Usando promises

Uma `Promise` é um objeto que representa a eventual conclusão ou falha de uma operação assíncrona. Como a maioria das pessoas consomem promises já criadas, este guia explicará o consumo de promises devolvidas antes de explicar como criá-las.

Essencialmente, uma promise é um objeto retornado para o qual você adiciona callbacks, em vez de passar callbacks para uma função.

Por exemplo, em vez de uma função old-style que espera dois callbacks, e chama um deles em uma eventual conclusão ou falha:

```
function successCallback(result) {
  console.log("It succeeded with " + result);
}

function failureCallback(error) {
  console.log("It failed with " + error);
}

doSomething(successCallback, failureCallback);
```

…funções modernas retornam uma promise e então você pode adicionar seus callbacks:

```
const promise = doSomething();
promise.then(successCallback, failureCallback);
```

…ou simplesmente:

```
doSomething().then(successCallback, failureCallback);
```

Nós chamamos isso de *chamada de função assíncrona*. Essa convenção tem várias vantagens. Vamos explorar cada uma delas.

Garantias

Ao contrário dos callbacks com retornos de funções old-style, uma promise vem com algumas garantias:

- Callbacks nunca serão chamados antes da conclusão da execução atual do loop de eventos do JavaScript.
- Callbacks adicionadas com .then mesmo *depois* do sucesso ou falha da operação assíncrona, serão chamadas, como acima.
- Multiplos callbacks podem ser adicionados chamando-se .then várias vezes, para serem executados independentemente da ordem de inserção.

Mas o benefício mais imediato das promises é o encadeamento.

Encadeamento

Uma necessidade comum é executar duas ou mais operações assíncronas consecutivas, onde cada operação subsequente começa quando a operação anterior é bem sucedida, com o resultado do passo anterior. Nós conseguimos isso criando uma *cadeia de promises*.

Aqui está a mágica: a função `then` retorna uma nova promise, diferente da original:

```
const promise = doSomething();
const promise2 = promise.then(successCallback, failureCallback);
```

ou

```
const promise2 = doSomething().then(successCallback, failureCallback);
```

Essa segunda promise representa a conclusão não apenas de `doSomething()`, mas também do `successCallback` ou `failureCallback` que você passou, que podem ser outras funções assíncronas que retornam uma promise. Quando esse for o caso, quaisquer callbacks adicionados a `promise2` serão enfileiradas atrás da promise retornada por `successCallback` ou `failureCallback`.

Basicamente, cada promise representa a completude de outro passo assíncrono na cadeia.

Antigamente, realizar operações assíncronas comuns em uma linha levaria à clássica pirâmide da desgraça:

```
doSomething(function(result) {
  doSomethingElse(result, function(newResult) {
    doThirdThing(newResult, function(finalResult) {
      console.log('Got the final result: ' + finalResult);
    }, failureCallback);
  }, failureCallback);
}, failureCallback);
```

Ao invés disso, com funções modernas, nós atribuímos nossas callbacks às promises retornadas, formando uma *cadeia de promise*:

```
doSomething().then(function(result) {
  return doSomethingElse(result);
})
.then(function(newResult) {
  return doThirdThing(newResult);
})
.then(function(finalResult) {
  console.log('Got the final result: ' + finalResult);
})
.catch(failureCallback);
```

Os argumentos para `then` são opcionais, e `catch(failureCallback)` é uma abreviação para `then(null, failureCallback)`. Você pode também pode ver isso escrito com arrow functions:

```
doSomething()
.then(result => doSomethingElse(result))
.then(newResult => doThirdThing(newResult))
.then(finalResult => {
  console.log(`Got the final result: ${finalResult}`);
})
.catch(failureCallback);
```

Importante: Sempre retorne um resultado, de outra forma as callbacks não vão capturar o resultado da promise anterior.

Encadeando depois de um catch

É possivel encadear *depois* de uma falha, i.e um `catch`. Isso é muito útil para realizar novas ações mesmo depois de uma falha no encadeamento. Leia o seguinte exemplo:

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

```
    console.log('Initial');

    resolve();
})
.then(() => {
    throw new Error('Something failed');

    console.log('Do this');
})
.catch(() => {
    console.log('Do that');
})
.then(() => {
    console.log('Do this whatever happened before');
});
```

Isso vai produzir o seguinte texto:

```
Initial
Do that
Do this whatever happened before
```

Observe que o texto "Do this" não foi impresso por conta que o erro "Something failed" causou uma rejeição.

Propagação de erros

Na pirâmide da desgraça vista anteriormente, você pode se lembrar de ter visto `failureCallback` três vezes, em comparação a uma única vez no fim da corrente de promises:

```
doSomething()
.then(result => doSomethingElse(result))
.then(newResult => doThirdThing(newResult))
.then(finalResult => console.log(`Got the final result: ${finalResult}`))
.catch(failureCallback);
```

Basicamente, uma corrente de promises para se houver uma exceção, procurando por catch handlers no lugar. Essa modelagem de código segue bastante a maneira de como o código síncrono funciona:

```
try {
  const result = syncDoSomething();
  const newResult = syncDoSomethingElse(result);
```

```
  const finalResult = syncDoThirdThing(newResult);
  console.log(`Got the final result: ${finalResult}`);
} catch(error) {
  failureCallback(error);
}
```

Essa simetria com código assíncrono resulta no *syntactic sugar* `async` / `await` presente no ECMAScript 2017:

```
async function foo() {
  try {
    const result = await doSomething();
    const newResult = await doSomethingElse(result);
    const finalResult = await doThirdThing(newResult);
    console.log(`Got the final result: ${finalResult}`);
  } catch(error) {
    failureCallback(error);
  }
}
```

É construído sobre promises, por exemplo, `doSomething()` é a mesma função que antes. Leia mais sobre a sintaxe aqui .

Por pegar todos os erros, até mesmo exceções jogadas(*thrown exceptions*) e erros de programação, as promises acabam por solucionar uma falha fundamental presente na pirâmide da desgraça dos callbacks. Essa característica é essencial para a composição funcional das operações assíncronas.

Criando uma Promise em torno de uma callback API antiga

Uma `Promise` pode ser criada do zero utilizando o seu construtor. Isto deve ser necessário apenas para o envolvimento de APIs antigas.

Em um mundo ideal, todas as funções assíncronas já retornariam promises. Infelizmente, algumas APIs ainda esperam que os retornos de sucesso e/ou falha sejam passados da maneira antiga. O exemplo por excelência é o `setTimeout()` (en-US) function:

```
setTimeout(() => saySomething("10 seconds passed"), 10000);
```

Misturar chamadas de retorno e promises de *old-style* é problemático. Se `saySomething` falhar ou contiver um erro de programação, nada o captura.

Por sorte nós podemos envolvê-la em uma promise. É uma boa prática envolver funções problemáticas no menor nivel possível, e nunca chamá-las diretamente de novo:

```
const wait = ms => new Promise(resolve => setTimeout(resolve, ms));

wait(10000).then(() => saySomething("10 seconds")).catch(failureCallback);
```

Basicamente, um construtor de promises pega uma função executora que nos deixa resolver ou rejeitar uma promise manualmente. Desde que `setTimeout` não falhe, nós deixamos a rejeição de fora neste caso.

Composição

`Promise.resolve()` e `Promise.reject()` são atalhos para se criar manualmente uma promise que já foi resolvida ou rejeitada, respectivamente. Isso pode ser útil em algumas situações.

`Promise.all()` e `Promise.race()` são duas ferramentas de composição para se executar operações assíncronas em paralelo.

Uma composição sequencial é possível usando JavaScript de uma forma esperta:

```
[func1, func2].reduce((p, f) => p.then(f), Promise.resolve());
```

Basicamente reduzimos um vetor de funções assíncronas a uma cadeia de promises equivalentes a: `Promise.resolve().then(func1).then(func2);`

Isso também pode ser feito com uma função de composição reutilizável, que é comum em programação funcional:

```
const applyAsync = (acc,val) => acc.then(val);
const composeAsync = (...funcs) => x => funcs.reduce(applyAsync, Promis
```

A função composeAsync aceitará qualquer número de funções como argumentos e retornará uma nova função que aceita um valor inicial a ser passado pelo pipeline de composição. Isso é benéfico porque alguma, ou todas as funções, podem ser assíncronas ou síncronas, e é garantido de que serão executadas na ordem correta.

```
const transformData = composeAsync(func1, asyncFunc1, asyncFunc2, func2
```

```
transformData(data);
```

No ECMAScript 2017, uma composição sequencial pode ser feita de forma mais simples com async/await:

```
for (const f of [func1, func2]) {
  await f();

}
```

Cronometragem

Para evitar surpresas, funções passadas para `then` nunca serão chamadas sincronamente, mesmo com uma função já resolvida:

```
Promise.resolve().then(() => console.log(2));
console.log(1); // 1, 2
```

Ao invés de rodar imediatamente, a função passada é colocada em uma micro tarefa, o que significa que ela roda depois que a fila estiver vazia no final do atual processo de evento de loop do Javascript, ou seja: muito em breve:

```
const wait = ms => new Promise(resolve => setTimeout(resolve, ms));

wait().then(() => console.log(4));
Promise.resolve().then(() => console.log(2)).then(() => console.log(3));
console.log(1); // 1, 2, 3, 4
```

Ver também

- `Promise.then()`
- Promises/A+ specification
- Venkatraman.R - JS Promise (Part 1, Basics)
- Venkatraman.R - JS Promise (Part 2 - Using Q.js, When.js and RSVP.js)
- Venkatraman.R - Tools for Promises Unit Testing
- Nolan Lawson: We have a problem with promises — Common mistakes with promises

Last modified: 6 de set. de 2021, by MDN contributors