프로미스

프로미스란?

자바스크립트는 비동기 처리를 위한 하나의 패턴으로 콜백 함수를 사용한다. 하지만 전통적인 콜백 패턴은 콜백 헬로 인해 가독성이 나쁘고 비동기 처리 중 발생한 에러의 처리가 곤란하며 여러 개의 비동기 처리를 한번에 처리하는 데도 한계가 있다.

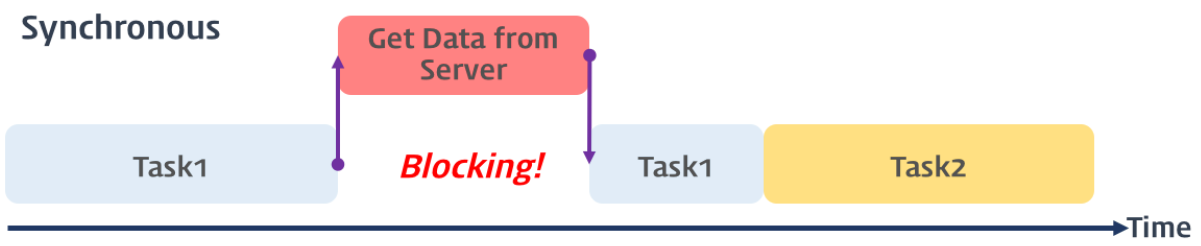
ES6에서는 비동기 처리를 위한 또 다른 패턴으로 프로미스(Promise)를 도입했다. 프로미스는 전통적인 콜백 패턴이 가진 단점을 보완하며 비동기 처리 시점을 명확하게 표현할 수 있다는 장점이 있다.

콜백 패턴의 단점

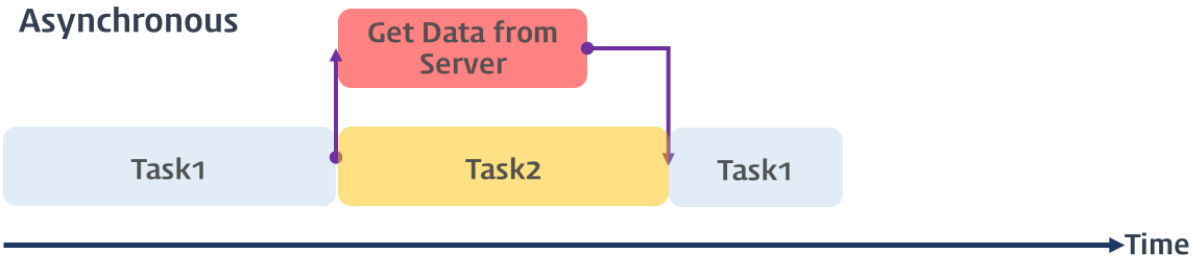
콜백 헬

먼저 동기식 처리 모델과 비동기식 처리 모델에 대해 간단히 살펴보자.

동기식 처리 모델(Synchronous processing model)은 직렬적으로 태스크(task)를 수행한다. 즉, 태스크는 순차적으로 실행되며 어떤 작업이 수행 중이면 다음 태스크는 대기하게 된다. 예를 들어 서버에서 데이터를 가져와서 화면에 표시하는 태스크를 수행할 때, 서버에 데이터를 요청하고 데이터가 응답될 때까지 이후의 태스크들은 블로킹된다.



비동기식 처리 모델(Asynchronous processing model 또는 Non-Blocking processing model)은 병렬적으로 태스크를 수행한다. 즉, 태스크가 종료되지 않은 상태라 하더라도 대기하지 않고 즉시 다음 태스크를 실행한다. 예를 들어 서버에서 데이터를 가져와서 화면에 표시하는 태스크를 수행할 때, 서버에 데이터를 요청한 이후 서버로부터 데이터가 응답될 때까지 대기하지 않고(Non-Blocking) 즉시 다음 태스크를 수행한다. 이후 서버로부터 데이터가 응답되면 이벤트가 발생하고 이벤트 핸들러가 데이터를 가지고 수행할 태스크를 계속해 수행한다. 자바스크립트의 대부분의 DOM 이벤트와 Timer 함수(setTimeout, setInterval), Ajax 요청은 비동기식 처리 모델로 동작한다.

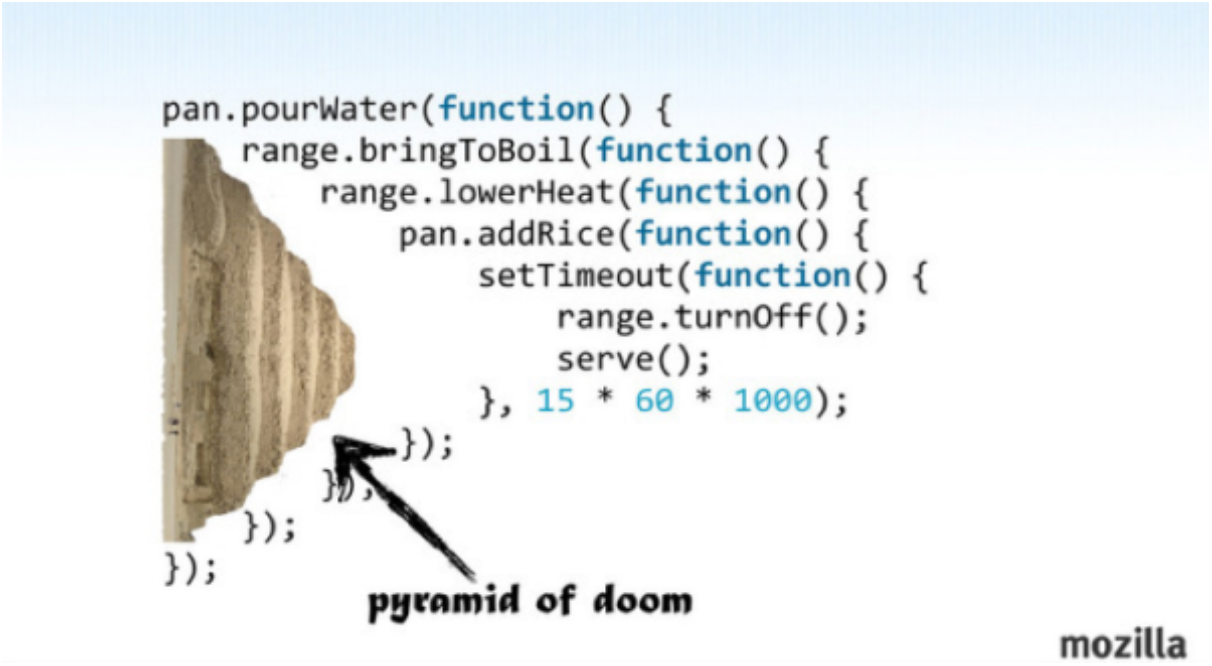


자바스크립트에서 빈번하게 사용되는 비동기식 처리 모델은 요청을 병렬로 처리하여 다른 요청이 블로킹(blocking, 작업 중단)되지 않는 장점이 있다.

하지만 비동기 처리를 위해 콜백 패턴을 사용하면 처리 순서를 보장하기 위해 여러 개의 콜백 함수가 네스팅(nesting, 중첩)되어 복잡도가 높아지는 **콜백 헬(Callback Hell)**이 발생하는 단점이 있다. 콜백 헬은 가독성을 나쁘게 하며 실수를 유발하는 원인이 된다. 아래는 콜백 헬이 발생하는 전형적인 사례이다.

```
step1(function(value1) {
  step2(value1, function(value2) {
    step3(value2, function(value3) {
      step4(value3, function(value4) {
        step5(value4, function(value5) {
            // value5를 사용하는 처리
        });
      });
    });
```

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



콜백 헬이 발생하는 이유에 대해 살펴보자. 비동기 처리 모델은 실행 완료를 기다리지 않고 즉시 다음 태스크를 실행한다. 따라서 비동기 함수(비동기를 처리하는 함수) 내에서 처리 결과를 반환(또는 전역 변수에의 할당)하면 기대한 대로 동작하지 않는다. 다음 코드를 살펴보자.

```
<!DOCTYPE html>
<html>
<body>
  <script>
    // 비동기 함수
    function get(url) {
      // XMLHttpRequest 객체 생성
      const xhr = new XMLHttpRequest();

      // 서버 응답 시 호출될 이벤트 핸들러
      xhr.onreadystatechange = function () {
        // 서버 응답 완료가 아니면 무시
        if (xhr.readyState !== XMLHttpRequest.DONE) return;
```

```
        if (xhr.status === 200) { // 정상 응답
          console.log(xhr.response);
          // 비동기 함수의 결과에 대한 처리는 반환할 수 없다.
          return xhr.response; // ①
        } else { // 비정상 응답
          console.log('Error: ' + xhr.status);
        }
      };

      // 비동기 방식으로 Request 오픈
      xhr.open('GET', url);
      // Request 전송
      xhr.send();
    }

    // 비동기 함수 내의 readystatechange 이벤트 핸들러에서 처리 결
과를 반환(①)하면 순서가 보장되지 않는다.
    const res = get('http://jsonplaceholder.typicode.com/po
sts/1');
    console.log(res); // ② undefined
  </script>
</body>
</html>
```

비동기 함수 내의 readystatechange 이벤트 핸들러에서 처리 결과를 반환(①)하면 순서가 보장되지 않는다. 즉, ②에서 get 함수가 반환한 값을 참조할 수 없다. 그 이유에 대해 살펴보자.

get 함수가 호출되면 get 함수의 실행 컨텍스트가 생성되고 호출 스택(실행 컨텍스트 스택)에서 실행된다. get 함수가 반환하는 xhr.response는 readystatechange 이벤트 핸들러가 반환한다. readystatechange 이벤트는 발생하는 시점을 명확히 알 수 없지만 반드시 get 함수가 종료한 이후 발생한다. get 함수의 마지막 문인 `xhr.send();`가 실행되어야 request를 전송하고 request를 전송해야 readystatechange 이벤트가 발생할 수 있기 때문이다.

get 함수가 종료하면 곧바로 console.log(②)가 호출되어 호출 스택에 들어가 실행된다. console.log가 호출되기 직전에 readystatechange 이벤트가 이미 발생했다하더라도 이

벤트 핸들러는 console.log보다 먼저 실행되지 않는다.

readystatechange 이벤트의 이벤트 핸들러는 이벤트가 발생하면 즉시 실행되는 것이 아니다. 이벤트가 발생하면 일단 태스크 큐로 들어가고 호출 스택이 비면 그때 이벤트 루프에 의해 호출 스택으로 들어가 실행된다. console.log 호출 시점 이전에 readystatechange 이벤트가 이미 발생했다하더라도 get 함수가 종료하면 곧바로 console.log가 호출되어 호출 스택에 들어가기 때문에 readystatechange 이벤트의 이벤트 핸들러는 console.log가 종료되어 호출 스택에서 빠진 이후 실행된다. 만약 get 함수 이후에 console.log가 100번 호출된다면 readystatechange 이벤트의 이벤트 핸들러는 모든 console.log가 종료한 이후에나 실행된다.

때문에 get 함수의 반환 결과를 가지고 후속 처리를 할 수 없다. 즉, 비동기 함수의 처리 결과를 반환하는 경우, 순서가 보장되지 않기 때문에 그 반환 결과를 가지고 후속 처리를 할 수 없다. 즉, 비동기 함수의 처리 결과에 대한 처리는 비동기 함수의 콜백 함수 내에서 처리해야 한다. 이로 인해 콜백 헬이 발생한다.

만일 비동기 함수의 처리 결과를 가지고 다른 비동기 함수를 호출해야 하는 경우, 함수의 호출이 중첩(nesting)이 되어 복잡도가 높아지는 현상이 발생하는데 이를 **Callback Hell**이라 한다.

Callback Hell은 코드의 가독성을 나쁘게 하고 복잡도를 증가시켜 실수를 유발하는 원인이 되며 **에러 처리가 곤란**하다.

에러 처리의 한계

콜백 방식의 비동기 처리가 갖는 문제점 중에서 가장 심각한 것은 에러 처리가 곤란하다는 것이다. 아래의 코드를 살펴보자.

```
try {
  setTimeout(() => { throw new Error('Error!'); }, 1000);
} catch (e) {
  console.log('에러를 캐치하지 못한다..');
  console.log(e);
}
```

try 블록 내에서 setTimeout 함수가 실행되면 1초 후에 콜백 함수가 실행되고 이 콜백 함수는 예외를 발생시킨다. 하지만 이 예외는 catch 블록에서 캐치되지 않는다. 그 이유에 대해

알아보자.

비동기 처리 함수의 콜백 함수는 해당 이벤트(timer 함수의 tick 이벤트, XMLHttpRequest의 readystatechange 이벤트 등)가 발생하면 태스트 큐(Task queue)로 이동한 후 호출 스택이 비어졌을 때, 호출 스택으로 이동되어 실행된다. setTimeout 함수는 비동기 함수이므로 콜백 함수가 실행될 때까지 기다리지 않고 즉시 종료되어 호출 스택에서 제거된다. 이후 tick 이벤트가 발생하면 setTimeout 함수의 콜백 함수는 태스트 큐로 이동한 후 호출 스택이 비어졌을 때 호출 스택으로 이동되어 실행된다. 이때 setTimeout 함수는 이미 호출 스택에서 제거된 상태이다. 이것은 setTimeout 함수의 콜백 함수를 호출한 것은 setTimeout 함수가 아니다라는 것을 의미한다. setTimeout 함수의 콜백 함수의 호출자(caller)가 setTimeout 함수라면 호출 스택에 setTimeout 함수가 존재해야 하기 때문이다.

예외(exception)는 호출자(caller) 방향으로 전파된다. 하지만 위에서 살펴본 바와 같이 setTimeout 함수의 콜백 함수를 호출한 것은 setTimeout 함수가 아니다. 따라서 setTimeout 함수의 콜백 함수 내에서 발생시킨 에러는 catch 블록에서 캐치되지 않아 프로세스는 종료된다.

이러한 문제를 극복하기 위해 Promise가 제안되었다. Promise는 ES6에 정식 채택되어 **IE를 제외한** 대부분의 브라우저가 지원하고 있다.

프로미스의 생성

프로미스는 Promise 생성자 함수를 통해 인스턴스화한다. Promise 생성자 함수는 비동기 작업을 수행할 콜백 함수를 인자로 전달받는데 이 콜백 함수는 resolve와 reject 함수를 인자로 전달받는다.

```
// Promise 객체의 생성
const promise = new Promise((resolve, reject) => {
  // 비동기 작업을 수행한다.

  if (/* 비동기 작업 수행 성공 */) {
    resolve('result');
  }
  else { /* 비동기 작업 수행 실패 */
    reject('failure reason');
```

```
  }
});
```

Promise는 비동기 처리가 성공(fulfilled)하였는지 또는 실패(rejected)하였는지 등의 상태(state) 정보를 갖는다.

상태	의미	구현
pending	비동기 처리가 아직 수행되지 않은 상태	resolve 또는 reject 함수가 아직 호출되지 않은 상태
fulfilled	비동기 처리가 수행된 상태 (성공)	resolve 함수가 호출된 상태
rejected	비동기 처리가 수행된 상태 (실패)	reject 함수가 호출된 상태
settled	비동기 처리가 수행된 상태 (성공 또는 실패)	resolve 또는 reject 함수가 호출된 상태

Promise 생성자 함수가 인자로 전달받은 콜백 함수는 내부에서 비동기 처리 작업을 수행한다. 이때 비동기 처리가 성공하면 콜백 함수의 인자로 전달받은 resolve 함수를 호출한다. 이때 프로미스는 'fulfilled' 상태가 된다. 비동기 처리가 실패하면 reject 함수를 호출한다. 이때 프로미스는 'rejected' 상태가 된다. Promise를 사용하여 비동기 함수를 정의해보자.

```
const promiseAjax = (method, url, payload) => {
  return new Promise((resolve, reject) => {
    const xhr = new XMLHttpRequest();
    xhr.open(method, url);
    xhr.setRequestHeader('Content-type', 'application/jso
n');
    xhr.send(JSON.stringify(payload));

    xhr.onreadystatechange = function () {
      // 서버 응답 완료가 아니면 무시
      if (xhr.readyState !== XMLHttpRequest.DONE) return;

      if (xhr.status >= 200 && xhr.status < 400) {
        // resolve 메소드를 호출하면서 처리 결과를 전달
        resolve(xhr.response); // Success!
```

```
      } else {
        // reject 메소드를 호출하면서 에러 메시지를 전달
        reject(new Error(xhr.status)); // Failed...
      }
    };
  });
};
```

위 예제처럼 비동기 함수 내에서 Promise 객체를 생성하고 그 내부에서 비동기 처리를 구현한다. 이때 비동기 처리에 성공하면 resolve 메소드를 호출한다. 이때 resolve 메소드의 인자로 비동기 처리 결과를 전달한다. 이 처리 결과는 Promise 객체의 후속 처리 메소드로 전달된다. 만약 비동기 처리에 실패하면 reject 메소드를 호출한다. 이때 reject 메소드의 인자로 에러 메시지를 전달한다. 이 에러 메시지는 Promise 객체의 후속 처리 메소드로 전달된다.

프로미스의 후속 처리 메소드

Promise로 구현된 비동기 함수는 Promise 객체를 반환하여야 한다. Promise로 구현된 비동기 함수를 호출하는 측(promise consumer)에서는 Promise 객체의 후속 처리 메소드(then, catch)를 통해 비동기 처리 결과 또는 에러 메시지를 전달받아 처리한다. Promise 객체는 상태를 갖는다고 하였다. 이 상태에 따라 후속 처리 메소드를 체이닝 방식으로 호출한다. Promise의 후속 처리 메소드는 아래와 같다.

thenthen 메소드는 두 개의 콜백 함수를 인자로 전달 받는다. 첫 번째 콜백 함수는 성공(fulfilled, resolve 함수가 호출된 상태) 시 호출되고 두 번째 함수는 실패(rejected, reject 함수가 호출된 상태) 시 호출된다.**then 메소드는 Promise를 반환한다.catch**예외(비동기 처리에서 발생한 에러와 then 메소드에서 발생한 에러)가 발생하면 호출된다. catch 메소드는 Promise를 반환한다.

앞에서 프로미스로 정의한 비동기 함수 get을 사용해 보자. get 함수는 XMLHttpRequest 객체를 통해 Ajax 요청을 수행하므로 브라우저에서 실행하여야 한다.

```
<!DOCTYPE html>
<html>
<body>
<!DOCTYPE html>
```

```
<html>
<body>
  <pre class="result"></pre>
  <script>
    const $result = document.querySelector('.result');
    const render = content => { $result.textContent = JSON.
stringify(content, null, 2); };

    const promiseAjax = (method, url, payload) => {
      return new Promise((resolve, reject) => {
        const xhr = new XMLHttpRequest();
        xhr.open(method, url);
        xhr.setRequestHeader('Content-type', 'application/j
son');
        xhr.send(JSON.stringify(payload));

        xhr.onreadystatechange = function () {
          if (xhr.readyState !== XMLHttpRequest.DONE) retur
n;

          if (xhr.status >= 200 && xhr.status < 400) {
            resolve(xhr.response); // Success!
          } else {
            reject(new Error(xhr.status)); // Failed...
          }
        };
      });
    };

    /*
      비동기 함수 promiseAjax은 Promise 객체를 반환한다.
      Promise 객체의 후속 메소드를 사용하여 비동기 처리 결과에 대한
후속 처리를 수행한다.
    */
    promiseAjax('GET', 'http://jsonplaceholder.typicode.co
m/posts/1')
      .then(JSON.parse)
      .then(
```

```
        // 첫 번째 콜백 함수는 성공(fulfilled, resolve 함수가 호출된 상태) 시 호출된다.
        render,
        // 두 번째 함수는 실패(rejected, reject 함수가 호출된 상태) 시 호출된다.
        console.error
      );
  </script>
</body>
</html>
```

프로미스의 에러 처리

위 예제의 비동기 함수 get은 Promise 객체를 반환한다. 비동기 처리 결과에 대한 후속 처리는 Promise 객체가 제공하는 후속 처리 메서드 then, catch, finally를 사용하여 수행한다. 비동기 처리 시에 발생한 에러는 then 메서드의 두 번째 콜백 함수로 처리할 수 있다.

```
const wrongUrl = 'https://jsonplaceholder.typicode.com/XXX/1';

// 부적절한 URL이 지정되었기 때문에 에러가 발생한다.
promiseAjax(wrongUrl)
  .then(res => console.log(res), err => console.error(err)); // Error: 404
```

비동기 처리에서 발생한 에러는 Promise 객체의 후속 처리 메서드 catch를 사용해서 처리할 수도 있다.

```
const wrongUrl = 'https://jsonplaceholder.typicode.com/XXX/1';

// 부적절한 URL이 지정되었기 때문에 에러가 발생한다.
promiseAjax(wrongUrl)
  .then(res => console.log(res))
  .catch(err => console.error(err)); // Error: 404
```

catch 메서드를 호출하면 내부적으로 `then(undefined, onRejected)`을 호출한다. 위 예제는 내부적으로 다음과 같이 처리된다.

```
const wrongUrl = 'https://jsonplaceholder.typicode.com/XXX/1';

// 부적절한 URL이 지정되었기 때문에 에러가 발생한다.
promiseAjax(wrongUrl)
  .then(res => console.log(res))
  .then(undefined, err => console.error(err)); // Error: 404
```

단, then 메서드의 두 번째 콜백 함수는 첫 번째 콜백 함수에서 발생한 에러를 캐치하지 못하고 코드가 복잡해져서 가독성이 좋지 않다.

```
promiseAjax('https://jsonplaceholder.typicode.com/todos/1')
  .then(res => console.xxx(res), err => console.error(err));
  // 두 번째 콜백 함수는 첫 번째 콜백 함수에서 발생한 에러를 캐치하지 못한다.
```

catch 메서드를 모든 then 메서드를 호출한 이후에 호출하면 비동기 처리에서 발생한 에러(reject 함수가 호출된 상태)뿐만 아니라 then 메서드 내부에서 발생한 에러까지 모두 캐치할 수 있다.

```
promiseAjax('https://jsonplaceholder.typicode.com/todos/1')
  .then(res => console.xxx(res))
  .catch(err => console.error(err)); // TypeError: console.xxx is not a function
```

또한 then 메서드에 두 번째 콜백 함수를 전달하는 것보다 catch 메서드를 사용하는 것이 가독성이 좋고 명확하다. 따라서 에러 처리는 then 메서드에서 하지 말고 catch 메서드를 사용하는 것을 권장한다.

프로미스 체이닝

비동기 함수의 처리 결과를 가지고 다른 비동기 함수를 호출해야 하는 경우, 함수의 호출이 중첩(nesting)이 되어 복잡도가 높아지는 콜백 헬이 발생한다. 프로미스는 후속 처리 메소드를 체이닝(chainning)하여 여러 개의 프로미스를 연결하여 사용할 수 있다. 이로써 콜백 헬을 해결한다.

Promise 객체를 반환한 비동기 함수는 프로미스 후속 처리 메소드인 then이나 catch 메소드를 사용할 수 있다. 따라서 then 메소드가 Promise 객체를 반환하도록 하면(then 메소드는 기본적으로 Promise를 반환한다.) 여러 개의 프로미스를 연결하여 사용할 수 있다.

아래는 서버로 부터 특정 포스트를 취득한 후, 그 포스트를 작성한 사용자의 아이디로 작성된 모든 포스트를 검색하는 예제이다.

```
<!DOCTYPE html>
<html>
<body>
  <pre class="result"></pre>
  <script>
    const $result = document.querySelector('.result');
    const render = content => { $result.textContent = JSON.
stringify(content, null, 2); };

    const promiseAjax = (method, url, payload) => {
      return new Promise((resolve, reject) => {
        const xhr = new XMLHttpRequest();
        xhr.open(method, url);
        xhr.setRequestHeader('Content-type', 'application/j
son');
        xhr.send(JSON.stringify(payload));

        xhr.onreadystatechange = function () {
          if (xhr.readyState !== XMLHttpRequest.DONE) retur
n;

          if (xhr.status >= 200 && xhr.status < 400) {
            resolve(xhr.response); // Success!
```

```
          } else {
            reject(new Error(xhr.status)); // Failed...
          }
        };
      });
    };

    const url = 'http://jsonplaceholder.typicode.com/posts';

    // 포스트 id가 1인 포스트를 검색하고 프로미스를 반환한다.
    promiseAjax('GET', `${url}/1`)
      // 포스트 id가 1인 포스트를 작성한 사용자의 아이디로 작성된 모든 포스트를 검색하고 프로미스를 반환한다.
      .then(res => promiseAjax('GET', `${url}?userId=${JSON.parse(res).userId}`))
      .then(JSON.parse)
      .then(render)
      .catch(console.error);
  </script>
</body>
</html>
```

프로미스의 정적 메소드

Promise는 주로 생성자 함수로 사용되지만 함수도 객체이므로 메소드를 갖을 수 있다. Promise 객체는 4가지 정적 메소드를 제공한다.

Promise.resolve/Promise.reject

Promise.resolve와 Promise.reject 메소드는 존재하는 값을 Promise로 래핑하기 위해 사용한다.

정적 메소드 Promise.resolve 메소드는 인자로 전달된 값을 resolve하는 Promise를 생성한다.

```
const resolvedPromise = Promise.resolve([1, 2, 3]);
resolvedPromise.then(console.log); // [ 1, 2, 3 ]
```

위 예제는 아래 예제와 동일하게 동작한다.

```
const resolvedPromise = new Promise(resolve => resolve([1,
2, 3]));
resolvedPromise.then(console.log); // [ 1, 2, 3 ]
```

Promise.reject 메소드는 인자로 전달된 값을 reject하는 프로미스를 생성한다.

```
const rejectedPromise = Promise.reject(new Error('Erro
r!'));
rejectedPromise.catch(console.log); // Error: Error!
```

위 예제는 아래 예제와 동일하게 동작한다.

```
const rejectedPromise = new Promise((resolve, reject) => re
ject(new Error('Error!')));
rejectedPromise.catch(console.log); // Error: Error!
```

Promise.all

Promise.all 메소드는 프로미스가 담겨 있는 배열 등의 이터러블을 인자로 전달 받는다. 그리고 전달받은 모든 프로미스를 병렬로 처리하고 그 처리 결과를 resolve하는 새로운 프로미스를 반환한다. 아래 예제를 살펴보자.

```
Promise.all([
  new Promise(resolve => setTimeout(() => resolve(1), 300
0)), // 1
  new Promise(resolve => setTimeout(() => resolve(2), 200
0)), // 2
```

```
  new Promise(resolve => setTimeout(() => resolve(3), 100
0))  // 3
]).then(console.log) // [ 1, 2, 3 ]
  .catch(console.log);
```

Promise.all 메소드는 3개의 프로미스를 담은 배열을 전달받았다. 각각의 프로미스는 아래와 같이 동작한다.

- 첫번째 프로미스는 3초 후에 1을 resolve하여 처리 결과를 반환한다.

- 두번째 프로미스는 2초 후에 2을 resolve하여 처리 결과를 반환한다.

- 세번째 프로미스는 1초 후에 3을 resolve하여 처리 결과를 반환한다.

Promise.all 메소드는 전달받은 모든 프로미스를 병렬로 처리한다. 이때 모든 프로미스의 처리가 종료될 때까지 기다린 후 아래와 모든 처리 결과를 resolve 또는 reject한다.

- 모든 프로미스의 처리가 성공하면 **각각의 프로미스가 resolve한 처리 결과를 배열에 담아 resolve하는 새로운 프로미스를 반환**한다. 이때 첫번째 프로미스가 가장 나중에 처리되어도 Promise.all 메소드가 반환하는 프로미스는 첫번째 프로미스가 resolve한 처리 결과부터 차례대로 배열에 담아 그 배열을 resolve하는 새로운 프로미스를 반환한다. 즉, **처리 순서가 보장된다.**

- 프로미스의 처리가 하나라도 실패하면 가장 먼저 실패한 프로미스가 reject한 에러를 reject하는 새로운 프로미스를 즉시 반환한다.

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

위 예제의 경우, 세번째 프로미스가 가장 먼저 실패하므로 세번째 프로미스가 reject한 에러가 catch 메소드로 전달된다.

Promise.all 메소드는 전달 받은 이터러블의 요소가 프로미스가 아닌 경우, Promise.resolve 메소드를 통해 프로미스로 래핑된다.

```
Promise.all([
  1, // => Promise.resolve(1)
  2, // => Promise.resolve(2)
  3  // => Promise.resolve(3)
]).then(console.log) // [1, 2, 3]
  .catch(console.log);
```

아래는 github id로 github 사용자 이름을 취득하는 예제이다.

```
const githubIds = ['jeresig', 'ahejlsberg', 'ungmo2'];

Promise.all(githubIds.map(id => fetch(`https://api.github.com/users/${id}`)))
  // [Response, Response, Response] => Promise
  .then(responses => Promise.all(responses.map(res => res.json())))
  // [user, user, user] => Promise
  .then(users => users.map(user => user.name))
  // [ 'John Resig', 'Anders Hejlsberg', 'Ungmo Lee' ]
  .then(console.log)
  .catch(console.log);
```

위 예제의 Promise.all 메소드는 fetch 함수가 반환한 3개의 프로미스의 배열을 인수로 전달받고 이 프로미스들을 병렬 처리한다. 모든 프로미스의 처리가 성공하면 Promise.all 메소드는 각각의 프로미스가 resolve한 3개의 Response 객체가 담긴 배열을 resolve하는 새로운 프로미스를 반환하고 후속 처리 메소드 then에는 3개의 Response 객체가 담긴 배열이 전달된다. 이때 json 메소드는 프로미스를 반환하므로 한번 더 Promise.all 메소드를 호출해야 하는 것에 주의하자. 두번째 호출한 Promise.all 메소드는 github로 부터 취득한 3개의 사용자 정보 객체가 담긴 배열을 resolve하는 프로미스를 반환하고 후속 처리 메소드 then에는 3개의 사용자 정보 객체가 담긴 배열이 전달된다.

Promise.race

Promise.race 메소드는 Promise.all 메소드와 동일하게 프로미스가 담겨 있는 배열 등의 이터러블을 인자로 전달 받는다. 그리고 Promise.race 메소드는 Promise.all 메소드처럼 모든 프로미스를 병렬 처리하는 것이 아니라 가장 먼저 처리된 프로미스가 resolve한 처리 결과를 resolve하는 새로운 프로미스를 반환한다.

```
Promise.race([
  new Promise(resolve => setTimeout(() => resolve(1), 300
0)), // 1
  new Promise(resolve => setTimeout(() => resolve(2), 200
0)), // 2
  new Promise(resolve => setTimeout(() => resolve(3), 100
0))  // 3
]).then(console.log) // 3
  .catch(console.log);
```

에러가 발생한 경우는 Promise.all 메소드와 동일하게 처리된다. 즉, Promise.race 메소드에 전달된 프로미스 처리가 하나라도 실패하면 가장 먼저 실패한 프로미스가 reject한 에러를 reject하는 새로운 프로미스를 즉시 반환한다.

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