

프로미스

callback hell

여러 개의 비동기 작업을 순차적으로 수행해야 할 때는 콜백 함수가 중첩되어 코드의 깊이가 깊어지는 현상이 발생한다. 이러한 현상을 콜백 지옥(callback hell) 이라고 부른다.

숫자 **n** 을 파라미터로 받아와서 다섯 번에 걸쳐 1초마다 1씩 더해서 출력하는 작업을 **setTimeout** 함수로 구현한 코드

```
function increaseAndPrint(n, callback) {
  setTimeout(() => {
    const increased = n + 1;
    console.log(increased);
    if (callback) {
      callback(increased); // 콜백함수 호출
    }
  }, 1000);
}

increaseAndPrint(0, n => {
  increaseAndPrint(n, n => {
    increaseAndPrint(n, n => {
      increaseAndPrint(n, n => {
        increaseAndPrint(n, n => {
          console.log('끝!');
        });
      });
    });
  });
});
```



이러한 콜백 함수의 코드 형태는 콜백 함수가 중첩되면서 들여쓰기 수준이 깊어져 코드의 **가독성을 떨어뜨리며 코드의 흐름을 파악하기 어려워진다**. 또한 콜백 함수마다 에러 처리를 따로 해줘야 하고, **에러가 발생한 위치를 추적하기 힘들게 된다**.

프로미스

- 프로미스 = 자바스크립트 비동기 처리에 사용되는 객체
- 비동기 처리란 '특정 코드의 실행이 완료될 때까지 기다리지 않고 다음 코드를 먼저 수행하는 자바스크립트의 특성'입니다.

프로미스의 3가지 상태

프로미스를 사용할 때 알아야 하는 가장 기본적인 개념이 바로 프로미스의 상태(states)다. 여기서 말하는 상태란 프로미스의 처리 과정을 의미합니다. `new Promise()`로 프로미스를 생성하고 종료될 때까지 3가지 상태를 갖는다.

1. **Pending(대기)** : 비동기 처리 로직이 아직 완료되지 않은 상태
2. **Fulfilled(이행)** : 비동기 처리가 완료되어 프로미스가 결과 값을 반환해준 상태
3. **Rejected(실패)** : 비동기 처리가 실패하거나 오류가 발생한 상태

1. Pending(대기)

아래와 같이 `new Promise()` 메서드를 호출하면 대기(Pending) 상태가 된다

```
new Promise();
```

2. Fulfilled(이행)

`new Promise()` 메서드를 호출할 때 콜백 함수를 선언할 수 있고, 콜백 함수의 인자는 `resolve`, `reject`가 된다.

```
new Promise(function(resolve, reject) {  
    //콜백 함수의 인자 resolve를 아래와 같이 실행하면 이행(Fulfilled) 상
```

```
    resolve();  
  });
```



Promise 생성자 안에 들어가는 콜백 함수를 executor 라고 부른다.

이행 상태가 되면 아래와 같이 then()을 이용하여 처리 결과 값을 받을 수 있다.

```
//프로미스는 별도로 함수로 감싸서 사용하는 것이 일반적이다.  
function getData() {  
  return new Promise(function(resolve, reject) {  
    var data = 100;  
    resolve(data);  
  });  
}  
// resolve()의 결과 값 data를 resolvedData로 받음  
getData().then(function(resolvedData) {  
  console.log(resolvedData); // 100  
});
```



프로미스의 '이행' 상태를 다르게 표현하자면 '완료' 봐도 된다.

Rejected(실패)

reject를 아래와 같이 호출하면 실패(Rejected) 상태가 됩니다.

```
new Promise(function(resolve, reject) {  
  reject();  
});
```

실패 상태가 되면 실패한 이유(실패 처리의 결과 값)를 catch()로 받을 수 있다.

```
function getData() {
  return new Promise(function(resolve, reject) {
    reject(new Error("Request is failed"));
  });
}

// reject()의 결과 값 Error를 err에 받음
getData().then().catch(function(err) {
  console.log(err); // Error: Request is failed
});
```

```
Error: Request is failed
    at 연습.html:12:12
    at new Promise (<anonymous>)
    at getData (연습.html:11:10)
    at 연습.html:17:1
```

프로미스 핸들러

프로미스는 성공/실패 결과를 `.then / .catch / .finally` 핸들러를 통해 받아 다음 후속 작업을 수행할 수 있다.

프로미스 핸들러는 프로미스의 상태에 따라 실행되는 콜백 함수라고 보면 된다.

- `.then()` : 프로미스가 이행(fulfilled)되었을 때 실행할 콜백 함수를 등록하고, 새로운 프로미스를 반환
- `.catch()` : 프로미스가 거부(rejected)되었을 때 실행할 콜백 함수를 등록하고, 새로운 프로미스를 반환
- `.finally()` : 프로미스가 이행되거나 거부될 때 상관없이 실행할 콜백 함수를 등록하고, 새로운 프로미스를 반환

프로미스 체이닝

- 프로미스 핸들러를 연달아 연결하는 것
- 여러 개의 비동기 작업을 순차적으로 수행할 수 있다

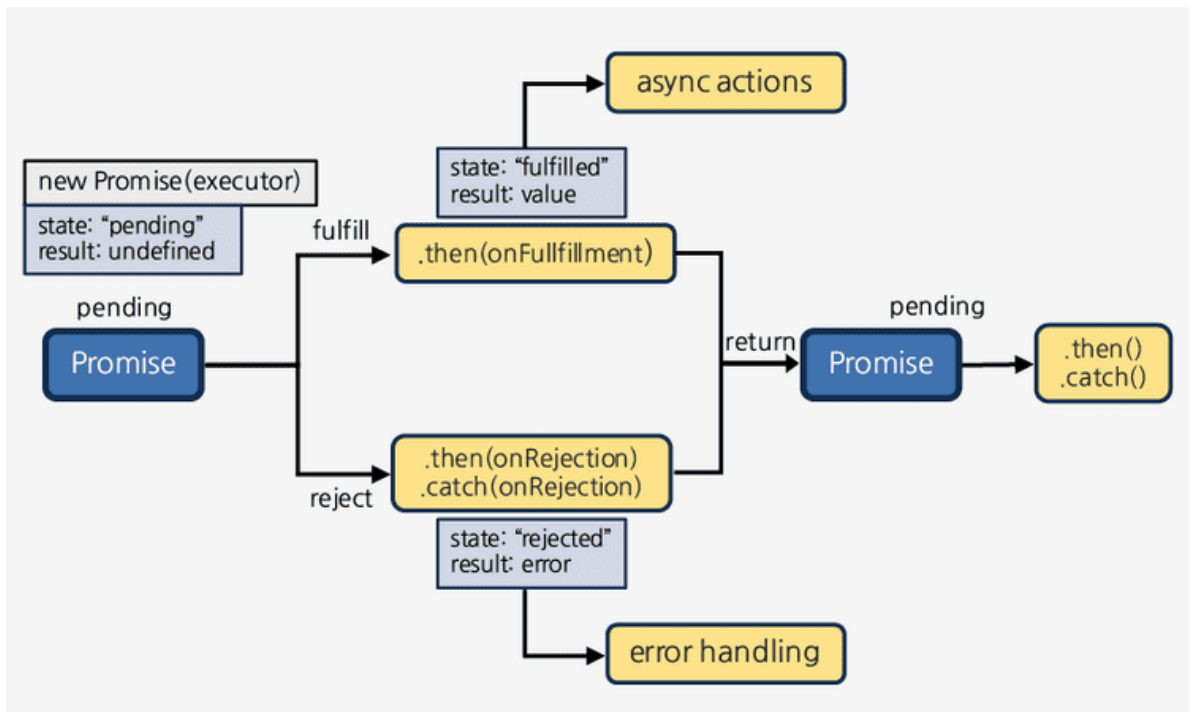
```
function doSomething() {
  return new Promise((resolve, reject) => {
    resolve(100)
  });
}

doSomething()
  .then((value1) => {
    const data1 = value1 + 50;
    return data1
  })
  .then((value2) => {
    const data2 = value2 + 50;
    return data2
  })
  .then((value3) => {
    const data3 = value3 + 50;
    return data3
  })
  .then((value4) => {
    console.log(value4); // 250 출력
  })
```



then 핸들러에서 값을 리턴 하면, 그 반환 값은 자동으로 프로미스 객체로 감싸져 반환

프로미스 처리 흐름



callback hell이 프로미스로 개선된 비동기 처리 문법

```
function increaseAndPrint(n) {
  return new Promise((resolve, reject) => {
    setTimeout(() => {
      const increased = n + 1;
      console.log(increased);
      resolve(increased);
    }, 1000)
  })
}

increaseAndPrint(0)
  .then((n) => increaseAndPrint(n))
  .then((n) => increaseAndPrint(n))
  .then((n) => increaseAndPrint(n))
  .then((n) => increaseAndPrint(n)); // 체이닝 기법
```

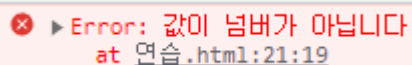
만일 연결된 이행 핸들러에서 중간에 오류가 있는 처리를 행한다면 예외 처리를 함으로써 catch 핸들러에 점프하도록 설정할 수 있다

```
function doSomething(arg) {
  return new Promise((resolve, reject) => {
    resolve(arg)
  });
}

doSomething('100A')
  .then((value1) => {
    const data1 = value1 + 50; // 숫자에 문자를 연산

    if (isNaN(data1))
      throw new Error('값이 넘버가 아닙니다')

    return data1
  })
  .then((value2) => {
    const data2 = value2 + 50;
    return data2
  })
  .catch((err) => {
    console.error(err);
  })
```



Error: 값이 넘버가 아닙니다
at 연습.html:21:19

catch 핸들러 다음으로 then 핸들러가 이어서 체이닝 되어 있다면, 에러가 처리되고 가까운 then 핸들러로 제어 흐름이 넘어가 실행이 이어지게 된다.

```
new Promise((resolve, reject) => {
  throw new Error("에러 발생!");
})
  .catch(function(error) {
```

```

        console.log("에러가 잘 처리되었습니다. 정상적으로 실행이 이어집니다.")
    })
    .then(() => {
        console.log("다음 핸들러가 실행됩니다.")
    })
    .then(() => {
        console.log("다음 핸들러가 또 실행됩니다.")
    });

```

에러가 잘 처리되었습니다. 정상적으로 실행이 이어집니다.

다음 핸들러가 실행됩니다.

다음 핸들러가 또 실행됩니다.

프로미스 정적 메서드

정적 메서드는 객체를 초기화 & 생성하지 않고도 바로 사용할 수 있기 때문에 비동기 처리를 보다 효율적이고 간편하게 구현할 수 있도록 도와준다.

Promise.resolve()

- 보통 프로미스의 `resolve()` 메서드는, 프로미스를 생성자로 만들고 그안의 콜백 함수의 매개변수를 통해 호출하여 사용해왔다.
- 프로미스 객체와 전혀 연관없는 함수 내에서 필요에 따라 프로미스 객체를 반환하여 핸들러를 이용할 수 있도록 응용이 가능
- 이 방법은 비동기 작업을 수행하지 않는 함수에서도 프로미스의 장점을 활용하고 싶은 경우에 유용하다.

```

// 프로미스 객체와 전혀 연관없는 함수
function getRandomNumber() {
    const num = Math.floor(Math.random() * 10); // 0 ~ 9 사이의 랜덤 숫자
    return num;
}

// Promise.resolve() 를 사용하여 프로미스 객체를 반환하는 함수

```



```
function getPromiseNumber() {
  const num = getRandomNumber(); // 일반 값
  return Promise.resolve(num); // 프로미스 객체
}

// 핸들러를 이용하여 프로미스 객체의 값을 처리하는 함수
getPromiseNumber()
  .then((value) => {
    console.log(`랜덤 숫자: ${value}`);
  })
  .catch((error) => {
    console.error(error);
  });
```

랜덤 숫자: 2

Promise.reject()

주어진 사유로 거부하는 Promise 객체를 반환한다.

```
// 주어진 사유로 거부되는 프로미스 생성
const p = Promise.reject(new Error('error'));

// 거부 사유를 출력
p.catch(error => console.error(error)); // Error: error
```

❌ ▶ Error: error
at 연습.html:11:26

Promise.all()

- 배열, Map, Set에 포함된 여러개의 프로미스 요소들을 한꺼번에 비동기 작업을 처리해야 할때 굉장히 유용한 프로미스 정적 메소드이다

- 모든 프로미스 비동기 처리가 이행(fulfilled) 될 때까지 기다리고 모든 프로미스가 완료 되면 그때 then 핸들러가 실행하는 형태
- 가장 대표적인 사용 예시로 여러 개의 API 요청을 보내고 모든 응답을 받아야 하는 경우에 사용할 수 있다.

```
// 1. 서버 요청 API 프로미스 객체 생성 (fetch)
const api_1 = fetch("https://jsonplaceholder.typicode.com/use
const api_2 = fetch("https://jsonplaceholder.typicode.com/use
const api_3 = fetch("https://jsonplaceholder.typicode.com/use

// 2. 프로미스 객체들을 묶어 배열로 구성
const promises = [api_1, api_2, api_3];

// 3. Promise.all() 메서드 인자로 프로미스 배열을 넣어,
// 모든 프로미스가 이행될 때까지 기다리고, 결과값을 출력
Promise.all(promises)
  .then((results) => {
    // results는 이행된 프로미스들의 값들을 담은 배열.
    // results의 순서는 promises의 순서와 일치.
    console.log(results); // [users1, users2, users3]
  })
  .catch((error) => {
    // 어느 하나라도 프로미스가 거부되면 오류를 출력
    console.error(error);
  });
```

```
▼ (3) [Response, Response, Response] ⓘ
  ▶ 0: Response {type: 'cors', url: 'https://json
  ▶ 1: Response {type: 'cors', url: 'https://json
  ▶ 2: Response {type: 'cors', url: 'https://json
      length: 3
  ▶ [[Prototype]]: Array(0)
```

Promise.allSettled()

- `Promise.all()` 메서드의 업그레이드 버전으로, 주어진 모든 프로미스가 처리되면 모든 프로미스 각각의 상태와 값 (또는 거부 사유)을 모아놓은 배열을 반환한다.

```
// 1초 후에 1을 반환하는 프로미스
const p1 = new Promise(resolve => setTimeout(() => resolve(1)

// 2초 후에 에러를 발생시키는 프로미스
const p2 = new Promise((resolve, reject) => setTimeout(() =>

// 3초 후에 3을 반환하는 프로미스
const p3 = new Promise(resolve => setTimeout(() => resolve(3)

// 세 개의 프로미스의 상태와 값 또는 사유를 출력
Promise.allSettled([p1, p2, p3])
  .then(result => console.log(result));
```

```
▼ (3) [{...}, {...}, {...}] 1
  ▶ 0: {status: 'fulfilled', value: 1}
  ▶ 1: {status: 'rejected', reason: Error: error at http://127.0
  ▶ 2: {status: 'fulfilled', value: 3}
    length: 3
  ▶ [[Prototype]]: Array(0)
```

Promise.any()

`Promise.any()` 는 주어진 모든 프로미스 중 **하나라도 완료**되면 바로 반환하는 정적 메서드이다.

```
const promise1 = new Promise((resolve, reject) => {
  setTimeout(() => {
    reject("promise1 failed");
  }, 3000);
});

const promise2 = new Promise((resolve, reject) => {
  setTimeout(() => {
    resolve("promise2 succeeded");
  }, 2000);
});

const promise3 = new Promise((resolve, reject) => {
  setTimeout(() => {
```

```

        reject("promise3 failed");
    }, 1000);
});

// promise1, promise2, promise3은 각각 3초, 2초, 1초 후에 거부되거
Promise.any([promise1, promise2, promise3])
    .then((value) => {
        console.log(value); // "promise2 succeeded"
    })
    .catch((error) => {
        console.error(error);
    });

```

```
promise2 succeeded
```

- `Promise.any()` 메서드의 결과로 `promise2`의 처리가 가장 먼저 도출됨을 볼 수 있다.
- 오로지 첫 번째로 이행(fulfilled) 된 프로미스만을 취급하기 때문에 나머지 `promise1`과 `promise3`의 거부(rejected)는 무시되게 된다.

Promise.race()

- `Promise.race()` 는 `Promise.any()` 와 같이 여러 개의 프로미스 중 가장 먼저 처리된 프로미스의 결과값을 반환하지만, 차이점이 존재한다.
- `Promise.any()` 는 가장 먼저 fulfilled(이행) 상태가 된 프로미스만을 반환하거나, 혹은 전부 rejected(실패) 상태가 된 프로미스(AggregateError)를 반환한다
- `Promise.race()` 는 fulfilled(이행), rejected(실패) 여부 상관없이 무조건 처리가 끝난 프로미스 결과 값을 반환한다.

```

const promise1 = new Promise((resolve, reject) => {
    setTimeout(() => {
        reject("promise1 failed");
    }, 3000);
});

const promise2 = new Promise((resolve, reject) => {
    setTimeout(() => {
        resolve("promise2 succeeded");
    }, 2000);
});

```

```

    }, 2000);
  });

  const promise3 = new Promise((resolve, reject) => {
    setTimeout(() => {
      reject("promise3 failed");
    }, 1000);
  });
  //Promise.race의 경우 실패여부 상관없이 결과 값을 반환하기 때문에
  //가장 먼저 결과가 반환되는 promise3의 결과 값을 반환
  Promise.race([promise1, promise2, promise3])
    .then((value) => {
      console.log(value);
    })
    .catch((error) => {
      console.error(error);
    });

```

❌ ▶ promise3 failed

콜백 못지않게 프로미스의 `then()` 메서드가 지나치게 체인되어 반복되면 코드가 장황해지고 가독성이 굉장히 떨어질 수 가 있다.

- 아래 코드는 fetch 함수를 사용하여 깃허브 API에서 유저 정보를 가져오고, then 메서드를 여러 번 연결하여 유저들의 로그인 이름을 심표로 구분한 문자열로 만들어 출력하는 비동기 작업을 수행한다
- 이런 식으로 then을 늘어뜨려 놓으면 코드가 길어지고, 각 then 메서드가 어떤 값을 반환하는지 파악하기 어렵게 된다
- 또한, catch 메서드가 마지막에 한 번만 사용되어 있기 때문에 중간에 발생할 수 있는 에러나 예외 상황에 대응하기 어렵다

```

fetch("https://api.github.com/users")
  .then((response) => {
    if (response.ok) {
      return response.json();
    } else {

```

```

        throw new Error("Network Error");
    }
})
.then((users) => {
    return users.map((user) => user.login);
})
.then((logins) => {
    return logins.join(", ");
})
.then((result) => {
    console.log(result);
})
.catch((error) => {
    console.error(error);
});

```

- async/await 키워드를 사용하면 비동기 작업을 마치 동기 작업처럼 쓸 수 있어서 코드가 간결하고 가독성이 좋아지게 된다.
- async/await 키워드는 ES8에서 도입된 비동기 처리를 위한 문법으로, 프로미스를 기반으로 하지만 then과 catch 메서드를 사용하지 않고 비동기 작업을 수행할 수 있다.

```

async function getData() {
    const response = await fetch('https://jsonplaceholder.typicode.com/users');
    const data = await response.json();
    console.log(data)
}
getData()

```