

6 Lý do Async/Await của Javascript đánh bại Promises



Trong trường hợp bạn đã quên mất, **Node.js** đã hỗ trợ `async/await` kể từ phiên bản 7.6. Nếu bạn chưa thử qua, bài viết này sẽ liệt kê các lý do cùng ví dụ để giải thích tại sao bạn nên chọn nó.

Async/await 101


Với những ai chưa hề nghe qua về **Async/await** thì đây là những giới thiệu ngắn gọn:


- Async/await là cách mới để viết code bất đồng bộ. Các phương pháp làm việc với code bất đồng bộ trước đây là sử dụng callback và promise.
- Async/await là khái niệm được xây dựng ở tầng trên promise. Do đó nó không thể sử dụng với callback thuần.
- Async/await cũng giống như promise, là non-blocking.
- Async/await làm cho code bất đồng bộ nhìn và chạy gần giống như code đồng bộ.


Cú pháp

Giả sử một hàm `getJSON` trả về một promise, promise đó chứa 1 vài đối tượng JSON. Ta cần gọi hàm đó, log các đối tượng JSON ra, sau đó trả về `"done"`.

Những việc làm hấp dẫn


Lập Trình Viên Java
 VNPT-IT 📍 Ho Chi Minh, Ha Noi, Da Nang, Others 💰 \$800 - \$1,500
 Java


AI ENGINEER
 CÔNG TY CỔ PHẦN GIẢI PHÁP CÔNG NGHỆ CARO 📍 Ha Noi 💰 Up to \$1,800
 AI


Full-Stack Software Engineer (PHP, MySQL, Python, ReactJS)
 NOT A SQUARE 📍 Ho Chi Minh 💰 \$500 - \$1,000
 PHP | MySQL | Python | Full-Stack | ReactJS

Đoạn code sau miêu tả quá trình trên, sử dụng promise.

```
const makeRequest = () =>
  getJSON()
    .then(data => {
      console.log(data)
      return "done"
    })

makeRequest()
```

Còn đây là đoạn code sử dụng async/await:

```
const makeRequest = async () => {
  console.log(await getJSON())
  return "done"
}

makeRequest()
```

Có 1 vài điểm khác biệt cần để ý:

1. Hàm có thêm từ khóa `async` phía trước. Từ khóa `await` chỉ được sử dụng bên trong hàm được định nghĩa với `async`. Bất cứ hàm `async` nào cũng sẽ trả về 1 promise một cách không tưởng minh, và giá trị resolve của promise sẽ là bất cứ cái gì mà hàm return (trong trường hợp này là chuỗi `"done"`).
2. Nhận xét trên cũng đồng nghĩa với việc ta không thể sử dụng `await` phía trước đoạn code chứa từ khóa `async`

```
// this will not work in top level
// await makeRequest()
```

```
// this will work
makeRequest().then((result) => {
  // do something
})
```

`await getJSON()` có nghĩa là lời gọi `console.log` sẽ chờ đến khi promise `getJSON()` được xử lý và trả về giá trị.

Ưu điểm của Async/await là gì?

1. Code ngắn và sạch hơn

Đơn giản nhất chính là số lượng code ta cần viết đã giảm đi đáng kể. Trong ví dụ trên, rõ ràng rằng ta đã tiết kiệm được rất nhiều dòng code. Ta không cần viết `.then`, tạo 1 hàm anonymous để xử lý response, hay là đặt tên `data` cho 1 biến ta không sử dụng. Ta tránh được các khối code lồng nhau. Những lợi ích nho nhỏ này sẽ tích tụ dần dần trong những đoạn code lớn, những project thật và sẽ trở nên rất đáng giá.

2. Error handling

Async/await giúp ta xử lý cả error đồng bộ lẫn error bất đồng bộ theo cùng 1 cấu trúc. Tạm biệt `try/catch`. Với đoạn code dưới dùng promise, `try/catch` sẽ không bắt được lỗi nếu `JSON.parse` lỗi do nó xảy ra bên trong promise. Ta cần gọi `.catch` bên trong promise và lặp lại code xử lý error, điều mà chắc chắn sẽ trở nên rắc rối hơn cả `console.log` trong đoạn code production.

```
const makeRequest = () => {
  try {
    getJSON()
      .then(result => {
        // this parse may fail
        const data = JSON.parse(result)
        console.log(data)
      })
    // uncomment this block to handle asynchronous errors
    // .catch((err) => {
    //   console.log(err)
    // })
  } catch (err) {
    console.log(err)
  }
}
```

Bây giờ hãy nhìn vào đoạn code sử dụng `async/await`. Khối `catch` giờ sẽ xử lý các lỗi parsing.

```
const makeRequest = async () => {  
  try {  
    // this parse may fail  
    const data = JSON.parse(await getJSON())  
    console.log(data)  
  } catch (err) {  
    console.log(err)  
  }  
}
```

3. Câu lệnh điều kiện

Hãy xem thử 1 đoạn code như dưới đây. Đoạn code này sẽ fetch dữ liệu và quyết định trả về giá trị hay là lấy thêm dữ liệu.

```
const makeRequest = () => {  
  return getJSON()  
    .then(data => {  
    if (data.needsAnotherRequest) {  
      return makeAnotherRequest(data)  
        .then(moreData => {  
          console.log(moreData)  
          return moreData  
        })  
    } else {  
      console.log(data)  
      return data  
    }  
  })  
}
```

Đoạn code đã dần dần giống với mô hình “xyz hell” mà ta thường thấy. Tổng cộng code có 6 level nested. Khi sử dụng async/await, ta sẽ có đoạn code mới dễ đọc hơn.

```
const makeRequest = async () => {  
  const data = await getJSON()  
  if (data.needsAnotherRequest) {  
    const moreData = await makeAnotherRequest(data);  
    console.log(moreData)  
    return moreData  
  }  
}
```

```
} else {  
  console.log(data)  
  return data  
}  
}
```

4. Giá trị intermediate

Hẳn bạn đã từng lâm vào tình huống sau: bạn cần gọi `promise1`, sau đó sử dụng giá trị nó trả về để gọi `promise2` cuối cùng sử dụng kết quả trả về của cả 2 promise trên để gọi `promise3`. Code của bạn sẽ thành ra thế này.

```
const makeRequest = () => {  
  return promise1()  
    .then(value1 => {  
      // do something  
      return promise2(value1)  
        .then(value2 => {  
          // do something  
          return promise3(value1, value2)  
        })  
    })  
}
```

Nếu `promise3` không yêu cầu tham số `value1`, promise sẽ bớt lớp nest đi 1 chút. Nếu bạn theo chủ nghĩa cầu toàn, bạn có thể giải quyết bằng cách wrap cả 2 giá trị `value1` và `value2` bằng `Promise.all` tránh được các lớp nest giống như đoạn code dưới.

```
const makeRequest = () => {  
  return promise1()  
    .then(value1 => {  
      // do something  
      return Promise.all([value1, promise2(value1)])  
    })  
    .then([value1, value2] => {  
      // do something  
      return promise3(value1, value2)  
    })  
}
```

Phương pháp này đã hi sinh tính ngữ nghĩa để đổi lấy tính dễ đọc của code. Đơn giản vì chả có lý do gì mà `value1` & `value2` được đặt chung vào 1 mảng, ngoại trừ việc làm như thế sẽ tránh được promise bị nest.

Tuy nhiên cái logic này trở nên cực kì ngớ ngẩn khi ta sử dụng `async/await`.

```
const makeRequest = async () => {  
  const value1 = await promise1()  
  const value2 = await promise2(value1)  
  return promise3(value1, value2)  
}
```

5. Error Stack

Hình dung 1 đoạn code gọi đến nhiều `promise` theo chuỗi. Tại 1 vị trí nào đó, đoạn code sẽ quăng ra 1 error.

```
const makeRequest = () => {  
  return callAPromise()  
    .then(() => callAPromise())  
    .then(() => callAPromise())  
    .then(() => callAPromise())  
    .then(() => callAPromise())  
    .then(() => {  
      throw new Error("oops");  
    })  
}  
  
makeRequest()  
  .catch(err => {  
    console.log(err);  
    // output  
    // Error: oops at callAPromise.then.then.then.then.then (index.js:8:13)  
  })
```

Error Stack trả về từ chuỗi promise không thể giúp ta xác định error xảy ra ở đâu. Tệ hơn nữa, nó còn làm ta hiểu lầm rằng lỗi nằm ở hàm `callAPromise`

Tuy nhiên, với `async/await`, Error Stack sẽ chỉ ra được hàm nào chứa lỗi.

```
const makeRequest = async () => {
```

```

    await callAPromise()
    await callAPromise()
    await callAPromise()
    await callAPromise()
    await callAPromise()
    throw new Error("oops");
  }

  makeRequest()
    .catch(err => {
      console.log(err);
      // output
      // Error: oops at makeRequest (index.js:7:9)
    })

```

Khi bạn phát triển ứng dụng trên môi trường local, điều này thoát nhìn không có quá nhiều tác dụng. Tuy nhiên với production server, nó lại rất hữu ích với Error Logs. Với những tình huống đó, biết được error xảy ra trong `makeRequest` sẽ tốt hơn rất nhiều khi được báo rằng error nằm trong `then` phía sau `then` phía sau `then`

6. Debug

Điều tuyệt vời cuối cùng khi bạn làm việc với `async/await` đó là việc debug trở nên rất đơn giản. Debug với Promise chưa bao giờ là công việc dễ chịu vì 2 lý do sau:

1/ Bạn không thể đặt breakpoint trong arrow function trả về expression.

```

4
5   const makeRequest = () => {
6     return callAPromise()
7       .then(() => callAPromise())
8       .then(() => callAPromise())
9       .then(() => callAPromise())
10      .then(() => callAPromise())
11   }
12

```

2/ Nếu bạn đặt breakpoint bên trong khối `.then` và sử dụng short-cut debug như `step-over`, trình debug sẽ không chuyển đến khối `.then` kế tiếp bởi vì nó chỉ "step" ở các đoạn code đồng bộ. Với `async/await`, bạn không cần arrow function quá nhiều nữa, bạn hoàn toàn có thể step qua lời gọi `await y` như với code đồng bộ.

```
4
5  const makeRequest = async () => {
6    await callAPromise()
7    await callAPromise()
8    await callAPromise()
9    await callAPromise()
10   await callAPromise()
11  }
12
```

Kết luận

Async/await là 1 trong những tính năng mang tính cách mạng được thêm vào JavaScript trong vài năm gần đây. Nó giúp bạn nhận ra Promise còn thiếu sót như thế nào, cũng như cung cấp giải pháp thay thế.

Có thể bạn quan tâm:

- [ES6 là gì? Những nổi bật và sự thay đổi tuyệt vời của ES6](#)
- [Mẹo với Javascript \(ES6\) và thủ thuật để làm cho code sạch hơn, ngắn hơn, và dễ đọc hơn \(Phần 2\)](#)
- [12 tips hay cho JavaScript](#)

*Xem thêm **việc làm JavaScript Developer** trên **TopDev***

TopDev via **Hackernoon**

Method Chaining trong JavaScript là gì

20 câu hỏi phỏng vấn Javascript dành cho Intern/Fresher