



**THỰC HỌC – THỰC NGHIỆP**

## JAVASCRIPT NÂNG CAO

BẤT ĐỒNG BỘ VÀ XỬ LÝ BẤT ĐỒNG BỘ  
(PHẦN 2)



- ⦿ Giải thích được khái niệm Callback Hell
- ⦿ Có thể xử lý code để tránh rơi vào tình huống tạo callback hell
- ⦿ Giải thích được khái niệm Promise
- ⦿ Có thể xử lý được bài toán bất đồng bộ bằng Promise

- 📖 Giải thích được khái niệm Callback Hell.
- 📖 Phương pháp xử lý để tránh Callback Hell
- 📖 Giải thích được khái niệm Promise
- 📖 Sử dụng được Promise




- 📖 Giải thích được khái niệm bất đồng bộ trong Javascript.
- 📖 Nguyên lý xử lý bất đồng bộ trong javascript
- 📖 Giải thích được khái niệm Callback
- 📖 Sử dụng được Callback





PHẦN 1:  
KHÁI NIỆM CALLBACK HELL

- ❑ Trong quá trình xử lý bất đồng bộ trong javascript, chúng ta đã được học cách giải quyết bằng callback, tuy nhiên nếu xử lý không tốt thì rất dễ xảy ra tình huống nhiều hàm callback lồng nhau.



```
1 function hell(win) {
2   // for listener purpose
3   return function() {
4     loadLink(win, REMOTE_SRC+'/assets/css/style.css', function() {
5       loadLink(win, REMOTE_SRC+'/lib/async.js', function() {
6         loadLink(win, REMOTE_SRC+'/lib/easyXDM.js', function() {
7           loadLink(win, REMOTE_SRC+'/lib/json2.js', function() {
8             loadLink(win, REMOTE_SRC+'/lib/underscore.min.js', function() {
9               loadLink(win, REMOTE_SRC+'/lib/backbone.min.js', function() {
10                loadLink(win, REMOTE_SRC+'/dev/base_dev.js', function() {
11                  loadLink(win, REMOTE_SRC+'/assets/js/deps.js', function() {
12                    loadLink(win, REMOTE_SRC+'/src/' + win.loader_path + '/loader.js', function() {
13                      async.eachSeries(SCRIPTS, function(src, callback) {
14                        loadScript(win, BASE_URL+src, callback);
15                      });
16                    });
17                  });
18                });
19              });
20            });
21          });
22        });
23      });
24    });
25  };
26 }
```

- ❑ Giống như hình minh họa callback hell là trạng thái code lồng nhau nhiều cấp dẫn đến hình thành hình kim tự tháp như hình minh họa, khái niệm callback hell chỉ cách code không tối ưu.
- ❑ Các vấn đề của callback hell:
  - ❖ Code nhìn phức tạp, khó nắm bắt, khó đọc
  - ❖ Khó debug lỗi
  - ❖ Giảm thẩm mỹ của code
  - ❖ Khó maintain

## ❑ Ví dụ 1:

```
function thuc_day(viecnaodo){
    viecnaodo();
}

function danh_rang(viecnaodo){
    viecnaodo();
}

function di_an_sang(viecnaodo){
    viecnaodo();
}

// Code không tối ưu
function main(){
    thuc_day(function(){
        danh_rang(function(){
            di_an_sang(function(){
                console.log('OMG!!!!');
            });
        });
    });
}
main();
```



OMG!!!!





## PHẦN 2: CÁC CÁCH XỬ LÝ CALLBACK HELL

## ❑ Các cách xử lý Callback Hell:

- ❖ Đặt tên cho hàm callback
- ❖ Thiết kế ứng dụng theo dạng module
- ❖ Định nghĩa hàm trước khi gọi
- ❖ Sử dụng Promises (ES2015 - ES6)
- ❖ Sử dụng Async/Await (ES2017 – ES8)

- ❑ Khi thực hiện viết hàm callback thì đa phần chúng ta sẽ không đặt tên (hàm vô danh) điều này không nên, hãy đặt tên cho chúng.

```
var form = document.querySelector('form')
form.onsubmit = function (submitEvent) {
  var name = document.querySelector('input').value
  request({
    uri: "http://example.com/upload",
    body: name,
    method: "POST"
  }, function (err, response, body) {
    var statusMessage = document.querySelector('.status')
    if (err) return statusMessage.value = err
    statusMessage.value = body
  })
}
```

✗ Không nên

```
var form = document.querySelector('form')
form.onsubmit = function formSubmit (submitEvent) {
  var name = document.querySelector('input').value
  request({
    uri: "http://example.com/upload",
    body: name,
    method: "POST"
  }, function postResponse (err, response, body) {
    var statusMessage = document.querySelector('.status')
    if (err) return statusMessage.value = err
    statusMessage.value = body
  })
}
```

✓ Nên

- ❑ Ý tưởng này được Isaac Schlueter (cha đẻ của npm) đưa ra thông qua phát biểu:

*"Hãy viết một chương trình thành những mô-đun nhỏ mỗi mô-đun phụ trách một việc cụ thể, và rồi hợp nhất chúng lại với nhau tạo thành một khối vững chắc hơn. Code sẽ không thể bị callback hell nếu tuân theo quy tắc này."*

- ❑ Với phiên bản đang áp dụng cho môn học này (ES5), chúng ta có thể tạo ra các đối tượng để thực hiện module hóa các hàm callback

```
var form = document.querySelector('form')
form.onsubmit = function (submitEvent) {
  var name = document.querySelector('input').value
  request({
    uri: "http://example.com/upload",
    body: name,
    method: "POST"
  }, function (err, response, body) {
    var statusMessage = document.querySelector('.status')
    if (err) return statusMessage.value = err
    statusMessage.value = body
  })
}
```

✗ Không nên

```
// tạo đối tượng formObject
var formObject = {
  // tạo phương thức formSubmit
  formSubmit: function () {
    var name = document.querySelector('input').value
    request({
      uri: "http://example.com/upload",
      body: name,
      method: "POST"
    }, this.postResponse()); // call phương thức postResponse
  },

  // tạo phương thức postResponse
  postResponse: function (err, response, body) {
    var statusMessage = document.querySelector('.status')
    if (err) return statusMessage.value = err
    statusMessage.value = body
  }
}

var form = document.querySelector('form');
form.onsubmit = formObject.formSubmit();
```

✓ Nên

- ❑ Định nghĩa hàm trước sau đó khi cần sử dụng làm callback thì chỉ cần gọi tên hàm

```
var form = document.querySelector('form')
form.onsubmit = function (submitEvent) {
  var name = document.querySelector('input').value
  request({
    uri: "http://example.com/upload",
    body: name,
    method: "POST"
  }, function (err, response, body) {
    var statusMessage = document.querySelector('.status')
    if (err) return statusMessage.value = err
    statusMessage.value = body
  })
}
```

✗ Không nên

```
function formSubmit() {
  var name = document.querySelector('input').value;
  request({
    uri: "http://example.com/upload",
    body: name,
    method: "POST"
  }, postResponse()); // call phương thức postResponse
}

function postResponse(err, response, body) {
  var statusMessage = document.querySelector('.status')
  if (err) return statusMessage.value = err
  statusMessage.value = body
}

var form = document.querySelector('form');
form.onsubmit = formSubmit();
```

✓ Nên



## PHẦN 3: KHÁI NIỆM PROMISE

- ❑ **Promise** là một đối tượng đặc biệt dùng cho các xử lý bất đồng bộ. Nó đại diện cho một xử lý bất đồng bộ và chứa kết quả cũng như các lỗi xảy ra từ việc xử lý bất đồng bộ.
- ❑ Một **Promise** đại diện cho một giá trị mà ta không cần phải biết ngay khi khởi tạo. Bằng các sử dụng **Promise** ta có thể kết hợp với các hàm xử lý khác để sử dụng kết quả sau khi thực thi xử lý bất đồng bộ mà nó đang đại diện. Vì vậy mà ta có thể lập trình bất đồng bộ gần giống với kiểu lập trình đồng bộ - tức là đợi xử lý bất đồng bộ xong mới thực thi các thao tác mà cần sử dụng tới kết quả của xử lý đó. Để có thể làm được việc đó thay vì trả ra kết quả của việc xử lý đồng bộ, **Promise** sẽ trả ra một *promise* khác. Bằng promise mới này ta lại có thể lặp lại việc sử dụng kết quả của thao tác xử lý lúc trước để làm đầu vào cho các thao tác xử lý lúc sau.

Nguồn: [https://developer.mozilla.org/vi/docs/Web/JavaScript/Reference/Global\\_Objects/Promise](https://developer.mozilla.org/vi/docs/Web/JavaScript/Reference/Global_Objects/Promise)



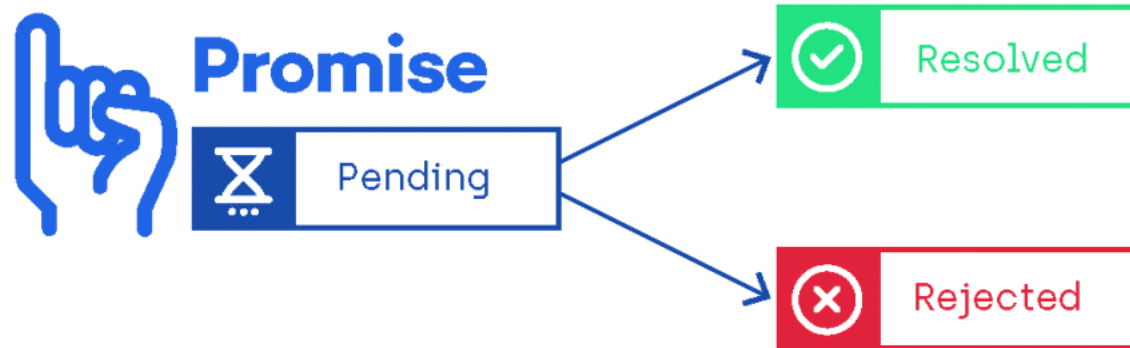
- ❑ Giải thích một cách ngắn gọn: Một object **Promise** đại diện cho một giá trị ở thời điểm hiện tại có thể chưa tồn tại, nhưng sẽ được xử lý và có giá trị vào một thời gian nào đó trong tương lai. Việc này sẽ giúp bạn viết các dòng code xử lý không đồng bộ trông có vẻ đồng bộ hơn.

- ❑ Tạo một Promise: tạo mới đối tượng Promise bằng cú pháp **new Promise**. Và hàm constructor của Promise nhận 2 tham số là 2 hàm **resolve** và **reject**.

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

## ❑ Các trạng thái của Promise:

- ❖ Pending (đang xử lý)
- ❖ Fulfilled (resolve - đã hoàn thành)
- ❖ Rejected (đã bị từ chối)



- ❑ Trạng thái Pending (đang xử lý): trạng thái đã tạo ra promise tuy nhiên chưa được thực thi

```
> var promise = new Promise(function(resolve, reject){ });  
console.log(promise);  
  
▼ Promise {<pending>} i  
  ► __proto__: Promise  
    [[PromiseStatus]]: "pending"  
    [[PromiseValue]]: undefined
```

- ❑ Trạng thái Fulfilled (resolve - đã hoàn thành): Promise đã được thực thi và return kết quả resolve(). Khi hàm resolve được trả về (chỉ nhận 1 giá trị duy nhất) – promise được phép gọi đến lệnh .then() – để tiếp tục thực hi

```
> var promise = new Promise(function(resolve, reject){
    setTimeout(function(){
        console.log(1);
        return resolve(2);
    }, 1000);
});

console.log(promise);

promise.then(function(data){
    console.log(data);
    console.log(promise);
});
```

► Promise {<pending>}

◀ ► Promise {<pending>}

1

2

▼ Promise {<resolved>: 2} ⓘ

- \_\_proto\_\_: Promise
- [[PromiseStatus]]: "resolved"
- [[PromiseValue]]: 2

- ❑ Trạng thái Reject (đã bị từ chối): Promise đã được thực thi và return kết quả reject(). Khi hàm reject được trả về (chỉ nhận 1 giá trị duy nhất) – promise được phép gọi đến lệnh .catch() – để tiếp tục thực hiện logic tiếp theo.
- ❑ Nếu không thực hiện hàm .catch() thì message báo lỗi sẽ chuyển thành error

```
> var promise = new Promise(function(resolve, reject){
  setTimeout(function(){
    console.log(1);
    return reject('message báo lỗi');
  }, 1000);
});

console.log(promise);

promise.catch(function(message){
  console.log(message);
  console.log(promise);
});
```

► Promise {<pending>}

◀ ► Promise {<pending>}

1

message báo lỗi

▼ Promise {<rejected>: "message báo lỗi"} ⓘ

- \_\_proto\_\_: Promise
- [[PromiseStatus]]: "rejected"
- [[PromiseValue]]: "message báo lỗi"

- ❑ Ví dụ 2: thực hiện chuyển màu màn hình sang màu vàng sau 1 giây, sau đó 1 giây chuyển sang màu xanh lá cây.

```
// Sử dụng Callback
setTimeout(function () {
    document.querySelector('body').style.background = 'yellow';
    setTimeout(function() {
        document.querySelector('body').style.background = 'green';
    }, 1000);
}, 1000);
```

```
// Sử dụng promise
function setColorPerTime(color, time) {
    return new Promise(function (resolve) {
        setTimeout(function () {
            document.querySelector('body').style.background = color;
            resolve();
        }, time);
    })
}

setColorPerTime('yellow', 1000)
    .then(function () {
        setColorPerTime('green', 1000);
    })
    .catch();
```

- ❑ Ví dụ 3: Đặt vấn đề bạn mong muốn sẽ đạt điểm môn “Lập trình javascript nâng cao” là trên 9 điểm tổng kết để đạt danh hiệu “sinh viên xuất sắc” – nếu đạt được điều kiện này nhà trường sẽ trao giải cho bạn một chuyến du lịch Singapore.
- ❑ Để biết được kết quả bạn có được đi Singapore hay không thì cần phải chờ sau khi thi hết block (1.5 tháng). Do đó chúng ta cần tạo ra 1 kịch bản và các hành động có thể xảy ra sau thời gian này (tạo ra promise)



# CÁCH TẠO RA PROMISE

```
<h2>Loading...</h2>

<script>
  var h2 = document.querySelector('h2');
  function quatrinhHoc(totalMark, time){
    return new Promise(function(resolve, reject){
      setTimeout(function(){
        if(totalMark >= 9){
          resolve(totalMark);
        }else{
          reject('Không đủ điều kiện sv xuất sắc');
        }
      }, time);
    })
  }

  var diemTongKet = prompt("Nhập điểm tổng kết:");
  // thời gian tính bằng mili giây (đại diện cho 1.5 tháng)
  var time = prompt("Thời gian delay: ");
  quatrinhHoc(parseInt(diemTongKet), parseInt(time))
    .then(function(mark){
      h2.innerHTML = 'Đi singapore, vì điểm tổng kết = ' + mark;
    })
    .catch(function(err){
      h2.innerHTML = err;
    })
  })
</script>
```

This page says

Nhập điểm tổng kết:

Cancel
OK

This page says

Thời gian delay:

Cancel
OK

**Loading...**



**Đi singapore, vì điểm tổng kết = 9**

❑ Thực hiện nhiều lệnh `.then()` liên tiếp: điều này hoàn toàn có thể thực hiện được với một điều kiện là tham số của hàm `.then()` là các hàm.

❖ Tuy nhiên có 1 chú ý là khi ta không sử dụng promise trong các hàm đến việc mất đi sự đồng bộ

```
var promise = new Promise(function(resolve, reject){
  setTimeout(function(){
    console.log(1);
    return resolve();
  }, 1000);
});
console.log(promise);

promise
  .then(function(){
    setTimeout(function(){
      console.log(2);
    }, 4000);
  })
  .then(function(){
    setTimeout(function(){
      console.log(3);
    }, 3000);
  })
  .then(function(){
    setTimeout(function(){
      console.log(4);
    }, 2000);
  });
```

► *Promise {<pending>}*

- 1 Sử dụng promise nên hiển thị đầu tiên
- 4 Hàm thông thường timeout 2000
- 3 Hàm thông thường timeout 3000
- 2 Hàm thông thường timeout 4000

❑ Thực hiện liên tiếp các Promise liên nhau cho các nghiệp vụ cần

```
var promise = new Promise(function(resolve, reject){
  setTimeout(function(){
    console.log(1);
    return resolve();
  }, 1000);
});
console.log(promise);

promise
  .then(function(){
    return new Promise(function(resolve, reject){
      setTimeout(function(){
        console.log(2);
        return resolve();
      }, 4000);
    })
  })
  .then(function(){
    return new Promise(function(resolve, reject){
      setTimeout(function(){
        console.log(3);
        return resolve();
      }, 3000);
    })
  })
  .then(function(){
    return new Promise(function(resolve, reject){
      setTimeout(function(){
        console.log(4);
        return resolve();
      }, 2000);
    })
  })
}
```

► *Promise {<pending>}*

1

2

3

4

- ❑ Một tính năng rất hay của **Promise** đó là có thể thực hiện nhiều promise cùng 1 lúc mà các hoạt động của chúng ko liên quan gì với nhau, nhưng kết quả của những promise đó cần thiết cho 1 hoạt động sau cùng.
- ❑ Hàm **Promise.all**, tham số truyền vào là 1 mảng các promise cần thực hiện. Khi đó, tham số của **.then()** chính là 1 mảng chứa các kết quả từ các promise.

- ❑ Nâng cấp ví dụ 3: điểm Assignment 1 được nhập sau 1s, điểm assignment2 được nhập sau 10s, sau đó tính điểm trung bình.

```
function assignment1(){
    return new Promise(function(resolve, reject){
        // chờ 1s để nhập điểm assignment 1
        setTimeout(function(){
            var ass1 = prompt("Nhập điểm assignment 1");
            return resolve(parseInt(ass1));
        }, 1000);
    });
}

function assignment2(){
    return new Promise(function(resolve, reject){
        // chờ 5s để nhập điểm assignment 2
        setTimeout(function(){
            var ass2 = prompt("Nhập điểm assignment 2");
            return resolve(parseInt(ass2));
        }, 5000);
    });
}

Promise.all([
    assignment1(),
    assignment2()
])
.then(function(result){
    // result là 1 mảng chứa các giá trị trả về từ các resolve
    console.log(result);
    var avgScore = (result[0] + result[1])/2;
    alert("Điểm trung bình của bạn là: " + avgScore);
});
```

This page says

Nhập điểm assignment 1

Cancel
OK

This page says

Nhập điểm assignment 2

Cancel
OK

This page says

Điểm trung bình của bạn là: 8

OK

- ☐ Tránh được Callback Hell
- ☐ Code rõ ràng, dễ đọc
- ☐ Dễ debug hơn
- ☐ Giải quyết được hầu hết các vấn đề bất đồng bộ

- ☑ Giải thích được khái niệm Callback Hell
- ☑ Biết các xử lý code để tránh gây ra Callback Hell
- ☑ Giải thích được khái niệm Promise
- ☑ Sử dụng được Promise trong xử lý bất đồng bộ







thank  
you!