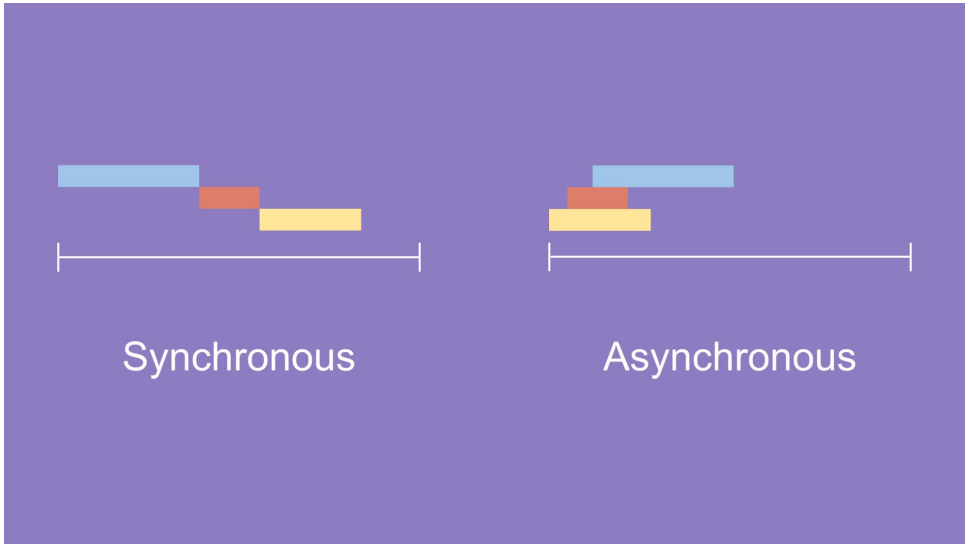
javascript 동기/비동기

김유진

동기, 비동기



동기 (Synchronous)

: 요청을 보낸 후 해당 요청의 **응답을 받아야** 다음 동작 실행

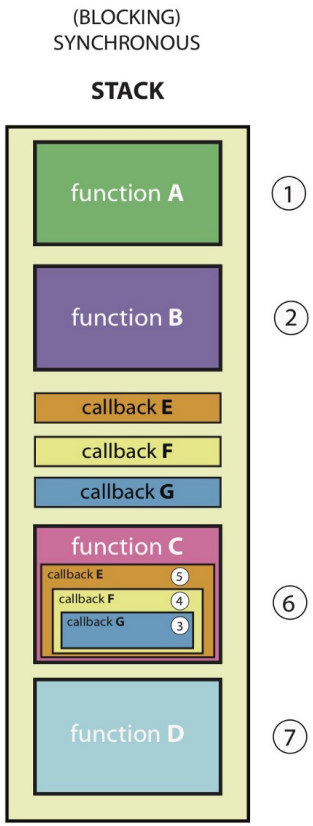
비동기 (Asynchronous)

: 요청을 보낸 후 **응답과 관계없이** 다음 동작을 실행

자바스크립트의 특성

- Single-threaded
- 동기적 (Synchronous)
- 콜스택
- blocking

-> 비동기적으로도 조작 가능



비동기적으로 처리하기

비동기적 처리

: 특정 코드의 연산이 끝날 때까지 코드의 실행을 멈추지 않고, 순차적으로 다음 코드를 먼저 실행하는 자바스크립트의 특성

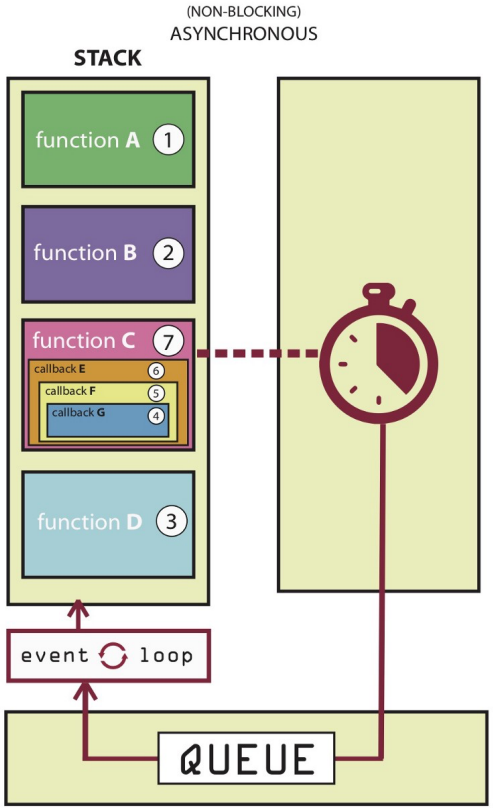
-> 즉시 처리하지 못하는 이벤트들 (ex 서버 통신, setTimeout)

-> Web API를 이용하여 콜백큐로 보내고

-> 이벤트 루프를 통해 콜스택이 비었을 경우 실행

-> Web API 로 들어오는 순서보다 어떤 이벤트가 **먼저 처리되느냐**가 중요

-> **실행 순서가 불명확함**



비동기 처리의 문제점

```
console.log('1');
setTimeout(function() {
 console.log('2')
}, 3000)
console.log('3')
// 1->3->2 순으로 출력
```

```
function getData() {
 var tableData;
 $.get('url', function (response) {
  tableData = response;
 })
 // $.get()로 데이터를 요청하고 받아올 때까지 기다리지 않고 return 해버림
 return tableData;
}
console.log(tableData) // 따라서 undefined
```

-> 비동기 **흐름 컨트롤이 어렵다**는 문제점

해결책, 콜백함수

콜백 함수

: 특정 함수에 **매개변수로 전달**된 함수

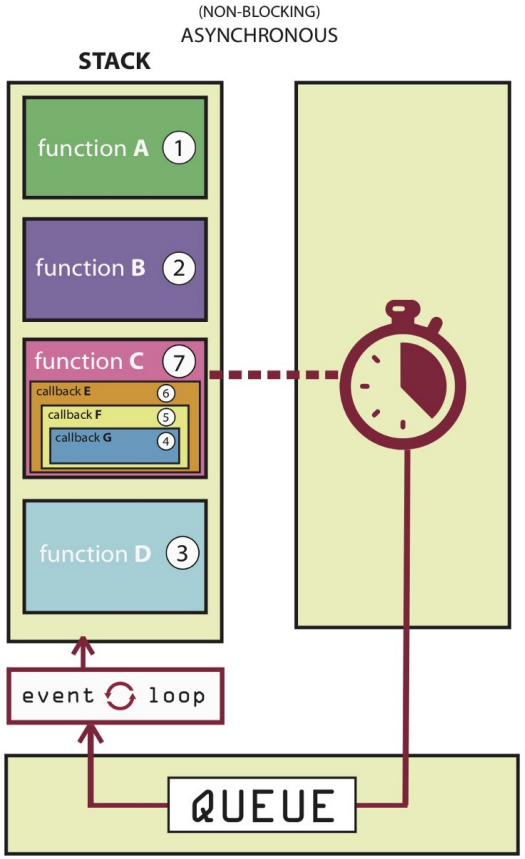
전달된 함수 안에서 호출, 실행됨

-> 콜백함수 E를 넘겨받은 함수 C가 E의 제어권을 갖게 되는 것 (오른쪽)

비동기 요청이 끝나면

-> 해당 요청의 결과는 **콜백큐**로 보내짐

-> **이벤트 루프**를 통해 처리



콜백함수 예제

```
function getData(callback) {
 $.get('url', function (response) {
  callback(response) // calllback 함수에 인자 response 넘겨줌
 })
}
getData(function (tableData) {
 console.log(tableData)
}) // callback
```

-> 처리되어야 하는 이벤트들을 **순차적으로** 콜백 함수로 넣어주는 방식

콜백함수의 문제점

콜백지옥

```
// 콜백지옥, 가독성이 떨어짐
$.get('url', function (response) {
 parseValue(response, fuction(id) {
  auth(id, function (result) {
   display(result, function (text) {
    console.log(text)
   })
  })
 })
})
```

```
doSomething(function(result) {
  doSomethingElse(result, function(newResult) {
    doThirdThing(newResult, function(finalResult) {
      console.log('Got the final result: ' + finalResult);
    }, failureCallback);
  }, failureCallback);
}, failureCallback);
```

1. 콜백함수가 언제 해결될지 정확한 **예측 불가**
   -> 모든 종속함수 **중첩** (콜백지옥)

1. 가독성이 떨어짐
2. **모든 콜백에서 각각 에러 핸들링** 필요
3. 로직 변경 어려움

해결책 예제

```
function parseValueDone(id) {
 auth(id, authDone);
}
function authDone(result) {
 display(result, displayDone)
}
function displayDone(text) {
 console.log(text)
}
$.get('url', function (response) {
 parseValue(response, parseValueDone)
})
```

코딩 패턴으로 해결도 가능 -> 각 콜백 함수를 분리

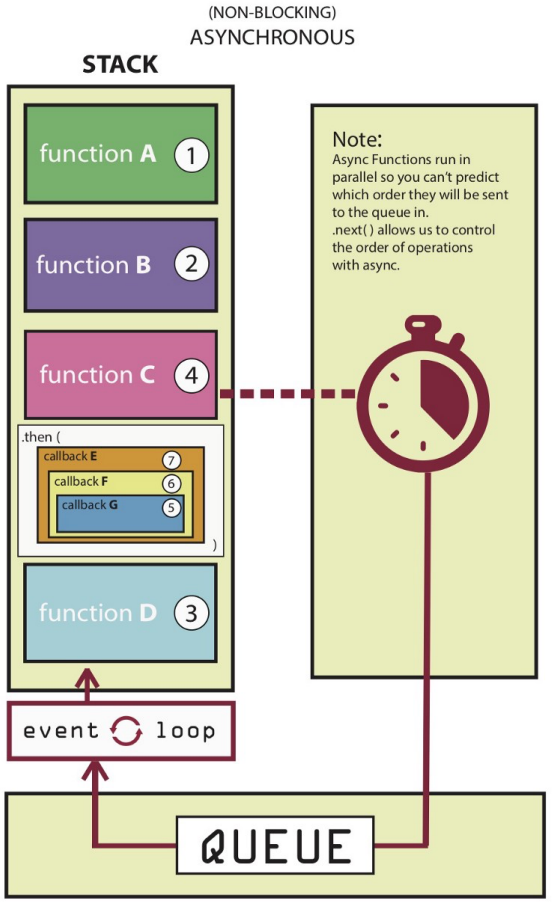
또 다른 해결책, Promise

Promise 객체

: 비동기 작업이 맞이할 미래의 완료 또는 실패와 그 결과값을 나타냄

-> 결과값 돌려받을 수 있으므로 이후 처리 쉽게 컨트롤 가능

- 한 블록 내에 많은 중첩 함수 쓰지 않아도 됨
- 더 module 지향적
- 읽기 쉽다는 비동기 프로그래밍의 장점을 포함



Promise 사용

```
new Promise();
```

new 키워드로 메소드 호출

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

Promise의 **인자로 콜백 함수**를 넘겨줌
이 콜백함수는 다시 ***resolve, reject*** 2개의 함수를 **인자로** 받음

- **resolve** 는 비동기 처리 성공일 때 실행
- **reject** 는 비동기 처리 실패일 때 실행

Promise의 상태값

상태 = **Promise의 처리과정**

Pending 대기

: 비동기 처리 로직 아직 완료되지 않은 상태

Fulfilled 성공

: 비동기 처리 로직 완료되어 Promise가 결과값 반환해준 상태

Settled

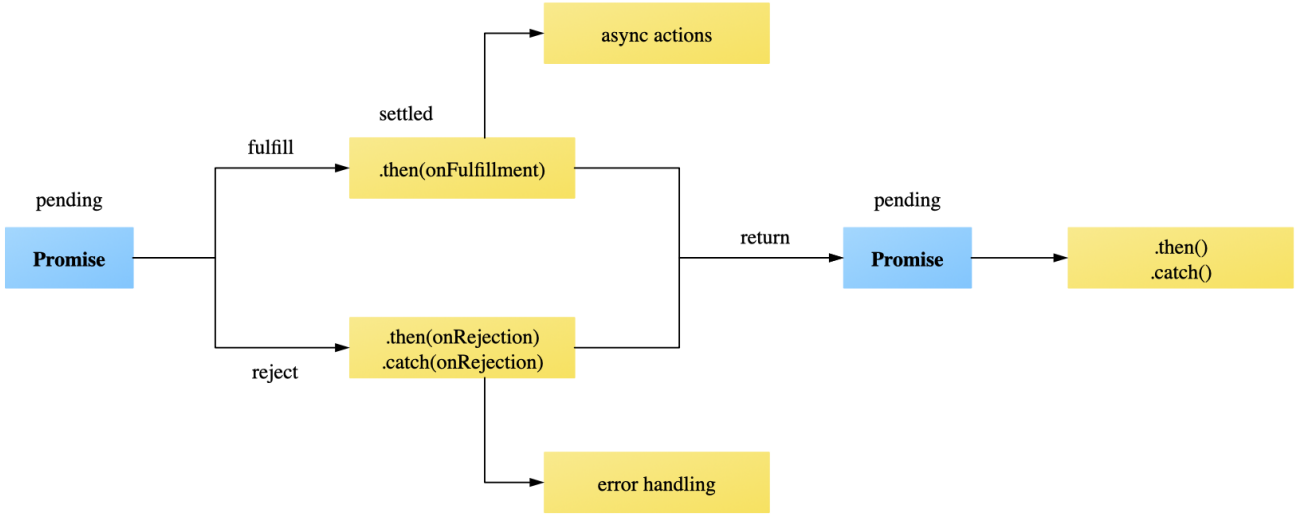
: 결과 값이 성공 혹은 실패로 반환된 상태

-> 한번 settled 된 값은 재실행되지 않음

Rejected 실패

: 비동기 처리가 실패하거나 오류가 발생한 상태

Promise의 처리과정



new Promise() 메소드 호출 시 바로 Pending

1. 콜백함수의 인자 **resolve** 실행 시 -> Fulfilled (Settled) -> **then()** 메소드 이용해서 **처리 결과값** 받음
2. 콜백함수의 인자 **reject** 호출 시 -> Rejected (Settled) -> **catch()** 메소드 이용해서 **실패처리 결과값** 받음

Promise 예제

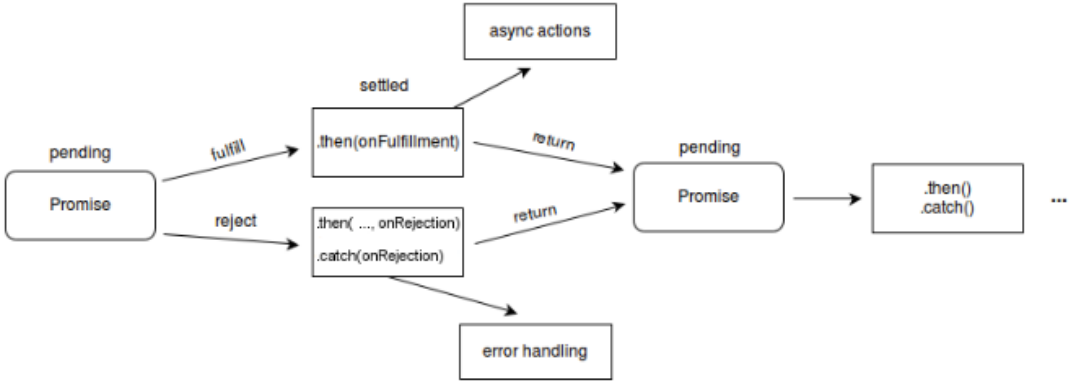
```
function getData(callback) {
 return new Promise(function(resolve, reject) {
  $.get('url', function(response) {
   if (response) {
    resolve(response) // 성공
   }
   reject(new Error("Request is fail")) // 실패
  })
 })
}


// resolve 시 then 메소드, reject 시 catch 메소드의 인자로 넘어간다.
getData().then(function (data) {
 console.log(data)
}).catch(function (err) {
 console.log(err)
});


// new Promise로 생성된 인스턴스는 객체이기 때문에 변수로 할당하거나 함수의 인자로 사용 가능
cosnt promise = new Promise((res, rej) => {
  setTimeout(() => {
    res(111);
  }, 1000);
});
```

then() 메소드는 다시 **Promise를 반환**함

-> **연속적으로** then 메소드 사용 가능

Promise Chaining

then 메소드를 연속적으로 사용한 Promise Chaining




```
doSomething() // doSomething의 반환 값은 then 함수의 인자로 전달된다.
  .then(res => doSomethingElse(result)) // doSomethingElse의 반환 값도 마찬가지
  .then(res => doThirdThing(newResult)) // doThirdThing 여기도 마찬가지
  .then(res => console.log(`Got the final result: ${finalResult}`)) // 위와 동일..
  .catch(err => console.error(err));
// 위의 Promise 객체들 중 하나라도 에러가 발생하면 catch 메소드로 넘겨진다.
```

여러 개의 Promise 객체들 중 **하나라도** 에러 발생 시 catch 메소드로 넘어감

```
new Promise(function(resolve, reject){
  setTimeout(function() {
    resolve(1);
  }, 2000);
})
.then(function(result) {
  console.log(result); // 1
  return result + 10;
})
.then(function(result) {
  console.log(result); // 11
  return result + 20;
})
.then(function(result) {
  console.log(result); // 31
});
```

then 메소드에서 반환되는 새로운 Promise 객체가 **다음 *then*의 인자로** 넘어감

Promise Chaining 사례

```
var userInfo = {
  id: 'test@abc.com',
  pw: '****'
};

function parseValue() {
  return new Promise({
    // ...
  });
}
function auth() {
  return new Promise({
    // ...
  });
}
function display() {
  return new Promise({
    // ...
  });
}

getData(userInfo)
  .then(parseValue)
  .then(auth)
  .then(diaplay);
```

```
new Promise((res, rej) => {
    console.log('Initial');

    res(); // resolve된 후 then 실행
})
.then(() => {
    throw new Error('Something failed');

    console.log('Do this'); // error 발생으로 실행되지 않는다.
})
.catch(err => {
    console.log(err); // throw new Error의 인자 값이 넘어온자.
})
.then(() => {
    console.log('Do this, whatever happened before'); // catch 구문 이후 chaining
});


// 출력값
Initial
Something failed // ERROR
Do this, whatever happened before // catch 메소드 이후 then 메소드 실행
```

catch() 이후에도 chaining 가능

Promise 에러처리

1. **then() 메소드의 두번째 인자**
   : Promise 의 ***reject*** **가 호출**되어 실패 상태일 경우 실행됨

```
getData().then(
  handleSuccess,
  handleError
);
```

```
function getData() {
  return new Promise(function(resolve, reject) {
    reject('failed');
  });
}

// 1. then()의 두 번째 인자로 에러를 처리하는 코드
getData().then(function() {
  // ...
}, function(err) {
  console.log(err);
});
```

Js Copy

```
// then()의 두 번째 인자로는 감지하지 못하는 오류
function getData() {
  return new Promise(function(resolve, reject) {
    resolve('hi');
  });
}

getData().then(function(result) {
  console.log(result);
  throw new Error("Error in then()"); // Uncaught (in promise) Error: Error in then
}, function(err) {
  console.log('then error : ', err);
});
```

-> then의 첫번째 **콜백 함수 내부의 오류**는 제대로 잡아내지 못하는 단점

Promise 에러처리

2. catch 메소드 이용

: 마찬가지로 ***reject*** **가 호출**되어 실패 상태일 경우 실행됨

```
getData().then().catch();
```

Js Copy

```
function getData() {
  return new Promise(function(resolve, reject) {
    reject('failed');
  });
}
// 2. catch()로 에러를 처리하는 코드
getData().then().catch(function(err) {
  console.log(err);
});
```

Js Copy

```
// catch()로 오류를 감지하는 코드
function getData() {
  return new Promise(function(resolve, reject) {
    resolve('hi');
  });
}

getData().then(function(result) {
  console.log(result); // hi
  throw new Error("Error in then()");
}).catch(function(err) {
  console.log('then error : ', err); // then error :  Error: Error in then()
});
```

Js Copy

더 많은 예외 처리 가능 -> 가급적 catch 사용

더 새로운 해결책, async / await

Async / Await 구문

: async function 선언은 AsyncFunction 객체를 반환하는 하나의 **비동기 함수**를 정의함
비동기 함수는 암시적으로 **Promise**를 사용하여 **결과를 반환**함

- Promise 를 **대체하는 것은 아님**
- 콜백과 Promise의 **단점을 보완**하고 있음
- Promise를 이용하지만 then, catch 메소드로 컨트롤 하지 않고 동기적 코드처럼 **반환값을 변수에 할당**

장점

- 동기적으로 코드 작성 가능 (코드의 절차적 작성)
  - 비동기적 사고에서 벗어남
- 가독성이 훨씬 좋음

async / await 사용

```
async function 함수명() {
  await 비동기_처리_메서드_명();
}
```

- await 키워드는 async 함수에서만 사용 가능
- Promise 객체 반환하는 함수에 사용 가능
- 여러 개의 비동기 처리 코드 다룰 때 효과적

에러처리

- try catch 구문 사용

-> 네트워크 통신 오류, 일반적 오류도 모두 처리 가능

async / await 예제

```
function fetchUser() {
  var url = 'https://jsonplaceholder.typicode.com/users/1'
  return fetch(url).then(function(response) {
    return response.json();
  });
}

function fetchTodo() {
  var url = 'https://jsonplaceholder.typicode.com/todos/1';
  return fetch(url).then(function(response) {
    return response.json();
  });
}
```

Js Copy

```
async function logTodoTitle() {
  try {
    var user = await fetchUser();
    if (user.id === 1) {
      var todo = await fetchTodo();
      console.log(todo.title); // delectus aut autem
    }
  } catch (error) {
    console.log(error);
  }
}
```

Js Copy

여러 개의 비동기 처리 코드 다루기
try catch 에러처리

async / await 예제

```
console.log(0);

function promise() {
  console.log(1);

  return new Promise(resolve => {
    setTimeout(() => {

      console.log(2);
      resolve('resolved');
    }, 2000);
  });
}

console.log(3);

async function asyncCall() {
  try {
      console.log(4);

      const result = await promise(); // Promise가 settled될 때까지 기다린 후 resolve된 값을 할당한다.

      console.log(result);
      console.log(5);
  } catch(err) {
    console.error(err); // error 발생 시 catch 블락에서 잡히도록 handling
  }
}
```

```
console.log(6);

asyncCall();

// 출력 값
0
3
6
4  // asyncCall 호출
1  // promise 함수 호출
2  // 2초 후 setTimeout 콜백 함수 호출
resolved // resolve함수 호출
5  // await 후 다음 코드 실행
```

먼저 완료되어야할 이벤트들의 순서대로, 동기적 코드처럼 작성 가능 (비동기 상황은 어떤 이벤트가 먼저 완료될지 순서 불명확함)

실제 사용

- MDN async function 에서 예제 살펴보면 유용

https://developer.mozilla.org/ko/docs/Web/JavaScript/Reference/Statements/async_function

- ajax 요청 (axios, fetch 함수)

-> 자동으로 Promise 반환함 -> 반환된 객체를 async / await 으로 처리

- Redux Saga

참고자료

(모든 이미지, 코드 출처)

https://medium.com/@yoohl/자바스크립트-비동기-동기-ac9495e42d0

https://velog.io/@jiwon/Javascript는-동기일까-비동기일까

https://velog.io/@yejinh/비동기-파헤치기

https://medium.com/better-programming/is-javascript-synchronous-or-asynchronous-what-the-hell-is-a-promise-7aa9dd8f3bfb

https://pro-self-studier.tistory.com/89

https://joshua1988.github.io/web-development/javascript/javascript-asynchronous-operation/

https://joshua1988.github.io/web-development/javascript/promise-for-beginners/

https://joshua1988.github.io/web-development/javascript/js-async-await/