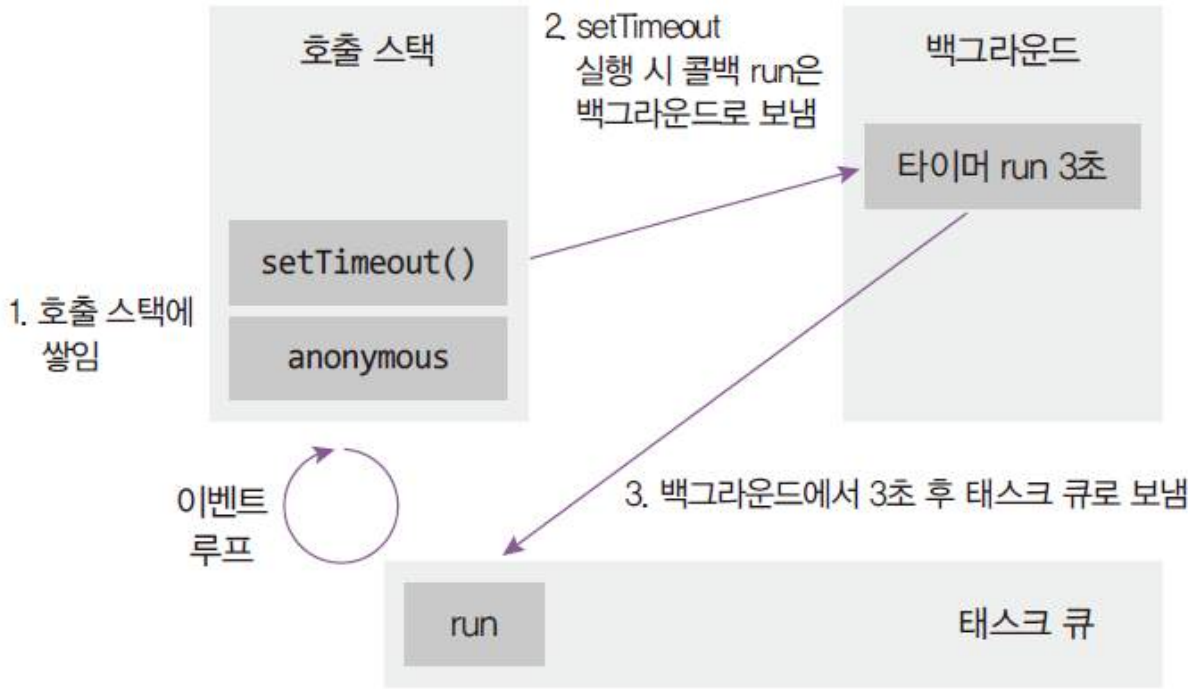
```
function run() {
  console.log('3초 후 실행');
}
console.log('시작');
setTimeout(run, 3000);
console.log('끝');
```

» 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 콜백 함수들(위의 예제에서는 run)을 관리하고, 호출할 순서를 결정하는 역할
- 태스크 큐: 이벤트 발생 후 호출되어야 할 콜백 함수들이 순서대로 기다리는 공간
- 백그라운드: 타이머나 I/O 작업 콜백, 이벤트 리스너들이 대기하는 공간. 여러 작업이 동시에 실행될 수 있음

# 2. 이벤트 루프

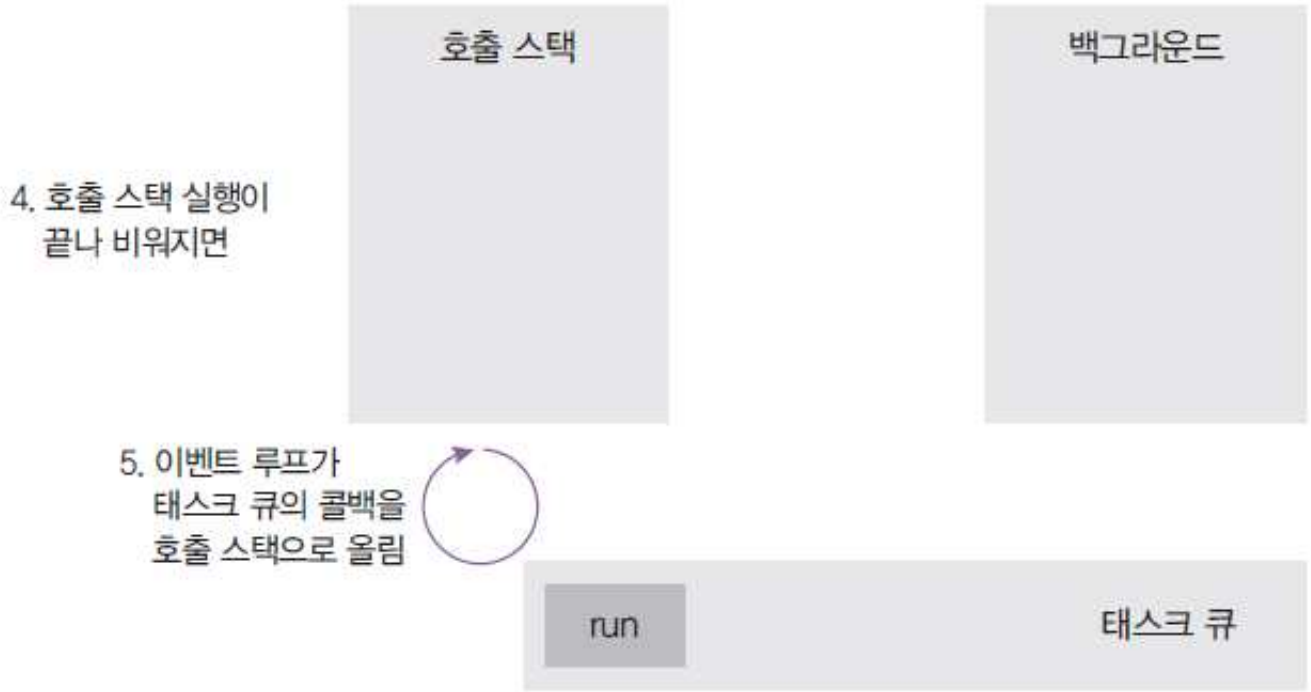
그림 1-7 이벤트 루프 2


» 예제 코드에서 setTimeout이 호출될 때 콜백 함수 run은 백그라운드로

- 백그라운드에서 3초를 보냄
- 3초가 다 지난 후 백그라운드에서 태스크 큐로 보내짐

# 2. 이벤트 루프

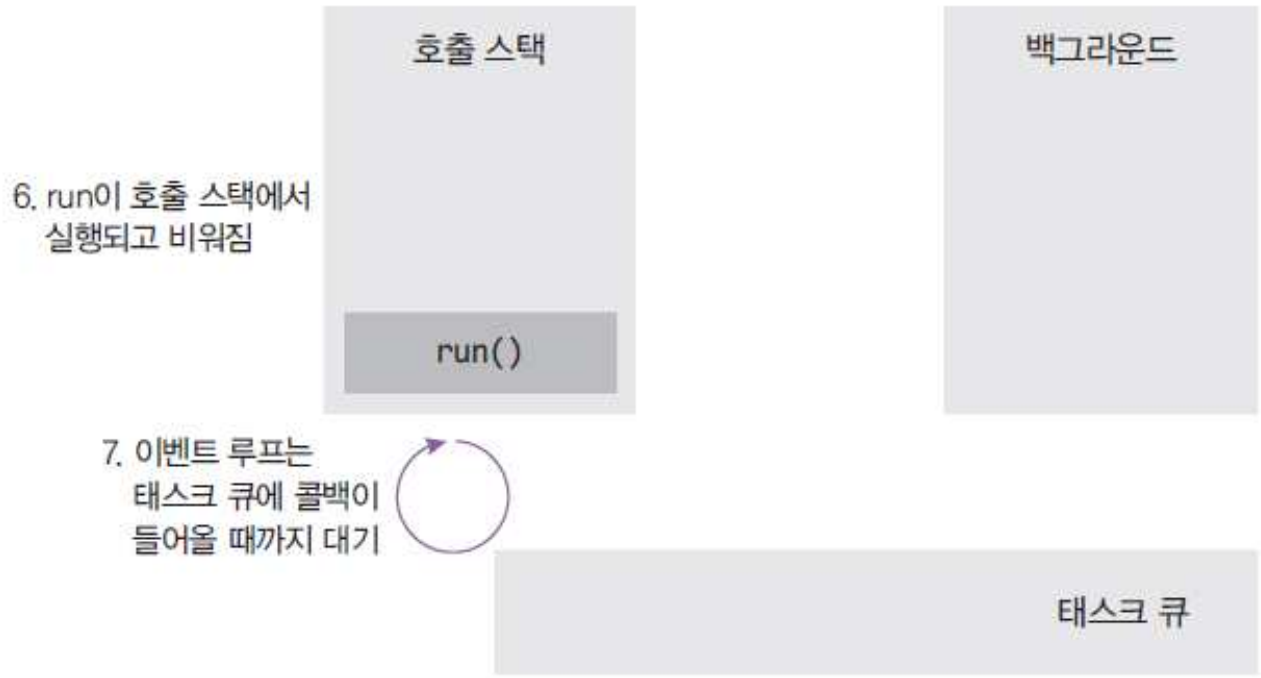
그림 1-7 이벤트 루프 2




- 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 대기하게 됨 -> 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

# 2.2 ES2015+

# 1. const, let

## ES2015 이전에는 var로 변수를 선언

- ES2015부터는 const와 let이 대체
- 가장 큰 차이점: 블록 스코프(var은 함수 스코프)

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

if (true) {
  const y = 3;
}
console.log(y); // Uncaught ReferenceError: y 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

## » 호이스팅 문제

- 변수의 선언과 초기화를 분리, 변수의 선언을 항상 컨텍스트 내 최상위로 끌어올리는 것
- var의 경우 호이스팅 시 undefined로 변수를 초기화

```
var amIHandsome = true;
var reaction = "You are worldwide handsome!";

function showReaction() {
  if (amIHandsome) {
    console.log(reaction);
    var reaction = "I am more handsome than you";
  }
}
showReaction();
```

```
function showReaction() {
  var reaction = undefined;
  if (amIHandsome) {
    console.log(reaction); // undefined
    var reaction = "I am more handsome than you";
  }
}
```

# 1. const, let

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

let b = 0;
b = 1; // 1

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

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

# 3. 객체 리터럴

## ES5 시절의 객체 표현 방법

- 속성 표현 방식에 주목

```
var sayNode = function() {
  console.log('Node');
};
var es = 'ES';
var oldObject = {
  sayJS: function() {
    console.log('JS');
  },
  sayNode: sayNode,
};
oldObject[es + 6] = 'Fantastic';
oldObject.sayNode(); // Node
oldObject.sayJS(); // JS
console.log(oldObject.ES6); // Fantastic
```

# 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

```
const newObject = {
  sayJS() {
    console.log('JS');
  },
  sayNode,
  [es + 6]: 'Fantastic',
};
newObject.sayNode(); // Node
newObject.sayJS(); // JS
console.log(newObject.ES6); // Fantastic
```

# 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)

```
const obj = (x, y) => {
    return {x, y};
};

const obj = (x, y) => {x, y};
```

```
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: 'zero',
  friends: ['nero', 'hero', 'xero'],
  logFriends: function () {
    var that = this; // relationship1을 가리키는 this를 that에 저장
    this.friends.forEach(function (friend) {
      console.log(that.name, friend);
    });
  },
};
relationship1.logFriends();
```

» 화살표 함수가 기존 function() {}을 대체하는 건 아님(this가 달라짐)

- logFriends 메서드의 this 값에 주목
- forEach의 function의 this와 logFriends의 this는 다름
- that이라는 중간 변수를 이용해서 logFriends의 this를 전달

# 4. 화살표 함수

```
const relationship2 = {
  name: 'zero',
  friends: ['nero', 'hero', 'xero'],
  logFriends() {
    this.friends.forEach(friend => {
      console.log(this.name, friend);
    });
  },
};
relationship2.logFriends();
```

» forEach의 인자로 화살표 함수가 들어간 것에 주목

- forEach의 화살표함수의 this와 logFriends의 this가 같아짐
- 화살표 함수는 자신을 포함하는 함수의 this를 물려받음
- 물려받고 싶지 않을 때: function() {}을 사용

# 5. 구조분해 할당

```
var candyMachine = {
  status: {
    name: 'node',
    count: 5,
  },
  getCandy: function () {
    this.status.count--;
    return this.status.count;
  },
};
var getCandy = candyMachine.getCandy;
var count = candyMachine.status.count;
```

```
const example = {a:123, b:{c:248, d: 135}}
const a = example.a;
const d = example.b.d;
```

- var getCandy와 var count에 주목
  - candyMachine부터 시작해서 속성을 찾아 들어가야 함

# 5. 구조분해 할당

```
const candyMachine = {
  status: {
    name: 'node',
    count: 5,
  },
  getCandy() {
    this.status.count--;
    return this.status.count;
  },
};
const { getCandy, status: { count } } = candyMachine;
```

```
const example = {a:123, b:{c:248, d: 135}}

const {a, b: {d}} = example;
console.log(a);
console.log(d);
```

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

# 5. 구조분해 할당

```
const candyMachine = {
  status: {
    name: 'node',
    count: 5,
  },
  getCandy() {
    this.status.count--;
    return this.status.count;
  },
};
const { getCandy, status: { count } } = candyMachine;
```

```
const example = {a:123, b:{c:248, d: 135}}

const {a, b: {d}} = example;
console.log(a);
console.log(d);
```

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

# 5. 구조분해 할당

» 배열도 구조분해 할당 가능

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

```
const array = ['nodejs', {}, 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 Zero = function(type, firstName, lastName) {
  Human.apply(this, arguments);
  this.firstName = firstName;
  this.lastName = lastName;
};
```

```
Zero.prototype = Object.create(Human.prototype);
Zero.prototype.constructor = Zero; // 상속하는 부분
Zero.prototype.sayName = function() {
  alert(this.firstName + ' ' + this.lastName);
};
var oldZero = new Zero('human', 'Zero', 'Cho');
Human.isHuman(oldZero); // 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 Zero extends Human {
  constructor(type, firstName, lastName) {
    super(type);
    this.firstName = firstName;
    this.lastName = lastName;
  }

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

const newZero = new Zero('human', 'Zero', 'Cho');
Human.isHuman(newZero); // 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');
  }
});
// 다른 코드가 들어갈 수 있음
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 = 'zero';
    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 = 'zero';
      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 = 'zero';
      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 = 'zero';
  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 = 'zero';
    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 = 'zero';
    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();
}
```

# 수고하셨습니다♥