




















Promise, async, await

- 1. [] [] [] []
- 2. Callback [] []
- 3. Callback -> Promise -> async/await

1.

#

-                   

- 在 3000ms 后，a 的值是 10。
- 在 3000ms 后，a 的值是 10，并且打印 'Finished'。

```
let a = 10
```

```
setTimeout(function callback() {
  console.log('a : ', a)
}, 3000)
```

```
console.log('Finished')
```

```
//Finished
```

```
//a: 10
```

异步编程

- 异步编程是指，在 JavaScript 中，某些操作不会立即执行，而是会在稍后的时间执行。
- 在 JavaScript 中，异步编程通常是指使用回调函数、Promise、async/await 等方式。
- 在 JavaScript 中，API (Application Programming Interface) 是指应用程序接口。
- 在 JavaScript 中，API 通常是指 setImmediate, setTimeout, XMLHttpRequest, fetch 等 Web API。

同步编程 异步编程 混合编程

在 JavaScript 中，同步编程是指，在代码执行过程中，遇到需要等待的操作时，会一直等待，直到操作完成后再继续执行后续代码。而异步编程是指，在代码执行过程中，遇到需要等待的操作时，不会一直等待，而是会在稍后的时间继续执行后续代码。

那么，在 JavaScript 中，如何实现异步编程呢？

在 JavaScript 中，实现异步编程通常有两种方式：一种是使用回调函数，另一种是使用 Promise。回调函数是指在代码执行过程中，遇到需要等待的操作时，将后续代码作为回调函数传递给该操作，待操作完成后，再调用该回调函数。Promise 是一种对象，它代表一个未来可能发生的操作，可以通过 then 方法链式调用，实现异步编程。

同步编程 异步编程 混合编程

1. API 同步 (同步编程 异步编程 混合编程)

在 JavaScript 中，同步编程是指，在代码执行过程中，遇到需要等待的操作时，会一直等待，直到操作完成后再继续执行后续代码。而异步编程是指，在代码执行过程中，遇到需要等待的操作时，不会一直等待，而是会在稍后的时间继续执行后续代码。

```
function fetchDataFromAPI() {
  console.log("API 호출 시작 ...");
  // 3초 동안 API 호출을 기다리는 시간
  setTimeout(() => {
    console.log("API 호출 완료!");
    // API 호출 결과
  }, 3000);

  console.log("API 호출이 끝났습니다.");
}

fetchDataFromAPI();
```

<API>

```
API 호출 시작 ...
API 호출 완료 ...
(3초 동안)
API 호출 완료 !
```

이 코드는 API 호출을 시작하고, 3초 동안 기다린 후, API 호출이 완료되었음을 로그에 출력합니다.

2. 파일 읽기 (fs 모듈을 사용하여)

Node.js에서 파일을 읽기 위해서는 fs 모듈을 사용해야 합니다. fs 모듈은 Node.js의 기본 모듈 중 하나이며, 파일 시스템과 관련된 작업을 수행하는 데 사용됩니다.

<fs> (Node.js 모듈)

```
const fs = require('fs');

console.log("파일 읽기 시작");

fs.readFile('example.txt', 'utf8', (err, data) => {
  if (err) {
    console.log("파일 읽기 실패:", err);
  } else {
    console.log("파일 읽기 성공:", data);
  }
});
```

```
console.log("👋 👋 👋 👋 👋 👋 ");
```

<[]>

```
👋 👋 👋  
👋 👋 👋 👋 👋 👋  
(👋 👋 👋 )  
👋 👋 : (👋 👋 👋 )
```

👋 👋 👋 👋 👋 👋 👋 👋 👋. 👋 👋 👋 👋 👋, 👋 👋 👋
👋 👋 👋 👋 👋 👋 👋.

3. 👋 👋 👋

👋 👋 👋 👋 👋 👋 👋 👋 👋 👋. 👋 👋, 👋 👋
👋 👋 👋 👋 👋 👋 👋 UI 👋 👋 👋 👋 👋.

<[]>

```
document.getElementById("submitButton").addEventListener("click", function() {  
  console.log("👋 👋 👋 ...");  
  
  // 👋 👋 👋 👋 👋  
  setTimeout(() => {  
    console.log("👋 👋 👋 👋 !");  
  }, 2000);  
  
  console.log("👋 👋 👋 👋 ...");  
});
```

<[]>

```
👋 👋 👋 ...  
👋 👋 👋 👋 ...  
(2👋 👋 )  
👋 👋 👋 👋 !
```

👋 👋 👋 👋 👋 👋 👋 👋 👋. 👋 👋 👋 👋 👋 👋
👋 👋 👋 👋 👋 👋.

阻塞 非阻塞 异步

=> 阻塞 阻塞 阻塞 阻塞 阻塞, 阻塞 阻塞 阻塞 阻塞 阻塞 阻塞,
阻塞 setTimeout WEB API 阻塞 阻塞 阻塞 task queue 阻塞 阻塞
阻塞.

- 阻塞 阻塞 阻塞 阻塞 阻塞 阻塞 阻塞 阻塞.
- 阻塞 阻塞(event loop), 阻塞 阻塞(task queue), 阻塞 阻塞(job queue) 阻塞 阻塞.
- API 阻塞 阻塞 阻塞 阻塞 阻塞 阻塞 阻塞.
- 阻塞 阻塞 阻塞 阻塞 阻塞 阻塞 阻塞 阻塞(API 阻塞 阻塞 阻塞) 阻塞
阻塞 阻塞 阻塞 阻塞 阻塞 阻塞 阻塞 阻塞.

image.png

```
request("user-data", (userData) => {  
  console.log("userData")  
  saveUsers(userData)  
});  
  
console.log("DOM")  
console.log("阻塞")
```

request 阻塞 阻塞 阻塞 阻塞 阻塞, userData 阻塞 task queue
阻塞 阻塞 阻塞.

2. Callback 함수

1. 콜백 함수 (Callback)

콜백 함수?

콜백 함수란 어떤 함수를 호출할 때, 그 함수가 끝나고 난 후에 수행할 작업을 전달하는 함수를 말합니다.

```
function sayHello() {  
  console.log("Hello!");  
}  
  
function executeCallback(callback) {  
  console.log("Before callback");  
  callback(); // 콜백 함수 호출  
  console.log("After callback");  
}  
  
executeCallback(sayHello);
```

<예제>

1. executeCallback 함수 호출, sayHello 함수가 호출됨
2. console.log("Before callback") 실행
3. sayHello()가 호출되어 "Hello!" 출력
4. console.log("After callback") 실행

<결과>

```
Before callback  
Hello!  
After callback
```

2. 堆棧 (Call Stack) 是什麼？

堆棧(Call Stack)

堆棧是 JavaScript 引擎用來追蹤函式調用的棧結構。當一個函式被調用時，它會被推入堆棧。當該函式執行完畢後，它就會從堆棧中彈出。堆棧中的每個元素都包含函式的上下文、參數和返回地址。堆棧的增長和縮減是動態的，它反映了當前正在執行的函式調用序列。

3. 事件循環 (Callback Queue) 是什麼？

事件循環(Callback Queue)

事件循環是 JavaScript 引擎用來處理所有待執行的任務的循環。它會不斷地從任務队列中取出任務並執行它們。任務队列中的任務包括：DOM 事件、setTimeout 和 setInterval 的回调函式、Promise 的 then 和 catch 方法等。事件循環的執行順序是：先執行主線任務，然後執行所有在任務队列中的任務。

```
console.log("Start");

setTimeout(() => {
  console.log("Timeout callback");
}, 1000);

console.log("End");
```

<輸出結果>

- console.log("Start") 執行 → "Start" 輸出
- setTimeout 執行 → 將任務推入任務队列 → 等待 1000ms 後執行
- console.log("End") 執行 → "End" 輸出
- 1000ms 後，任務队列中的任務被執行 → "Timeout callback" 輸出

<輸出結果>

Start
End
Timeout callback

4. 타이머 타이머

타이머 타이머 타이머

타이머 타이머

타이머 타이머 타이머, 타이머 타이머 타이머 타이머

타이머 타이머 타이머 타이머 타이머 타이머 타이머 타이머

타이머 타이머 + 타이머 타이머 + 타이머 타이머

```
console.log("Start");

setTimeout(() => {
  console.log("Callback 1");
}, 1000);

setTimeout(() => {
  console.log("Callback 2");
}, 500);

console.log("End");
```

<타이머 타이머>

1. console.log("Start") 타이머 → 타이머 타이머 타이머 → "Start" 타이머
2. setTimeout 타이머 → 타이머 타이머 타이머 타이머 타이머 (500ms 타이머 1000ms)
3. console.log("End") 타이머 → 타이머 타이머 타이머 → "End" 타이머
4. 500ms 타이머, 타이머 타이머 타이머(Callback 2) 타이머 타이머 타이머 타이머 → 타이머 타이머 → "Callback 2" 타이머
5. 1000ms 타이머, 타이머 타이머 타이머(Callback 1) 타이머 타이머 타이머 타이머 → 타이머 타이머 → "Callback 1" 타이머

< >

Start

End

Callback 2

Callback 1

#

1. (Call Stack)

JavaScript 栈 结构 存储 函数 调用 记录, 每个 函数 调用 记录 组成 调用 栈(Call Stack) 结构. 栈 遵循 先进 后出 原则 进行 函数 调用 和 返回.

2. (Callback Queue)

当 遇到 异步 操作 函数 (如: setTimeout, Promise) 时 函数 不会 立即 执行 而是 被 放入 回调 队列 中 等待 执行. 当 异步 操作 完成 后 函数 会被 放入 回调 队列. 当 回调 队列 不为 空 时 事件 循环 会 从 回调 队列 中 取出 函数 并 放入 调用 栈 中 执行.

3. (Event Loop)

事件 循环 会 不断 检查 回调 队列 是否为 空, 如果 不为 空 则 从 回调 队列 中 取出 函数 并 放入 调用 栈 中 执行.

4. (Callback 函数)

回调 函数 是指 在 主 函数 内部 定义 的 函数, 通常 用于 处理 异步 操作 的 结果.

#

```
function asd(asdf, callback) {  
  console.log("a");  
}
```

```
callback(); // sdf() 실행  
console.log("b");  
}  
  
function sdf() {  
  console.log("c");  
}  
  
asd("name", sdf); // 실행
```

실행 순서(시간 순서)

asd("name", sdf) 실행: 먼저 asd 실행 후 -> console.log("a") 실행: "a" 실행 -> 실행 후,
console.log 실행 후 실행 순서

callback() 실행: callback()은 실행 후 sdf 실행 후 sdf() 실행 -> sdf 실행 후 실행 후 -> sdf
실행 후 console.log("c")은 실행 후 "c" 실행. ->실행 후, sdf 실행 후 실행 순서.

먼저 asd 실행 후 console.log("b")은 실행. -> "b" 실행 -> console.log("b")은 실행 후 실행 순서 -
> asd 실행 후 실행 후 실행 순서

<실행 순서>

```
a  
c  
b
```

실행 순서 (setTimeout, Promise.then, fetch)은 실행 순서 실행 순서
실행 순서 후 실행 후 실행 순서 실행. 실행, 실행 후 실행 순서
실행 순서 후 실행 순서 실행.

실행 순서 실행

asd 실행

```
asd
```

console.log("a") 실행

```
asd  
console.log
```

sdf

```
asd
sdf
```

console.log("c")

```
asd
sdf
console.log
```

sdf → console.log("b")

```
asd
console.log
```

asd

```
(  )
```

3. Callback -> Promise -> async/await

1. Callback

callback 是函数调用的回调函数。

比如 setTimeout 2 秒后 Dain 回调函数调用的回调函数。

```
function getName(cb) {  
  setTimeout(() => {  
    cb("Dain");  
  }, 2000);  
}
```

比如 setTimeout 2 秒后 Dain 回调函数调用的回调函数。

```
getName((name) => {  
  console.log(name);  
})  
// 2 秒后 Dain
```

比如 setTimeout 2 秒后 Dain 回调函数调用的回调函数？

比如 setTimeout 2 秒后 Dain 回调函数调用的回调函数。

```
getName((name) => {  
  console.log(name);  
})  
  
getName((name) => {  
  console.log(name);  
})
```

```

getName((name) => {
  console.log(name);
})
// 2초 뒤
// Dain
// Dain
// Dain

```

이름을 받아 2초 뒤 console.log(name)을 호출하는 함수를 만들어주세요.

이름, 나이, 주소를 받아 2초 뒤 console.log(name, age, address)를 호출하는 함수를 만들어주세요.

```

function getName(cb) {
  setTimeout(() => {
    cb("Dain");
  }, 2000);
}

function getAge(cb) {
  setTimeout(() => {
    cb(30);
  }, 2000);
}

function getAddress(cb) {
  setTimeout(() => {
    cb("Seoul");
  }, 2000);
}

```

이름, 나이, 주소를 받아 2초 뒤 console.log(name, age, address)를 호출하는 함수를 만들어주세요.

```

getName((name) => {
  getAge((age) => {
    getAddress((address) => {
      console.log(name, age, address)
    })
  })
})

```

이름, 나이, 주소를 받아 2초 뒤 console.log(name, age, address)를 호출하는 함수를 만들어주세요.

이름은 3초 안에, 나이 2초 안에 6초 안에 Dain 30 Seoul 로그를 찍는다.

이름을 먼저 출력한다. 이름은 3초 안에, 나이 2초 안에 출력한다.

이름과 나이를 한 줄에 출력하는 방법은?

2. Promise

Promise는 비동기 작업을 동기적으로 처리할 수 있게 해준다.

이름, 나이, 주소를 각각 getName, getAge, getAddress로 받아 Promise로 반환하는 함수를 만든다.

```
function getName() {
  return new Promise((resolve) => {
    setTimeout(() => {
      resolve("Dain");
    }, 2000);
  })
}

function getAge() {
  return new Promise((resolve) => {
    setTimeout(() => {
      resolve(30);
    }, 2000);
  })
}

function getAddress() {
  return new Promise((resolve) => {
    setTimeout(() => {
      resolve("Seoul");
    }, 2000);
  })
}
```

이 Promise를 사용하여 이름과 나이를 2초 후에 resolve된 값을 받아, 출력한다.

이름과 나이를 한 줄에 출력하는 방법은?

```
getName().then((res) => {
  console.log(res);
})

getAge().then((res) => {
  console.log(res);
})

getAddress().then((res) => {
  console.log(res);
})
```

□□ □□ □□□ □□ □□ □□□ □□□ □□□□.

□□□ □□□ □□ □□ □□□ □ □□□.

```
Promise
  .all([getName(), getAge(), getAddress()])
  .then((res) => {
    const [name, age, address] = res;
    console.log(name, age, address)
  })
```

Promise.all([Promise.resolve(1), Promise.resolve(2), Promise.resolve(3), Promise.resolve(4), Promise.resolve(5)]).

getName, getAge, getAddress 각각 Promise.all로 묶어서
 .then, Promise.all로 묶어서 실행할 수 있다. 이때 Promise
 2개 Dain 30 Seoul을 각각 Promise로 묶어서 실행하면
 각각의 결과를 얻을 수 있다.

3. async/await

□ □□ □ □□□ □□□ □□ □□.

1. 在 `main` 函数中，使用 `std::async` 来启动异步任务。

await [] [] resolve [] [] [] [] [] [] [] [] 6 [] [] Dain 30
Seoul [] [] [] [].


```
(async () => {  
  const name = await getName();  
  const age = await getAge();  
  const address = await getAddress();  
  
  console.log(name, age, address);  
})();
```



Promise의 async/await은 코드를 좀 더 읽기 쉽습니다. 코드를 좀 더 쉽게 읽을 수 있습니다. 코드를 좀 더 쉽게 읽을 수 있습니다 Promise의 코드를 좀 더 쉽게 읽을 수 있습니다. 코드를 좀 더 쉽게 읽을 수 있습니다 asyene/await을 사용합니다.