Aula 9 - Execução Assíncrona de Código

aTip Learn - Lógica de Programação com JavaScript

Resumo da Aula

Nesta aula, exploraremos comunicação assíncrona em JavaScript com fetch para APIs externas. Aprenderemos a usar callbacks e promises para gerenciar fluxos assíncronos, e async/await para simplificar a escrita de código. Abordaremos também técnicas avançadas com promises, como execução concorrente de chamadas a APIs para melhorar a eficiência do código.

Objetivos

- Entender a execução assíncrona de código por meio de callbacks.
- Aprender a usar promises para gerenciar fluxos assíncronos.
- Utilizar async/await para simplificar a escrita de código assíncrono.
- Realizar chamadas a APIs externas com fetch.

Conteúdo

Funções de Callback

Uma função de callback é um tipo de função que geralmente passamos como parâmetro para outras funções. O objetivo da função de callback é realizar uma ação depois que a função principal terminar de executar, ou quando algum evento ocorrer.

Nós já vimos alguns exemplos de funções de callback em nossa aula sobre páginas Web interativas. Por exemplo, quando usamos `addEventListener` para adicionar um evento a um elemento HTML, passamos uma função de callback que será executada quando o evento ocorrer.

```
const botao = document.getElementById('botao');

function callbackClique() {
  alert('O botão foi clicado!');
}

botao.addEventListener('click', callbackClique);
```

Nesse exemplo, a função `callbackClique` é uma função de callback que será executada quando o botão for clicado. Perceba que quando passamos a função de callback para `addEventListener`, não usamos os parênteses `()` após o nome da função. Isso ocorre porque queremos passar a referência da função, e não executá-la imediatamente. Ou seja, se utilizássemos os parênteses, a função seria executada no momento em que passamos ela como parâmetro, e não quando o evento ocorrer.

Podemos ainda passar uma função anônima como callback, sem a necessidade de declarar uma função separada:

```
const botao = document.getElementById('botao');

botao.addEventListener('click', function() {
  alert('O botão foi clicado!');
});
```

O que são operações assíncronas?

Operações assíncronas são operações que não são executadas imediatamente, e que podem levar algum tempo para serem concluídas. Um exemplo comum de operação assíncrona é a leitura de um arquivo em disco, ou o envio de uma requisição para um servidor via rede.

Para evitar que o programa fique bloqueado aguardando a execução de uma operação assíncrona, o JavaScript utiliza callbacks, promises e async/await para permitir que o programa continue sendo executado enquanto a operação assíncrona é processada.

O exemplo abaixo simula uma operação assíncrona utilizando a função `setTimeout` (https://developer.mozilla.org/en-US/docs/Web/API/setTimeout), que aguarda um determinado tempo antes de executar a função de callback:

```
console.log('Início da execução');

setTimeout(function() {
  console.log('Função de callback executada após 2 segundos');
}, 2000);

console.log('Fim da execução');
```

No exemplo acima, a função `setTimeout` aguarda 2 segundos antes de executar a função de callback. Enquanto isso, o programa continua executando as instruções seguintes, sem esperar a função de callback ser executada. Dessa forma as mensagens de inicío e fim da execução são mostradas imediatamente, e a mensagem da função de callback é exibida após 2 segundos.

Callback Hell

Quando temos diversas funções assíncronas que dependem umas das outras, pode ocorrer o que chamamos de "Callback Hell", ou "Inferno dos Callbacks". Isso acontece quando temos muitas funções de callback aninhadas, tornando o código difícil de ler e manter.

Imagine que estamos desenvolvendo um código que lê um arquivo, envia o conteúdo desse arquivo para uma API, salva o resultado da API em um outro arquivo e por fim envia uma requisição para outra API com o caminho do arquivo salvo. O código poderia ficar assim:

```
lerArquivo('arquivo.txt', function(conteudo) {
  enviarParaAPI(conteudo, function(resultado) {
    salvarArquivo('resultado.txt', resultado, function() {
      enviarParaOutraAPI('resultado.txt', function(resposta) {
        console.log(resposta);
      });
    });
  });
});
```

Nesse exemplo temos quatro operações assíncronas que dependem uma das outras, o que faz com que o código tenha diversos níveis de aninhamento, se adicionarmos a esse código qualquer lógica extra para lidar com erros ou exceções, o código rapidamente se tornaria muito difícil de ler e manter.

Promises

Para evitar o "Callback Hell", a linguagem JavaScript introduziu as Promises, que são objetos que representam o resultado de uma operação assíncrona. Uma promise pode estar em três estados distintos:

- pendente: quando a operação assíncrona ainda não foi concluída.
- resolvida: quando a operação assíncrona foi concluída com sucesso.
- rejeitada: quando a operação assíncrona falhou.

Uma Promise pendente pode se tornar resolvida ou rejeitada, e uma vez que uma Promise é resolvida ou rejeitada, ela não pode mais mudar de estado. Uma Promise resolvida contém o valor de retorno da operação assíncrona, enquanto uma Promise rejeitada contém o motivo do erro.

Para acessarmos o valor de retorno de uma Promise, utilizamos o método `then`, que recebe uma função de callback que será executada quando a Promise for resolvida. Se a Promise for rejeitada, podemos utilizar o método `catch` para tratar o erro.

Imagine que temos uma função `lerArquivo` que retorna uma Promise que será resolvida com o conteúdo do arquivo, podemos utilizá-la da seguinte forma:

```
lerArquivo('arquivo.txt')
  .then(function(conteudo) {
    console.log('Conteúdo do arquivo:', conteudo);
  })
  .catch(function(erro) {
    console.error('Erro ao ler o arquivo:', erro);
  });
```

A função `lerArquivo` retorna uma Promise que será resolvida com o conteúdo do arquivo, e então utilizamos o método `then` para acessar o conteúdo do arquivo quando a Promise for resolvida, e o método `catch` para tratar qualquer erro que ocorrer durante a leitura do arquivo.

O método `then` pode retornar um valor ou uma outra `Promise`, que será encadeada com a Promise original. Isso nos permite encadear diversas operações assíncronas de forma mais legível e organizada. O nosso exemplo do "Callback Hell" poderia ser reescrito utilizando Promises da seguinte forma:

```
lerArquivo('arquivo.txt')
  .then(function(conteudo) {
    return enviarParaAPI(conteudo);
  })
  .then(function(resultado) {
    return salvarArquivo('resultado.txt', resultado);
  })
  .then(function() {
    return enviarParaOutraAPI('resultado.txt');
  })
  .then(function(resposta) {
    console.log(resposta);
  })
  .catch(function(erro) {
    console.error('Erro:', erro);
  });
```

Isso evita que precisemos aninhar diversas funções de callback, tornando o código mais legível e fácil de manter. Além disso, podemos utilizar um único método `catch` após os métodos `then` para tratar qualquer erro que ocorrer durante a execução das Promises.

Entretanto, é importante ter em mente que o uso de Promises não elimina a necessidade de funções de callback, uma vez que as Promises são construídas utilizando callbacks que são passados para os métodos `then` e `catch`. Pode parecer razoável utilizar Promises da seguinte forma:

```
let resposta = lerArquivo('arquivo.txt')
  .then(function(conteudo) {
    return enviarParaAPI(conteudo);
  })
  .then(function(resultado) {
    return salvarArquivo('resultado.txt', resultado);
  })
  .then(function() {
    return enviarParaOutraAPI('resultado.txt');
  });
```

O exemplo acima não está correto, pois a variável `resposta` irá conter o valor da resposta retornada pela chamada para a API, mas sim um objeto do tipo Promise que representa a operação assíncrona.

Criação de Promises

Em alguns momentos pode nos ser útil criar nossas próprias Promises, isso em geral acontece quando lidamos com algum código assíncrono que utiliza callbacks, e desejamos transformá-lo em uma Promise para melhorar a legibilidade e a manutenção do código.

Criar uma Promise é bastante simples, uma Promise nada mais é que um objeto criado por meio do construtor `new Promise(...)`. Esse construturo recebe como argumento uma função com os parâmetros `resolve` e `reject`, que são funções que devem ser chamadas para resolver ou rejeitar a Promise, respectivamente.

Por exemplo, se desejarmos criar uma Promise que aguarda 2 segundos e então retorna a data e hora atual, podemos fazer da seguinte forma:

```
function obterDataHoraEm2Segundos() {
  const promise = new Promise(function (resolve, reject) {
    setTimeout(function () {
      resolve(new Date());
    }, 2000);
  });

  return promise;
}

obterDataHoraEm2Segundos().then(function (dataHora) {
  console.log("Data e hora atual:", dataHora);
});
```

Nesse exemplo definimos uma função chamada `obterDataHoraEm2Segundos` que cria uma nova Promise, passarmos como argumento para essa Promise uma função que aguarda 2 segundos e então chama a função `resolve` que foi passada como argumento para a função, passando a data e hora atual para `resolve`. A chamada para `resolve` faz com que a Promise mude do estado pendente para resolvida, fazendo com que o método `then` seja chamado com o valor passado para `resolve`.

Caso desejássemos rejeitar a Promise, poderíamos chamar a função `reject` passando o motivo do erro:

```
function obterDataHoraEm2Segundos() {
  const promise = new Promise(function (resolve, reject) {
    setTimeout(function () {
      reject("Promise rejeitada!");
    }, 2000);
  });

  return promise;
}

obterDataHoraEm2Segundos()
  .then(function (dataHora) {
    console.log("Data e hora atual:", dataHora);
  })
  .catch(function (erro) {
    console.error("Erro ao obter a data e hora:", erro);
  });
```

Async/Await

As Promises são uma forma ferramenta que permite escrever código mais simples e mais legível para lidar com operações assíncronas, entretanto, ainda assim, as Promises fogem do fluxo de execução normal do código, executando o código passado para os métodos `then` e `catch` por meio de callbacks.

Para permitir que o código assíncrono seja escrito de forma mais parecida com o código síncrono, o JavaScript introduziu as palavras-chave `async` e `await`. A palavra-chave `async` é utilizada para declarar uma função assíncrona, enquanto a palavra-chave `await` é utilizada para aguardar a resolução de uma Promise.

Veja abaixo um exemplo parecido com algo que fizemos anteriormente, onde aguardamos 2 segundos antes de exibir uma mensagem no console:

```
function chamarConsoleEm2Segundos() {
  return new Promise(function (resolve, reject) {
    setTimeout(function () {
      console.log("Resolvendo a Promise");
      resolve();
    }, 2000);
  });
}

async function main() {
  console.log("Início da execução");
  await chamarConsoleEm2Segundos();
  console.log("Fim da execução");
}

main();
```

Nesse exemplo, a função `main` é declarada como assíncrona utilizando a palavra-chave `async`, fazemos isso pois toda função que utiliza a palavra-chave `await` em seu corpo deve ser declarada como assíncrona. A função `main` mostra uma mensagem de início da execução no console, chama a função `chamarConsoleEm2Segundos` e aguarda a resolução da Promise retornada por essa função utilizando a palavra-chave `await`, depois que essa Promise é resolvida exibimos uma mensagem de fim da execução no console.

Utilizando `async` e `await` podemos reescrever o código do exemplo do "Callback Hell" de forma mais legível e organizada:

```
async function main() {
  const conteudo = await lerArquivo('arquivo.txt');
  const resultado = await enviarParaAPI(conteudo);
  await salvarArquivo('resultado.txt', resultado);
  const resposta = await enviarParaOutraAPI('resultado.txt');
  console.log(resposta);
}
```

Nesse exemplo utilizamos a palavra-chave `await` para aguardar a resolução de cada Promise antes de prosseguir com a execução do código, tornando o código mais legível e fácil de manter. Dessa forma, podemos escrever código assíncrono de forma mais parecida com código síncrono, sem a necessidade de aninhar diversas funções de callback.

Try/Catch

Em JavaScript uma função pode retornar um resultado, ou lançar uma exceção caso ocorra algum erro. Para lidar com exceções, podemos utilizar a estrutura `try/catch`, que nos permite capturar exceções lançadas por uma função e tratar essas exceções de forma adequada.

É comum vermos a estrutura `try/catch` sendo utilizada em conjunto com Promises e async/await para tratar erros que ocorrem durante a execução de operações assíncronas. Isso ocorre pois as operações assíncronas são inerentementes mais propensas a erros, pois costumam depender de fatores externos, como acesso à rede ou leitura e gravação no sistema de arquivos.

Entretanto, é importante ter em mente que mesmo funções síncronas podem lançar exceções, assim, é sempre uma boa prática utilizar `try/catch` para tratar exceções, independentemente se a função é assíncrona ou não. Veja o exemplo da função `JSON.parse`, que lança uma exceção caso o JSON passado como argumento seja inválido:

```
function lerTextJSON(texto) {
  try {
    return JSON.parse(texto);
  } catch (erro) {
    console.error('Erro ao ler o JSON:', erro);
    return null;
  }
}

const textValido = '{"nome": "João"}';
const textInvalido = '{nome: "João"}';

console.log(lerTextJSON(textValido));
console.log(lerTextJSON(textInvalido));
```

Quando uma função lança uma exceção, o fluxo de execução é interrompido e a pilha de chamadas de função vai sendo desempilhada até que a exceção seja capturada por um bloco `try/catch` que a trate. Caso a exceção não seja capturada, o programa é encerrado e a exceção é exibida no console.

Fetch

Documentação: https://developer.mozilla.org/pt-BR/docs/Web/API/Fetch_API

A API Fetch é um conjunto de métodos e objetos presentes no NodeJS e em navegadores Web modernos que nos permite realizar requisições HTTP de forma assíncrona. A API Fetch é baseada em Promises, o que a torna uma ferramenta poderosa para lidar com chamadas a APIs externas.

Para realizar uma requisição utilizando a API Fetch, utilizamos a função `fetch`, que recebe como argumento a URL para a qual desejamos realizar a requisição. A função `fetch` retorna uma Promise que será resolvida com um objeto do tipo `Response` que contém a resposta da requisição.

No exemplo abaixo realizamos uma requisição para uma API pública que possui informações ficticías de uma lista de tarefas:

```
async function obterLista() {
  const resposta = await fetch('https://jsonplaceholder.typicode.com/todos');
  const lista = await resposta.json();
  console.log(lista);
}

obterLista();
```

Nesse exemplo a função `obterLista` chama a função `fetch` passando a URL da API de tarefas, aguarda a resolução da Promise retornada por `fetch`. A resolução dessa Promise corresponde ao envio da requisição e o recebimento da resposta, que é armazenada na variável `resposta`. Em seguida, utilizamos o método `json` do objeto `Response` para obter o conteúdo da resposta no formato JSON, e armazenamos esse conteúdo na variável `lista`.

A função `fetch` possui diversos recursos, permitindo que configuremos a requisição de acordo com nossas necessidades. Por exemplo, podemos passar um objeto de configuração como segundo argumento para `fetch`, onde podemos definir o método HTTP, os cabeçalhos, o corpo da requisição, entre outros.