02

Usando AsyncAwait

Transcrição

Deixamos o código mais bonito separando um método diferente, com a função de criar tarefas, de consolidação, e deixamos o clique de botão mais limpo. Porém ainda nos preocupamos com a recuperação do contexto de execução da *thread* principal, usando-o sempre que uma nova tarefa é encadeada, e isto está ficando complicado. Daqui em diante, sempre que tivermos uma função sendo executada de maneira assíncrona, lança-se a execução da tarefa, cujo resultado ainda nos preocupa. Depois, no contexto original, o resultado é processado...

Temos uma *task*, cujo resultado encadeia na tarefa uma outra, que utiliza o contexto de execução original da *thread* para, nesta tarefa, recuperar o resultado daquela *task*. É um trabalho braçal que não acontece apenas com a gente, e sim com toda a comunidade .NET. E não se trata apenas de tarefas assíncronas como esta, em que há uso intenso de CPU para a consolidação de vários dados. Na verdade, existem muitas outras tarefas assíncronas, como o download de um arquivo, aguardar um drive de rede retornar um documento, fazer *upload*, não somente orientadas à rede, e sim várias outras tarefas assíncronas envolvendo esta mesma construção.

A Microsoft, percebendo isto, lançou um recurso novo na versão `5` do C#, que torna esta escrita de aplicações assíncronas mais simples. O nome deste recurso é *AsyncAwait*, que facilita a construção de todos estes ornamentos. Para começarmos a usá-lo, precisamos indicar ao compilador a tarefa ou método a ser executado de forma assíncrona. No caso, será o clique do botão (`BtnProcessar_Click()`), para indicar que sua função é assíncrona, adicionando-se um codificador em seu cabeçalho:

```
private async void BtnProcessar_Click(object sender, RoutedEventArgs e)
```

No momento em que incluímos `async` , o compilador nos traz avisos, dicas e *warnings*. Aqui, somos informados de que marcamos o método do clique do botão como assíncrono, mas não estamos usando este recurso. Apesar de existir uma tarefa assíncrona marcada com o modificador (`async`), ainda estamos utilizando o *task* de maneira convencional. Como fazemos para usarmos este recurso novo sem termos que ficar encadeando, pegando resultados, na forma convencional?

O recurso se chama `AsyncAwait` e não é à toa. A primeira palavra chave é o `async` , e a segunda é o `await` . Podemos usá-lo para consolidar as contas, pois trata-se de uma *task*, a qual sempre pode ser aguardada.

```
await ConsolidarContas(contas);
```

Daqui em diante, voltamos a executar no contexto da *thread* inicial. Ao utilizarmos `ConsolidarContas()` temos uma tarefa que retorna uma lista de `string` , e nossa preocupação não é com a tarefa, e sim com seu resultado, a lista de `string` . Usando o `await` , podemos armazenar apenas o resultado desta tarefa em uma variável. Vamos também comentar a linha `var resultado = task.Result;` para não haver conflito entre nomes.

```
var resultado = await ConsolidarContas(contas)

  ConsolidarContas(contas)
  .ContinueWith(task => {
    var fim = DateTime.Now;
    //var resultado = task.Result;
    AtualizarView(resultado, fim - inicio);
    }, taskSchedulerUI)
```

```
    .ContinueWith(task =>
    {
      BtnProcessar.IsEnabled = true;
      }, taskSchedulerUI);
```

Nosso resultado não é uma *task* de lista de `string` como está definido em nossa função. O resultado é apenas uma lista de `string` pois aqui, na verdade, é como se estivessemos dentro de um `ContinueWith()`. Mas todo o ornamento de criar-se uma *task*, entre outros, se dá pelo compilador, deixando o código muito menos indentado, com menos níveis e uma "cara mais natural", com que estamos acostumados.

Ao executarmos no contexto original, não precisamos mais nos preocupar em guardar o contexto no início da função, porque o compilador já fará isto. Não precisamos nos preocupar com os encadeamentos referentes ao fim, atualização de *view* e clique do botão. Sendo assim, vamos removê-los de `ConsolidarContas()`, deletando-se este também:

```
private async void BtnProcessar_Click(object sender, RoutedEventArgs e)
{
  BtnProcessar.IsEnabled = false;

  var contas = r_Repositorio.GetContaClientes();

  AtualizarView(new List<string>(), TimeSpan.Zero);

  var inicio = DateTime.Now;

  var resultado = await ConsolidarContas(contas)

  var fim = DateTime.Now;
  AtualizarView(resultado, fim - inicio);
  BtnProcessar.IsEnabled = true;
}
```

Estamos com o código assíncrono usando o `AsyncAwait` do C# `5`, , vamos executar a aplicação e verificar seu funcionamento. Clicaremos em "*Start*" e, assim que ela abrir, em "Fazer Processamento". No Gerenciador de Tarefas, vemos a CPU sendo utilizada em `100%`. Voltamos à aplicação, ela está respondendo aos movimentos do mouse, e o resultado é exibido. O código está muito mais limpo, com os mesmos recursos desenvolvidos até agora sendo utilizados, transformando-se as tarefas de forma paralela, e agora usando-se o `AsyncAwait`.

Vamos melhorar o `ConsolidarContas()` também. Já aprendemos que uma *task* pode ter retorno, em vez de mapearmos as contas para uma lista de tarefas que populam uma lista, que é o que fazemos em

```
var contaResultado = r_Servico.ConsolidarMovimentacao(conta);
resultado.Add(contaResultado);
```

Vamos mapear as contas para o resultado da consolidação, sem popularmos outra lista. Substituiremos o código acima para:

```
return r_Servico.ConsolidarMovimentacao(conta);
```

Com uma expressão lambda bastante simples, fazendo somente o `return` de uma expressão (neste caso, de `r_Servico.ConsolidarMovimentacao(conta)`), podemos omitir este termo, bem como o corpo de função, que são as chaves, e também o ponto e vírgula (" ; "). Deixamos o código assim, muito mais limpo:

```
var tasks = contas.Select(conta =>
  {
    return Task.Factory.StartNew(() => r_Servico.ConsolidarMovimentacao(conta));
  });
```

Podemos fazer o mesmo com o `Task.Factory.StartNew`, deixando o código desta forma:

```
var tasks = contas.Select(conta =>
  Task.Factory.StartNew(() => r_Servico.ConsolidarMovimentacao(conta))
);
```

Já que não vamos mais utilizar o resultado, pois estamos mapeando as contas para o resultado de sua consolidação, deletaremos a linha `var resultado = new List<string>();`, utilizando o recurso de `AsyncAwait`, transformando-se a tarefa abaixo em assíncrona:

```
private async Task<List<string>> ConsolidarContas(IEnumerable<ContaCliente> contas)
{
  var tasks = contas.Select(conta =>
    Task.Factory.StartNew(() => r_Servico.ConsolidarMovimentacao(conta))
  );

  return Task.WhenAll
}
```

O que vamos esperar são todas as tarefas mapeadas anteriormente. Quando utilizamos a sobrecarga `WhenAll`, não com uma lista de tarefas e sim com lista de tarefas com mesmo retorno, o que ela nos retorna é uma tarefa que retorna um *array* deste tipo. Se possuímos várias tarefas que retornam uma `string` e estamos aguardando todas, o que vamos receber como resultado é uma tarefa que retorna um *array* deste tipo de retorno. Vamos apertar `F12` e veremos a sobrecarga que estamos utilizando agora:

```
...public static Task<TResult[]> WhenAll<TResult>(IEnumerable<Task<TResult>> tasks);
```

Ou seja, temos várias tarefas (`tasks`), todas retornando o mesmo tipo, e com o resultado, teremos uma tarefa que retorna um *array* deste tipo. Como podemos utilizar isto? Guardaremos o `WhenAll` de todas estas tarefas em uma variável. Aqui, não queremos uma *task*, e sim o resultado, portanto podemos usar apenas o `await`, tendo como resultado um *array* de `string`:

```
private async Task<string[]> ConsolidarContas(IEnumerable<ContaCliente> contas)
{
  var tasks = contas.Select(conta =>
    Task.Factory.StartNew(() => r_Servico.ConsolidarMovimentacao(conta))
    );

    return await Task.WhenAll(tasks);
}
```

Teremos um problema em `AtualizarView`, porque ele recebe uma lista. Vamos transformá-la em uma `IEnumerable`, fazer mais algumas mudanças, e o código ficou mais elegante no `ConsolidarContas` também:

```
private void AtualizarView(IEnumerable<String> result, TimeSpan elapsedTime)
{
  var tempoDecorrido = $"{ elapsedTime.Seconds }.{ elapsedTime.Milliseconds} segundos!";
  var mensagem = $"Processamento de {result.Count()} clientes em {tempoDecorrido}";

  LstResultados.ItemSource = result;


  TxtTempo.Text = mensagem;
}
```

Vamos executar a aplicação mais uma vez. ByteBank aberto, faremos o processamento, acompanhando-o pelo Gerenciador de Tarefas, que nos mostra que há muito acontecendo, e tudo está funcionando como esperado. A app está ficando muito boa e o código está ficando limpo também. Ainda temos o que melhorar: quando estamos executando uma tarefa, por mais que a tela continue respondendo, nada está sendo avisado ao usuário, indicando-se o andamento do processamento, por exemplo, como outros programas costumam fazer. Vamos melhorar isto?