프로미스(8월 7~8일)

배정	
상태	8월
속성	@2021년 8월 8일 오전 12:11
언어	

ES6에서 새롭게 도입한 패턴이다. 전통적인 콜백 패턴이 가진 단점을 보완하며 비동기 처리 시점을 명확하게 표현 할 수 있음.

기존은 콜백함수를 통해 비동기처리를 하였는데, 처리결과에 따른 후속처리를 수행함에 따라 비동기 함수가 비동기 처리결과를 가지고 또 비동기 함수를 호출해야 한다면, 콜백함수 호출이 중첩되어 복잡도가 높아지는 현상을 '콜백 헬' 이라 한다. (이거나왔어 또 처리해줘, 이거나왔어 또 처리해줘... ⇒ 지옥)

프로미스의 생성

```
// 프로미스 생성
const promise = new Promise((resolve, reject) => {
  // Promise 함수의 콜백 함수 내부에서 비동기 처리를 수행한다.
  if (/* 비동기 처리 성공 */) {
    resolve('result');
  } else { /* 비동기 처리 실패 */
    reject('failure reason');
  }
});
```

프로미스의 상태

Aa 상태	의미	상태 변경 조건	열
pending	비동기 처리가 아직 수행되지 않는 상태	프로미스 생성 직후	대기
fulfilled(then)	비동기 처리가 수행된 상태(성공)	resolve 함수 호출	이행
rejected(catch)	비동기 처리가 수행된 상태(실패)	reject 함수 호출	거부

프로미스란?

💡 비동기 처리 상태와 처리 결과를 관리하는 객체

프로미스의 후속 처리 메서드

then은 후속처리 할때 성공했을때와, catch는 후속처리 할때 실패했을때를 받음. 또 finally는 프로미스 상태와 상관없이 공통적으로 수행해야 할 처리 내용이 있을때 유용함. (finally 메서드도 then/catch 메서드와 마찬가지로 언제나 프로미스를 반환함.)

Promise.then

fulfilled 상태와 rejected 상태 모두(그냥 fulfilled는 무조건 잡아냄)

Promise.catch

rejected 상태.(원래 rejected된 상태랑 then에서 rejected된 상태까지 모두 잡아냄)

Promise.finally

finally 메서드의 콜백함수는 프로미스의 성공 또는 실패와 상관없이 무조건 호출됨.

프로미스의 상태와 상관없이 공통적으로 수행해야 할 처리 내용이 있을때 유용함.

```
const promiseGet = url => {
  return new Promise((resolve, reject) => {
    const xhr = new XMLHttpRequest();
    xhr.open('GET', url);
    xhr.send();

    xhr.onload = () => {
      if (xhr.status === 200) {
        // 성공적으로 응답을 전달받으면 resolve 함수를 호출한다.
        resolve(JSON.parse(xhr.response));
      } else {
        // 에러 처리를 위해 reject 함수를 호출한다.
        reject(new Error(xhr.status));
      }
    };
  });
};

// promiseGet 함수는 프로미스를 반환한다.
promiseGet('https://jsonplaceholder.typicode.com/posts/1')
.then(res => console.log(res))
.catch(err => console.error(err))
.finally(() => console.log('Bye!'));
```

어떤 콘솔로그에 공통적으로 수행해야할 처리가 있을때, (성공, 에러) 상관없이 'Bye!' 가 무조건 한번 호출됨. 이렇게, 프로미스는 then, catch, finally 후속처리 메서드를 통해 콜백헬을 해결함.

프로미스의 정적메서드

Promise는 주로 생성자 함수로 사용되지만 함수도 객체이므로 메서드를 가질 수 있음.
Promise는 5가지 정적 메서드를 제공합니다.

1. resolve & reject

▼ promise.all

여러개 비동기 처리를 병렬로 이용할때 처리함. 여러개의 promise 결과를 볼때 유용함.

안에 있는 parameter가 배열임(사용자 정보의 객체가 담긴 배열)

인수로 전달받은 모든 promise가 '이행(fullfiled)' 상태가 되면 종료하는데, 하나라도 '거부(reject)' 상태가 되면 catch로 잡아 종료된다.

```
const requestData1 = () => new Promise(resolve => setTimeout(() => resolve(1), 3000));
const requestData2 = () => new Promise(resolve => setTimeout(() => resolve(2), 2000));
const requestData3 = () => new Promise(resolve => setTimeout(() => resolve(3), 1000));

Promise.all([requestData1(), requestData2(), requestData3()])
  .then(console.log) // [ 1, 2, 3 ] ⇒ 약 3초 소요
  .catch(console.error);
```

▼ promise.race

모든 promise가 '이행(fullfield)' 상태가 되는 것을 기다리는 것이 아니라 , 가장 먼저 fullflled 상태가 된 프로미스의 처리결과를 resolve하는 새로운 프로미스를 반환함.

```
// 가장 먼저 반환되는것을 반환 시켜줌..
// 예제1번은 1초가 제일 빠름 --> 3이 반환, 예제2번도 마찬가지..
Promise.race([
  new Promise(resolve => setTimeout(() => resolve(1), 3000)), // 1
  new Promise(resolve => setTimeout(() => resolve(2), 2000)), // 2
  new Promise(resolve => setTimeout(() => resolve(3), 1000)) // 3
])
  .then(console.log) // 3
  .catch(console.log);
//
------------------------------------------------------------------------
Promise.race([
```

```
  new Promise((_, reject) => setTimeout(() => reject(new Error('Error 1')), 3000)),
  new Promise((_, reject) => setTimeout(() => reject(new Error('Error 2')), 2000)),
  new Promise((_, reject) => setTimeout(() => reject(new Error('Error 3')), 1000))
])
  .then(console.log)
  .catch(console.log); // Error: Error 3
```

▼ promise.allSettled

비동기 처리결과를 배열로 반환해줌. ES11에 도입된 메서드. 익스플로러에서는 작동이안됨.

```
Promise.allSettled([
  new Promise(resolve => setTimeout(() => resolve(1), 2000)),
  new Promise((_, reject) => setTimeout(() => reject(new Error('Error!')), 1000))
]).then(console.log);
```

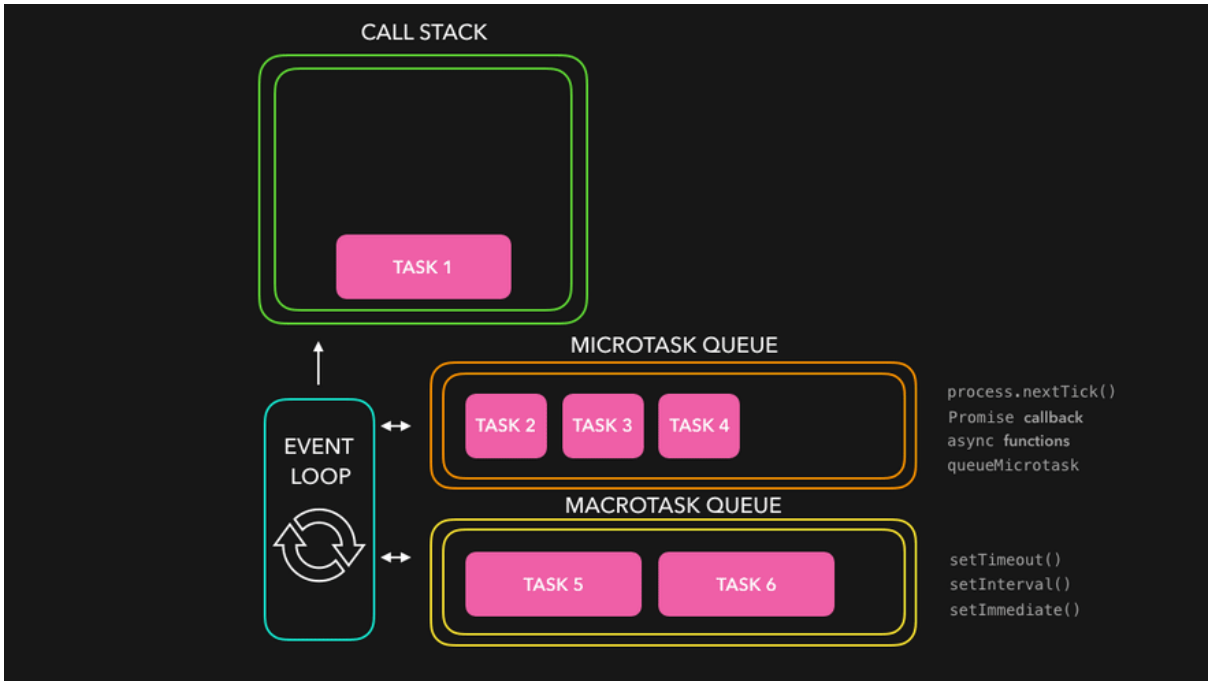
프로미스의 처리 결과를 나타내는 객체

> **fulfilled 상태)**
> status 프로퍼티 : 비동기 처리 상태 , value 프로퍼티 : 처리 결과

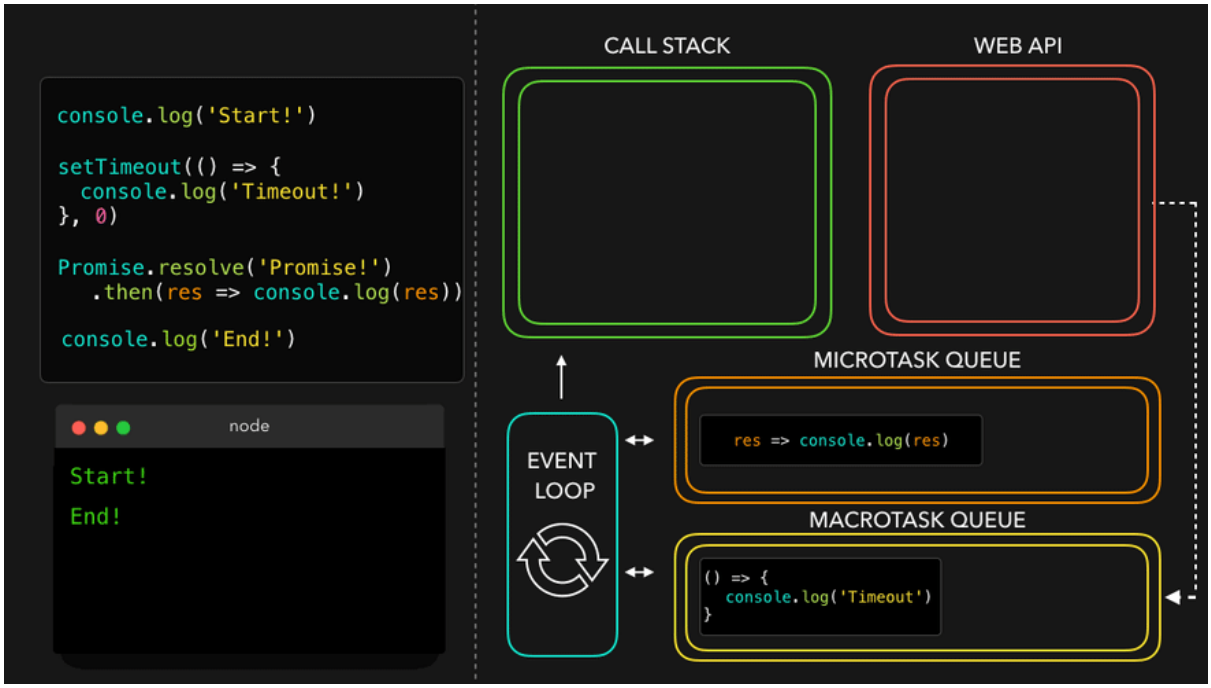
> **rejected 상태)**
> status 프로퍼티 : 비동기 처리 상태 , reason 프로퍼티 : 에러

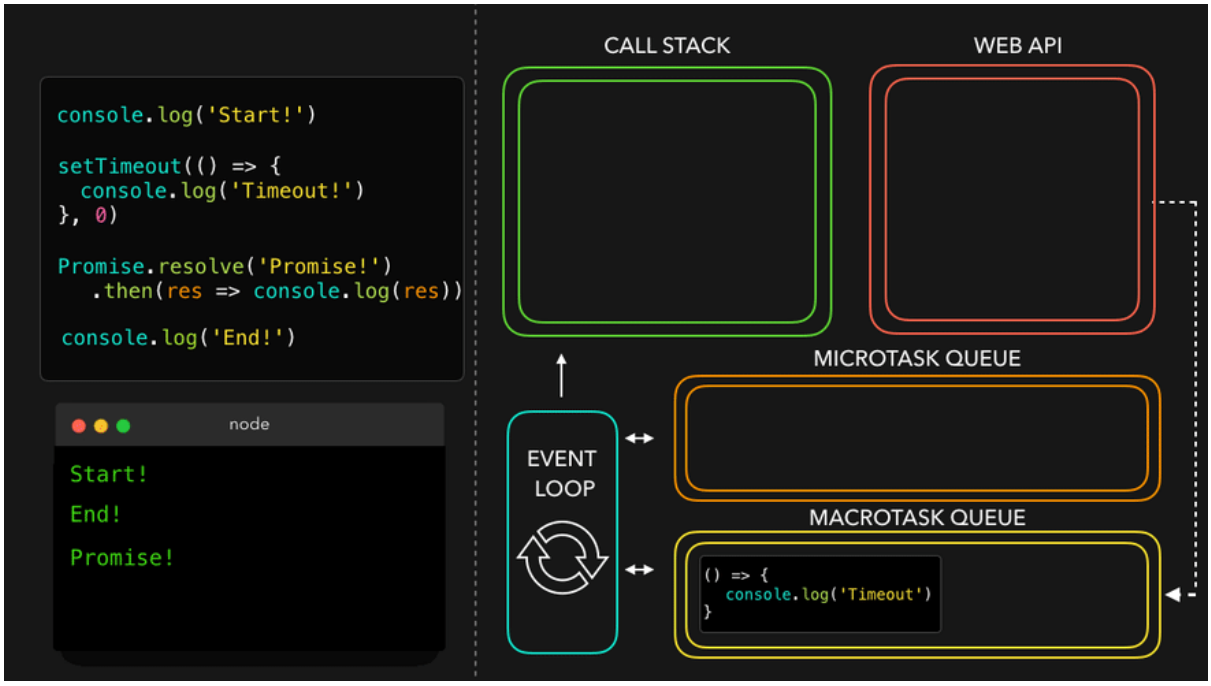
마이크로태스크 큐

'마이크로 태스크 큐'가 '매크로 태스크 큐'보다 우선순위가높음. 먼저 마이크로 태스크큐에서 콜스택이 들어가고, 그뒤로 매크로 테스크 큐에서 콜스택이 들어간다.


CALL STACK
TASK 1
MICROTASK QUEUE
TASK 2
TASK 3
TASK 4
process.nextTick()
Promise callback
async functions
queueMicrotask
EVENT LOOP
MACROTASK QUEUE
TASK 5
TASK 6
setTimeout()
setInterval()
setImmediate()

예시 코드 움짤


console.log('Start!')
setTimeout(() => {
console.log('Timeout!')
}, 0)
Promise.resolve('Promise!')
.then(res => console.log(res))
console.log('End!')
node
Start!
End!
CALL STACK
WEB API
MICROTASK QUEUE
res => console.log(res)
EVENT LOOP
MACROTASK QUEUE
() => {
console.log('Timeout')
}



→ Microtask queue에 있는 작업이 먼저 콜스택에 들어가고, 그 다음 Macrotask queue에 있는 작업이 콜스택에 들어감을 알 수 있다.

fetch

fetch함수는 XMLHttpRequest 객체보다 사용법이 간단하고 프로미스를 지원하기 때문에 비동기 처리를 위한 콜백패턴의 단점에서 자유롭다. 익스플로러를 제외한 대부분 모던 브라우저에 사용 가능함.

```
const promise = fetch(url [, options])

fetch('https://jsonplaceholder.typicode.com/todos/1')
  .then(response => console.log(response));
```

HTTP 요청을 전송할 **URL**과 HTTP 요청 **메서드**, HTTP 요청 **헤더**, **페이로드** 등을 설정한 객체를 전달.

fetch 함수는 HTTP 응답을 나타내는 response 객체를 래핑한 promise 객체를 반환함.

```
const request = {
  get(url) {
    return fetch(url);
  },
  post(url, payload) {
    return fetch(url, {
      method: 'POST',
      headers: { 'content-Type': 'application/json' },
```

```
      body: JSON.stringify(payload)
    });
  },
  patch(url, payload) {
    return fetch(url, {
      method: 'PATCH',
      headers: { 'content-Type': 'application/json' },
      body: JSON.stringify(payload)
    });
  },
  delete(url) {
    return fetch(url, { method: 'DELETE' });
  }
};
```

▼ GET요청

```
request.get('https://jsonplaceholder.typicode.com/todos/1')
  .then(response => response.json())
  .then(todos => console.log(todos))
  .catch(err => console.error(err));
// {userId: 1, id: 1, title: "delectus aut autem", completed: false}
```

▼ POST요청

```
request.post('https://jsonplaceholder.typicode.com/todos', {
  userId: 1,
  title: 'JavaScript',
  completed: false
}).then(response => response.json())
  .then(todos => console.log(todos))
  .catch(err => console.error(err));
// {userId: 1, title: "JavaScript", completed: false, id: 201}
```

▼ PATCH 요청

```
request.patch('https://jsonplaceholder.typicode.com/todos/1', {
  completed: true
}).then(response => response.json())
  .then(todos => console.log(todos))
  .catch(err => console.error(err));
// {userId: 1, id: 1, title: "delectus aut autem", completed: true}
```

▼ DELETE 요청

```
request.delete('https://jsonplaceholder.typicode.com/todos/1')
  .then(response => response.json())
  .then(todos => console.log(todos))
```

```
  .catch(err => console.error(err));
// {}
```