Conceitos gerais de programação assíncrona

Samuel Amaro

16 de julho de 2021

Neste artigo, nós vamos ver um número de conceitos importantes relativos à programação assíncrona e como ela se parece em navegadores modernos e em JavaScript. Você deve entender estes conceitos antes de trabalhar com outros artigos neste módulo.

Objetivo: Entender os conceitos básicos da programação assíncrona e como ela se manifesta em navegadores e JavaScript.

Assícrono?

Normalmente, o código de um programa é executado de forma direta, com uma coisa acontecendo por vez. Se uma função depende do resultado de outra função, ela tem que esperar o retorno do resultado, e até que isso aconteça, o programa inteiro praticamente para de funcionar da perspectiva do usuário.

Usuários do Mac, por exemplo, conseguem ver isso como o cursor giratório em arco-íris (ou "beachball", como normalmente é chamado). Este cursor é o jeito do sistema operacional dizer: "o programa atual que você está usando teve que parar e esperar algo terminar de ser executado, e estava demorando tanto que fiquei preocupado se você estava pensando no que aconteceu."

Essa é uma situação frustrante, e não faz bom uso do poder de processamento do computador — especialmente em uma era em que computadores tem múltiplos núcleos de processamento disponíveis. Não há sentido em ficar esperando por algo quando você pode deixar outra tarefa ser executada em um núcleo de processador diferente e deixar que ele te avise quando terminar. Isso te permite fazer mais coisas por enquanto, o que é a base da **programação assincrona**. Depende do ambiente de programação que você está usando (navegadores da Web, no caso de desenvolvimento da Web) para fornecer APIs que permitem executar essas tarefas de forma assíncrona.

Bloqueio de código

Técnicas **async**(assíncronas) são muito úteis, principalmente na programação web. Quando um aplicativo web é executado em um navegador e executa um pedaço de código rigoroso sem retornar o controle para o navegador, ele pode parecer que travou. Isso é chamado de **blocking**; o navegador está bloqueado de continuar a manusear a entrada do usuário e de realizar outras tarefas até que o aplicativo web retorne o controle do processador.

Vamos dar uma olhadinha em alguns exemplos para que você entenda o blocking.

No nosso exemplo simple-sync.html, nós adicionamos um evento de click em um botão para que, quando clicado, ele executa uma tarefa pesada (calcula 10 milhões de datas e depois imprime a última delas no console) e depois adiciona um parágrafo no DOM:

```
1
2  const btn = document.querySelector('button');
3  btn.addEventListener('click', () => {
4    let myDate;
5    for(let i = 0; i < 10000000; i++) {
6      let date = new Date();
7      myDate = date
8    }
9
10   console.log(myDate);
11
12   let pElem = document.createElement('p');
13   pElem.textContent = 'This is a newly-added paragraph.';
14   document.body.appendChild(pElem);
15 });
```

Listing 1: exemplo simples de sync

Quando o exemplo for executado, abra seu console JavaScript e depois clique no botão — você verá qua o parágrafo não aparece até que o programa termine de calcular as datas e imprimir a última no console. O código é executado na ordem em que ele aparece na fonte, e a operação seguinte só é executada depois que a primeira for terminada.

Nota: Nota: O exemplo anterior não é muito realistico. Você nunca calcularia 10 milhões de datas em um aplicativo real! Mas isso serve par te dar um apoio sobre o assunto.

No nosso segundo exemplo simple-sync-ui-blocking.html (veja aqui), nós simulamos algo mais realistico que você pode encontrar em uma página real. Nós bloqueamos a interatividade do usuário na renderização da UI. Neste exemplo, nós temos dois botões:

- Um botão "Fill canvas"que quando for clicado renderiza 10 milhão de círculos azuis no elemento `<canvas>`.
- Um botão "Clique-me"que mostra um alerta quando clicado.

```

function expensiveOperation() {
  for(let i = 0; i < 1000000; i++) {
    ctx.fillStyle = 'rgba(0,0,255, 0.2)';
    ctx.beginPath();
    ctx.arc(random(0, canvas.width), random(0, canvas.height), 10,
    degToRad(0), degToRad(360), false);
    ctx.fill()
  }
}

fillBtn.addEventListener('click', expensiveOperation);

alertBtn.addEventListener('click', () =>
  alert('You clicked me!')
);
```

Listing 2: exemplo simples de bloqueio de sicronização da interface do usuario

Se você clicar no primeiro botão e imediatamente no segundo, você verá que a mensagem de alerta não aparece até que os círculos sejam totalmente renderizados. A primeira operação bloqueia a segunda até a sua finalização.

Nota: OK, no nosso caso, isso é ruim e estamos bloqueando o código de propósito, mas isso é um problema comum que desenvolvedores de aplicativos reais sempre tentam resolver.

E por quê isso acontece? A resposta é que o JavaScript é **single threaded**. E é neste ponto que precisamos introduzir a você o conceito de **threads**.

Threads

Uma **thread** é basicamente um único processo que um programa pode usar para concluir tarefas. Cada thread só pode fazer uma tarefa de cada vez:

```
          Tarefa A --> Tarefa B --> Tarefa C
```

Listing 3: thread

Cada tarefa será executada sequencialmente; uma tarefa tem que ser concluída antes que a próxima possa ser iniciada.

Como foi dito anteriormente, muitos computadores possuem múltiplos núcleos, para que possam fazer múltiplas coisas de uma vez só. Linguagens de programação que podem suportar múltiplas threads podem usar múltiplos processadores para concluir múltiplas tarefas simultâneamente:

```
1          Thread 1: Tarefa A --> Tarefa B
2          Thread 2: Tarefa C --> Tarefa D
```

Listing 4: thread

JavaScript é single Threaded

JavaScript é tradicionalmente single-threaded. Mesmo com múltiplos núcleos de processamento, você só pode fazê-lo executar tarefas em uma única thread, chamada de **main thread**(thread principal). Nosso exemplo de cima é executado assim:

```
1          Main thread: Renderelizar circulos no canvas --> Mostrar
       alert()
```

Depois de um tempo, o JavaScript ganhou algumas ferramentas para ajudar em tais problemas. As `Web workers` te permitem mandar parte do processamento do JavaScript para uma thread separada. Você geralmente usaria uma worker para executar um processo pesado para que a UI não seja bloqueada.

```
1          Main thread: Tarefa A --> Tarefa C
2       Worker thread: Tarefa pesada B
```

Com isso em mente, dê uma olhada em `simple-sync-worker.html`, com o seu console JavaScript aberto. Isso é uma nova versão do nosso exemplo que calcula 10 milhões de datas em uma tread worker separada. Agora, quando você clica no botão, o navegador é capaz de mostrar o parágrafo antes que as datas sejam terminadas. A primeira opreção não bloqueia a segunda.

Código Assícrono

Web workers podem ser bem úteis, mas elas tem as suas limitações. Uma delas é que elas não são capazes de acessar a DOM — você não pode fazer com que uma worker faça algo diretamente para atualizar a UI. Nós não poderíamos renderizar nossos 1 milhão de círculos azuis na nossa worker; basicamente ela pode apenas fazer cálculos de números.

O segundo problema é que, mesmo que o código executado em uma worker não cause um bloqueio, ele ainda é um código síncrono. Isso se torna um

problema quando uma função depende dos resultados de processos anteriores para funcionar. Considere os diagramas a seguir:

```
Main thread: Tarefa A --> Tarefa B
```

Nesse caso, digamos que a tarefa A está fazendo algo como pegar uma imagem do servidor e que a tarefa B faz algo com essa imagem, como colocar um filtro nela. Se você iniciar a tarefa A e depois tentar executar a tarefa B imediatamente, você obterá um erro, porque a imagem não estará disponível ainda.

```
Main thread:   Tarefa A --> Tarefa B --> | Tarefa D|
Worker thread: Tarefa C ----------------> |         |
```

Neste caso, digamos que a tarefa D faz uso dos resultados das tarefas B e C. Se nós pudermos garantir que esses resultados estejam disponíveis ao mesmo tempo, então tudo talvez esteja bem, mas isso não é garantido. Se a tarefa D tentar ser executada quando um dos resultados não estiver disponível, ela retornará um erro.

Para consertarmos tais problemas, os browsers nos permitem executar certas operações de modo assíncrono. Recursos como `Promises` te permitem executar uma operação e depois esperar pelo resultado antes de executar outra operação:

```
Main Thread: Tarefa A                    Tarefa B
Promisse:           |___operacao async___|
```

Já que a operação está acontecendo em outro lugar, a main thread não está bloqueada enquanto a operação assíncrona está sendo processada.

Nós vamos começar a olhar em como podemos escrever código assíncrono no próximo artigo.

Conclusão

O design moderno de software gira em torno do uso de programação assíncrona, para permitir que os programas façam mais de uma coisa por vez. Ao usar APIs mais novas e mais poderosas, você encontrará mais casos em que a única maneira de fazer as coisas é assincronamente. Costumava ser difícil escrever código assíncrono. Ainda é preciso se acostumar, mas ficou muito mais fácil. No restante deste módulo, exploraremos ainda mais por que o código assíncrono é importante e como projetar o código que evita alguns dos problemas descritos acima.