Using async/await with a forEach loop

Asked 8 years, 6 months ago Modified 2 months ago Viewed 2.4m times

3272

Are there any issues with using `async`/`await` in a `forEach` loop? I'm trying to loop through an array of files and `await` on the contents of each file.

```
import fs from 'fs-promise'

async function printFiles () {
  const files = await getFilePaths() // Assume this works fine

  files.forEach(async (file) => {
    const contents = await fs.readFile(file, 'utf8')
    console.log(contents)
  })
}

printFiles()
```

This code does work, but could something go wrong with this? I had someone tell me that you're not supposed to use `async`/`await` in a higher-order function like this, so I just wanted to ask if there was any issue with this.

javascript node.js promise async-await ecmascript-2017

Share
Improve this question
Follow

edited Mar 12, 2021 at 12:19

Nikita Fedyashev
18.9k ● 13 ● 56 ● 103

asked Jun 1, 2016 at 18:55

Saad
53.6k ● 21 ● 76 ● 114

Gotcha! Wasted a few hours figuring this behavior out... Old fashioned iteration `for i = ...` never fails. – Ronnie Smith Mar 11 at 22:22

Here every time the async function returns. and the compiler can't wait for the async function till the end of the function execution that's why every async doesn't work. – Daxesh Italiya May 6 at 12:52

35 Answers

Sorted by: Highest score (default)

1 2 Next

Sure the code does work, but I'm pretty sure it doesn't do what you expect it to do. It just fires off multiple asynchronous calls, but the `printFiles` function does

5722 immediately return after that.

Reading in sequence

If you want to read the files in sequence, **you cannot use** `forEach` indeed. Just use a modern `for … of` loop instead, in which `await` will work as expected:

```
async function printFiles () {
  const files = await getFilePaths();

  for (const file of files) {
    const contents = await fs.readFile(file, 'utf8');
    console.log(contents);
  }
}
```

Reading in parallel

If you want to read the files in parallel, **you cannot use** `forEach` indeed. Each of the `async` callback function calls does return a promise, but you're throwing them away instead of awaiting them. Just use `map` instead, and you can await the array of promises that you'll get with `Promise.all`:

```
async function printFiles () {
  const files = await getFilePaths();

  await Promise.all(files.map(async (file) => {
    const contents = await fs.readFile(file, 'utf8')
    console.log(contents)
  }));
}
```

Share

Improve this answer

Follow

edited Feb 8, 2020 at 23:30

mesqueeb

6,264 ● 7 ● 51 ● 85

answered Jun 1, 2016 at 19:02

Bergi

663k ● 158 ● 1k ● 1.5k

110 Could you please explain why does `for ... of ...` work? – Demonbane Aug 15, 2016 at 18:04

261 ok i know why... Using Babel will transform `async` / `await` to generator function and using `forEach` means that each iteration has an individual generator function, which has nothing to do with the others. so they will be executed independently and has no context of `next()` with others. Actually, a simple `for()` loop also works because the iterations are also in one single generator function. – Demonbane Aug 15, 2016 at 19:21

52 @Demonbane: In short, because it was designed to work :-) `await` suspends the current *function* evaluation, including all control structures. Yes, it is quite similar to generators in that regard (which is why they are used to polyfill async/await). – Bergi Aug 15, 2016 at 23:28

8 @arve0 Not really, an `async` function is quite different from a `Promise` executor callback, but yes the `map` callback returns a promise in both cases. – Bergi Mar 29, 2017 at 16:25

6 @Taurus If you don't intend to await them, then `for…of` would work equally to `forEach`. No, I really mean that paragraph to emphasise that there is no place for `.forEach` in modern JS code. – Bergi Mar 20, 2018 at 13:24

672

With ES2018, you are able to greatly simplify all of the above answers to:

```
async function printFiles () {
  const files = await getFilePaths()

  for await (const contents of files.map(file => fs.readFile(file, 'utf8'))) {
    console.log(contents)
  }
}
```

See spec: proposal-async-iteration

Simplified:

```
  for await (const results of array) {
    await longRunningTask()
  }
  console.log('I will wait')
```

2018-09-10: This answer has been getting a lot of attention recently, please see Axel Rauschmayer's blog post for further information about asynchronous iteration.

Share

Improve this answer

Follow

edited Jun 22, 2022 at 17:42

Steffan

734 ● 1 ● 11 ● 26

answered Jun 15, 2018 at 11:17

Cisco

22.8k ● 6 ● 43 ● 65

16 I don't think this answer address the initial question. `for-await-of` with a synchronous iterable (an array in our case) doesn't cover the case of iterating concurrently an array using asynchronous operations in each iteration. If I'm not mistaken, using `for-await-of` with a synchronous iterable over non-promise values is the same as using a plain `for-of`. – Antonio Val Jan 9, 2019 at 10:30

3 How we delegates `files` array to the `fs.readFile` here? It tooks from iterable? – Vadim Shvetsov Jan 17, 2019 at 13:34

4 This answer has the same issue as the OP: It accesses all files in parallel. The serialized printing of results merely hides it. – jib Feb 18, 2021 at 13:52

2 Less characters does not mean it is simpler. This is mostly convoluted and unreadable. – user13548229 Aug 14, 2021 at 16:40

15 This answer is wrong. `files.map()` returns an array of promises, **not an asynchronous iterator**, for which `for await` was made! It will cause unhandled-rejection crashes! – Bergi Dec 13, 2021 at 15:57

209

Instead of `Promise.all` in conjunction with `Array.prototype.map` (which does not guarantee the order in which the `Promise`s are resolved), I use `Array.prototype.reduce`, starting with a resolved `Promise`:

```
async function printFiles () {
  const files = await getFilePaths();

  await files.reduce(async (promise, file) => {
    // This line will wait for the last async function to finish.
    // The first iteration uses an already resolved Promise
    // so, it will immediately continue.
    await promise;
    const contents = await fs.readFile(file, 'utf8');
    console.log(contents);
  }, Promise.resolve());
}
```

Share

Improve this answer

Follow

edited Mar 3, 2019 at 3:38

answered Mar 26, 2018 at 19:48

Timothy Zorn

3,201 ● 2 ● 21 ● 18

4 This works perfectly, thank you so much. Could you explain what is happening here with `Promise.resolve()` and `await promise;`? – parrker9 Mar 28, 2018 at 20:48

2 This is pretty cool. Am I right in thinking the files will be read in order and not all at once? – GollyJer Jun 9, 2018 at 0:24

8 @parrker9 `Promise.resolve()` returns an already resolved `Promise` object, so that `reduce` has a `Promise` to start with. `await promise;` will wait for the last `Promise` in the chain to resolve. @GollyJer The files will be processed sequentially, one at a time. – Timothy Zorn Jun 17, 2018 at 15:00

4 @Shay, You mean sequential, not synchronous. This is still asynchronous - if other things are scheduled, they will run in between the iterations here. – Timothy Zorn May 30, 2019 at 16:51

6 If you need the async processes to finish as quickly as possible and you don't care about them being completed sequentially, try one of the provided solutions with a good amount of upvotes which uses `Promise.all`. Example: `Promise.all(files.map(async (file) => { /* code */ }));` – Timothy Zorn Jan 31, 2020 at 16:03

151

```
files.forEach(async (file) => {
    const contents = await fs.readFile(file, 'utf8')
})
```

The issue is, the promise returned by the iteration function is ignored by `forEach()`. `forEach` does not wait to move to the next iteration after each async code execution is completed. All the `fs.readFile` functions will be invoked in the same round of the event loop, which means they are started in parallel, not in sequential, and the execution continues immediately after invoking forEach(), without waiting for all the `fs.readFile` operations to complete. Since forEach does not wait for each promise to resolve, the loop actually finishes iterating before promises are resolved. You are expecting that after `forEach` is completed, all the async code is already executed but that is not the case. You may end up trying to access values that are not available yet.

you can test the behaviour with this example code

```
const array = [1, 2, 3];

const sleep = (ms) => new Promise(resolve => setTimeout(resolve, ms));
const delayedSquare = (num) => sleep(100).then(() => num * num);

const testForEach = (numbersArray) => {
  const store = [];
  // this code here treated as sync code
  numbersArray.forEach(async (num) => {
    const squaredNum = await delayedSquare(num);
    // this will console corrent squaredNum value
    // console.log(squaredNum) will log after console.log("store",store)
    console.log(squaredNum);
    store.push(squaredNum);
  });
  // you expect that store array is populated as [1,4,9] but it is not
  // this will return []
  console.log("store",store);
};
testForEach(array);
// Notice, when you test, first "store []" will be logged
// then squaredNum's inside forEach will log
```

Run code snippet Expand snippet

the solution is using the for-of loop.

```
for (const file of files){
    const contents = await fs.readFile(file, 'utf8')
}
```

Share

Improve this answer

Follow

edited Jun 1, 2023 at 1:54

answered Feb 1, 2022 at 19:28

Yilmaz

48.7k ● 18 ● 210 ● 264

`Since forEach does not wait for each promise to resolve` - What do you mean by this. That is the purpose of the await keyword. Does that mean that forEach returns a list a promises, `const promise_list = forEach(async() => {some async call....})` – Jamie Marshall Sep 17 at 22:00

The `await` keyword works inside the async function, but `forEach()` continues executing the next iteration without waiting for the current one to finish. So even though you're using await, the forEach() function doesn't know what `await` is – Yilmaz Sep 18 at 0:54

`forEach()` is designed for synchronous operations. It doesn't handle asynchronous operations, which is why it doesn't know or handle `await` properly. – Yilmaz Sep 18 at 0:57

javascript has microtask queue where promises are handled javascript.info/microtask-queue – Yilmaz Sep 18 at 1:08

so each promise inside `forEach` is passed to `microtask` queue. while promises are handled here, since `forEach` is treated as sync code, it keeps iterating. The Event Loop processes this microtask queue as soon as it finishes any synchronous tasks on the call stack. that is why if you check the code example, after `forEach` iteration is over, `store` array is empty, because promises from `microtask` has not been executed on call stack – Yilmaz Sep 18 at 1:13

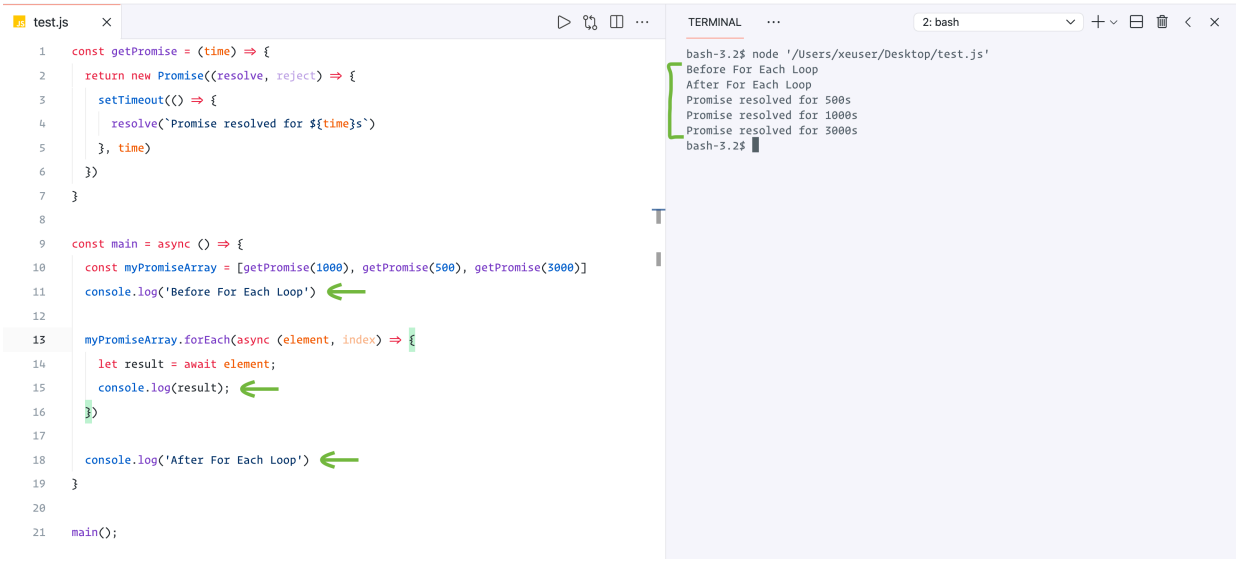
71

Picture worth 1000 words - For Sequential Approach Only

Background : I was in similar situation last night. I used async function as foreach argument. The result was un-predictable. When I did testing for my code 3 times, it ran without issues 2 times and failed 1 time. (something weird)

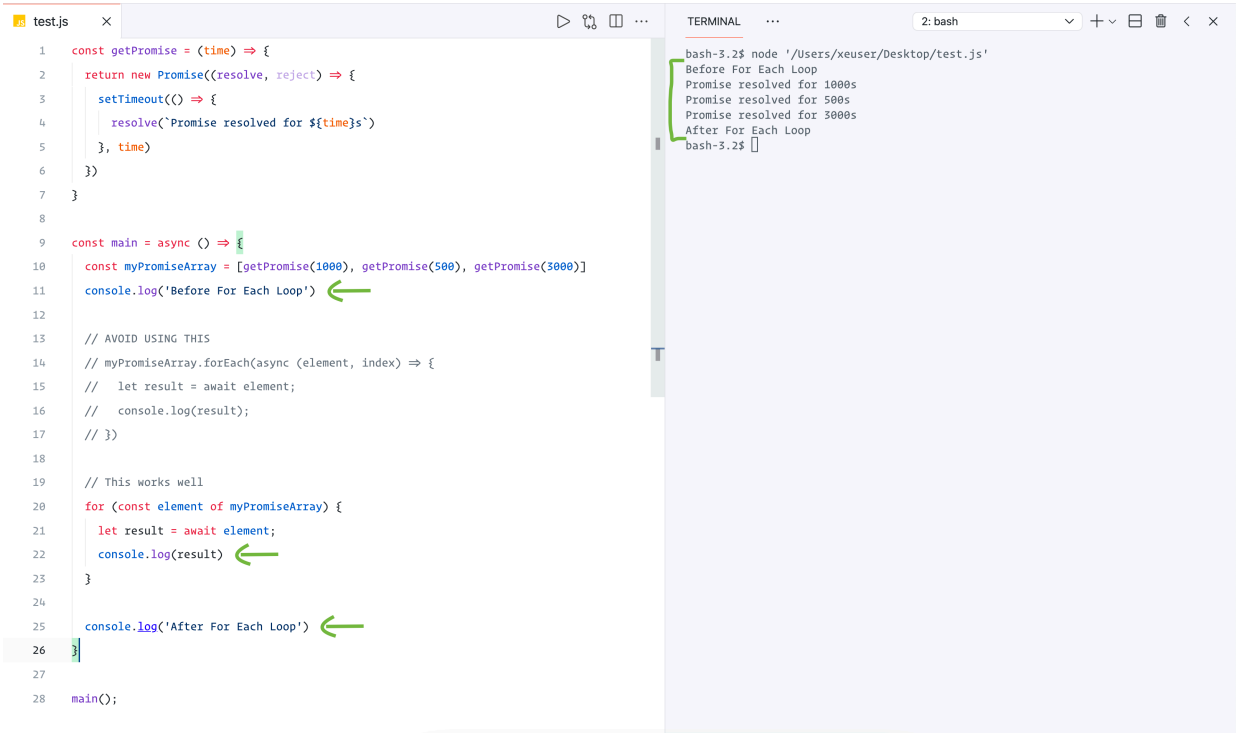
Finally I got my head around & did some scratch pad testing.

Scenario 1 - How un-sequential it can get with async in foreach




```
const getPromise = (time) => {
  return new Promise((resolve, reject) => {
    setTimeout(() => {
      resolve(`Promise resolved for ${time}s`)
    }, time)
  })
}

const main = async () => {
  const myPromiseArray = [getPromise(1000), getPromise(500), getPromise(3000)]
  console.log('Before For Each Loop')

  myPromiseArray.forEach(async (element, index) => {
    let result = await element;
    console.log(result);
  })

  console.log('After For Each Loop')
}

main();
```

Scenario 2 - Using `for - of` loop as @Bergi above suggested




```
const getPromise = (time) => {
  return new Promise((resolve, reject) => {
    setTimeout(() => {
      resolve(`Promise resolved for ${time}s`)
    }, time)
  })
}

const main = async () => {
  const myPromiseArray = [getPromise(1000), getPromise(500), getPromise(3000)]
  console.log('Before For Each Loop')

  // AVOID USING THIS
  // myPromiseArray.forEach(async (element, index) => {
  //   let result = await element;
  //   console.log(result);
  // })

  // This works well
  for (const element of myPromiseArray) {
    let result = await element;
    console.log(result)
  }

  console.log('After For Each Loop')
}

main();
```

If you are little old school like me, you could simply use the classic for loop, that works too :)

```
const getPromise = (time) => {
  return new Promise((resolve, reject) => {
    setTimeout(() => {
      resolve(`Promise resolved for ${time}s`)
```

```
    }, time)
  })
}

const main = async () => {
  const myPromiseArray = [getPromise(1000), getPromise(500), getPromise(3000)]
  console.log('Before For Each Loop')

  // AVOID USING THIS
  // myPromiseArray.forEach(async (element, index) => {
  //   let result = await element;
  //   console.log(result);
  // })

  // This works well too - the classic for loop :)
  for (let i = 0; i < myPromiseArray.length; i++) {
    const result = await myPromiseArray[i];
    console.log(result);
  }

  console.log('After For Each Loop')
}

main();
```

I hope this helps someone, good day, cheers!

Share Improve this answer Follow

answered May 19, 2021 at 4:41

krupesh Anadkat

2,111 ● 1 ● 23 ● 33

I suggest using the phrase 'Before/After Loop' would make it less confusing when it's not a 'For Each Loop'. – close Mar 2, 2022 at 11:49

The brother is out here just writing code using Githubs official like an absolute heathen. I'm not even mad. To each their own. Nonetheless, I would cache the `length` to speed that for loop up and prevent recalculations between every iteration. – User_coder Apr 20, 2022 at 0:03

62

The p-iteration module on npm implements the Array iteration methods so they can be used in a very straightforward way with async/await.

An example with your case:

```
const { forEach } = require('p-iteration');
const fs = require('fs-promise');

(async function printFiles () {
  const files = await getFilePaths();

  await forEach(files, async (file) => {
    const contents = await fs.readFile(file, 'utf8');
    console.log(contents);
```

```
  });
})();
```

Share

Improve this answer

Follow

edited Oct 17, 2019 at 5:59

answered Jul 10, 2017 at 8:15

Antonio Val

3,340 ● 1 ● 15 ● 27

61

Here are some `forEachAsync` prototypes. Note you'll need to `await` them:

```
Array.prototype.forEachAsync = async function (fn) {
    for (let t of this) { await fn(t) }
}

Array.prototype.forEachAsyncParallel = async function (fn) {
    await Promise.all(this.map(fn));
}
```

Note while you may include this in your own code, you should not include this in libraries you distribute to others (to avoid polluting their globals).

To use:

```
await myArray. forEachAsyncParallel( async (item) => { await
myAsyncFunction(item) })
```

Note you'll need a modern too like `esrun` to use await at the top level in TypeScript.

Share

Improve this answer

Follow

edited Jan 15 at 2:54

mikemaccana

122k ● 110 ● 424 ● 528

answered Mar 22, 2018 at 15:11

MM

Matt

2,282 ● 15 ● 23

2 usage : await myArray. forEachAsyncParallel(async (item) => { await myAsyncFunction(item) }) – Damien Romito Jun 2, 2021 at 15:32

@Matt, isn't it a problem to await `fn` in case it wasn't asynchronous? what if the given input was a synchronous function? stackoverflow.com/a/53113299/18387350 – Normal Jun 22, 2022 at 4:43

@Normal no. Awaiting a synchronous value just wraps it in a promise and then unwraps it. – Aluan Haddad Dec 29, 2023 at 19:07

40

@Bergi has already gave the answer on how to handle this particular case properly. I'll not duplicate here.

I'd like to address the difference between using `forEach` and `for` loop when it comes to `async` and `await`

how `forEach` works

Let's look at how `forEach` works. According to ECMAScript Specification, MDN provides an implementation which can be used as a polyfill. I copy it and paste here with comments removal.

```
Array.prototype.forEach = function (callback, thisArg) {
  if (this == null) { throw new TypeError('Array.prototype.forEach called on
null or undefined'); }
  var T, k;
  var O = Object(this);
  var len = O.length >>> 0;
  if (typeof callback !== "function") { throw new TypeError(callback + ' is not
a function'); }
  if (arguments.length > 1) { T = thisArg; }
  k = 0;
  while (k < len) {
    var kValue;
    if (k in O) {
      kValue = O[k];
      callback.call(T, kValue, k, O); // pay attention to this line
    }
    k++;
  }
};
```

Let's back to your code, let's extract the callback as a function.

```
async function callback(file){
  const contents = await fs.readFile(file, 'utf8')
  console.log(contents)
}
```

So, basically `callback` returns a promise since it's declared with `async`. Inside `forEach`, `callback` is just called in a normal way, if the callback itself returns a promise, the javascript engine will not wait it to be resolved or rejected. Instead, it puts the `promise` in a job queue, and continues executing the loop.

How about `await fs.readFile(file, 'utf8')` inside the `callback`?

Basically, when your async `callback` gets the chance to be executed, the js engine will pause until `fs.readFile(file, 'utf8')` to be resolved or rejected, and resume execution of the async function after fulfillment. So the `contents` variable store the actual result from `fs.readFile`, not a `promise`. So, `console.log(contents)` logs out the file content not a `Promise`

Why `for ... of` works?

when we write a generic `for of` loop, we gain more control than `forEach`. Let's refactor `printFiles`.

```
async function printFiles () {
  const files = await getFilePaths() // Assume this works fine

  for (const file of files) {
    const contents = await fs.readFile(file, 'utf8')
    console.log(contents)
    // or await callback(file)
  }
}
```

When evaluate `for` loop, we have `await` promise inside the `async` function, the execution will pause until the `await` promise is settled. So, you can think of that the files are read one by one in a determined order.

Execute sequentially

Sometimes, we really need the the async functions to be executed in a sequential order. For example, I have a few new records stored in an array to be saved to database, and I want them to be saved in sequential order which means first record in array should be saved first, then second, until last one is saved.

Here is an example:

```
const records = [1, 2, 3, 4];

async function saveRecord(record) {
  return new Promise((resolved, rejected) => {
    setTimeout(()=> {
      resolved(`record ${record} saved`)
    }, Math.random() * 500)
  });
}

async function forEachSaveRecords(records) {
  records.forEach(async (record) => {
    const res = await saveRecord(record);
    console.log(res);
  })
}

async function forofSaveRecords(records) {
  for (const record of records) {
    const res = await saveRecord(record);
    console.log(res);
  }
}
(async () => {
  console.log("=== for of save records ===")
  await forofSaveRecords(records)
```

```
  console.log("=== forEach save records ===")
  await forEachSaveRecords(records)
})()
```

Run code snippet Expand snippet

I use `setTimeout` to simulate the process of saving a record to database - it's asynchronous and cost a random time. Using `forEach`, the records are saved in an undetermined order, but using `for..of`, they are saved sequentially.

Share Improve this answer Follow

answered Nov 9, 2021 at 16:34

sam
1,807 ● 13 ● 15

So in short: `foreach` doesn't handle callbacks in asynchronous way, therefore no waiting. – ado387 Jun 7, 2022 at 17:50

I appreciate your effort. I am working on some puppeteer things, and I was wondering why my async, await is not working. your answer clarified my doubt. Like foreach, the issue is identical for the map, filter, etc. as well. – KAmit Aug 28, 2022 at 15:52

32

This solution is also memory-optimized so you can run it on 10,000's of data items and requests. Some of the other solutions here will crash the server on large data sets.

In TypeScript:

```
export async function asyncForEach<T>(array: Array<T>, callback: (item: T,
index: number) => Promise<void>) {
        for (let index = 0; index < array.length; index++) {
            await callback(array[index], index);
        }
    }
```

How to use?

```
await asyncForEach(receipts, async (eachItem) => {
    await ...
})
```

Share

Improve this answer

Follow

edited Oct 12, 2021 at 11:25

mudamudamuda
3 ● 2

answered Apr 16, 2020 at 17:18

Oliver Dixon
7,400 ● 5 ● 70 ● 97

I think it will be helpful if you can complete this example :) in the how to use section. For my case: await asyncForEach(configuration.groupNames, async (groupName) => { await AddUsersToGroup(configuration, groupName); }) – Ido Bleicher Oct 14, 2021 at 8:10

Thanks, nice solution!! – JulienRioux Mar 12, 2022 at 17:50

What if I need to return an array of type U ? – Russo Nov 1, 2023 at 11:37

22

A simple drop-in solution for replacing a `forEach()` await loop that is not working is replacing `forEach` with `map` and adding `Promise.all(` to the beginning.

For example:

```
await y.forEach(async (x) => {
```

to

```
await Promise.all(y.map(async (x) => {
```

An extra `)` is needed at the end.

Share

Improve this answer

Follow

edited Feb 10, 2021 at 19:00

answered Jan 27, 2021 at 21:57

yeah22

490 ● 5 ● 12

2 Not quite the same. Promise.all will run all the promises *concurrently*. A for loop is meant to be sequential. – srmark Oct 7, 2021 at 11:06

20

In addition to @Bergi's answer, I'd like to offer a third alternative. It's very similar to @Bergi's 2nd example, but instead of awaiting each `readFile` individually, you create an array of promises, each which you await at the end.

```
import fs from 'fs-promise';
async function printFiles () {
  const files = await getFilePaths();

  const promises = files.map((file) => fs.readFile(file, 'utf8'))

  const contents = await Promise.all(promises)

  contents.forEach(console.log);
}
```

Note that the function passed to `.map()` does not need to be `async`, since `fs.readFile` returns a Promise object anyway. Therefore `promises` is an array of

Promise objects, which can be sent to `Promise.all()`.

In @Bergi's answer, the console may log file contents in the order they're read. For example if a really small file finishes reading before a really large file, it will be logged first, even if the small file comes *after* the large file in the `files` array. However, in my method above, you are guaranteed the console will log the files in the same order as the provided array.

Share

Improve this answer

Follow

edited Oct 11, 2019 at 0:57

answered Feb 23, 2018 at 0:47

chharvey

9,256 ● 9 ● 59 ● 99

13

it's pretty painless to pop a couple methods in a file that will handle asynchronous data in a serialized order and give a more conventional flavour to your code. For example:

```
module.exports = function () {
  var self = this;

  this.each = async (items, fn) => {
    if (items && items.length) {
      await Promise.all(
        items.map(async (item) => {
          await fn(item);
        }));
    }
  };

  this.reduce = async (items, fn, initialValue) => {
    await self.each(
      items, async (item) => {
        initialValue = await fn(initialValue, item);
      });
    return initialValue;
  };
};
```

now, assuming that's saved at './myAsync.js' you can do something similar to the below in an adjacent file:

```
...
/* your server setup here */
...
var MyAsync = require('./myAsync');
var Cat = require('./models/Cat');
var Doje = require('./models/Doje');
var example = async () => {
  var myAsync = new MyAsync();
  var doje = await Doje.findOne({ name: 'Doje', noises: [] }).save();
  var cleanParams = [];
```

```
  // FOR EACH EXAMPLE
  await myAsync.each(['bork', 'concern', 'heck'],
    async (elem) => {
      if (elem !== 'heck') {
        await doje.update({ $push: { 'noises': elem }});
      }
    });

  var cat = await Cat.findOne({ name: 'Nyan' });

  // REDUCE EXAMPLE
  var friendsOfNyanCat = await myAsync.reduce(cat.friends,
    async (catArray, friendId) => {
      var friend = await Friend.findById(friendId);
      if (friend.name !== 'Long cat') {
        catArray.push(friend.name);
      }
    }, []);
  // Assuming Long Cat was a friend of Nyan Cat...
  assert(friendsOfNyanCat.length === (cat.friends.length - 1));
}
```

Share

Improve this answer

Follow

edited Sep 26, 2017 at 9:07

answered Sep 22, 2017 at 23:03

Jay Edwards

1,025 ● 1 ● 13 ● 23

3 Minor addendum, don't forget to wrap your await/asyncs in try/catch blocks!! – Jay Edwards Sep 26, 2017 at 9:08

11

Bergi's solution works nicely when `fs` is promise based. You can use `bluebird`, `fs-extra` or `fs-promise` for this.

However, solution for node's native `fs` libary is as follows:

```
const result = await Promise.all(filePaths
    .map( async filePath => {
      const fileContents = await getAssetFromCache(filePath, async function() {

        // 1. Wrap with Promise
        // 2. Return the result of the Promise
        return await new Promise((res, rej) => {
          fs.readFile(filePath, 'utf8', function(err, data) {
            if (data) {
              res(data);
            }
          });
        });
      });

      return fileContents;
    }));
```

Note: `require('fs')` compulsorily takes function as 3rd arguments, otherwise throws error:

```
TypeError [ERR_INVALID_CALLBACK]: Callback must be a function
```

Share Improve this answer Follow

answered May 26, 2019 at 22:08

master_dodo

1,339 ● 3 ● 21 ● 34

11

It is not good to call an asynchronous method from a loop. This is because each loop iteration will be delayed until the entire asynchronous operation completes. That is not very performant. It also averts the advantages of parallelization benefits of `async` / `await`.

A better solution would be to create all promises at once, then get access to the results using `Promise.all()`. Otherwise, each successive operation will not start until the previous one has completed.

Consequently, the code may be refactored as follows;

```
const printFiles = async () => {
  const files = await getFilePaths();
  const results = [];
  files.forEach((file) => {
    results.push(fs.readFile(file, 'utf8'));
  });
  const contents = await Promise.all(results);
  console.log(contents);
}
```

Share Improve this answer Follow

answered Apr 2, 2021 at 13:18

Johnz

280 ● 4 ● 13

12 It is also not good to open thousands of files at once to read them concurrently. One always has to do an assessment whether a sequential, parallel, or mixed approach is better. Sequential loops are not fundamentally bad, `await` actually makes them possible in the first place. Also they do not "aver the benefits" of asynchronous execution, as you can still run multiple such loops at once (e.g. two concurrent calls to `printFiles`). – Bergi Apr 2, 2021 at 15:46

10

One important **caveat** is: The `await + for .. of` method and the `forEach + async` way actually have different effect.

Having `await` inside a real `for` loop will make sure all async calls are executed one by one. And the `forEach + async` way will fire off all promises at the same time, which is faster but sometimes overwhelmed(**if you do some DB query or visit some web services with volume restrictions** and do not want to fire 100,000 calls at a time).

You can also use `reduce + promise` (less elegant) if you do not use `async/await` and want to make sure files are read **one after another**.

```
files.reduce((lastPromise, file) =>
 lastPromise.then(() =>
   fs.readFile(file, 'utf8')
 ), Promise.resolve()
)
```

Or you can create a forEachAsync to help but basically use the same for loop underlying.

```
Array.prototype.forEachAsync = async function(cb){
    for(let x of this){
        await cb(x);
    }
}
```

Share Improve this answer Follow

answered Sep 24, 2017 at 20:00

LeOn - Han Li

10.2k ● 2 ● 68 ● 63

Have a look at How to define method in javascript on Array.prototype and Object.prototype so that it doesn't appear in for in loop. Also you probably should use the same iteration as native `forEach` - accessing indices instead of relying on iterability - and pass the index to the callback. – Bergi Nov 16, 2017 at 13:57

You can use `Array.prototype.reduce` in a way that uses an async function. I've shown an example in my answer: stackoverflow.com/a/49499491/2537258 – Timothy Zorn Mar 26, 2018 at 19:54

10

You can use `Array.prototype.forEach`, but async/await is not so compatible. This is because the promise returned from an async callback expects to be resolved, but `Array.prototype.forEach` does not resolve any promises from the execution of its callback. So then, you can use forEach, but you'll have to handle the promise resolution yourself.

Here is a way to read and print each file in series using `Array.prototype.forEach`

```
async function printFilesInSeries () {
  const files = await getFilePaths()
```

```
  let promiseChain = Promise.resolve()
  files.forEach((file) => {
    promiseChain = promiseChain.then(() => {
      fs.readFile(file, 'utf8').then((contents) => {
        console.log(contents)
      })
    })
  })
  await promiseChain
}
```

Here is a way (still using `Array.prototype.forEach`) to print the contents of files in parallel

```
async function printFilesInParallel () {
  const files = await getFilePaths()

  const promises = []
  files.forEach((file) => {
    promises.push(
      fs.readFile(file, 'utf8').then((contents) => {
        console.log(contents)
      })
    )
  })
  await Promise.all(promises)
}
```

Share Improve this answer Follow

answered May 20, 2020 at 20:57

richytong
2,442 ● 1 ● 13 ● 23

2 The first senario is ideal for loops that needs to be ran in serie and you cant use for of
– Mark Odey May 29, 2020 at 19:03

9

Just adding to the original answer

- The parallel reading syntax in the original answer is sometimes confusing and difficult to read, maybe we can write it in a different approach

```
async function printFiles() {
  const files = await getFilePaths();
  const fileReadPromises = [];

  const readAndLogFile = async filePath => {
    const contents = await fs.readFile(file, "utf8");
    console.log(contents);
    return contents;
  };

  files.forEach(file => {
```

```
    fileReadPromises.push(readAndLogFile(file));
  });

  await Promise.all(fileReadPromises);
}
```

- For sequential operation, not just **for...of**, normal for loop will also work

```
async function printFiles() {
  const files = await getFilePaths();

  for (let i = 0; i < files.length; i++) {
    const file = files[i];
    const contents = await fs.readFile(file, "utf8");
    console.log(contents);
  }
}
```

Share Improve this answer Follow

answered Dec 1, 2019 at 16:59

gsaandy

611 ● 6 ● 8

9

The OP's original question

> Are there any issues with using async/await in a forEach loop? ...

was covered to an extent in @Bergi's selected answer, which showed how to process in serial and in parallel. However there are other issues noted with parallelism -

1. Order -- @chharvey notes that -

> For example if a really small file finishes reading before a really large file, it will be logged first, even if the small file comes after the large file in the files array.

2. Possibly opening too many files at once -- A comment by Bergi under another answer

> It is also not good to open thousands of files at once to read them concurrently. One always has to do an assessment whether a sequential, parallel, or mixed approach is better.

So let's address these issues showing actual code that is brief and concise, and does *not* use third party libraries. Something easy to cut, paste, and modify.

Reading in parallel (all at once), printing in serial (as early as possible per file).

The easiest improvement is to perform full parallelism as in @Bergi's answer, but making a small change so that each file is *printed as soon as possible while preserving order*.

```
async function printFiles2() {
  const readProms = (await getFilePaths()).map((file) =>
    fs.readFile(file, "utf8")
  );
  await Promise.all([
    await Promise.all(readProms),                    // branch 1
    (async () => {                                   // branch 2
      for (const p of readProms) console.log(await p);
    })(),
  ]);
}
```

Above, two separate branches are run concurrently.

- branch 1: Reading in parallel, all at once,
- branch 2: Reading in serial to force order, but waiting no longer than necessary

That was easy.

Reading in parallel with a concurrency limit, printing in serial (as early as possible per file).

A "concurrency limit" means that no more than `N` files will ever being read at the same time.
Like a store that only allows in so many customers at a time (at least during COVID).

First a helper function is introduced -

```
function bootablePromise(kickMe: () => Promise<any>) {
  let resolve: (value: unknown) => void = () => {};
  const promise = new Promise((res) => { resolve = res; });
  const boot = () => { resolve(kickMe()); };
  return { promise, boot };
}
```

The function `bootablePromise(kickMe:() => Promise<any>)` takes a function `kickMe` as an argument to start a task (in our case `readFile`) but is not started immediately.

`bootablePromise` returns a couple of properties

- `promise` of type `Promise`
- `boot` of type function `()=>void`

`promise` has two stages in life

1. Being a promise to start a task
2. Being a promise complete a task it has already started.

`promise` transitions from the first to the second state when `boot()` is called.

`bootablePromise` is used in `printFiles` --

```
async function printFiles4() {
  const files = await getFilePaths();
  const boots: (() => void)[] = [];
  const set: Set<Promise<{ pidx: number }>> = new Set<Promise<any>>();
  const bootableProms = files.map((file,pidx) => {
    const { promise, boot } = bootablePromise(() => fs.readFile(file, "utf8"));
    boots.push(boot);
    set.add(promise.then(() => ({ pidx })));
    return promise;
  });
  const concurLimit = 2;
  await Promise.all([
    (async () => {                                    // branch 1
      let idx = 0;
      boots.slice(0, concurLimit).forEach((b) => { b(); idx++; });
      while (idx<boots.length) {
        const { pidx } = await Promise.race([...set]);
        set.delete([...set][pidx]);
        boots[idx++]();
      }
    })(),
    (async () => {                                    // branch 2
      for (const p of bootableProms) console.log(await p);
    })(),
  ]);
}
```

As before there are two branches

- branch 1: For running and handling concurrency.
- branch 2: For printing

The difference now is the no more than `concurLimit` Promises are allowed to run concurrently.

The important variables are

- `boots`: The array of functions to call to force its corresponding Promise to transition. It is used only in branch 1.
- `set`: There are Promises in a random access container so that they can be easily removed once fulfilled. This container is used only in branch 1.
- `bootableProms`: These are the same Promises as initially in `set`, but it is an array not a set, and the array is never changed. It is used only in branch 2.

Running with a mock `fs.readFile` that takes times as follows (filename vs. time in ms).

```
const timeTable = {
  "1": 600,
  "2": 500,
  "3": 400,
  "4": 300,
  "5": 200,
  "6": 100,
};
```

test run times such as this are seen, showing the concurrency is working --

```
[1]0--0.601
[2]0--0.502
[3]0.503--0.904
[4]0.608--0.908
[5]0.905--1.105
[6]0.905--1.005
```

Available as executable in the typescript playground sandbox

Share
Improve this answer
Follow

edited Nov 22, 2022 at 7:37

LinuxDisciple
2,369 ● 17 ● 20

answered Jan 17, 2022 at 20:42

Craig Hicks
2,518 ● 29 ● 40

9

You can use a simple traditional for loop like this

```
for(let i = 0; i< products.length; i++){
    await <perform some action like database read>
}
```

Share Improve this answer Follow

answered Sep 4, 2023 at 4:50

Shubham Goel
2,156 ● 20 ● 28

The question was more about using the `forEach` method specifically. But if you were to do it this style, I would recommend using a `for..of` loop instead. `for (const product of products) { await <perform some action like database read> }` – Saad Sep 4, 2023 at 21:54

@Saad sometimes you also need an index... – Diego Nov 20 at 19:10

8

Both the solutions above work, however, Antonio's does the job with less code, here is how it helped me resolve data from my database, from several different child refs and then pushing them all into an array and resolving it in a promise after all is done:

```
Promise.all(PacksList.map((pack)=>{
    return fireBaseRef.child(pack.folderPath).once('value',(snap)=>{
        snap.forEach( childSnap => {
            const file = childSnap.val()
            file.id = childSnap.key;
            allItems.push( file )
        })
    })
})).then(()=>store.dispatch( actions.allMockupItems(allItems)))
```

Share Improve this answer Follow

answered Aug 26, 2017 at 10:47

Hooman Askari
1,616 ● 17 ● 30

8

Like @Bergi's response, but with one difference.

`Promise.all` rejects all promises if one gets rejected.

So, use a recursion.

```
const readFilesQueue = async (files, index = 0) {
    const contents = await fs.readFile(files[index], 'utf8')
    console.log(contents)

    return files.length <= index
        ? readFilesQueue(files, ++index)
        : files

}

const printFiles async = () => {
    const files = await getFilePaths();
    const printContents = await readFilesQueue(files)

    return printContents
}

printFiles()
```

PS

`readFilesQueue` is outside of `printFiles` cause the side effect* introduced by `console.log`, it's better to mock, test, and or spy so, it's not cool to have a function that returns the content(sidenote).

Therefore, the code can simply be designed by that: three separated functions that are "pure"** and introduce no side effects, process the entire list and can easily be modified to handle failed cases.

```
const files = await getFilesPath()

const printFile = async (file) => {
    const content = await fs.readFile(file, 'utf8')
    console.log(content)
}

const readFiles = async = (files, index = 0) => {
    await printFile(files[index])

    return files.lengh <= index
        ? readFiles(files, ++index)
        : files
}

readFiles(files)
```

Future edit/current state

Node supports top-level await (this doesn't have a plugin yet, won't have and can be enabled via harmony flags), it's cool but doesn't solve one problem (strategically I work only on LTS versions). How to get the files?

Using composition. Given the code, causes to me a sensation that this is inside a module, so, should have a function to do it. If not, you should use an IIFE to wrap the role code into an async function creating simple module that's do all for you, or you can go with the right way, there is, composition.

```
// more complex version with IIFE to a single module
(async (files) => readFiles(await files())(getFilesPath)
```

Note that the name of variable changes due to semantics. You pass a functor (a function that can be invoked by another function) and recieves a pointer on memory that contains the initial block of logic of the application.

But, if's not a module and you need to export the logic?

Wrap the functions in a async function.

```
export const readFilesQueue = async () => {
    // ... to code goes here
}
```

Or change the names of variables, whatever...

`*` by side effect menans any colacteral effect of application that can change the statate/behaviour or introuce bugs in the application, like IO.

`**` by "pure", it's in apostrophe since the functions it's not pure and the code can be converged to a pure version, when there's no console output, only data manipulations.

Aside this, to be pure, you'll need to work with monads that handles the side effect, that are error prone, and treats that error separately of the application.

Share

Improve this answer

Follow

edited May 3, 2020 at 14:46

answered Dec 21, 2019 at 1:11

lukaswilkeer
345 ● 4 ● 15

7

Currently the Array.forEach prototype property doesn't support async operations, but we can create our own poly-fill to meet our needs.

```
// Example of asyncForEach Array poly-fill for NodeJs
// file: asyncForEach.js
// Define asynForEach function
async function asyncForEach(iteratorFunction){
  let indexer = 0
  for(let data of this){
    await iteratorFunction(data, indexer)
    indexer++
  }
}
// Append it as an Array prototype property
Array.prototype.asyncForEach = asyncForEach
module.exports = {Array}
```

And that's it! You now have an async forEach method available on any arrays that are defined after these to operations.

Let's test it...

```
// Nodejs style
// file: someOtherFile.js

const readline = require('readline')
Array = require('./asyncForEach').Array
const log = console.log
```

```
// Create a stream interface
function createReader(options={prompt: '>'}){
  return readline.createInterface({
    input: process.stdin
    ,output: process.stdout
    ,prompt: options.prompt !== undefined ? options.prompt : '>'
  })
}
// Create a cli stream reader
async function getUserIn(question, options={prompt:'>'}){
  log(question)
  let reader = createReader(options)
  return new Promise((res)=>{
    reader.on('line', (answer)=>{
      process.stdout.cursorTo(0, 0)
      process.stdout.clearScreenDown()
      reader.close()
      res(answer)
    })
  })
}

let questions = [
  `What's your name`
  ,`What's your favorite programming language`
  ,`What's your favorite async function`
]
let responses = {}

async function getResponses(){
// Notice we have to prepend await before calling the async Array function
// in order for it to function as expected
  await questions.asyncForEach(async function(question, index){
    let answer = await getUserIn(question)
    responses[question] = answer
  })
}

async function main(){
  await getResponses()
  log(responses)
}
main()
// Should prompt user for an answer to each question and then
// log each question and answer as an object to the terminal
```

We could do the same for some of the other array functions like map...

```
async function asyncMap(iteratorFunction){
  let newMap = []
  let indexer = 0
  for(let data of this){
    newMap[indexer] = await iteratorFunction(data, indexer, this)
    indexer++
  }
  return newMap
}
```

```
Array.prototype.asyncMap = asyncMap
```

... and so on :)

Some things to note:

- Your iteratorFunction must be an async function or promise
- Any arrays created before `Array.prototype.<yourAsyncFunc> = <yourAsyncFunc>` will not have this feature available

Share

Improve this answer

Follow

edited Apr 26, 2019 at 9:22

mikemaccana

122k ● 110 ● 424 ● 528

answered Mar 12, 2019 at 23:31

Beau

91 ● 1 ● 4

7

Today I came across multiple solutions for this. Running the async await functions in the forEach Loop. By building the wrapper around we can make this happen.

More detailed explanation on how it works internally, for the native forEach and why it is not able to make a async function call and other details on the various methods are provided in link here

The multiple ways through which it can be done and they are as follows,

Method 1 : Using the wrapper.

```
await (()=>{
     return new Promise((resolve,reject)=>{
       items.forEach(async (item,index)=>{
           try{
               await someAPICall();
           } catch(e) {
              console.log(e)
           }
           count++;
           if(index === items.length-1){
             resolve('Done')
           }
         });
     });
    })();
```

Method 2: Using the same as a generic function of Array.prototype

Array.prototype.forEachAsync.js

```
if(!Array.prototype.forEachAsync) {
    Array.prototype.forEachAsync = function (fn){
```

```
      return new Promise((resolve,reject)=>{
        this.forEach(async(item,index,array)=>{
            await fn(item,index,array);
            if(index === array.length-1){
                resolve('done');
            }
        })
      });
    };
  }
```

Usage :

```
require('./Array.prototype.forEachAsync');

let count = 0;

let hello = async (items) => {

// Method 1 - Using the Array.prototype.forEach

    await items.forEachAsync(async () => {
         try{
               await someAPICall();
          } catch(e) {
             console.log(e)
          }
        count++;
    });

    console.log("count = " + count);
}

someAPICall = () => {
    return new Promise((resolve, reject) => {
        setTimeout(() => {
            resolve("done") // or reject('error')
        }, 100);
    })
}

hello(['', '', '', '']); // hello([]) empty array is also be handled by default
```

Method 3 :

Using Promise.all

```
  await Promise.all(items.map(async (item) => {
        await someAPICall();
        count++;
    }));

    console.log("count = " + count);
```

Method 4 : Traditional for loop or modern for loop

```
// Method 4 - using for loop directly

// 1. Using the modern for(.. in..) loop
   for(item in items){

        await someAPICall();
        count++;
    }

//2. Using the traditional for loop

    for(let i=0;i<items.length;i++){

        await someAPICall();
        count++;
    }


    console.log("count = " + count);
```

Share

Improve this answer

Follow

edited Nov 27, 2019 at 5:42

answered Nov 24, 2019 at 20:31

Pranav Kumar H M

987 ● 7 ● 10

Your methods 1 and 2 are simply incorrect implementations where `Promise.all` should have been used - they do not take any of the many edge cases into account. – Bergi Nov 24, 2019 at 21:48

@Bergi: Thanks for the valid comments, Would you please explain me why method 1 and 2 are incorrect. It also serves the purpose. This works very well. This is to say that all these methods are possible, based on the situation one can decide on choosing one. I have the running example for the same. – Pranav Kumar H M Nov 25, 2019 at 14:15

It fails on empty arrays, it doesn't have any error handling, and probably more problems. Don't reinvent the wheel. Just use `Promise.all`. – Bergi Nov 25, 2019 at 15:25

In certain conditions where its not possible it will be helpful. Also error handling is done by forEach api by default so no issues. Its taken care ! – Pranav Kumar H M Nov 26, 2019 at 5:57

No, there are no conditions where `Promise.all` is not possible but `async` / `await` is. And no, `forEach` absolutely doesn't handle any promise errors. – Bergi Nov 26, 2019 at 13:22

6

To see how that can go wrong, print console.log at the end of the method.

Things that can go wrong in general:

- Arbitrary order.
- printFiles can finish running before printing files.
- Poor performance.

These are not always wrong but frequently are in standard use cases.

Generally, using forEach will result in all but the last. It'll call each function without awaiting for the function meaning it tells all of the functions to start then finishes without waiting for the functions to finish.

```
import fs from 'fs-promise'

async function printFiles () {
  const files = (await getFilePaths()).map(file => fs.readFile(file, 'utf8'))

  for(const file of files)
    console.log(await file)
}

printFiles()
```

This is an example in native JS that will preserve order, prevent the function from returning prematurely and in theory retain optimal performance.

This will:

- Initiate all of the file reads to happen in parallel.
- Preserve the order via the use of map to map file names to promises to wait for.
- Wait for each promise in the order defined by the array.

With this solution the first file will be shown as soon as it is available without having to wait for the others to be available first.

It will also be loading all files at the same time rather than having to wait for the first to finish before the second file read can be started.

The only draw back of this and the original version is that if multiple reads are started at once then it's more difficult to handle errors on account of having more errors that can happen at a time.

With versions that read a file at a time then then will stop on a failure without wasting time trying to read any more files. Even with an elaborate cancellation system it can be hard to avoid it failing on the first file but reading most of the other files already as well.

Performance is not always predictable. While many systems will be faster with parallel file reads some will prefer sequential. Some are dynamic and may shift under load, optimisations that offer latency do not always yield good throughput under heavy contention.

There is also no error handling in that example. If something requires them to either all be successfully shown or not at all it won't do that.

In depth experimentation is recommended with console.log at each stage and fake file read solutions (random delay instead). Although many solutions appear to do the same in simple cases all have subtle differences that take some extra scrutiny to squeeze out.

Use this mock to help tell the difference between solutions:

```
(async () => {
  const start = +new Date();
  const mock = () => {
    return {
      fs: {readFile: file => new Promise((resolve, reject) => {
        // Instead of this just make three files and try each timing
arrangement.
        // IE, all same, [100, 200, 300], [300, 200, 100], [100, 300, 200],
etc.
        const time = Math.round(100 + Math.random() * 4900);
        console.log(`Read of ${file} started at ${new Date() - start} and will
take ${time}ms.`)
        setTimeout(() => {
          // Bonus material here if random reject instead.
          console.log(`Read of ${file} finished, resolving promise at ${new
Date() - start}.`);
          resolve(file);
        }, time);
      })},
      console: {log: file => console.log(`Console Log of ${file} finished at
${new Date() - start}.`)},
      getFilePaths: () => ['A', 'B', 'C', 'D', 'E']
    };
  };

  const printFiles = (({fs, console, getFilePaths}) => {
    return async function() {
      const files = (await getFilePaths()).map(file => fs.readFile(file,
'utf8'));

      for(const file of files)
        console.log(await file);
    };
  })(mock());

  console.log(`Running at ${new Date() - start}`);
  await printFiles();
  console.log(`Finished running at ${new Date() - start}`);
})();
```

Share

Improve this answer

Follow

edited Oct 14, 2019 at 19:16

answered Oct 14, 2019 at 18:35

jgmjgm
4,675 ● 1 ● 28 ● 20

5

Using Task, futurize, and a traversable List, you can simply do

```
async function printFiles() {
  const files = await getFiles();

  List(files).traverse( Task.of, f => readFile( f, 'utf-8'))
    .fork( console.error, console.log)
}
```

Here is how you'd set this up

```
import fs from 'fs';
import { futurize } from 'futurize';
import Task from 'data.task';
import { List } from 'immutable-ext';

const future = futurizeP(Task)
const readFile = future(fs.readFile)
```

Another way to have structured the desired code would be

```
const printFiles = files =>
  List(files).traverse( Task.of, fn => readFile( fn, 'utf-8'))
    .fork( console.error, console.log)
```

Or perhaps even more functionally oriented

```
// 90% of encodings are utf-8, making that use case super easy is prudent

// handy-library.js
export const readFile = f =>
  future(fs.readFile)( f, 'utf-8' )

export const arrayToTaskList = list => taskFn =>
  List(files).traverse( Task.of, taskFn )

export const readFiles = files =>
  arrayToTaskList( files, readFile )

export const printFiles = files =>
  readFiles(files).fork( console.error, console.log)
```

Then from the parent function

```
async function main() {
  /* awesome code with side-effects before */
  printFiles( await getFiles() );
  /* awesome code with side-effects after */
}
```

If you really wanted more flexibility in encoding, you could just do this (for fun, I'm using the proposed Pipe Forward operator)

```
import { curry, flip } from 'ramda'

export const readFile = fs.readFile
  |> future,
  |> curry,
  |> flip

export const readFileUtf8 = readFile('utf-8')
```

PS - I didn't try this code on the console, might have some typos... "straight freestyle, off the top of the dome!" as the 90s kids would say. :-p

Share

Improve this answer

Follow

edited Apr 3, 2018 at 22:51

answered Feb 28, 2018 at 4:41

Babakness

3,114 ● 3 ● 20 ● 28

3

As other answers have mentioned, you're probably wanting it to be executed in sequence rather in parallel. Ie. run for first file, wait until it's done, *then* once it's done run for second file. That's not what will happen.

I think it's important to address *why* this doesn't happen.

Think about how `forEach` works. I can't find the source, but I presume it works something like this:

```
const forEach = (arr, cb) => {
  for (let i = 0; i < arr.length; i++) {
    cb(arr[i]);
  }
};
```

Now think about what happens when you do something like this:

```
forEach(files, async logFile(file) {
  const contents = await fs.readFile(file, 'utf8');
  console.log(contents);
});
```

Inside `forEach`'s `for` loop we're calling `cb(arr[i])`, which ends up being `logFile(file)`. The `logFile` function has an `await` inside it, so maybe the `for` loop will wait for this `await` before proceeding to `i++`?

No, it won't. Confusingly, that's not how `await` works. From the docs:

> An await splits execution flow, allowing the caller of the async function to resume execution. After the await defers the continuation of the async function, execution of subsequent statements ensues. If this await is the last expression executed by its function execution continues by returning to the function's caller a pending Promise for completion of the await's function and resuming execution of that caller.

So if you have the following, the numbers won't be logged before `"b"`:

```
const delay = (ms) => {
  return new Promise((resolve) => {
    setTimeout(resolve, ms);
  });
};

const logNumbers = async () => {
  console.log(1);
  await delay(2000);
  console.log(2);
  await delay(2000);
  console.log(3);
};

const main = () => {
  console.log("a");
  logNumbers();
  console.log("b");
};

main();
```

Circling back to `forEach`, `forEach` is like `main` and `logFile` is like `logNumbers`. `main` won't stop just because `logNumbers` does some `await`ing, and `forEach` won't stop just because `logFile` does some `await`ing.

Share Improve this answer Follow

answered Dec 23, 2020 at 19:51

Adam Zerner

19.1k ● 17 ● 99 ● 168

2

Similar to Antonio Val's `p-iteration`, an alternative npm module is `async-af`:

```
const AsyncAF = require('async-af');
const fs = require('fs-promise');

function printFiles() {
  // since AsyncAF accepts promises or non-promises, there's no need to await
here
  const files = getFilePaths();

  AsyncAF(files).forEach(async file => {
```

```
    const contents = await fs.readFile(file, 'utf8');
    console.log(contents);
  });
}

printFiles();
```

Alternatively, `async-af` has a static method (log/logAF) that logs the results of promises:

```
const AsyncAF = require('async-af');
const fs = require('fs-promise');

function printFiles() {
  const files = getFilePaths();

  AsyncAF(files).forEach(file => {
    AsyncAF.log(fs.readFile(file, 'utf8'));
  });
}

printFiles();
```

However, the main advantage of the library is that you can chain asynchronous methods to do something like:

```
const aaf = require('async-af');
const fs = require('fs-promise');

const printFiles = () => aaf(getFilePaths())
  .map(file => fs.readFile(file, 'utf8'))
  .forEach(file => aaf.log(file));

printFiles();
```

`async-af`

Share Improve this answer Follow

answered Jun 21, 2018 at 16:55

Scott Rudiger
1,292 ● 12 ● 16

2

Here is a great example for using async in forEach loop.

Write your own asyncForEach

```
async function asyncForEach(array, callback) {
    for (let index = 0; index < array.length; index++) {
        await callback(array[index], index, array)
    }
}
```

You can use it like this

```
await asyncForEach(array, async function(item,index,array){
     //await here
   }
)
```

Share Improve this answer Follow

answered Feb 11, 2021 at 1:45

Jellow

101 • 6

1

If you'd like to iterate over all elements concurrently:

```
async function asyncForEach(arr, fn) {
  await Promise.all(arr.map(fn));
}
```

If you'd like to iterate over all elements non-concurrently (e.g. when your mapping function has side effects or running mapper over all array elements at once would be too resource costly):

Option A: Promises

```
function asyncForEachStrict(arr, fn) {
  return new Promise((resolve) => {
    arr.reduce(
      (promise, cur, idx) => promise
        .then(() => fn(cur, idx, arr)),
      Promise.resolve(),
    ).then(() => resolve());
  });
}
```

Option B: async/await

```
async function asyncForEachStrict(arr, fn) {
  for (let idx = 0; idx < arr.length; idx += 1) {
    const cur = arr[idx];

    await fn(cur, idx, arr);
  }
}
```

Share Improve this answer Follow

answered Nov 19, 2020 at 10:45

Wojciech Maj

1,102 • 8 • 23

1 Your option a involves the `Promise` constructor antipattern. – Bergi Nov 19, 2020 at 15:02

1

For TypeScript users, a `Promise.all(array.map(iterator))` wrapper with working types

- Using `Promise.all(array.map(iterator))` has correct types since the TypeScript's stdlib support already handles generics.
- However copy pasting `Promise.all(array.map(iterator))` every time you need an async map is obviously suboptimal, and `Promise.all(array.map(iterator))` doesn't convey the intention of the code very well - so most developers would wrap this into an `asyncMap()` wrapper function. However doing this requires use of generics to ensure that values set with `const value = await asyncMap()` have the correct type.

```
export const asyncMap = async <ArrayItemType, IteratorReturnType>(
  array: Array<ArrayItemType>,
  iterator: (
    value: ArrayItemType,
    index?: number
  ) => Promise<IteratorReturnType>
): Promise<Array<IteratorReturnType>> => {
  return Promise.all(array.map(iterator));
};
```

And a quick test:

```
it(`runs 3 items in parallel and returns results`, async () => {
  const result = await asyncMap([1, 2, 3], async (item: number) => {
    await sleep(item * 100);
    return `Finished ${item}`;
  });
  expect(result.length).toEqual(3);
  // Each item takes 100, 200 and 300ms
  // So restricting this test to 300ms plus some leeway
}, 320);
```

`sleep()` is just:

```
const sleep = async (timeInMs: number): Promise<void> => {
  return new Promise((resolve) => setTimeout(resolve, timeInMs));
};
```

edited Dec 30, 2022 at 22:53

answered Dec 12, 2022 at 12:01

mikemaccana

122k ● 110 ● 424 ● 528

Share

Improve this answer

Follow

If anyone has feedback on this answer please let me know - I generally believe most programmers wouldn't want to copy paste `Promise.all(array.map(iterator))` rather than just have a single function, and sadly wrapping `Promise.all(array.map(iterator))` without generics won't have the correct types. The answer also isn't a duplicate, and should be helpful to anyone using async/await and TS, so if there's something I can improve (which there seems to be from the voting so far) please tell me. – mikemaccana Dec 30, 2022 at 21:42

2 I'm not sure this is what OP wanted. It runs all promises in **parallel**, only waiting at the end for all to finish (in unpredictable order). Sometimes that's great, sometimes you really want each iteration to finish (including `await ...`) before starting next iteration.
– Beni Cherniavsky-Paskin Nov 30, 2023 at 9:47

Yep, parallel was intended for the purposes of speed, hence the `it runs 3 items in parallel and returns results` test. I can add a version for series, one moment...
– mikemaccana Dec 8, 2023 at 0:33

1 2 Next

Highly active question. Earn 10 reputation (not counting the association bonus) in order to answer this question. The reputation requirement helps protect this question from spam and non-answer activity.