Asynchronous Programming

Ba Nguyễn (Updated 2021)

Sync vs Async

Synchronous - Chương trình sẽ thực thi các câu lệnh được thực hiện lần lượt theo thứ tự trong mã, câu lệnh trước đó phải hoàn thành mới xử lý câu lệnh tiếp theo.

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

for (let i = 0; i < 100; i++) {
    console.log(" ... ");
}

console.log("End");

// Start  ...  End
```

Sync vs Async

Ưu điểm của ***synchronous***:

- Các câu lệnh trong trương trình được thực thi lần lượt sẽ dễ kiểm soát hơn
- Nếu một câu lệnh có lỗi chương trình sẽ dừng mà không chạy tiếp

Nhược điểm của ***synchronous***:

- Khi chương trình cần thao tác với dữ liệu bên ngoài (như truy vấn dữ liệu, lấy dữ liệu từ server, đọc ghi file, ...) và mỗi thao tác cần một khoảng thời gian nhất định để thực thi. Khi đó, nếu tất cả thao tác được xử lý đồng bộ sẽ cần rất nhiều thời gian để hoàn thành

Sync vs Async

Asynchronous - Chương trình sẽ thực thi tất cả câu lệnh cùng một lúc, các câu lệnh sau có thể chạy mà không cần câu lệnh trước đó đã thực hiện xong hay chưa. Rất nhiều chức năng có sẵn trong JavaScript cho phép lập lịch các thao tác *bất đồng bộ*.

```
function asynchronous(process) {
    console.log("Start");

    setTimeout(() => {
        process();
    }, 2000);

    console.log("Running");
}

let end = () => console.log("End");

asynchronous(end); // Start Running End
```

Sync vs Async

Ưu điểm của ***asynchronous***:

- Tối ưu thời gian chạy của chương trình
- Luồng xử lý của chương trình không bị ảnh hưởng bởi các thao tác cần nhiều thời gian

Nhược điểm của ***asynchronous***:

- Các câu lệnh không thực hiện theo đúng thứ tự, đồng thời kết quả trả về cũng không đúng thứ tự khiến cho việc kiểm soát và gỡ lỗi rất khó khăn

Các cách lập trình bất đồng bộ trong JavaScript:

- *Callback*
- *Promise*
- *Async/await*

setTimeout()

setTimeout() cho phép "đặt lịch" cho một hành động nào đó, sẽ được thực thi sau một khoảng thời gian nhất định

Cú pháp:

```
let timer = setTimeout(func | code[, delay][, args]);
clearTimeout(timer);
```

Ví dụ:

```
let timmer = setTimeout(() => {
    alert("Hello babe");
}, 1000); // thông báo alert() sau 1s
```

```
let timer = setTimeout(function hi() {
    alert("Hello babe");
    timer = setTimeout(hi, 1000);
}, 1000); // lặp thông báo alert() mỗi 1s
```

setInterval()

setInterval() cho phép "đặt lịch" cho một hành động nào đó, sẽ được thực thi lặp đi lặp lại sau mỗi khoảng thời gian nhất định

Cú pháp:

```
let timer = setInterval(func | code[, delay][, args]);
clearInterval(timer);
```

Ví dụ:

```
let timer = setInterval(() => {
    console.log("I love you!");
}, 1000); // mỗi 1s in ra "I love you!"
```

```
let seconds = 1;
let timer = setInterval(() => {
    console.log(seconds++);
}, 1000); // mỗi 1s in ra seconds
```

Callback

Cách tiếp cận đơn giản với lập trình bất đồng bộ đó là cung cấp cho các hàm bất đồng bộ thêm một tham số được gọi là hàm *callback*, khi nó thực hiện xong tác vụ, hàm *callback* sẽ được gọi cùng với kết quả đó.

```
function asynchronous(callback) {
    console.log("Start process");
    setTimeout(() => {
        let data = doSomething();
        callback(data);
    }, 10000);
}

asynchronous((data) => console.log(data));
```

Handling errors

Với các quy trình bất đồng bộ, một công việc bắt đầu tại một thời điểm nhưng kết thúc ở một thời điểm khác. Khi công việc hoàn thành nó gọi lại hàm *callback* với kết quả của nó, nhưng cũng có trường hợp công việc đó không thể hoàn thành (*lỗi*), hàm *callback* có thể sử dụng thêm một tham số lỗi để xử lý trường hợp đó

```
function asynchronous(callback) {
    console.log("I'm doing my job");
    // if error
    callback(error, null); // error = error, data = null
    // else
    callback(null, data); // error = null, data = data
}
```

Handling errors

Với các quy trình bất đồng bộ, một công việc bắt đầu tại một thời điểm nhưng kết thúc ở một thời điểm khác. Khi công việc hoàn thành nó gọi lại hàm *callback* với kết quả của nó, nhưng cũng có trường hợp công việc đó không thể hoàn thành (*lỗi*), hàm *callback* có thể sử dụng thêm một tham số lỗi để xử lý trường hợp đó

```
asynchronous((error, data) ⇒ {
    if (error) {
        console.log("Failure");
    } else {
        console.log(data);
    }
});
```

Callback hell

Một hạn chế của việc sử dụng *callback* trong lập trình bất đồng bộ là khi muốn thực hiện nhiều thao tác bất đồng bộ, việc lồng *callback* dẫn tới tình trạng mã rất khó đọc, khó bảo trì, ... (được gọi là *callback hell*)

```
asynchronous(() => {
    asynchronous(() => {
        asynchronous(() => {
            asynchronous(() => {
                asynchronous(() => {
                    asynchronous(() => {
                        // lol
                    })
                })
            })
        })
    })
});
```

Promise

ES6 cung cấp cách thức mới để lập trình bất đồng bộ, khắc phục được nhược điểm của *callback* đó là `Promise`. `Promise` ***là đối tượng đặc biệt, đại diện cho một sự kiện sẽ xảy ra trong tương lai***, nó thực hiện một thao tác nào đó và tại thời điểm hoàn thành, nó sẽ thông báo đến tất cả các mã *đang chờ nhận kết quả* từ nó.

Khởi tạo một `Promise` mới:

```
let promise = new Promise((resolve, reject) ⇒ {
    // Do something here
});
```

Promise

`Promise` nhận một hàm *callback* (được gọi là *executor*), nó sẽ tự động thực thi khi `Promise` mới được khởi tạo. Tham số `resolve`, `reject` là *callback* đặc biệt, khi *executor* thực thi xong và tạo ra kết quả (bất kể thành công hay có lỗi) nó sẽ gọi một trong 2 hàm `resolve` hoặc `reject`:

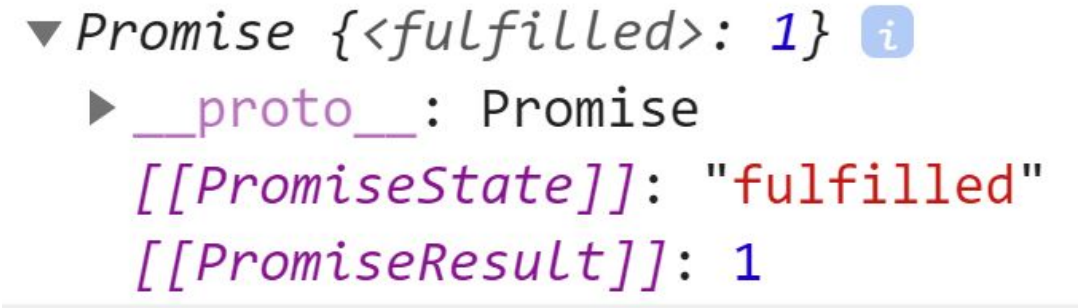
- Nếu thao tác thành công, kết thúc `Promise` với kết quả - `resolve(data)`
- Nếu có lỗi, kết thúc `Promise` với lỗi - `reject(error)`

```
let promise = new Promise((resolve, reject) ⇒ {
    let { error, data } = doSomething();
    if (data) {
        resolve(data);
    }
    reject(error);
});
```

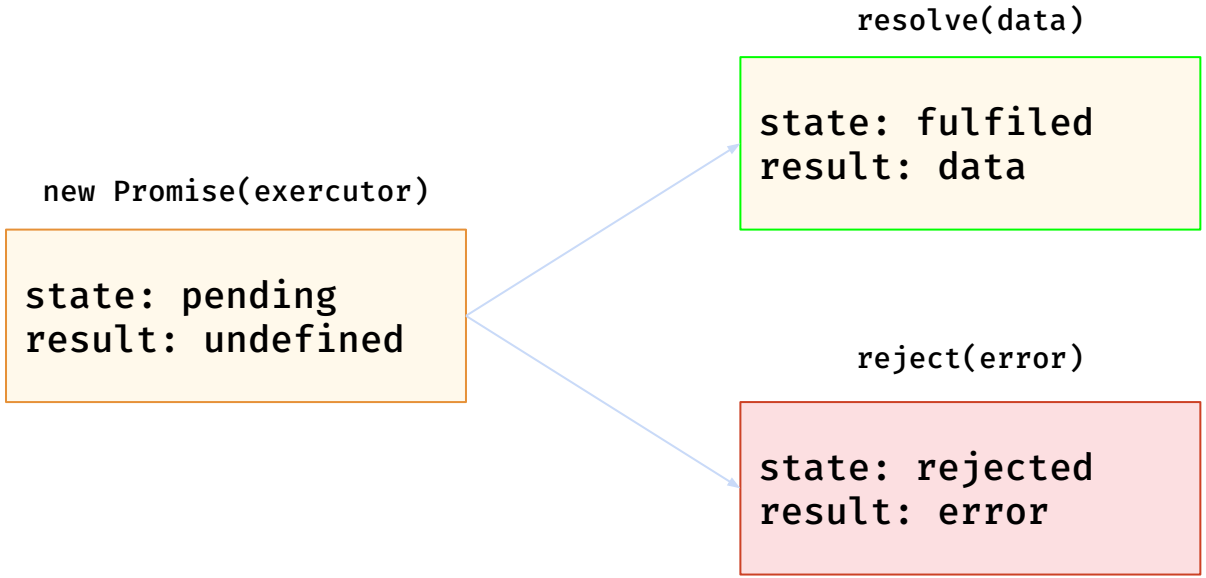
Promise

Mỗi đối tượng `Promise` được tạo đi kèm 2 thuộc tính đặc biệt:

- `state` - trạng thái của `Promise`, giá trị khi khởi tạo là `pending`, khi `resolve` được gọi giá trị của nó là `fulfiled` và `rejected` khi `reject` được gọi
- `result` - kết quả của `Promise`, giá trị khi khởi tạo là `undefined`, khi `resolve(data)` được gọi giá trị của nó là `data` và `error` khi `reject(error)` được gọi



Promise



Promise

💡 Note

- Chỉ có thể có 1 lệnh trả về kết quả là `resolve` hoặc `reject`
- Khi gọi `reject`, nên trả về một đối tượng `Error()`
- Có thể gọi `resolve` hoặc `reject` ngay lập tức

```
let promise = Promise.resolve("LoL");
let rj = Promise.reject(new Error("LoL"));
```

Promise Handler

Sau khi gọi `resolve` hoặc `reject`, `Promise` trả về một kết quả (bất kể hoàn thành hay lỗi). Để xử lý kết quả đó, `Promise` cung cấp các trình xử lý:

- `then()` - xử lý kết quả của `Promise`, bao gồm cả hoàn thành hoặc lỗi
- `catch()` - xử lý trường hợp lỗi (thay cho `then()`)
- `finally()` - luôn chạy bất kể hoàn thành hoặc lỗi, khác một chút với `then()`

```
let promise = new Promise((r, e) => {
    // do something
})
    .then((success) => {})
    .catch((error) => {})
    .finaly(() => {});
```

Promise Handler

Phương thức `then()` nhận 2 tham số là 2 hàm *callback* tương ứng với 2 trường hợp hoàn thành hoặc lỗi, mỗi hàm được gọi với giá trị tương ứng

```
let promise = new Promise((r, e) ⇒ {
    // do something
}).then(
    function (success) {
        // Success, do something with data
    },
    function (error) {
        // Fail, do something with error
    }
);
```

Promise Handler

Phương thức `catch()` là cú pháp ngắn gọn thay thế cho trường hợp lỗi của `then()`

```
let promise = new Promise((r, e) ⇒ {
    // do something
})
    .then(function (success) {
        // Success, do something with data
    })
    .catch(function (error) {
        // Fail, do something with eror
    });
```

Lab

1. Viết hàm `capitalize(param, ms)` nhận vào 2 tham số, trả về một `Promise` (sau số milisecond là tham số `ms`). Nếu `param` là chuỗi, chuyển đổi nó thành dạng `capitalize` và gọi `resolve` với giá trị đó, nếu `param` không phải chuỗi, `reject` nó với thông báo lỗi. Hiển thị kết quả lên màn hình `console`
2. Sửa đổi hàm `capitalize(param, ms)`, `param` là một mảng chuỗi, chuyển đổi tất cả chuỗi trong mảng thành dạng `capitalize` và `resolve` mảng đó. Nếu một phần tử không phải là chuỗi, `reject` với thông báo lỗi. Hiển thị kết quả lên màn hình `console`

Promise chaining

Các trình xử lý `then()`, `catch()`, `finally()` cũng trả về một `Promise` mới, có thể kết hợp nhiều trình xử lý để tạo thành một chuỗi các tác vụ

```
let promise = Promise.resolve(1)
    .finally(() => console.log("F1"))
    .then((r) => r + 1)
    .catch((e) => console.log(e))
    .then((r) => r + 2)
    .then((r) => r + 3)
    .then((r) => console.log(r))
    .finally(() => console.log("Finally")); // F1 7 Finally
```

Promise API

`Promise.all()` nhận vào một danh sách các `Promise`, chờ đợi tất cả chúng hoàn thành và một `Promise` mới chứa kết quả của chúng. Nếu một trong các `Promise` gọi `reject()`, `Promise.all()` cũng sẽ `reject` với lỗi đó

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

Promise API

`Promise.race()` nhận vào một danh sách `Promise`, tuy nhiên nó chỉ đợi kết quả sớm nhất trả về (bất kể thành công hay lỗi) và bỏ qua những `Promise` khác

```
let promise = Promise.race([
    Promise.resolve(1),
    Promise.reject("Failure"),
    Promise.resolve(2),
]).then((r) ⇒ console.log(r));
// 1
```

Promise API

`Promise.any()` nhận vào một danh sách `Promise`, nó đợi `Promise` đầu tiên thành công (khác `Promise.race()`), nếu tất cả đều `reject`, nó sẽ `reject` về một danh sách lỗi.

```
let promise = Promise.race([
    Promise.resolve(1),
    Promise.reject("Failure"),
    Promise.resolve(2),
]).then((r) ⇒ console.log(r));
// 1
```

Promise API

`Promise.allSettled()` nhận vào một danh sách `Promise`, chờ đợi tất cả `Promise` hoàn thành công việc và trả về toàn bộ kết quả, bao gồm cả thành công và lỗi

```
let promise = Promise.allSettled([
    Promise.resolve(1),
    Promise.reject("Failure"),
]).then(console.log);
/* [
    {status: "fulfilled", value: 1},
    {status: "rejected", reason: "Failure"},
] */
```

Async/Await

`Async` function tự động đặt kết quả trả về từ một hàm vào một `Promise`, thêm từ khóa `async` vào trước khai báo hàm

```
async function lol() {
    return "😘";
}

lol().then((r) ⇒ console.log(r));
// 😘

function lol() {
    return Promise.resolve("😘");
}
```

Async/Await

Keyword `await` ***chỉ sử dụng được trong*** `async` ***function***, nó chờ một `Promise` hoàn thành và trả về kết quả.

```
async function demo() {
    let result = await new Promise((resolve) => {
        setTimeout(() => resolve("🙂"), 2000);
    });

    console.log(result);
}

demo(); // after 2 second 🙂
```

Async/Await

Khi sử dụng `async/await` có thể thay thế cho `then()`

```
async function demo() {
    let result = await new Promise((r) ⇒
        setTimeout(() ⇒ r("Success"), 2000));
    console.log(result);
}

function demo() {
    return new Promise((r) ⇒
        setTimeout(() ⇒ r("Success"), 2000));
}
demo().then((r) ⇒ console.log(r));
```