Promise là gì?

Promise là một *cơ chế* trong JavaScript giúp bạn thực thi các tác vụ bất đồng bộ mà không rơi vào *callback hell* hay *pyramid of doom*, là tình trạng các hàm callback lồng vào nhau ở quá nhiều tầng. Các tác vụ bất đồng bộ có thể là gửi AJAX request, gọi hàm bên
trong `setTimeout`, `setInterval` hoặc `requestAnimationFrame`, hay thao tác với WebSocket hoặc Worker… Dưới đây là một callback hell điển hình.

```
api.getUser('pikalong', function(err, user) {
  if (err) throw err
  api.getPostsOfUser(user, function(err, posts) {
    if (err) throw err
    api.getCommentsOfPosts(posts, function(err, comments) {
      // vân vân và mây mây...
    })
  })
})
```

Ví dụ trên khi được viết lại bằng Promise sẽ là:

```
api.getUser('pikalong')

  .then(user => api.getPostsOfUser(user))

  .then(posts => api.getCommentsOfPosts(posts))

  .catch(err => { throw err })
```

Để tạo ra một promise object thì bạn dùng class Promise có sẵn trong trình duyệt như sau:

```
const p = new Promise( /* executor */ function(resolve, reject) {

  // Thực thi các tác vụ bất đồng bộ ở đây, và gọi
`resolve(result)` khi tác

  // vụ hoàn thành. Nếu xảy ra lỗi, gọi đến `reject(error)`.

})
```

Trong đó, `executor` là một hàm có hai tham số:

- `resolve` là hàm sẽ được gọi khi promise hoàn thành
- `reject` là hàm sẽ được gọi khi có lỗi xảy ra

Ví dụ:

```
api.getUser = function(username) {
  // Hàm api.getUser() trả về một promise object
  return new Promise((resolve, reject) => {
    // Gửi AJAX request
    http.get(`/users/${username}`, (err, result) => {

      // Nếu có lỗi bên trong callback, chúng ta gọi đến hàm
`reject()`
      if (err) return reject(err)

      // Ngược lại, dùng `resolve()` để trả dữ liệu về cho
`.then()`
      resolve(result)

    })
  })
}
```

Như vậy `api.getUser()` sẽ trả về một promise object. Chúng ta có thể truy xuất đến kết quả trả về bằng phương thức `.then()` như sau:

```
function onSuccess(user) { console.log(user) }

function onError(err) { console.error(error) }



api.getUser('pikalong')

  .then(onSuccess, onError)
```

Phương thức `.then(onSuccess, onError)` nhận vào hai hàm: `onSuccess` được gọi khi promise hoàn thành và `onError` được gọi khi có lỗi xảy ra. Bên trong tham số `onSuccess` bạn có thể trả về một giá trị đồng bộ, chẳng hạn như giá trị số, chuỗi, `null`, `undefined`, array hay object; hoặc một **promise object** khác. Các giá trị bất đồng bộ sẽ được bọc bên trong một Promise, cho phép bạn kết nối (chaining) nhiều promises lại với nhau.

```
promise()
```

```
  .then(() => { return 'foo' })
  .then(result1 => {
     console.log(result1) // 'foo'
     return anotherPromise()
  })
  .then(result2 => console.log(result2)) // `result2` sẽ là kết quả
của anotherPromise()
  .catch(err => {})
```

Trong ví dụ trên, bạn thấy đến phương thức `.catch()`. Phương thức này chỉ là *cú pháp bọc đường*(syntactic sugar) của `.then(null, onError)` mà thôi. Chúng ta sẽ nói thêm về `.catch()` ở bên dưới.

Tạo nhanh Promise với `Promise.resolve()` và `Promise.reject()`

Có những trường hợp bạn chỉ cần bọc một giá trị vào promise hay tự động reject. Thay vì dùng cú pháp `new Promise()` dài dòng, bạn có thể dùng hai phương thức tĩnh `Promise.resolve(result)` và `Promise.reject(err)`

```
const p = Promise.resolve(12)

  .then(result => console.log(result)) // 12

  .then(res => Promise.reject(new Error('Dừng lại nhanh')))

  .then(() => 'Cười thêm phát nữa là tym anh đứt phanh')

  .catch(err => console.error(err)) // Error: Dừng lại nhanh
```

Còn async/await là cái chi?

Được giới thiệu trong ES8, async/await là một *cơ chế* giúp bạn thực hiện các thao tác bất đồng bộ một cách *tuần tự* hơn. Async/await vẫn sử dụng Promise ở bên dưới nhưng mã nguồn của bạn (theo một cách nào đó) sẽ trong sáng và dễ theo dõi.

Để sử dụng, bạn phải khai báo hàm với từ khóa `async`. Khi đó bên trong hàm bạn có thể dùng `await`.

```
async function() {
  try {
    const user = await api.getUser('pikalong')
    const posts = await api.getPostsOfUser(user)
    const comments = await api.getCommentsOfPosts(posts)

    console.log(comments)
  } catch (err) {
    console.log(err)
  }
}
```

Cần lưu ý là kết quả trả về của async function luôn là một Promise.

```
async function hello() {
  return 1
}

console.log(hello() instanceof Promise) // true
hello().then(console.log) // 1
```

Căn bản về Promise và async/await là vậy. Hiện giờ, bạn đã có thể sử dụng Promise và async/await ở tất cả các trình duyệt hiện đại (trừ IE11 ra nhé, bạn vẫn cần polyfill cho nó). Hãy xem những trường hợp cần lưu ý khi sử dụng chúng.

“Kim tự tháp” Promises

Một lỗi chúng ta hay mắc phải khi mới làm quen với Promise, đó là tạo ra “kim tự tháp” promises như thế này.

```
api.getUser('pikalong')
  .then(user => {
     api.getPostsOfUser(user)
       .then(posts => {
         api.getCommentsOfPosts(posts)
         .then(comments => {
           console.log(comments)
         })
         .catch(err => console.log(err))
       })
       .catch(err => console.log(err))
  })
  .catch(err => console.log(err))
```

Lý do vì chúng ta quên mất tính chất liên kết (chaining) của promise, cho phép bên trong hàm `resolve` có thể trả về một giá trị đồng bộ hoặc **một promise** khác. Do đó cách giải quyết là:

```
api.getUser('pikalong')

  // Trả về một promise

  .then(user => api.getPostsOfUser(user))

  .then(posts => api.getCommentsOfPosts(posts))

  .catch(err => { throw err })
```

Theo Ehkoo, việc hiểu và sử dụng thành thạo tính liên kết là một trong những điểm **QUAN TRỌNG NHẤT** khi làm việc với Promise. Khi promise lồng vào nhau từ 2 tầng trở lên thì đã đến lúc bạn phải refactor lại rồi.

Luôn đưa vào `.then()` một hàm

Bạn thử đoán xem đoạn code sau sẽ in ra gì?

```
Promise.resolve(1)

  .then(2)

  .then(console.log)
```

Câu trả lời là `1` đó. Phương thức `.then` đòi hỏi tham số của nó phải là một hàm. Nếu bạn đưa vào `.then()` một giá trị, nó sẽ bị bỏ qua, giải thích tại sao đoạn code trên hiển thị `1`. Trường hợp tương tự:

```
Promise.resolve(1)

  .then(Promise.resolve(2))

  .then(console.log) // 1
```

Cách giải quyết:

```
Promise.resolve(1)

  .then(() => 2)
```

```
  // hoặc như thế này, mặc dù hơi dư thừa

  .then(() => Promise.resolve(2))

  .then(console.log) // 2
```
Chúng ta sẽ được kết quả như ý.

Cẩn thận với `this` khi dùng tham chiếu hàm

Giả sử bạn có đoạn code sau:

```
const add2 = x => x + 2




Promise.resolve(4).then(result => add2(result))
```
Hàm `onSuccess` không làm gì khác ngoài việc chuyển `result` vào cho `add2`, nên bạn có thể dùng tham chiếu hàm để đoạn code trên gọn hơn.

```
Promise.resolve(4).then(add2)
```
Bạn có thể nghĩ, vậy với phương thức của một đối tượng, ta cũng có thể đưa tham chiếu hàm vào `.then()`?

```
class User {
  constructor(user) {
    this.user = user
  }

  getUsername() {
    return this.user.username
  }
}

const u = new User({ username: 'pikalong' })
Promise.resolve()
  .then(u.getUsername)
  .then(console.log)
```
Nhưng bạn lại nhận được lỗi sau:

Unhandled rejection:[TypeError: Cannot read property ‘user’ of undefined]

Lý do là vì khi trong strict mode, biến ngữ cảnh `this` chỉ được xác định khi trực tiếp gọi phương thức của đối tượng đó, hoặc thông qua `.bind()`. Bạn có thể xem giải thích chi tiết hơn ở đây.

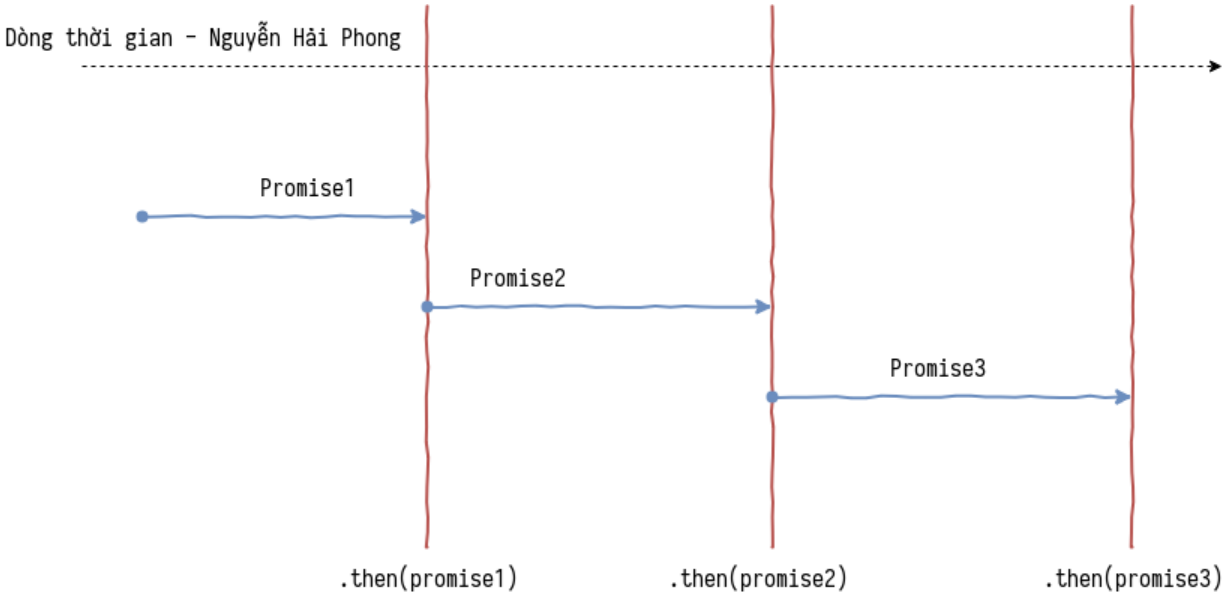
Để giải quyết lỗi này, bạn có thể dùng một trong những cách sau:

```
.then(() => u.getUsername())

// hoặc
.then(u.getUsername.bind(u))

// hoặc dùng hàm mũi tên khi khai báo phương thức trong class (cần plugin
// `transform-class-properties` của Babel)
class User {
  // ...
  getUsername = () => {
    return this.user.username
  }
}
```

Chạy các Promise tuần tự



Trong trường hợp muốn chạy các promises một cách tuần tự như sơ đồ ở trên, bạn có thể dùng hàm `Array.prototype.reduce` .

```
[promise1, promise2, promise3].reduce(function(currentPromise,
promise) {
  return currentPromise.then(promise)
}, Promise.resolve())

// Đoạn ở trên khi được viết dài dòng ra
Promise.resolve().then(promise1).then(promise2).then(promise3)
```

Async/await mang đến giải pháp "xinh đẹp" hơn, cho phép bạn truy xuất đến giá trị của các promises phía trước nếu cần thiết.

```
async function() {

  const res1 = await promise1()

  const res2 = await promise2(res1)

  const res3 = await promise3(res2)
```

Chạy nhiều Promises cùng lúc với Promise.all()

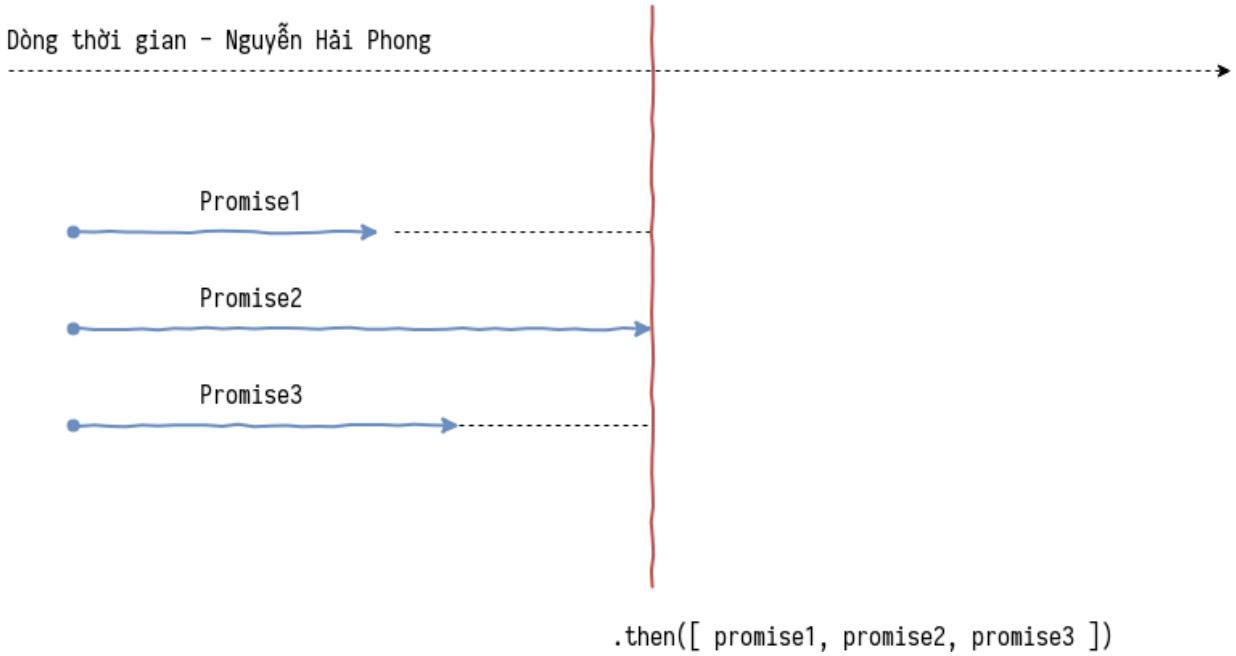
Lại có trường hợp bạn muốn thực thi và lấy ra kết quả của nhiều promises cùng lúc. Giải pháp "ngây thơ" sẽ là dùng vòng lặp, hoặc `.forEach`.

```
const userIds = [1, 2, 3, 4]

// api.getUser() là hàm trả về promise
const users = []
for (let id of userIds) {
  api.getUser(id).then(user => ([...users, user]))
}

console.log(users) // [], oát-đờ-heo?
```

Lý do là vì khi promise chưa kịp resolve thì dòng `console.log` đã chạy rồi. Chúng ta có thể sửa bằng cách dùng `Promise.all([promise1, promise2, ...])`. Phương thức này nhận vào một mảng các promises và chỉ resolve khi tất cả các promises này hoàn thành, hoặc reject khi một trong số chúng xảy ra lỗi.




```
const userIds = [1, 2, 3, 4]

Promise.all(usersIds.map(api.getUser))
  .then(function(arrayOfResults) {
    const [user1, user2, user3, user4] = arrayOfResults
  })
```

Nếu dùng async/await thì...

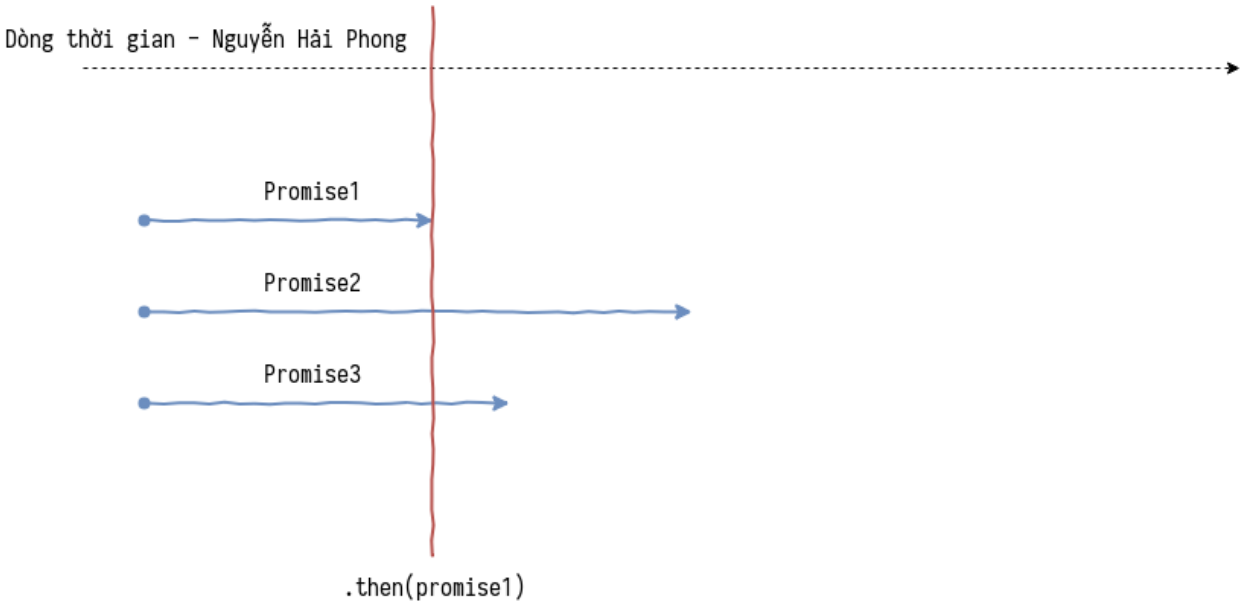
```
async function() {

  const userIds = [1, 2, 3, 4]

  const [user1, user2, user3, user4] = await
Promise.all(usersIds.map(api.getUser))

}
```

Đừng quên Promise.race()



Ngoài hai kiểu chạy tuần tự và song song ở trên, chúng ta còn có `Promise.race([promise1, promise2, ...])`. Phương thức này nhận vào một mảng các promises và sẽ resolve/reject ngay khi một trong số các promises này hoàn thành/xảy ra lỗi.

```
Promise.race([
  ping('ns1.example.com'),
  ping('ns2.example.com'),
  ping('ns3.example.com'),
  ping('ns4.example.com')
]).then(result => {})
```

Cẩn thận với `return` không tường minh

Xét hai đoạn mã sau:

```
api.getUser('pikalong')
  .then(user => {
    return api.getPostsByUser(user)
  })
  .then(console.log) // posts

api.getUser('pikalong')
  .then(user => {
    api.getPostsByUser(user)
  })
```

```
  .then(console.log) // undefined
```

Đoạn mã thứ hai trả về `undefined` vì trong JavaScript nếu một hàm không *công khai* trả về một giá trị, `undefined` mặc định sẽ được trả về (nguồn). Do đó, bạn cần lưu ý về giá trị `return` khi làm việc với Promise.

Phân biệt `.then(resolve, reject)` và `.then(resolve).catch(reject)`

Hàm `reject` trong `.then(resolve, reject)` chỉ có thể chụp được lỗi từ những `.then()` phía trước nó, mà không thể bắt được lỗi xảy ra trong hàm `resolve` cùng cấp.

```
api.getUser('pikalong')
  .then(user => { throw new Error('Lỗi rồi bạn ei') }, err => { /*
Không có gì ở đây cả */ })

api.getUser('pikalong')
  .then(user => { throw new Error('Lỗi rồi bạn ei') })
  .catch(err => console.log(err)) // Chụp được rồi bạn ei
```

Lưu ý là promise sẽ dừng quá trình thực thi khi bắt được lỗi

```
Promise.resolve()
  .then(() => { throw 'foo' })
  .then(() => { throw 'bar' }, err => { console.error("here", err)
})
  .catch(err => console.error('final', err))

// console:
// "here bar"
```

Truyền dữ liệu giữa các promises với nhau

Một trong những yếu điểm của Promise là không có cơ chế mặc định để bạn truyền dữ liệu giữa các promise objects với nhau. Nghĩa là:

```
api.getUser('pikalong')

  .then(user => api.getPostsByUser(user))

  .then(posts => {
```

```
    // Muốn sử dụng biến user ở trên thì làm sao đây?

  })
```

Một cách là dùng `Promise.all()`.

```
api.getUser('pikalong')
  .then(user => Promise.all([user, api.getPostsByUser(user)]))
  .then(results => {
     // Dùng kỹ thuật phân rã biến trong ES6. Bạn lưu ý chúng ta
dùng 1 dấu , để
     // tách ra phần tử thứ hai của mảng mà thôi
     const [ , posts ] = results

     // Lại tiếp tục truyền dữ liệu bao gồm [user, posts, comments]
xuống promise sau
     return Promise.all([...results,
api.getCommentsOfPosts(posts)])
  })
```

Hoặc, nếu bạn cảm thấy phân tách mảng khó dùng vì phải nhớ thứ tự của các giá trị thì ta có thể dùng object như sau:

```
api.getUser('pikalong')

  .then(user => api.getPostsByUser(user).then(posts => ({ user,
posts })))

  .then(results =>
api.getCommentsOfPosts(results.posts).then(comments => ({
...results, comments })))

  .then(console.log) // { users, posts, comments }
```

Lại một lần nữa, async/await lại tỏa sáng vì giúp bạn truy xuất đến kết quả của những promises phía trước.

```
async function() {

  const user = await api.getUser('pikalong')

  const posts = await api.getPostsOfUser(user)

  const comments = await api.getCommentsOfPosts(posts)

}
```

Cẩn thận nha, Promise không lazy

Với đoạn code sau:

```
console.log('before')
const promise = new Promise(function fn(resolve, reject) {
  console.log('hello')
  // ...
});
console.log('after')
```

Kết quả được in ra console lần lượt sẽ là:

```
before

hello

after
```

Bạn có thể thấy hàm `executor` của Promise được thực thi ngay lập tức. Điều này có thể dẫn đến những kết quả không mong muốn, chẳng hạn như:

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

  return http.get(`/api`, (err, result) =>  err ? reject(err) :
resolve(result))

})



button.onclick = e => getUsers
```

Cách giải quyết là đưa vào một hàm trả về promise.

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

  return http.get(`/api`, (err, result) =>  err ? reject(err) :
resolve(result))

})



button.onclick = e => getUsers()
```

Cuối cùng, `.finally()`

Bên cạnh `.then()` và `.catch()`, chúng ta còn có `.finally(onFinally)`. Phương thức này nhận vào một hàm và sẽ được kích hoạt dù cho promise trước nó hoàn thành hay xảy ra lỗi.

```
showLoadingSpinner()
api.getUser('pikalong')
  .then(user => {})
  .catch(err => {})
  .finally(hideLoadingSpinner)

// async/await
async function() {
  try {
    showLoadingSpinner()
    api.getUser('pikalong')
  } catch(err) {
  } finally {
    hideLoadingSpinner()
  }
}
```

Bạn có thể đọc thêm về `Promise.prototype.finally()` ở đây. Lưu ý là phương thức này hiện chỉ được hỗ trợ bởi Firefox, Chrome và Opera thôi nhé.

Bạn có thể thấy Promise và async/await không hoàn toàn thay thế mà hỗ trợ lẫn nhau. Mặc dù chúng ta có thể dùng async/await ở đa số các trường hợp, Promise vẫn là nền tảng cần thiết khi thực thi các tác vụ bất đồng bộ trong JavaScript. Do đó bạn nên xem xét và lựa chọn giải pháp phù hợp, tùy vào tình hình thực tế nhá.