

[在应用程序中打开 ↗](#)[报名](#)[登入](#)

即使使用 `async/await`, 原始的 `Promise` 仍然是编写最佳并发 javascript 的关键

丹尼尔·布莱恩 · [跟随](#)

阅读时间 9 分钟 · 2017 年 2 月 21 日

[听](#)[分享](#)

到了 es2017, `async/await` 指日可待。在上一篇文章中, [我建议充分掌握 `Promise`](#), 因为它们是构建 `async/await` 的基础。了解 `Promise` 有助于理解 `async/await` 基础的概念, 并帮助您编写更好的异步函数。

但即使您已经加入了异步潮流 (这是我个人喜欢的潮流), 并且您完全理解 `Promise`, 仍然有一些非常令人信服的理由继续将它们与异步函数一起使用。`Async/await` 绝对不会让您完全不必经常打破它们。为什么? 简单的:

1. 您可能仍然需要编写代码才能发送到浏览器
2. 使用纯 `async/await` 编写真正的并发代码有时是不可能或容易的。

为浏览器编写代码? 这不是 `Babel` 已经解决的问题了吗?

因此, 显然除非您纯粹为 Node 中的服务器端编写代码, 否则您将不得不考虑在浏览器中运行 JavaScript。`Babel` 提供了一种非常好的方法, 可以将 ES2015+ 编译为可以在旧版浏览器中运行的 javascript, 并且使用 Facebook 出色的[Regenerator](#), `Babel` 甚至可以编译 `async/await` 代码。

那么问题解决了吗? 嗯, 不完全是。

症结在于, 生成的代码不一定是您希望在客户端包中提供的代码。例如, 采用这个简单的异步函数, 它使用异步映射器函数串行映射数组:

```
异步函数serialAsyncMap(collection, fn) {  
  let result = [];
```

```

for (let item of collection) {
    result.push(await fn(item));
}

```

Babel/Regenerator 将其编译为以下 56 行函数:

```

var serialAsyncMap = function () {
    var _ref = _asyncToGenerator(regeneratorRuntime.mark(function
    _callee(collection, fn) {
        var result, _iterator, _isArray, _i, _ref2, item; }

    return regeneratorRuntime.wrap(function _callee$(_context) {
        while (1) {
            switch (_context.prev = _context.next) {
                case 0:
                    result = [];
                    _iterator = collection, _isArray =
                        Array.isArray(_iterator), _i = 0, _iterator = _isArray ? _iterator :
                        _iterator[Symbol.iterator]();
```

[请参阅此处查看完整代码](#)

这只是转译七行函数的结果。顺便说一句 - 那是在我们将其包含 regeneratorRuntime 或 _asyncToGenerator 纳入我们的捆绑包之前。因此, 虽然 Regenerator 是一项了不起的技术壮举, 但它并没有完全生成最精简的代码来发送到浏览器 - 我怀疑也不是最优化或性能最好的代码。它也很难阅读、理解和调试。

不管怎样, 考虑一下我们是否使用原始 Promise 编写了相同的函数:

```

函数serialAsyncMap(collection, fn) {
    让结果= [];
    让 Promise = Promise.resolve();

    for (let item of collection) {
        Promise = Promise.then(() => {
            return fn(item).then(result => {
                results.push(result);
            });
        });
    }
}
```

```

return Promise.then(() => {
    返回结果;
}) ;
}

```

或者稍微简洁的版本:

```

函数serialAsyncMap(collection, fn) {

    让结果= [];
    让 Promise = Promise.resolve();

    for (let item of collection) {
        Promise = Promise.then(() => fn(item))
            .then(result => results.push(result));
    }
    返回

    promise.then(() => 结果);
}

```

当然, 这里比纯 `async/await` 版本有更多的心理障碍, 但原始的 `Promise` 版本比 `Regenerator` 生成的代码明显更精简、更容易阅读、更容易调试。调试方面与源映射支持不太强的环境尤其相关 (通常是最需要调试的环境, 例如旧版本的 IE)。

还有其他选择吗?

`Regenerator` 有一些替代的 `async/await` 转译器, 它们采用异步代码并尝试转换为更传统的 `.then` 符号 `.catch`。根据我的经验, 这些转译器对于简单的函数来说工作得很好, `await` 然后 `return`, `try/catch` 最多可能只有一个块。但原始异步函数越复杂 (带有任何条件 `await` 语句或循环), 生成的代码最终就越像意大利面条。

至少对我来说, 这还不够好。如果我不能轻松地查看预转译的代码并想象转译的结果会是什么样子, 那么我能够轻松调试它的机会也相当渺茫。

`async/await` 还需要很长时间才能获得接近 100% 的浏览器支持, 因此在熟悉为客户端代码编写和编写 `Promise` 之前, 请不要屏住呼吸。

好吧, 所以我可能仍然需要为我的客户端编写 `Promise`, 但是只要我运行节点, 我就可以在我的服务器上使用 `async/await`, 对吧?

嗯, 是的, 也不是。

通常, 您可以在服务器端 JavaScript 中添加一些 `async function and` 语句, 也许可以发出一些 `http` 请求, 然后就可以开始了。`await` 您甚至可以并行化异步任务

`Promise.all` (尽管我认为这感觉有点不合适, 并且最好有特殊的语法来使用 `async/await` 运行并行函数)。

但是, 当您想要编写比“串行运行这些异步任务”或“并行运行这些异步任务”更复杂的东西时, 会发生什么?

给我一个例子...

考虑以下。我们想做披萨。

- 我们独立制作面团。
- 我们独立制作酱汁。
- 在我们决定哪种奶酪最适合做披萨之前, 我们希望能够品尝一下酱汁。

因此, 让我们从一个超级幼稚的纯异步/等待解决方案开始:

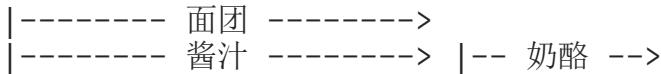
```
异步函数 makePizza(sauceType = 'red') {  
    让面团 = 等待 makeDough();  
    让酱=等待makeSauce(sauceType);  
    让奶酪=等待grateCheese(sauce.defineCheese());  
  
    面团.添加(酱汁);  
    面团.添加(奶酪);  
  
    返回面团;  
}
```

这有一个巨大的优势: 它非常简单, 并且非常容易阅读和理解。首先我们做面团, 然后做酱汁, 然后磨碎奶酪。简单的!

但这并不完全是最佳的。我们正在一件一件地做事情, 而实际上我们应该允许 javascript 引擎同时运行这些任务。所以而不是:

|----- 面团 -----> |----- 酱汁 -----> |-- 奶酪 -->

我们想要更多类似的东西:

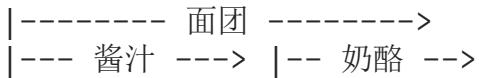


这样, 任务就能更快地完成。那么让我们再尝试一下:

```

异步函数 makePizza(sauceType = 'red') {
  let [ 面团, 酱 ] =
    等待 Promise.all([ makeDough(), makeSauce(sauceType) ]);
  让奶酪=等待grateCheese(sauce.defineCheese());
  面团.添加(酱汁);
  面团.添加(奶酪);
  返回面团;
}
  
```

好的, 所以我们的代码看起来有点时髦 `Promise.all`, 但至少现在是最佳的, 对吧? 噢.....不。在开始磨奶酪之前, 我正在等待面团和酱汁完成。如果我很快就做好酱汁怎么办? 现在我的执行看起来像这样:



请注意, 在我开始做奶酪之前, 我仍然在等待面团和酱汁的制作? 我只需要完成酱汁就可以开始做奶酪了, 所以我在这里浪费时间。因此, 让我们回到绘图板并尝试全力以赴地使用 `Promise` 而不是 `async/await`:

```

函数makePizza(sauceType = 'red') {
  让doughPromise = makeDough();
  让 saucePromise = makeSauce(sauceType);
  让cheesePromise = saucePromise.then(sauce => {
    return grateCheese(sauce.defineCheese());
  });

  return Promise.all([ 面团Promise, 酱料Promise, 奶酪Promise ])
    .then(([ 面团, 酱料, 奶酪 ]) => {
      面团.add(酱);
    })
}
  
```

```

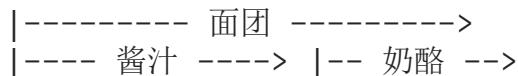
    面团.add(奶酪);

    返回面团;
  });

}

```

这样效果好多了。现在，每个任务将尽快完成，并且一旦其所有依赖项都完成。所以唯一阻止我磨碎奶酪的就是等待酱汁准备好。



但这样做时，我们必须完全选择不编写异步/等待代码，并全力以赴地兑现承诺。让我们试着稍微回顾一下 `async/await`:

```

异步函数 makePizza(sauceType = 'red') {

  让doughPromise = makeDough();
  让 saucePromise = makeSauce(sauceType);

  让酱=等待酱Promise;
  让奶酪=等待grateCheese(sauce.defineCheese());
  让面团=等待面团Promise;

  面团.添加(酱汁);
  面团.添加(奶酪);

  返回面团;
}

```

好吧，现在我们已经完全优化了，又回到了异步/等待状态.....但这仍然感觉像是倒退了一步。我们必须提前设置每个承诺，因此存在一些心理开销。我们还依赖于同时运行的这些承诺，因为这些任务是在等待任何任务之前提前设置的。通过阅读代码，这一点并不是很明显，并且将来可能会被意外地排除或破坏。所以这可能是我最不喜欢的实现。

让我们再试一次。我们可以再尝试一下:

```

异步函数 makePizza(sauceType = 'red') {

  让prepareDough = memoize(async () => makeDough());

```

```

    让 prepareSauce = memoize(async () => makeSauce(sauceType)) ;
    让 prepareCheese = memoize (async () => {
        返回 grateCheese ((等待prepareSauce ()) .确定Cheese ()) ;
    }) ;

    let [ 面团、酱汁、奶酪 ] =
        wait Promise.all([
            prepareDough(),prepareSauce(),prepareCheese()
        ]);

    面团.添加(酱汁);
    面团.添加(奶酪);

    返回面团;
}

```

这是我最喜欢的解决方案。我们不是提前设置隐式并发运行的 `Promise`, 而是设置三个记忆任务（保证每个任务仅运行一次）, 并调用它们以同时运行 `Promise.all`。

我们几乎完全避免在这里提及 `Promise`, 除了之外 `Promise.all`, 即使它们在幕后支持 `async/await`。我在另一篇关于记忆化和并发性的文章中更详细地讨论了这种模式。但在我看来, 这会带来最佳并发性和可读性/可维护性的完美组合。

当然, 我总是愿意被证明是错误的, 所以如果您有 `makePizza` 您喜欢的实现, 请告诉我!

所以我们做披萨的速度非常快, 你的意思是什么?

关键是, 如果您打算编写完全并发的代码, 即使在最新的 Node.js 版本中, 了解如何混合 `Promise` 和 `async/await` 仍然是一项绝对必要的技能。无论您最喜欢的实现 `makePizza` 是什么, 您仍然必须考虑如何将 `Promise` 链接在一起并组成, 以使函数运行时尽可能减少不必要的延迟。

`Async/await` 本身只能让您到目前为止, 如果您不知道 `Promise` 如何为您的代码提供动力, 那么您将陷入困境, 没有明显的方法来优化并发任务。

关于这一点.....

不要害怕编写助手来从业务逻辑中抽象出承诺/并发逻辑。一旦您了解了 `Promise` 的工作原理, 这样做可以从您的代码中删除大量意大利面条, 并使您的异步应用程序/业务逻辑函数的意图更加明显, 而不会用样板代码拥挤它们。

以这个函数为例, 它每十秒检查一次用户是否登录, 并在检测到以下情况时解决承诺:

```

function onUserLoggedIn(id) {
  return ajax(`user/${id}`).then(user => {
    if (user.state === 'logged_in') {
      return;
    }

    return new Promise(resolve => {
      return setTimeout(resolve, 10 * 1000));
    }).then(() => {
      return onUserLoggedIn(id);
    })
  });
}

```

这不是我想要发布的功能——业务逻辑与承诺/延迟逻辑紧密耦合。在我知道这个函数的作用之前，我必须阅读并理解整个混乱的内容。

为了改进这一点，我可以将异步/承诺逻辑拆分为一些单独的辅助函数，并使我的业务逻辑更加清晰：

```

函数延迟(时间) {
  返回新的Promise(解决=>{
    返回setTimeout(解决, 时间));
  });
}

函数直到(conditionFn,delayTime = 1000) {
  return Promise.resolve().then(() => {
    return conditionFn();
  }).then(result => {

    if (!result) {
      return delay(delayTime).then(() => {
        return直到(conditionFn,delayTime);
      });
    }
  });
}

```

或者这些助手的超级简洁版本：

```

让延迟 = 时间 =>
新 Promise(resolve =>

```

```

    setTimeout(resolve, time)
);

让直到 = (cond, 时间) =>
cond().then(结果 =>
    结果 || 延迟(时间).then(() =>
        直到(cond, 时间)
    )
);

```

然后 `onUserLoggedIn` 与控制流逻辑的耦合变得不那么紧密:

```

函数 onUserLoggedIn(id) {
    return until(() => {
        return ajax(`user/${id}`).then(user => {
            return user.state === 'logged_in';
        });
    }, 10 * 1000);
}

```

`onUserLoggedIn` 现在我对未来能够阅读和理解多了一点希望。只要我记住 `until` 实用函数的接口, 我就不必每次都重新理解它的逻辑。我可以将它放入一个 `promise-utils` 文件中, 而在很大程度上忘记它是如何工作的, 而专注于我的应用程序逻辑。

哦, 是的, 我们正在谈论异步/等待, 对吧? 好吧, 这是我们的幸运日 - 由于 `async/await` 和 `Promise` 是完全可互操作的, 我们只是无意中创建了一个可以继续使用的辅助函数, 即使使用异步函数也是如此:

```

异步函数onUserLoggedIn (id) {
    返回等待直到 (async () => {
        让用户=等待ajax (`user / $ {id}`);
        返回user.state ==='logged_in';
    }, 10 * 1000);
}

```

因此, 无论是基于 `Promise` 的代码还是基于 `async/await` 的代码, 同样的规则都是正确的 - 如果您发现并发逻辑渗透到您的异步业务函数中, 请务必考虑是否可以将其抽象一些。当然, 在合理范围内。

这里有一个相当大的现有抽象集合, 可能会有所帮助。

因此, 如果您从本文中获得任何内容, 那就是: 如果您正在编写异步/等待代码, 您不仅应该了解 `Promise` 的工作原理, 还应该准备好在必要时使用它们来构建异步/等待代码。单独使用 `Async/await` 并不能赋予您足够的能力来完全避免使用 `Promise` 进行思考。

谢谢!

— 丹尼尔

JavaScript

ES6

承诺

异步



跟随



丹尼尔·布莱恩编剧

2.9K 关注者

我写代码, 有时我的代码可以工作。丹尼尔@bluesuncorp.co.uk

丹尼尔·布莱恩的更多作品

 丹尼尔·布莱恩

使用 TypeScript 枚举的九种糟糕方法, 以及一种好方法。

TypeScript 枚举受到很多人的讨厌。这并不是没有道理的：他们有这么多潜在的步枪。

5 分钟阅读 · 2022 年 12 月 15 日

 106 14

```
tion formatName(firstName, lastName) {  
` ${firstName} ${lastName}`;  
  
tion initializeName(firstName, lastName) {  
` ${firstName[0].toUpperCase()} ${lastName[0].toUpperCase()}`;
```

 丹尼尔·布莱恩

JavaScript 树摇动, 像专业人士一样

JavaScript 中的 Tree Shaking 正在成为一种重要的实践, 以避免过大的包大小并提高性能。

7分钟阅读 · 2020年1月20日

👏 985

💬 2



丹尼尔·布莱恩

在 javascript 中获取 Windows 引用的所有已知方法

构建 zoid 和后机器人的最大挑战之一是弄清楚如何处理不同的窗口。两者的很大一部分.....

5分钟阅读 · 2017年2月3日

👏 第739章

💬 8





丹尼尔·布莱恩

在开始使用 `async/await` 之前了解 `Promise`

随着 Babel 现在支持开箱即用的 `async/await`, 并且 ES2016 (或 ES7) 即将到来, 越来越多的人正在意识到如何.....

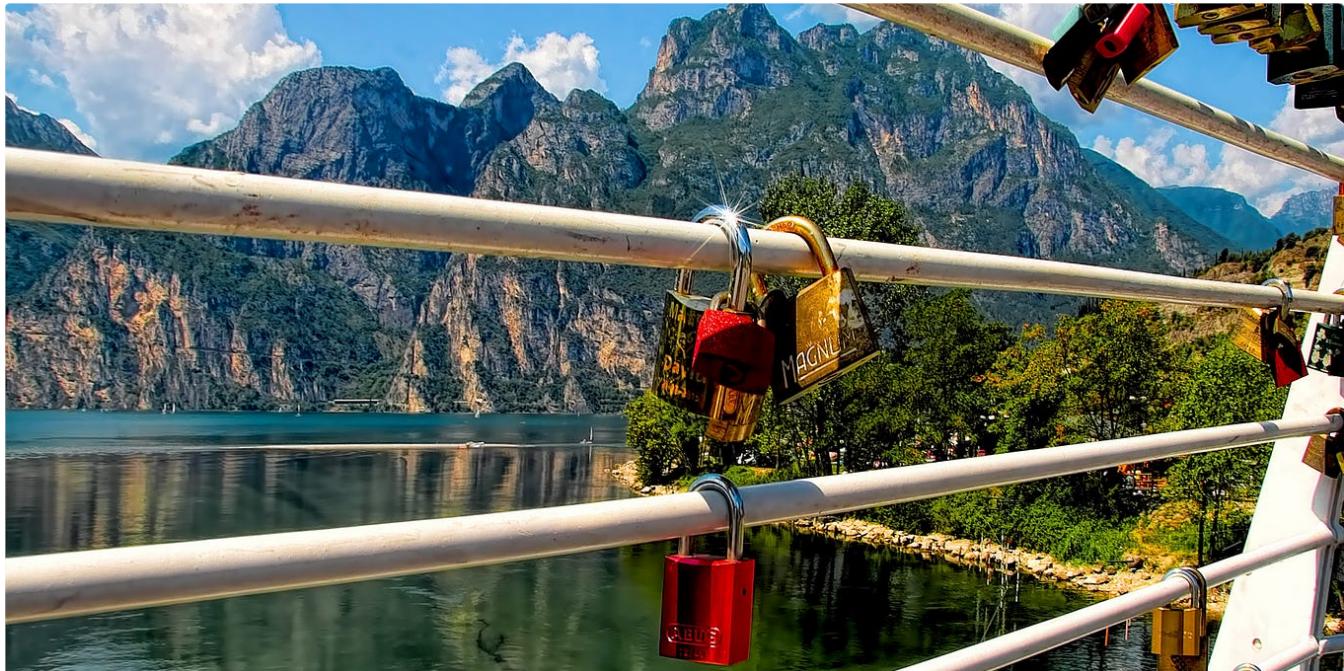
7分钟阅读 · 2016年1月19日

10.7K

41

[查看丹尼尔·布莱恩的全部内容](#)

媒体推荐



 丹尼尔·迪亚斯 在 极客文化

Promise.allSettled() 和 Promise.all() 有什么区别?

像专业人士一样使用 Promise

◆ · 3 分钟阅读 · 4月1日

 14





Express

JS

 乌特卡沙·巴克希 在 极客文化

如何使用 Express.js 将文件上传到 AWS S3?

在这篇短文中, 我将向您展示如何在 Express.js 应用程序中使用express-fileupload 将文件上传到 AWS S3。我们将使用 aws-sdk...

◆ · 4 分钟阅读 · 2月19日



42



列表



帮助您成长为软件开发人员的故事

19 个故事 · 196次保存



一般编码知识

20 个故事 · 104 次保存



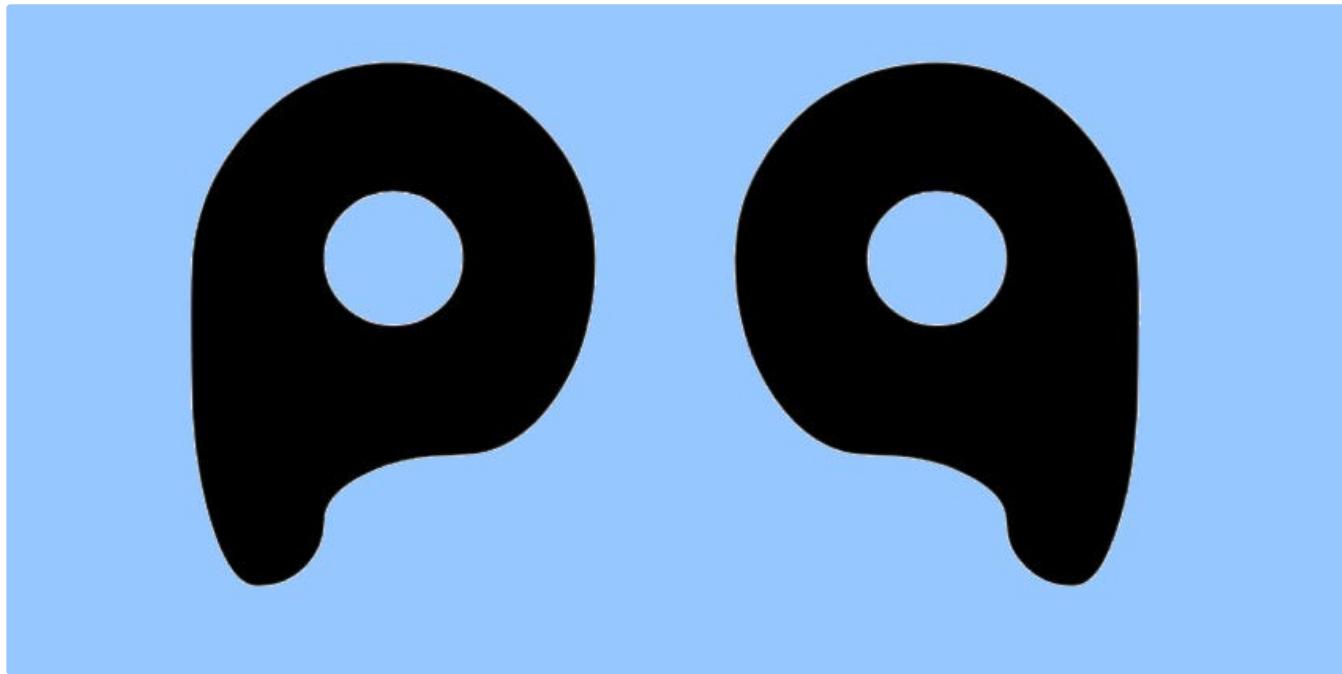
现代营销

32 个故事 · 29次扑救



中等出版物接受故事提交

145 个故事 · 263保存



熊猫任务 在 JavaScript 的简单英语

何时不在 React 中使用 useMemo

React 的 useMemo 钩子是一个强大的工具, 通过缓存昂贵的结果来优化应用程序的性能.....

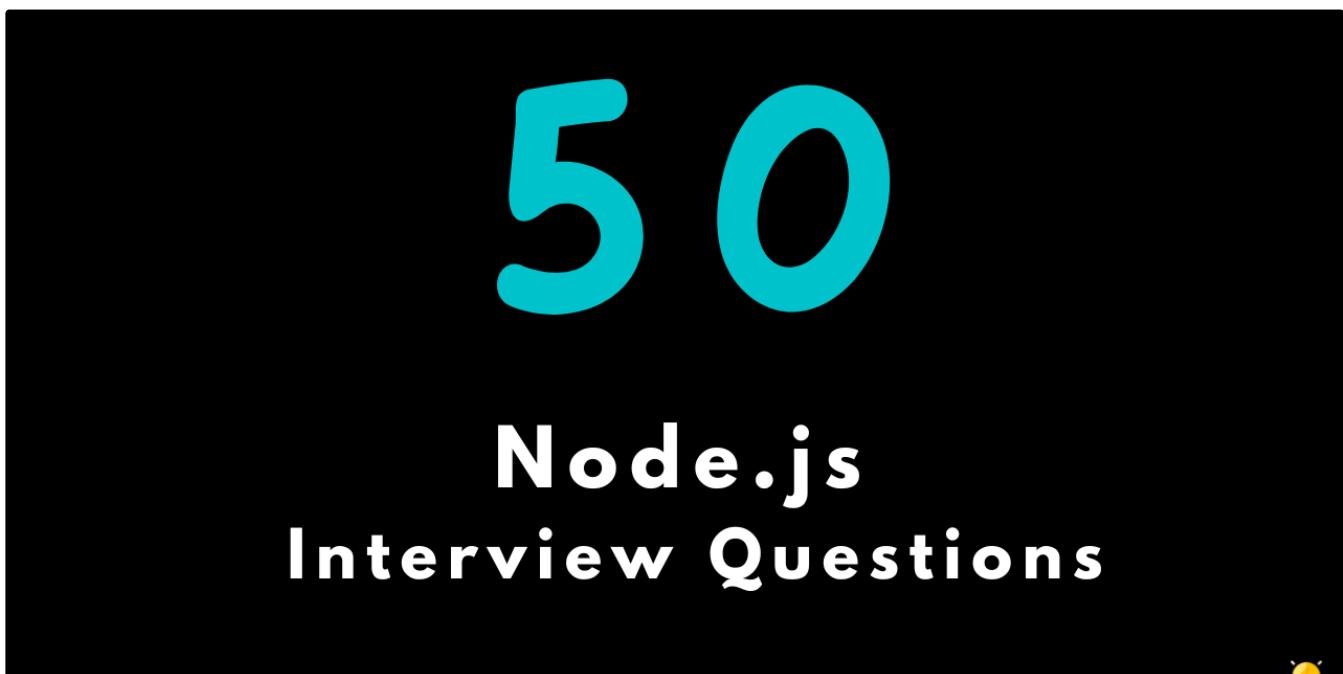
◆ · 4 分钟阅读 · 2月21日

 57

丹尼尔·迪亚斯 在 JavaScript 的简单英语

Map 和 WeakMap 有什么区别

JavaScript 提供了许多数据结构供开发人员选择。JavaScript 中最常用的两种数据结构是 Map...

 · 4 分钟阅读 · 3月29日 6



什雷·维杰瓦尔吉亚 在 前端周刊

50 个 Node JS 面试问题

把故事留到未来, 破解任何 Node JS 后端面试

◆ · 3 分钟阅读 · 2月13日

👏 265

💬 2



全栈技巧

20 个 Javascript 面试问题及代码答案。

我将开始一系列 JavaScript 面试问题, 这可能会对初级到中级开发人员有所帮助。

◆ · 6 分钟阅读 · 1月31日

👏 142

💬 4



查看更多推荐