

Entendendo Promises de uma vez por todas



Lucas Santos

Apr 8, 2019 · 19 min read

A large, bold, dark brown word "then" centered on a solid yellow rectangular background.

Créditos: fransciskim.com

Promises são um conceito essencial do JavaScript. Elas estão presentes em praticamente todo o ecossistema da linguagem.

Promises são um padrão de desenvolvimento que visam representar a conclusão de operações assíncronas. Elas não eram nativas do JavaScript até o ES6, quando houve uma implementação oficial na linguagem, antes delas, a maioria das funções usavam *callbacks*.

Neste artigo vamos estudar a fundo o que é uma Promise e como podemos entender seu funcionamento.

Vamos usar uma ferramenta muito legal chamado *Promisees* criada pelo Nicolás Bevacqua para podermos visualizar a execução de nossas Promises em tempo real de forma visual.

Antes de começar, queria deixar um *disclaimer* de que, apesar de saber que um artigo no Medium deveria ter mais ou menos 7 minutos para não ficar chato, este artigo é bem profundo e explicativo, então eu vou tentar colocar tudo o possível sobre Promises aqui para servir de guia para outros que possam estar procurando!

História das Promises

Promises remontam à década de 70 — como você pode ver neste artigo — e eram chamadas de **futures**, **deferred** ou **delays**. Pelo artigo, elas são definidas como:

Construtos usados para sincronizar a execução de um programa em linguagens de programação concorrentes. Eles descrevem um objeto que atua como um proxy para um resultado que é, inicialmente, desconhecido devido a sua computação não estar completa no momento da chamada.

De acordo com o que vemos na Internet, no JavaScript, as Promises fizeram sua primeira aparição em 2007 em uma biblioteca chamada `MochiKit` . Depois outras bibliotecas como o `Dojo` e o `jQuery` adotaram a mesma especificação pouco tempo depois.

Por fim, para padronizar todas as implementações, o grupo *CommonJS* escreveu a especificação chamada *Promises/A+* que visava ditar todas as regras necessárias para definir o que era uma Promise e sua interoperabilidade com outros sistemas.

No caso do NodeJS, nas primeiras versões, o runtime já implementava nativamente Promises, que foram removidas em favor de callbacks (que é a forma como conhecemos NodeJS no início), depois do lançamento do ES6 a plataforma

implementou nativamente a funcionalidade de Promises que já estava implementada no V8 desde algum tempo antes. Isto porque o padrão ES6 já implementa o modelo A+, que descrevemos antes, de forma nativa, portanto a grande maioria dos browsers já permite o uso de Promises sem nenhum tipo de biblioteca externa.

Fluxo Assíncrono

O JavaScript por si só é tido como uma linguagem que tem que lidar com várias chamadas e execuções que não acontecem no momento que o programador executou o código, por exemplo, a leitura de um arquivo no NodeJS de forma síncrona:

```
1 const fs = require('fs')
2
3 const texto = fs.readFileSync('./arquivo.txt')
```

sync.js hosted with ❤ by GitHub

[view raw](#)

Esta função é uma função síncrona, ou seja, quando a chamarmos, vamos pausar o que quer que esteja sendo executado e vamos realizar este processamento, depois vamos retornar o valor final. Desta forma estamos fazendo uma operação completamente síncrona. No nosso caso, vamos parar a execução do programa para buscar e ler o arquivo e depois vamos retornar seu resultado ao fluxo normal do programa.

Como queremos que nossas operações e nosso código rodem o mais rápido possível, queremos paralelizar o máximo de ações que conseguirmos. Ações de leitura de arquivos são consideradas lentas porque I/O é sempre mais lento que processamento em memória, vamos paralelizar a nossa função dizendo que queremos ler o arquivo de forma assíncrona:

```
1 const fs = require('fs')
2
3 fs.readFile('./arquivo.txt', (err, texto) => {
4   // fazer alguma coisa com o arquivo
5 })
```

async.js hosted with ❤ by GitHub

[view raw](#)

Agora o que estamos fazendo é passando um *callback* para a função *readFile* que deverá ser executado **após** a leitura do arquivo. Em essência — e abstraindo muito a funcionalidade — o que a função *readFile* faz é algo assim:

```

1  function readFile (path, callback) {
2    if (!path) throw new Error('Path is required')
3
4    // Leitura do arquivo de forma assíncrona
5    // Criando err como variável de erros e data como variável com o conteúdo
6
7    callback(err, dados)
8  }

```

readFileIntern.js hosted with ❤ by GitHub

[view raw](#)

Basicamente estamos registrando uma ação que vai ser executada após uma outra ação ser concluída, mas não sabemos quando essa ação será concluída. O que sabemos é apenas que em um momento ela será concluída, então o JavaScript utiliza o EventLoop — que não vamos cobrir neste artigo, mas vocês podem pesquisar aqui e aqui — para registrar um callback, basicamente o que estamos dizendo é: "Quando função X acabar, execute Y e me dê o resultado". Então estamos delegando a resolução de uma computação para outro método.

Uma outra opção

Muitas outras APIs nos fornecem uma outra opção ao se trabalhar com o fluxo assíncrono: os eventos.

Eventos são muito presentes no JavaScript, no front-end, quando escutamos por eventos de `click` em um botão com `elemento.addEventListener` ou no NodeJS quando podemos executar, por exemplo, um `fetch` que busca dados de uma API:

```

1  fetch('api')
2    .on('error', err => { /* faça algo */ })
3    .on('data', dados => { /* faça algo */ })

```

eventDriven.js hosted with ❤ by GitHub

[view raw](#)

O problema com a API de eventos é que o código fica literalmente solto, então é difícil manter uma linearidade de pensamento porque o código vai ficar pulando de parte para parte.

Por que Promises?

Se já tínhamos uma implementação de funções assíncronas, por que houve a preocupação de criar todo um novo padrão para podermos ter exatamente a mesma coisa? A questão aqui é mais a organização do código do que a funcionalidade.

Imagine que temos uma função que lê um arquivo, após este arquivo ser lido ela precisa escrever em um outro arquivo e ai executar outra função assíncrona. Nossa código ficaria assim:

```

1 const fs = require('fs')
2
3 fs.readFile('./arquivo.txt', (err, dados) => {
4     if (err) throw new Error(err)
5     fs.writeFile('./outroarquivo.txt', dados, (err) => {
6         if (err) throw new Error(err)
7         outraFuncaoAssincrona((err, dados) => {
8             const x = dados.split(',')
9             const y = x.map((e) => e.toUpperCase())
10            maisUmaFuncaoAssincrona(y, (err, resultado) => {
11                // continua
12            })
13        })
14    })
15})

```

[callbackHell.js](#) hosted with ❤ by GitHub

[view raw](#)

Veja que o código fica super complicado para ler... Isso é o que chamamos de *callback hell*



```

callbackhell.js
1 var floppy = require('floppy');
2
3 floppy.load('disk1', function (data1) {
4     floppy.prompt('Please insert disk 2', function() {
5         floppy.load('disk2', function (data2) {
6             floppy.prompt('Please insert disk 3', function() {
7                 floppy.load('disk3', function (data3) {
8                     floppy.prompt('Please insert disk 4', function() {
9                         floppy.load('disk4', function (data4) {
10                            floppy.prompt('Please insert disk 5', function() {
11                                floppy.load('disk5', function (data5) {
12                                    floppy.prompt('Please insert disk 6', function() {
13                                        floppy.load('disk6', function (data6) {
14                                            //if node.js would have existed in 1995
15                                            });
16                                        });
17                                    });
18                                });
19                            });
20                        });
21                    });
22                });
23            });
24        });
25    });
26});
27

```

Representação de um callback hell

As Promises foram um passo seguinte para que pudéssemos melhorar um pouco a execução do nosso código. Primeiramente vamos melhorar nosso código anterior,

podemos extrair as funções posteriores para outros blocos, melhorando um pouco a nossa visualização:

```

1  const fs = require('fs')
2
3  function callbackRead(err, dados) {
4      if (err) throw new Error(err)
5      fs.writeFile('./outroarquivo.txt', dados, callbackWrite)
6  }
7
8  function callbackWrite (err) {
9      if (err) throw new Error(err)
10     outraFuncaoAssincrona(callbackAsync1)
11 }
12
13 function callbackAsync1 (err, dados) {
14     const x = dados.split(',')
15     const y = x.map((e) => e.toUpperCase())
16     maisUmaFuncaoAssincrona(y, callbackAsync2)
17 }
18
19 function callbackAsync2 (err, resultado) {
20     // ...
21 }
22
23 fs.readFile('./arquivo.txt', callbackRead)

```

[callbackLessHell.js](#) hosted with ❤ by GitHub

[view raw](#)

Agora o problema é outro, estamos encadeando nossas funções e fica muito difícil entender todo o fluxo porque temos que passar em várias partes do código. Com Promises, nosso código ficaria assim:

```

1  const fs = require('fs')
2  const { promisify } = require('util')
3  const readFilePromise = promisify(fs.readFile)
4  const writeFilePromise = promisify(fs.writeFile)
5
6  function outraFuncaoAssincrona (parametro) {
7      return new Promise((resolve, reject) => {
8          resolve(parametro.split(','))
9      })
10 }
11
12 function maisUmaFuncaoAssincrona (parametro) {
13     return new Promise((resolve, reject) => {

```

```

14     // continua
15 }
16 }
17
18 readFilePromise('./arquivo.txt')
19 .then((err, dados) => {
20     writeFilePromise('./outroarquivo.txt', dados)
21     return dados
22 })
23 .then(outraFuncaoAssincrona)
24 .then(maisUmaFuncaoAssincrona)
25 .catch(console.error)

```

promises.js hosted with ❤ by GitHub

[view raw](#)

Veja que agora, apesar de nosso código não ter reduzido muito em tamanho, ele está mais legível, porque temos a implementação do `then`, de forma que conseguimos ver todo o *pipeline* de execução.

Promises

Promises, como já dissemos, definem uma ação que vai ser executada no futuro, ou seja, ela pode ser resolvida (com sucesso) ou rejeitada (com erro).

Há uma diferença entre lançar um erro e rejeitar uma promise. Lançar (dar um `throw`) no erro, vai parar a execução do seu código, é o equivalente a darmos um `return` em uma função. Porém rejeitar uma Promise fará com que o código continue sendo executado posteriormente

A anatomia de uma Promise segue a seguinte API:

```

1 // Criando uma promise
2 const p = new Promise((resolve, reject) => {
3     try {
4         resolve(funcaoX())
5     } catch (e) {
6         reject(e)
7     }
8 })
9
10 // Executando uma promise
11 p
12 .then((parametros) => /* sucesso */)
13 .catch((erro) => /* erro */)

```

```

14
15 // Tratando erros e sucessos no then
16 p
17 .then(resposta => { /* tratar resposta */ }, erro => { /* tratar erro */ })

```

[promiseAnatomy.js](#) hosted with ❤ by GitHub

[view raw](#)

Como podemos ver, toda a Promise retorna um método `then` e outro `catch`, utilizamos o `then` para tratar quando queremos **resolver** a Promise, e o `catch` quando queremos tratar os erros de uma Promise **rejeitada**. Tanto `then` quanto `catch` retornam **outra Promise** e é isso que permite que façamos o encadeamento de `then.then.then`.

Para criarmos uma Promise é muito simples, basta inicializar um `new Promise` que recebe uma função como parâmetro, esta função tem a assinatura `(resolve, reject) => {}`, então podemos realizar nossas tarefas assíncronas no corpo desta função, quando queremos retornar o resultado final fazemos `resolve(resultado)` e quando queremos retornar um erro fazemos `reject(erro)`.

Estados de uma Promise

Uma Promise pode assumir quatro estados principais:

- *Pending*: O estado inicial da Promise, ela foi iniciada mas ainda não foi realizada nem rejeitada
- *Fulfilled*: Sucesso da operação, é o que chamamos de uma Promise **realizada** (ou, em inglês, *resolved*) — eu, pessoalmente, prefiro o termo **resolvida**.
- *Rejected*: Falha da operação, é o que chamamos de uma Promise **rejeitada** (em inglês, *rejected*)
- *Settled*: É o estado final da Promise, quando ela já sabe se foi *resolved* ou *rejected*

Uma Promise que está pendente (*pending*) pode vir a se tornar uma Promise resolvida com um valor, ou então rejeitada por um motivo (que é o erro). Sempre que qualquer um dos dois casos acontecer, o método `then` da Promise será chamado e ele será o responsável por verificar se houve um erro ou um sucesso, chamando o método `resolve` em caso de sucesso, ou `reject` em caso de falha.

async actions

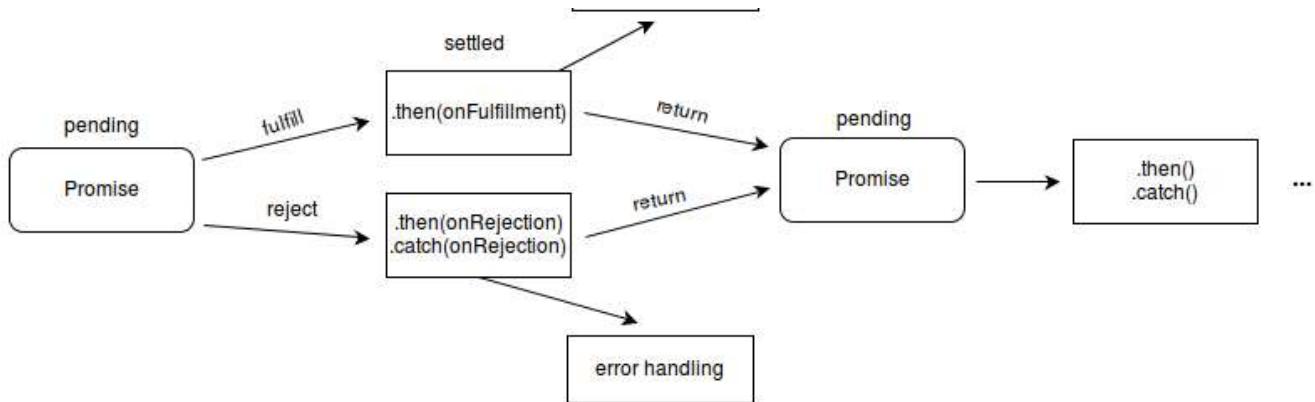


Diagrama de como uma Promise se comporta (Fonte: MDN)

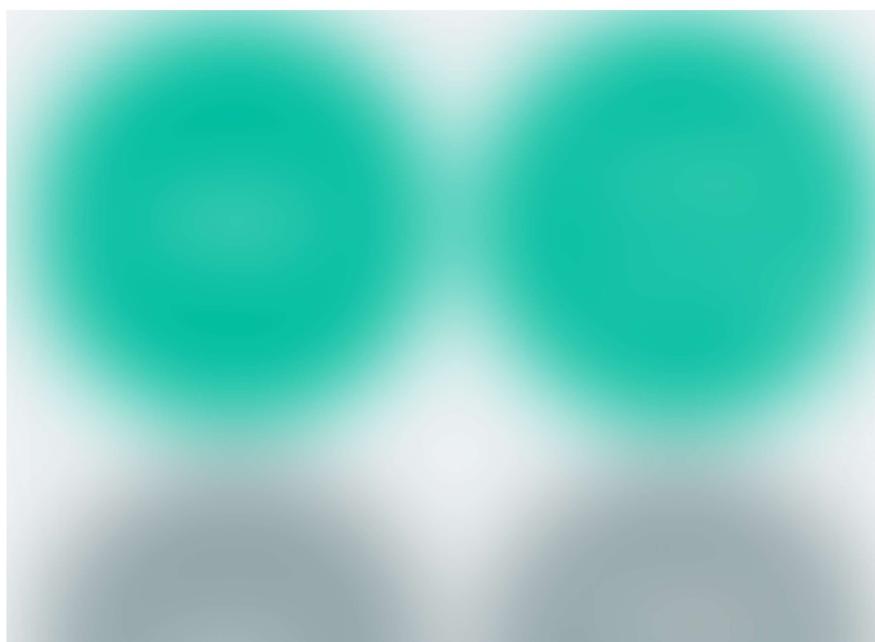
É importante mencionar que o método `catch` é somente um alias para o segundo parâmetro de `then`. Ele torna o código bem mais legível, então é considerado uma boa prática

Encadeamento

O encadeamento de Promises (com `then` e `catch`) é muito importante para entendermos o que está acontecendo, porque dependendo da forma como encadeamos nossas chamadas, vamos ter resultados diferentes.

Vamos tomar o seguinte exemplo:

Quando fazemos isto, estamos ligando tanto o bloco `then` quanto o `catch` na mesma Promise `p`, perceba que estamos passando dois parâmetros para a função `then` — ou então diretamente na Promise criada no segundo caso, não há diferenças. Vamos ter este mapa:

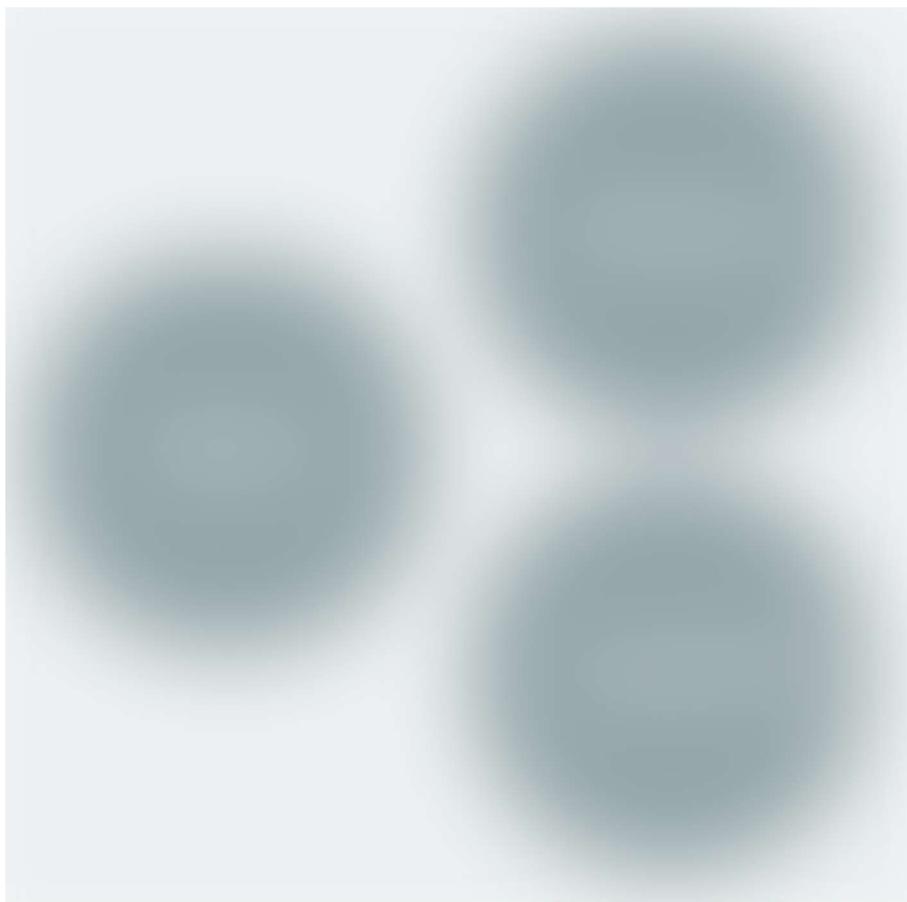




Ligando then e catch ao mesmo tempo

Vamos modificar um pouco o nosso código e fazer o bind do nosso `then` e `catch` separadamente:

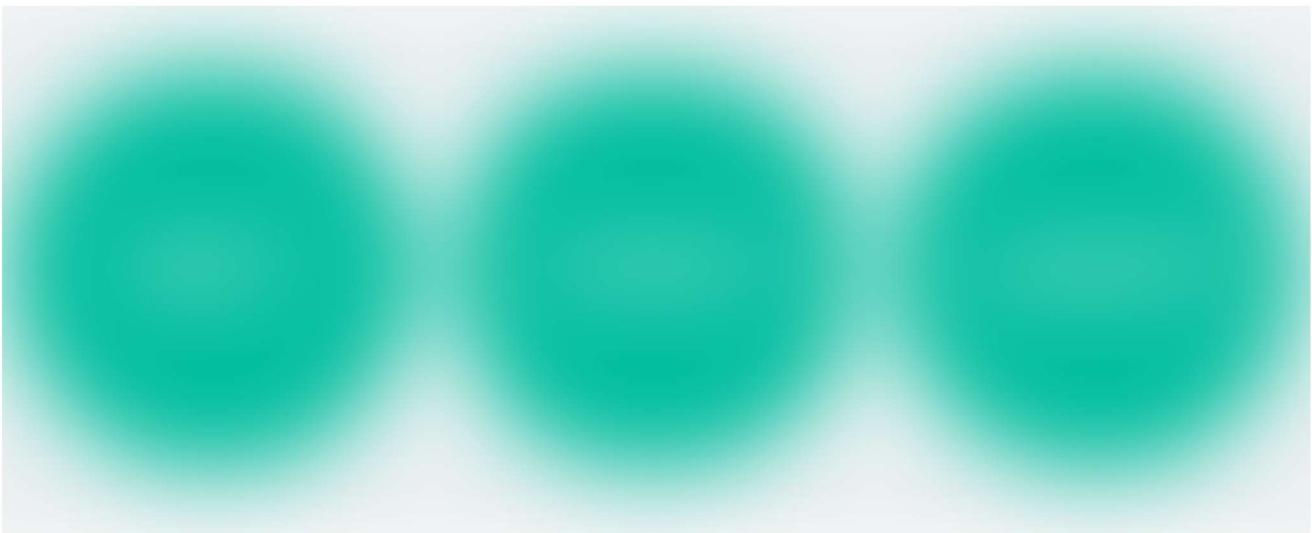
Isso nos dá dois *bindings* diferentes para a mesma Promise, apesar da semântica ser um pouco diferente, o resultado ainda é o mesmo que temos acima, porém com duas ligações diferentes:



Separando os bindings

Agora temos o terceiro caso, onde criamos o encadeamento de um `catch` no próprio `then` — isto porque, lembre-se, todo o `then` e `catch` retorna uma outra Promise para nós — vamos modificar o nosso código:

Isto fará com que a ligação do `catch` seja feita na Promise retornada por `then` e não na nossa Promise criada originalmente:



Encadeamento de Promises

Um `catch` para todos controlar

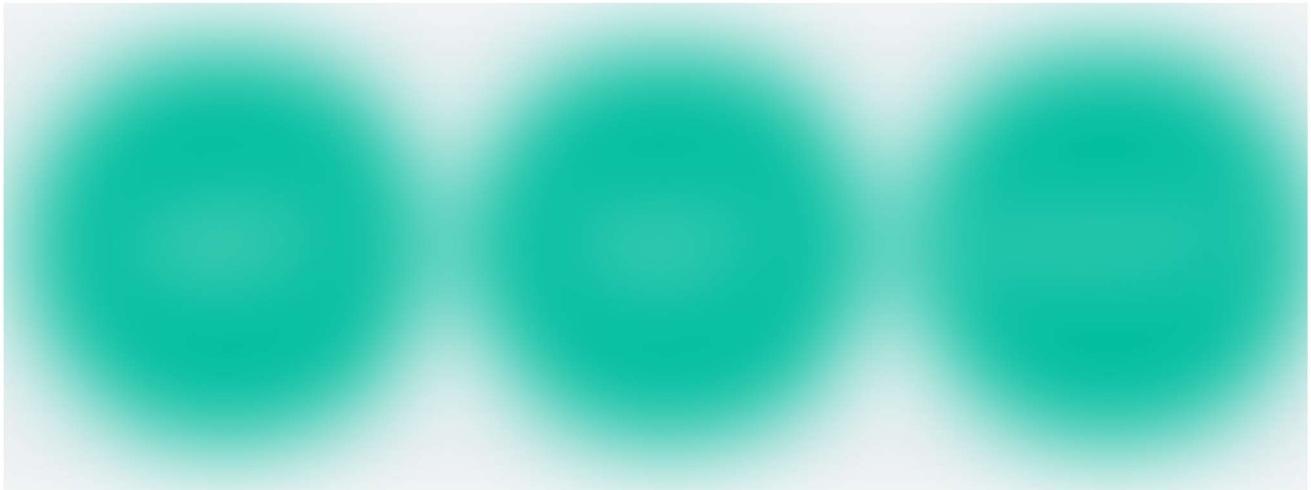
Essencialmente, tudo o que fizemos acima não tem muitas diferenças práticas, isto por conta de algo super importante que não acontecia nos *callbacks*.

Quando falamos de *callbacks* temos que pensar em funções externas. Um callback poderia aceitar uma função única que iria receber como parâmetro um objeto `err` e um `data`, que são respectivamente os erros ocorridos na função assíncrona que o chamou e os dados recebidos no caso de sucesso (muito próximo do nosso `then` e `catch`), porém esta função só iria capturar os erros **daquela** execução, ou seja, para cada *callback* teríamos que ter uma nova função de recuperação e de tratamento de erros ou então teríamos que tratar cada erro em uma função separada.

Com Promises isso não acontece, isto porque, independente do tratamento que damos a Promise, ele sempre vai buscar o primeiro tratamento de erros disponível, em outras palavras, todos os erros irão cair para o primeiro `catch` que encontrarem. Vamos a um exemplo.

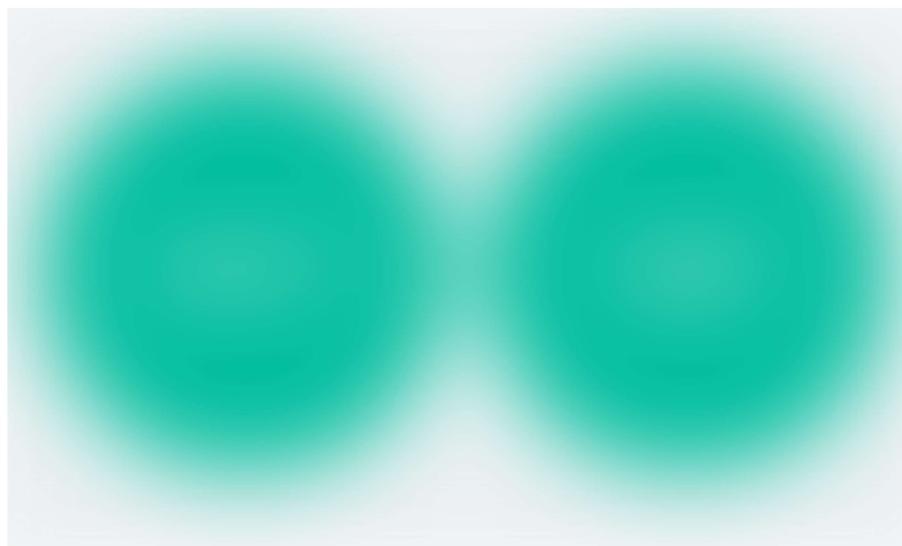
Temos uma função cara ou coroa, ela vai resolver a promise com sucesso se o valor de `Math.random()` — que te dá um número aleatório entre 0 e 1 — for maior que 0.5, caso contrário ela vai rejeitar essa Promise:

Colocamos um `then` e um `catch` simples, se for resolvida, vamos logar a mensagem no `stdout` se não, no `stderr`. Isso nos dá o seguinte mapa para um **sucesso** (quando tiramos um número maior que 0.5):



Nossa função log é executada, mas não a error

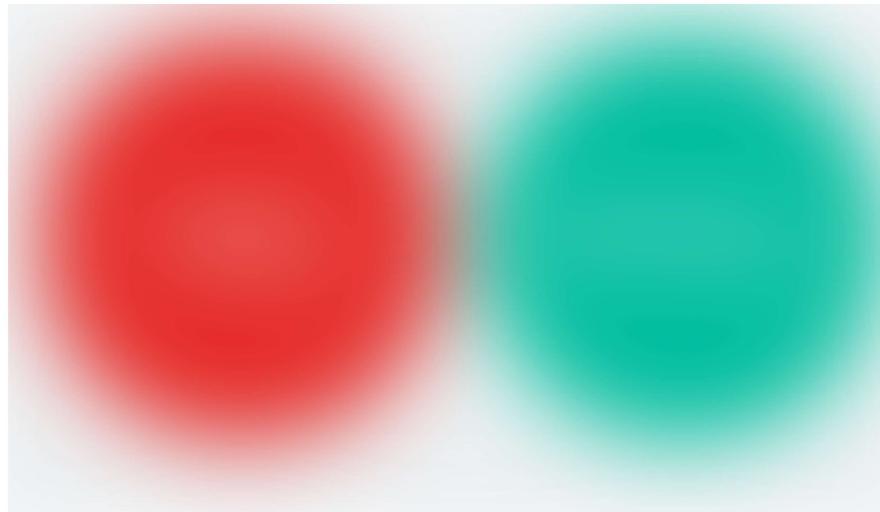
E vamos ter printado no console somente `yay`. Porque o `then` foi chamado já que resolvemos a Promise. Mas no nosso mapa anterior podemos ver que temos um `catch` ligado nele, isto acontece porque ligamos o `catch` no `then`, então ele computou que haveria uma chamada de tratamento de erros, mas como não houve uma rejeição ela não foi executada. Se colocássemos `p.then` e depois `p.catch`, ao invés de `p.then.catch`, teríamos o seguinte mapa de sucesso:



O log está ligado somente à Promise

Veja que agora o `catch` não foi computado porque ele não está ligado no `then`, mas sim no `p` original. Da mesma forma em um erro teríamos somente o `error()` sendo

executado:



O catch está ligado somente à Promise e não mais ao then

Agora o que acontece quando temos uma série de ações que queremos tomar depois?
Por exemplo:

Veja que aqui estamos executando 3 ações após a primeira Promise, a cada ação nós printamos na tela o que estamos fazendo e retornamos o mesmo valor para a próxima Promise — lembre-se que cada `then` retorna outra Promise, então todo o valor retornado dentro de um `then` é como se estivéssemos dando um `resolve(valor)` dentro de uma Promise — e por fim temos um handler de erro que deverá pegar todos os erros da primeira Promise e printar um `no` no console:



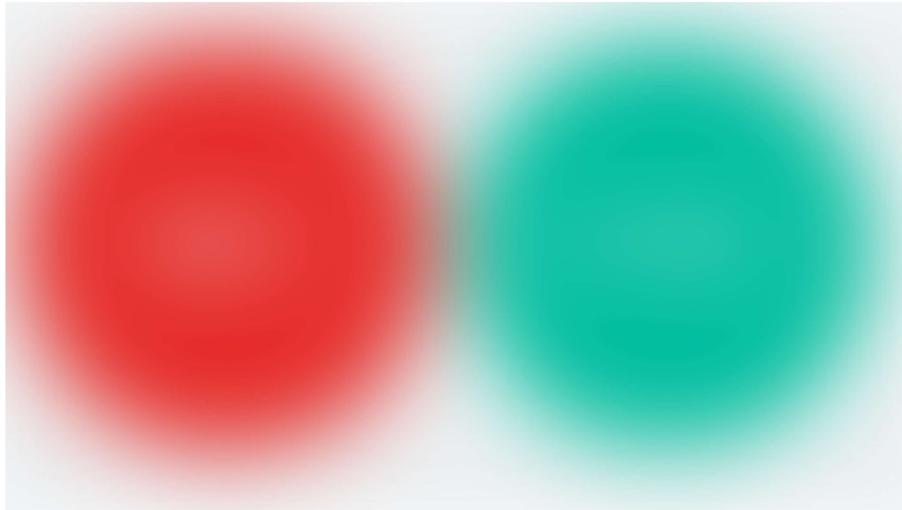
Fluxo de sucesso

Em um fluxo de sucesso vamos ter todas as ações executadas e o nosso `catch` contabilizado mas não executado, nossa saída seria algo assim:

```
yay da ação 1  
yay da ação 2
```

yay da ação 3

E para um fluxo de erro teríamos:



Com simplesmente um `no` no console, ou seja, ele pulou todos os `then`, e caiu diretamente no nosso handler de erro. O que acontece se colocamos um outro `catch` na jogada?

Veja que estamos declarando agora dois handlers de erro. O que deve acontecer é que, quando a Promise for rejeitada, ele deverá chamar o primeiro handler (`erro1`) e parar por ai, certo? Errado:



Cada `catch` só captura a primeira rejeição de uma Promise

O que aconteceu aqui? Nosso `catch erro1` foi executado, mas parece que todo o resto do fluxo seguiu normalmente! Lembre-se que "jogar" um erro é diferente de rejeitar uma Promise. O `throw` irá parar a execução do sistema, mas um `reject` irá manter o sistema sendo executado, por esta razão é possível ter múltiplos `catch` em uma Promise. Cada `catch` irá capturar o erro relativo às Promises anteriores, uma vez

capturado, o valor que ele retornar será passado para a próxima Promise que executará normalmente.

No caso acima vamos ter a seguinte saída no console:

```
Primeiro catch  
Error da ação 2  
Error da ação 3
```

E em um caso de sucesso vamos obter a mesma saída que já obtemos anteriormente, porque não vamos cair em nenhum bloco `catch`. Isso é importante porque muitos pensam que o `catch` é universal, mas na verdade, quando encadeados em outros `then`, o primeiro erro que acontece consome o primeiro `catch` e assim por diante.

Agora, se tivéssemos feito algo deste tipo:

Veja que estamos separando o que é sucesso do que é erro, então o nosso mapa de erros seria algo assim:



E isso significa que iríamos printar ambos os erros no console:

Primeiro catch
no

Perceberam como a ordem do encadeamento importa? E neste caso:

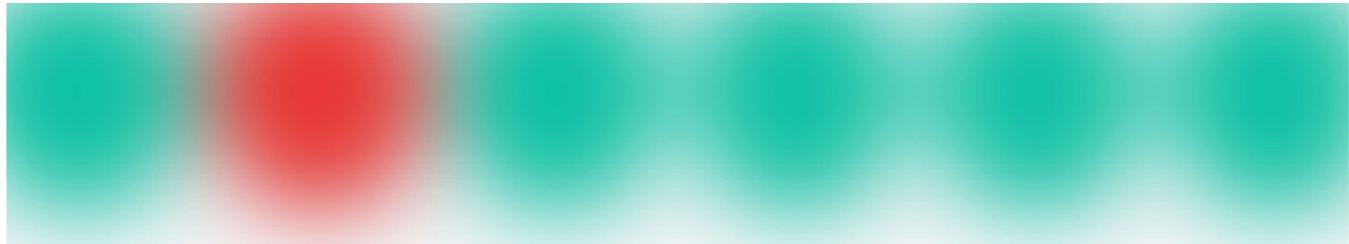
Aqui vamos ter alguns casos específicos, se `p` falha, então a função `erro1` e `erro2` devem ser executadas, mas não `erro3` de acordo com esse mapa:



Isto porque estamos criando um handler de erro acoplado à **Promise original** e outro que está acoplado ao resultado da execução posterior (os `then`). Então nossa saída seria algo assim:

Eu pego todos os erros dessa promise
Eu pego somente os erros que aconteceram até a ação 1
Eu executo normalmente
Eu executo normalmente

Isto acontece pelo mesmo motivo que falamos antes, o primeiro catch é consumido e as demais funções executam normalmente. Agora, se a Promise `p` é resolvida, então sempre vamos ter um erro na `acao1` — isto porque demos um `throw` dentro dela, e isso é o equivalente a dar um `reject` na Promise que este `then` retorna — e ai vamos ter outro mapa:



Veja que `erro1` não é executado de maneira nenhuma, porque a Promise `p` foi resolvida com sucesso, o que deu o erro foi uma de suas execuções posteriores, a `acao1`, e o `catch` com a função `erro1` não está ligado neste `then`. Então teríamos a seguinte saída no console:

```
Estou rejeitando o valor, o catch a seguir deve tratar
Eu pego somente os erros que aconteceram até a ação 1
Eu executo normalmente
Eu executo normalmente
```

Perceba que as demais Promises de `acao2` e `acao3` continuam executando em todos os casos.

Promise.**finally**

O ES9, lançado em 2018, trouxe uma nova funcionalidade às Promises, o `finally`. De acordo com a especificação, este método **sempre** será executado, independente se a Promise foi resolvida ou rejeitada. Isto foi criado para manter a ideia do `try/catch/finally` que já existe há décadas em outras linguagens e pode ser muito útil em diversos casos.

Em um bloco padrão de `try/catch/finally` temos a seguinte estrutura:

```
try {
  // código executado
} catch (erro) {
  // irá cair aqui se o código executado jogar um erro
} finally {
  // essa parte sempre vai ser executada
}
```

O mesmo funciona para as Promises. Vamos a um exemplo:

No caso de um sucesso, vamos ter a seguinte saída do console:

```
yay  
Eu sempre sou executado
```

Já no caso de um erro:

```
no  
Eu sempre sou executado
```

Ou seja, é como se tivéssemos sempre alguém ouvindo a finalização de nossas Promises para **sempre** executar um trecho de código. O método `finally` já está disponível desde a versão 10.3 do NodeJS e na maioria dos browsers.

Settled

Um estado importante de comentarmos aqui é o estado *Settled* de uma Promise. Como já falamos antes, este estado é quando temos uma Promise completamente resolvida, que já recebeu seus valores de *resolved* ou *reject*, ou seja, é uma Promise que já "acabou".

Uma Promise neste estado já teve seus handlers `then` e/ou `catch` executados. A partir deste ponto dizemos que ela está terminada, agora, se no futuro adicionarmos outro handler, digamos, outro `then`, na mesma Promise, o que acontece?

Vamos analisar o fluxo desta Promise:

1. A Promise é criada
2. O handler `then` é adicionado
3. Após 2s a Promise recebe a resposta do `resolve`
4. A Promise executa o handler e é dada como *settled*
5. Um novo handler é adicionado

Promises que já estão com o estado definido como *settled* são resolvidas imediatamente após a adição posterior de um novo handler, ou seja, nosso handler tardio de multiplicação vai retornar **na hora** o valor 2000:



Veja a segunda execução retornando no mesmo momento

Promises de Promises

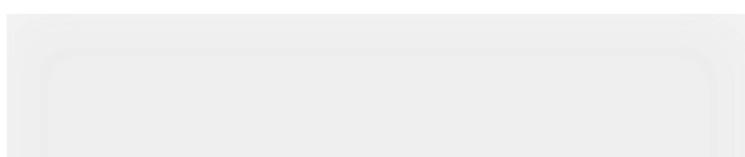
É possível para que uma Promise retorne outra Promise para ser resolvida, por exemplo, vamos imaginar que temos que pegar duas informações diferentes de APIs diferentes, mas uma depende da outra.

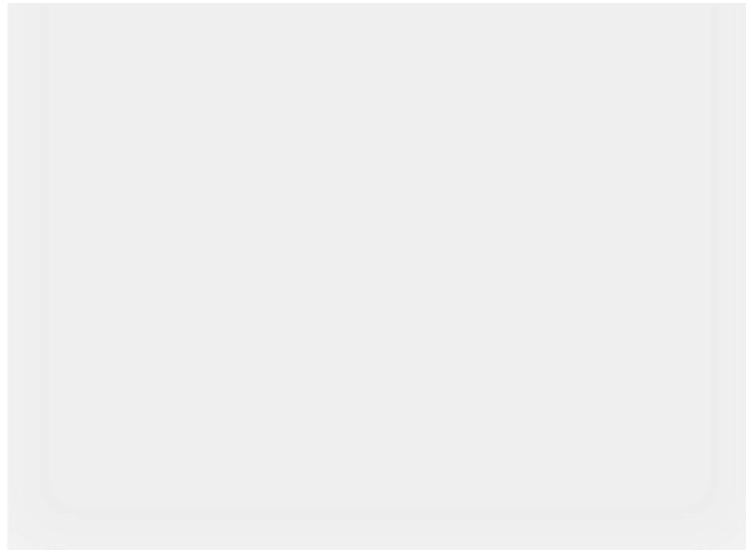
Quando retornamos uma Promise para outra Promise, só vamos ter a resolução completa do conjunto quando ambas as Promises entrarem no estado *settled*. Ou seja, se, dentro de uma Promise, chamamos outra Promise, a primeira só se resolverá após o retorno da segunda como *settled*:

O que acontece aqui é a Promise inteira só será resolvida após a execução do primeiro `fetch` e também do segundo `fetch`, que retorna uma Promise resolvida no último `then`. Vamos a outro exemplo mais simples.

Uma Promise que retorna outra Promise que pode ou não ser resolvida após 1s, usando nossa mesma função de moeda:

Veja como fica esta execução em tempo real:





Bloqueio de Promises

Veja que o primeiro `then` fica amarelo, porque ele está esperando a segunda Promise (a que tem o `setTimeout`) ser resolvida, isto significa que ele está **bloqueado**, na espera da segunda Promise. Quando a mesma retorna, todos os outros handlers são resolvidos instantaneamente.

Métodos de Promises

Além do `then`, `catch` e `finally` uma Promise também possui outros métodos estáticos muito úteis.

Promise.resolve e Promise.reject

Estes dois métodos são atalhos para quando queremos retornar uma Promise que sempre terá o mesmo valor, ou sempre resolvida, ou sempre rejeitada, de forma que não precisamos ficar criando todo o boilerplate de `new Promise...`.

Vamos imaginar que temos a seguinte Promise:

```
const p = new Promise((resolve) => resolve(1056))
```

Independente do que acontecer, a Promise **sempre** resolverá para o valor 1056. Ela nunca terá um `catch` e nunca lançará um erro... Então podemos simplesmente escrever assim:

```
const p = Promise.resolve(1056)
```

De forma similar podemos fazer com o `reject` :

```
const p = Promise.reject('Erro')
```

Promise.all

A ideia do método `all` é executar ações simultaneamente, ou seja, disparar uma série de Promises ao mesmo tempo e esperar pelo retorno de todas elas. Isto é muito útil quando, por exemplo, temos que pegar informações de várias APIs que não são relacionadas entre si.

Um exemplo mais real seria: "Buscar todos os dados de redes sociais de um usuário", podemos pegar todos os usuários desse cliente nas redes sociais e disparar um monte de Promises, uma para cada rede social e esperar a resposta.

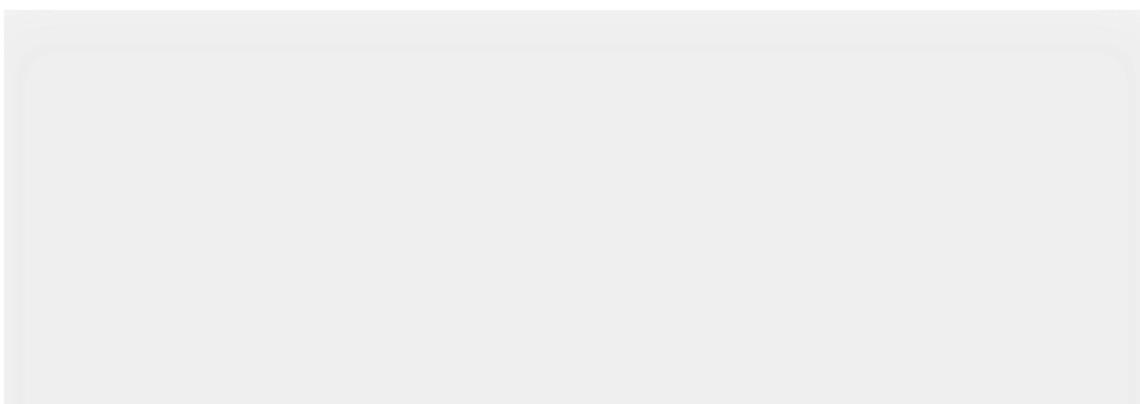
O método `Promise.all` é justamente isso. Ele recebe um Array de Promises não resolvidas e inicia todas elas. Ele só vai terminar em dois casos:

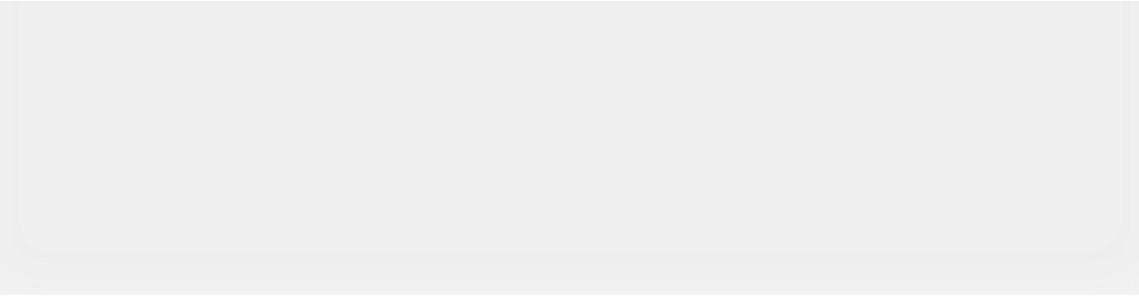
1. Todas as Promises do array foram resolvidas
2. Pelo menos uma Promise foi rejeitada

Ou seja, é um método tudo ou nada, se todas as Promises tiverem sucesso, o método terá sucesso, porém no primeiro erro, o método te devolverá um erro.

Vamos ver este snippet de código (também presente no Promisees):

Criamos um array de várias Promises, cada uma delas resolve em um momento diferente, porém nenhuma delas tem uma propriedade `b` ou `c` então elas serão rejeitadas naturalmente, veja a animação:



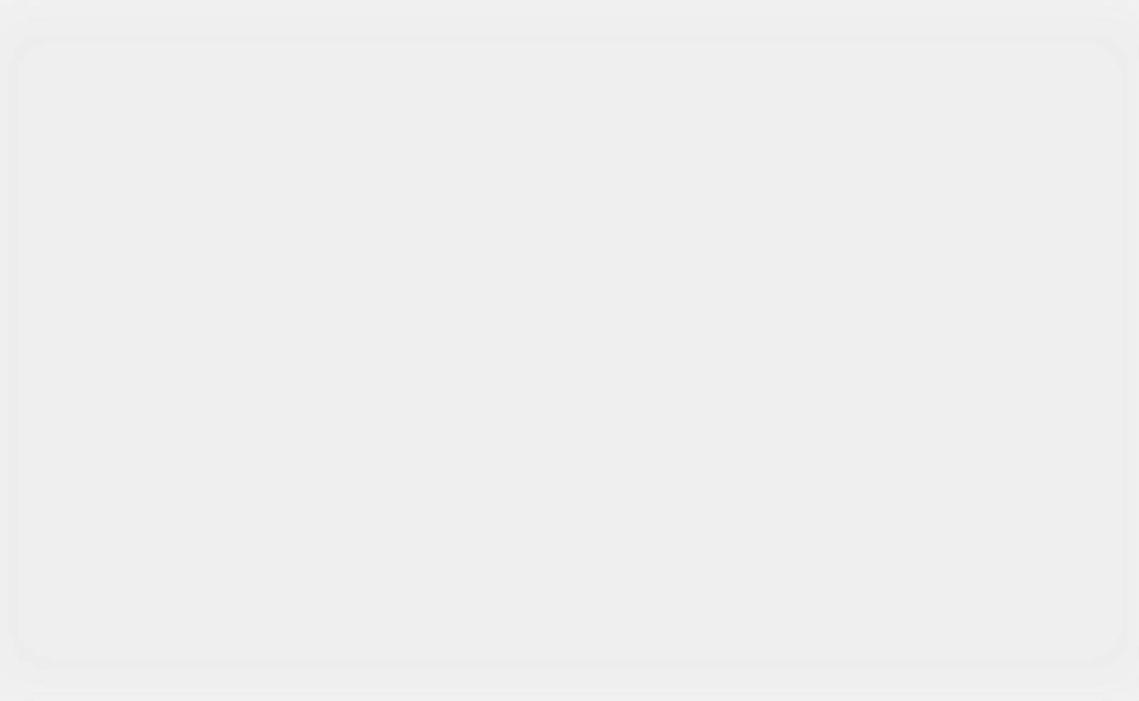


Ao ser rejeitado, todo o método é finalizado

Perceba que temos 3 `new()` ligados a um `[all]`, eles se resolvem em momentos diferentes, uma vez que **todos** são resolvidos, o método `then` é chamado, mas ele retorna um erro que rejeita a sequencia de Promises, nesse instante todo o método é finalizado e o array de Promises é dado como *settled*. Retornando o resultado do erro.

Vamos modificar o código para que elas passem:

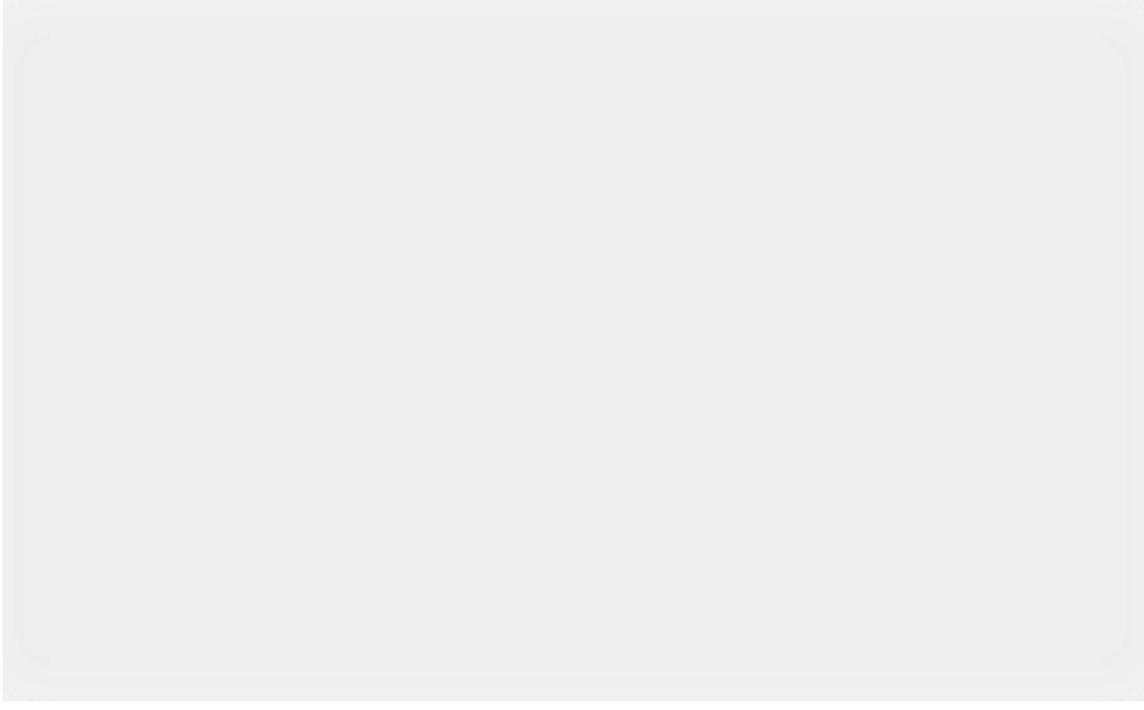
Veja como o mapa fica agora:



Agora é bem mais claro de ver que o `[all]` espera todas as Promises se resolverem antes de chamar seu handler, no caso de sucesso, o `Promise.all` retorna um array com todos os resultados das Promises enviadas.

Vamos ver o que acontece no caso de uma dessas Promises ser rejeitada:

Perceba como podemos ver exatamente o funcionamento do `Promise.all`:

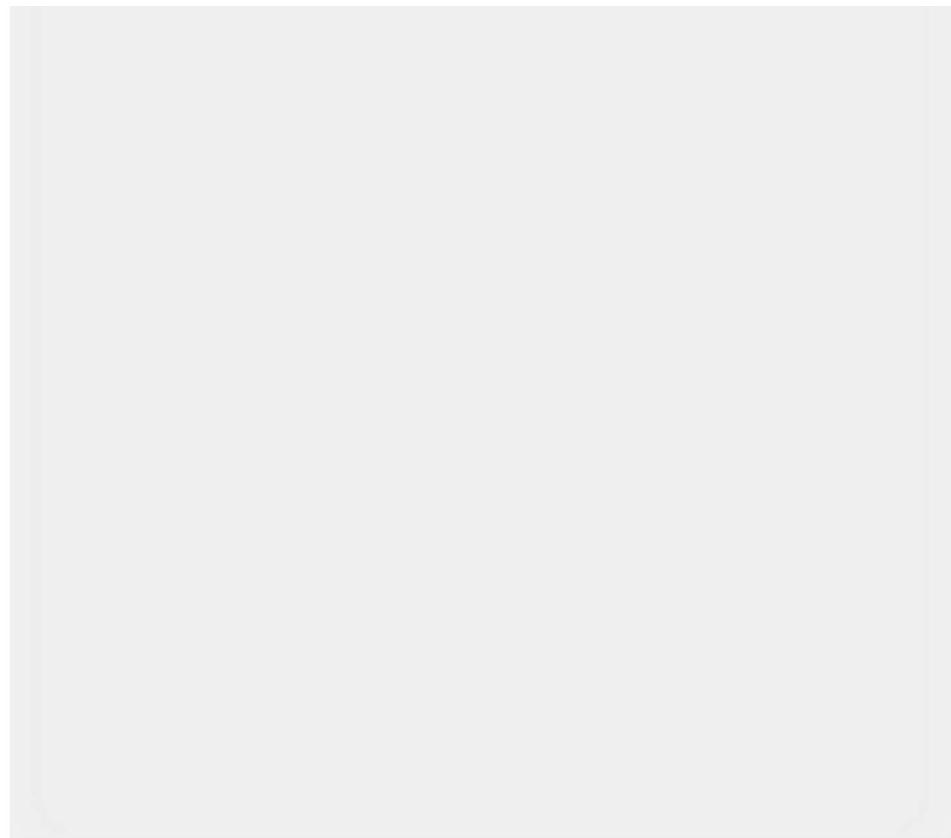


Quando a segunda Promise é rejeitada, todos os handlers são chamados imediatamente, ou seja, o método retorna o valor do erro da segunda Promise para o `catch` e ignora completamente a última Promise, ela ainda é executada, mas não tem seu valor consumido por ninguém.

Promise.race

O método `Promise.race` faz exatamente o que o nome diz, ele recebe um array de Promises, inicia todas elas, a que retornar Primeiro vai ser o retorno do método por completo. Ele é um caso especial do `Promise.all` onde, ao invés de esperar **todas** as Promises serem resolvidas, simplesmente retorna o primeiro resultado que obtiver. Veja este exemplo:

Temos dois arrays de Promises, um deles é resolvido em 4s e depois rejeitado em 8s, enquanto o outro é rejeitado em 2s e depois resolvido em 6s e 10s, vamos ver o mapa:



Perceba que, no primeiro caso, o método `[race]` aguarda duas Promises, enquanto o segundo aguarda 3. O primeiro tem sua Promise resolvida e ele já se torna verde, porque é o resultado que ele espera, então a segunda Promise (que é rejeitada) não é sequer consumida.

No segundo race (o que fica vermelho no final), temos uma Promise que é rejeitada logo de cara, então todas as demais Promises são ignoradas e o handler `catch` é chamado.

Async/Await

Não irei me estender demais sobre `async` e `await` por aqui porque já escrevi um artigo que fala sobre ele com mais detalhes. O que vou dizer aqui é somente uma desmistificação do que a maioria pensa sobre estas keywords.

Async e await são keywords que foram introduzidas no ES8 em 2017. Basicamente é um *syntax sugar* (uma firula de linguagem que foi adicionada somente para poder facilitar a escrita) do `then` e `catch`.

O motivo pela adição do `async/await` foi o mesmo da adição das Promises no JavaScript, o callback hell. Só que dessa vez tínhamos o Promise hell, onde ficávamos

aninhando Promises dentro de Promises eternamente e isso tornava tudo muito mais difícil de se ler.

A proposta de funções assíncronas é justamente nivelar todo mundo em um único nível. Escrever um código assim:

```
async function foo () {  
  if (Math.random() > 0.5) return 'yeah'  
  throw new Error('ops')  
}
```

É a mesma coisa que escrever isso:

```
const foo = new Promise((resolve, reject) => {  
  if (Math.random() > 0.5) return resolve('yeah')  
  reject('ops')  
})
```

A diferença é que podemos deixar tudo no mesmo nível, ao invés de escrevermos:

```
foo.then((resposta) => { ... }).catch((erro) => ...)
```

Podemos fazer isto (desde que estejamos dentro de outra função async):

```
async function bar () {  
  try {  
    const resposta = await foo()  
  } catch (erro) { throw erro }
```

Futuras implementações

O JavaScript é um padrão que está em constante mudança. Portanto já existem novas ideias e implementações para novos métodos de Promises, o mais legal é o `allSettled`.

Promise.allSettled

Este método veio para sanar um grande problema com o `Promise.all`. Em muitos casos reais, nós queremos executar várias Promises de forma paralela e trazer o resultado de **todas** elas, e não só o erro ou então só o array de sucessos, nós queremos tanto os erros, quanto os sucessos.

Esta proposta está em estágio 3 no comitê e tem grandes chances de entrar para o ESNext nas futuras edições.

Vejamos o exemplo — que está também na documentação — sobre o motivo desta proposta:

Este é um problema comum com o `Promise.all`, quando queremos pegar o resultado de todas as Promises, temos que fazer uma função de reflexão, que nada mais faz do que atribuir um handler para cada uma das Promises no array e jogar isso tudo dentro do `all`. Desta forma estamos sobrescrevendo o comportamento original da Promise pelo nosso próprio e retornando para cada valor um objeto com as descrições do que aconteceu.

A proposta pretende criar um método `allSettled` para abstrair a função `reflect`:

Ambos os casos irão nos dar um array de objetos no final com esta assinatura:

```
[  
  { status: 'resolved', value: 'valor da resolução' },  
  { status: 'rejected', reason: 'mensagem de erro' }  
]
```

Para mais informações veja a página da proposta.

Conclusão

O Objetivo da escrita deste artigo não foi somente para entrar mais a fundo em Promises como um todo, mas sim devido a uma grande dificuldade que notei em vários programadores (até mesmo experientes, incluindo eu mesmo) com o fluxo assíncrono do JavaScript.

Espero que, com este artigo, possamos entender de uma vez por todas o que são Promises e o que elas significam e qual é a importância desta adição para a linguagem

e porque todos deveriam saber Promises ao invés de callbacks.

Veja a segunda parte deste artigo aqui!

Se você viu algum erro, tem alguma sugestão ou quer adicionar algo no artigo, fala comigo através de qualquer uma das minhas redes sociais em <http://lsantos.dev> :D

Edição 09/05/2019

Para complementar este artigo, fui chamado pela Digital Innovation One para fazer um webinar sobre Promises, onde me baseei no que aprendemos aqui e mostrei de forma prática! Recomendo muito para acompanhar e acrescentar ao estudo:

Referências

- <https://github.com/tc39/proposal-promise-allSettled>
- <https://braziljs.org/blog/promises-no-javascript/>
- https://en.wikipedia.org/wiki/Futures_and_promises
- https://developer.mozilla.org/pt-BR/docs/Web/JavaScript/Reference/Global_Objects/Promise
- <https://developer.mozilla.org/en-US/docs/Web/JavaScript/EventLoop#Run-to-completion>

- https://developer.mozilla.org/pt-BR/docs/Web/JavaScript/Guide/Usando_promises
- <https://ponyfoo.com/articles/es6-promises-in-depth>
- https://nodejs.org/dist/latest-v8.x/docs/api/util.html#util_util_promisify_original
- <https://medium.freecodecamp.org/es9-javascripts-state-of-art-in-2018-9a350643f29c>

JavaScript Programming Development Nodejs Technology

About Help Legal

Get the Medium app

