USCS
UNIVERSIDADE DE SÃO CAETANO DO SUL

Linguagem de Programação

Prof. Renato Carioca Duarte

USCS
UNIVERSIDADE DE SÃO CAETANO DO SUL

Unidade 07

Operações Assíncronas

Operações Assíncronas

Operações Assíncronas

- É uma operação que não é executada imediatamente.
- Em vez disso, ela é colocada em uma fila e executada mais tarde, quando a CPU estiver disponível.

- As operações assíncronas são úteis para lidar com tarefas que podem demorar um tempo para serem concluídas, como fazer uma solicitação HTTP ou ler um arquivo do disco.
- Se essas tarefas fossem executadas de forma síncrona, elas bloqueariam o programa até que fossem concluídas, o que poderia prejudicar a experiência do usuário.

Operações Assíncronas

- Existem várias maneiras de lidar com operações assíncronas em JavaScript:

1. Callback
2. Promise
3. Async/Await

Callback

- Um callback é uma função que é passada como argumento para outra função e é chamada quando a operação assíncrona é concluída.

- Por exemplo, a função `setTimeout()` recebe um callback como argumento e executa o callback depois de um determinado número de milissegundos.

- Por exemplo, o seguinte código chama a função `setTimeout()` para executar o callback `myCallback()` depois de 1 segundo:

```
setTimeout(myCallback, 1000);
```

Callback

- Por exemplo, a função avisar() é declarada como uma função que exibe um alerta com a mensagem "Seu tempo acabou!! Tente novamente!!".
- A função setTimeout() é usada para chamar a função avisar() depois de 5000 milissegundos

```
function avisar() {
    alert('Seu tempo acabou!! Tente novamente!!');
};

setTimeout(avisar, 5000);
```

Promise

- As Promises são um padrão mais moderno para lidar com operações assíncronas.
- A Promise API fornece uma maneira de representar um resultado assíncrono.
- Elas representam um valor futuro que pode ser resolvido (sucesso) ou rejeitado (erro).
- Isso permite que você encadeie várias operações assíncronas de forma mais legível e controlada.

Promise

- Uma Promise é um objeto que pode ser em um dos seguintes estados:

  - Pending: A Promise ainda não foi resolvida.
  - Fulfilled: A Promise foi resolvida com um valor.
  - Rejected:A Promise foi rejeitada com um erro.

Promise

- Para criar uma Promise, você pode usar o construtor `Promise()`.
- Sintaxe do construtor:

```
new Promise((resolve: Function, reject: Function) => void)
```

- O construtor `Promise()` recebe 2 callbacks como argumento: resolve e reject.
- **resolve**: função para retornar o resultado da promise.
- **reject**: função para retornar o erro da promise.

Promise

- O exemplo abaixo cria uma instância de uma Promise, que representa uma operação assíncrona que será resolvida com o valor `10` após 1 segundo de atraso.

```
const promise = new Promise((resolve, reject) =>
  setTimeout(() => {
    resolve(10);
  }, 1000);
});
```

Promise

- O exemplo abaixo cria uma instância de uma Promise, que representa uma operação assíncrona que será resolvida com o valor `10` após 1 segundo de atraso.

```
const promise = new Promise((resolve, reject) => {
  setTimeout(() => {
    resolve(10);
  }, 1000);
});

promise.then((value) => {
  console.log(value);
});
```

Promise

- A função soma Assinc (a, b) simula uma adição assíncrona de dois números.
- Ela cria uma Promise que resolve com o resultado da adição após 1 segundo de atraso.

```
function somaAssinc(a, b) {
  return new Promise((resolve, reject) => {
    setTimeout(() => {
      const result = a + b;
      resolve(result);
    }, 10000);
  });
}
```

Promise

- Em seguida, usamos o método .then() para lidar com o resultado da Promise quando ela for resolvida.
- Se ocorrer um erro durante o processamento da Promise, podemos usar o método .catch() para lidar com isso.

```
somaAssinc(5, 3)
.then((sum) => {
    console.log(`A soma é: ${sum}`);
})
.catch((error) => {
    console.error(`Erro: ${error}`);
});
```

Promise

- No exemplo, estamos somando os números 5 e 3 de forma assíncrona e, quando a Promise for resolvida, imprimimos o resultado no console.
- Se algo der errado na Promise, capturamos o erro e o tratamos usando .catch().

Async/Await

- Async/Await é uma forma mais legível e simplificada de lidar com operações assíncronas em JavaScript.
- Ele é baseado em Promises e permite que você escreva código assíncrono de uma maneira que se parece mais com código síncrono, tornando-o mais fácil de entender e manter.
- Funcionamento:

1. Async Function:

- Ao declarar uma função como `async`, você está indicando que essa função conterá operações assíncronas.
- O uso do `async` antes da função automaticamente a transforma em uma Promise.

Async/Await

2.Await Keyword:

- Dentro de uma função `async`, você pode usar a palavra-chave `await` antes de uma expressão que retorna uma Promise.
- O `await` pausa a execução da função até que a Promise seja resolvida.
- Isso permite que você aguarde a conclusão de operações assíncronas sem bloquear a thread principal.

3. Tratamento de Erros:

- Assim como com Promises, você pode usar `try/catch` para lidar com erros em operações assíncronas dentro de funções assíncronas.

Async/Await

```
function atraso(ms) {
  return new Promise((resolve) => setTimeout(resolve, ms));
}

async function teste() {
  console.log("Iniciando...");
  await atraso(8000); // Aguarda por 2 segundos
  console.log("Após o atraso de 8 segundos.");
}

teste();
```

Async/Await

- Neste exemplo, temos uma função atraso que retorna uma Promise que resolve após um certo número de milissegundos.
- A função testa usa async/await para esperar por essa Promise de atraso e, durante esse tempo, outras operações podem ser executadas.
- Ao executar o código acima, você verá que "Iniciando..." será impresso, haverá um atraso de 2 segundos e, em seguida, "Após o atraso de 2 segundos." será impresso.

Vantagens do Async/Await

Código Mais Limpo:

- O `async/await` simplifica o encadeamento de Promises e evita o aninhamento excessivo de callbacks, o que pode tornar o código mais difícil de entender.

Tratamento de Erros:

- O tratamento de erros é mais semelhante ao código síncrono, com blocos `try/catch`, o que facilita a identificação e resolução de problemas.

Encadeamento de Operações:

- É mais fácil encadear várias operações assíncronas, melhorando a legibilidade do código.

Vantagens do Async/Await

- No entanto, é importante lembrar que o `async/await` não substitui completamente o uso de Promises.
- Ele é uma ferramenta complementar que torna o código mais legível em cenários assíncronos, mas você ainda precisa entender as Promises subjacentes para usá-lo efetivamente.

Dúvidas

??????