DNC

Material de Apoio

Programação Assíncrona

Programação Assíncrona

A programação assíncrona permite que um programa execute operações sem bloquear a execução das demais tarefas.

Em JavaScript, isso é especialmente importante devido ao seu modelo de execução de thread única, onde tarefas demoradas podem bloquear o fluxo do programa.

Callbacks, promises e a sintaxe async/await são as principais ferramentas disponíveis para implementar a lógica assíncrona.

Com a prática, você poderá escolher a abordagem mais adequada para cada situação e garantir que seu código permaneça limpo e eficiente.

Callbacks

Um callback é uma função passada como argumento para outra função, que é executada após a conclusão de uma operação.

```
function fetchData(callback) {
    setTimeout(() => {
        const data = { id: 1, name: 'Product' };
        callback(data);
    }, 2000);
}

fetchData((data) => {
    console.log(data);
});
```

Promises

Promises são uma maneira mais moderna de lidar com operações assíncronas. Uma promise é um objeto que representa a eventual conclusão (ou falha) de uma operação assíncrona e seu valor resultante.

```
function fetchData(callback) {
    setTimeout(() => {
        const data = { id: 1, name: 'Product' };
        callback(data);
    }, 2000);
}

fetchData((data) => {
    console.log(data);
});
```

Async/Await

Introduzido no ECMAScript 2017, async e await fornecem uma sintaxe mais clara e direta para trabalhar com promises. Uma função declarada como async retorna uma promise, e await só pode ser usado dentro de funções async.

```
async function fetchData() {
    return new Promise((resolve, reject) => {
        setTimeout(() => {
            const data = { id: 1, name: 'Product' };
            resolve(data);
        }, 2000);
    });
}

async function getData() {
    try {
        const data = await fetchData();
        console.log(data);
    } catch (error) {
        console.error(error);
    }
}

getData();
```

Múltiplas Promises

O método Promise.all permite que várias promises sejam executadas em paralelo e retorna uma única promise que resolve quando todas as promises passadas como argumento resolvem.

```
function fetchData(id) {
    return new Promise((resolve) => {
        setTimeout(() => {
            resolve({ id, name: `Product ${id}` });
        }, 1000);
    });
}

async function getAllData() {
    const ids = [1, 2, 3];
    const promises = ids.map(id => fetchData(id));
    try {
        const results = await Promise.all(promises);
        console.log(results);
    } catch (error) {
        console.error(error);
    }
}

getAllData();
```

Tratamento de Erros

Quando trabalhamos com operações assíncronas, é importante lidar corretamente com erros para garantir que o programa possa responder adequadamente a falhas.

```
async function fetchDataWithError() {
    return new Promise((resolve, reject) => {
        setTimeout(() => {
            reject(new Error('Failed to fetch data'));
        }, 2000);
    });
}

async function getData() {
    try {
        const data = await fetchDataWithError();
        console.log(data);
    } catch (error) {
        console.error('Error:', error.message);
    }
}

getData();
```

E aí, curtiu?

Esperamos que esse resumo tenham enriquecido sua perspectiva estratégica para enfrentar os desafios.

Salve esse PDF para consultar sempre que precisar.

DNC