03

Usando o CancellationTokenSource e o CancellationToken

Transcrição

Vamos alterar o documento do código da janela principal (`MainWindow`) para incluirmos o `CancellationTokenSource` . Logo no início do arquivo, criaremos um campo deste tipo e, por convenção, vamos chamá-lo de `cts` , acrônimo de "*Cancellation Token Source*".

```
public partial class MainWindow : Window
{
  private readonly ContaClienteRepository r_Repositorio;
  private readonly ContaClienteService r_Servico;
  private CancellationTokenSource _cts;
...
}
```

Não atribuiremos nenhum valor ao `cts` neste momento, fazendo-o somente quando o usuário clicar no botão de processamento, instanciando um novo `CancellationTokenSource` .

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

  _cts = new CancellationTokenSource();

  var contas = r_Repositorio.GetContaClientes();
...
}
```

Utilizamos o `CancellationTokenSource` para a emissão de um `CancellationToken()` a ser usado pelas tarefas, que por sua vez são usadas dentro de `ConsolidarContas` . Iremos criar um novo parâmetro neste método, que receberá um `CancellationToken` denominado `ct` - por padrão, quando se nomeiam estes objetos.

Da variável `resultadoConsolidacao` até o retorno `resultadoConsolidacao` , temos a tarefa, a unidade de trabalho. Para reproduzirmos o fluxo de execução dos slides do primeiro vídeo, colocaremos um `if()` , acessando-se a propriedade do `IsCancellationRequested` . Caso haja requisição de cancelamento, adotaremos a convenção criada pelo .NET lançando uma nova exceção do tipo `OperationCanceledException` (operação de exceção cancelada) e indicando que houve cancelamento durante a execução da tarefa.

Vamos verificar os construtores que temos para esta exceção: o *default*, sem parâmetros e com mensagem, como a maioria das exceções do .NET. E temos um - que é bem interessante - e recebe o `CancellationToken` como parâmetro. Se aquele que emitiu o cancelamento está disponível, vamos indicá-lo ao framework também, para informarmos que a exceção não foi lançada de forma aleatória, e sim por conta do `CancellationToken` . Adicionaremos este código antes do retorno do valor:

```
private async Task<String[]> ConsolidarContas(IEnumerable<ContaCliente> contas, IProgress<string> re
{
  var tasks = contas.Select(conta =>
    Task.Factory.StartNew(() =>
```

```
    {
      if (ct.IsCancellationRequested)
        throw new OperationCanceledException(ct);

      var resultadoConsolidacao = r_Servico.ConsolidarMovimentacao(conta);

      reportadorDeProgresso.Report(resultadoConsolidacao);

      if (ct.IsCancellationRequested)
        throw new OperationCanceledException(ct);

      return resultadoConsolidacao;
    })
  );
```

Ao alterarmos a assinatura do método de consolidação de contas, recebendo por parâmetro o `CancellationToken`, modificaremos também a sua chamada. O Visual Studio nos avisa que esquecemos de acrescentar um parâmetro. Faremos isto a partir da inclusão do `Token` criado pelo `cts` e acessado por sua propriedade de mesmo nome:

```
var resultado = await ConsolidarContas(contas, progress, _cts.Token);
```

Nossa aplicação se encontra mais engrenada, porém, ainda é necessário oferecermos ao usuário a opção de cancelamento. Acessamos o `CancellationTokenSource` invocando o método referente e que será responsável por notificar o `Token`. No momento em que se verificar a propriedade `IsCancellationRequested`, tudo aquilo que envolver o `Token` confirmará, ou não, o cancelamento.

```
private void BtnCancelar_Click(object sender, RoutedEventArgsEvent e)
{
  BtnCancelar.IsEnabled = false;

  _cts.Cancel();
}
```

Vamos conferir o funcionamento da aplicação clicando em "Fazer Processamento" quando ela for aberta. Notem que há certa demora para responder, e só depois indica o erro de exceção, o qual ocorre porque clicamos em "Cancelar" enquanto provavelmente se executava a seguinte linha:

```
var resultadoConsolidacao = r_Servico.ConsolidarMovimentacao(conta);
```

É exibida a mensagem "*System.OperationCanceledException:'A operação foi cancelada.*".

Antes do retorno, tivemos o lançamento da exceção, e vamos clicando em "*Continue*" (onde se localiza o botão "*Start*"), passando por todas elas. Em `if`, o erro aparece novamente, possivelmente porque havia outras *threads* executando este código, e travando naquele momento. Clicaremos em "*Continue*" e, depois, veremos que são várias *threads* pois utilizaremos uma máquina com `8` núcleos, de forma que o `TaskScheduler` deve ter colocado `8` tarefas para serem rodadas simultaneamente.

Continuamos clicando em "*Continue*"... Lançou-se uma exceção logo no começo de `if (ct.IsCancellationRequested)`, cuja *task* foi criada, sendo executada após o cancelamento por parte do usuário. Seguiremos clicando em "*Continue*", atingindo-s

a seguinte linha:

```
var resultado = await ConsolidarContas(contas, progress, _cts.Token);
`
```

Esta exceção vai passando por todas as tarefas, até o início da chamada `await ConsolidarContas()`.

O `_cts` utiliza por parâmetro o `CancellationToken`, e deve tratar esse tipo de exceção de operação cancelada. Vamos fazer isto? Ao clicarmos novamente em "*Continue*", o Visual Studio acabará se perdendo, já que houve uma exceção não tratada. Utilizaremos um *Try/Catch*, cujo `catch` não serve para qualquer exceção, e sim apenas para a operação cancelada. Assim, todo código referente às variáveis `resultado` e `fim` será realocado para dentro do bloco `try`:

```
try
{
  var resultado = await ConsolidarContas(contas, progress, _cts.Token);

  var fim = DateTime.Now;
  AtualizarView(resultado, fim - inicio);
}
catch (OperationCanceledException)
{
    throw;
}
finally
{
  BtnProcessar.IsEnabled = true;
  BtnCancelar.IsEnabled = false;
}
```

Criaremos o bloco `finally`, no qual o `BtnProcessar` e o `BtnCancelar` atuarão, e que ficará *disabled*, pois se a tarefa foi continuada, o botão precisará ser desativado novamente. No tratamento de exceção, podemos simplesmente mostrar ao usuário uma mensagem de que a execução foi cancelada.

```
try
{
  var resultado = await ConsolidarContas(contas, progress, _cts.Token);

  var fim = DateTime.Now;
  AtualizarView(resultado, fim - inicio);
}
catch (OperationCanceledException)
{
    TxtTempo.Text ="Operação cancelada pelo usuário";
}
finally
{
  BtnProcessar.IsEnabled = true;
  BtnCancelar.IsEnabled = false;
}
```

O `TxtTempo` informa quantas tarefas foram feitas e em quanto tempo, e mudaremos seu texto para "Operação cancelada pelo usuário". Ao fim destas alterações, rodaremos a aplicação novamente para conferirmos o que foi feito. Após o início do

processamento, clicaremos em "Cancelar", e a aplicação não nos responde imediatamente, por supostamente estar realizando o bloco de execução de `resultadoConsolidacao`.

Em `if`, lança-se uma exceção, clicaremos em "*Continue*", a linha seguinte lançará outra exceção, várias *threads* fazem o mesmo, porque todas elas verificam a propriedade `IsCancellationRequested`. Vimos isso anteriormente... Em `throw new OperationCanceledException(ct);`, foi executada uma tarefa que já se iniciou sendo cancelada, ou seja, ela deixou de ser executada logo no começo. Seguimos clicando em "*Continue*" exceção após exceção. A app continua respondendo, pois agora tratamos este tipo de exceção simplesmente notificando ao usuário que houve o cancelamento na operação.

Em seguida, fecharemos a aplicação. Pode parecer estranho cancelarmos a operação quando o `TaskScheduler` do .NET, que "orquestra" o momento em que as *tasks* deveriam ser executadas e agenda o momento oportuno para a execução delas, executando uma tarefa que já se inicia cancelada. Notamos isto porque logo no início, uma exceção é lançada. Isto ocorre pois não usamos uma sobrecarga `StartNew` que recebe como parâmetro um `CancellationToken`.

Pode-se utilizar uma sobrecarga que receba como parâmetro um *delegate* e, como segundo parâmetro, um `CancellationToken`. Aproveitaremos para usá-lo nesta construção, de forma que o `TaskScheduler`, antes de se preocupar em agendar a execução de uma tarefa, providenciando uma *thread* e preparando-a para executar o código, verificará se o `CancellationToken` já foi disparado. Então, ele não irá preparar todo este cenário para a execução desta *task*.

Por que é uma boa prática verificar-se isto logo no começo? Porque até o `TaskScheduler` começar a execução de uma *task* e a primeira linha, de fato, ser executada, ou seja, até que se verifique se houve cancelamento ou não, para posteriormente se executar a tarefa, há um intervalo de tempo, em que o usuário pode clicar no botão de cancelamento, até porque a *thread* principal está livre. O usuário pode clicar exatamente nesse momento crucial. É por isto que a verificação em relação ao cancelamento por parte do usuário é realizada logo no começo. No entanto, vamos observar que o bloco de código abaixo é repetitivo e "chato":

```
if (ct.IsCancellationRequested)
  throw new OperationCanceledException(ct);
```

Colocamos este `if` porque dentro deste bloco, temos a oportunidade de restituirmos a aplicação para um estado válido caso seja uma operação que precise ser retornada, o que significa que quando o usuário cancela a execução enquanto o processamento está sendo feito, haja a opção de se desfazer o que foi feito. Quando não fazemos nada, como neste caso (em que somente lançamos uma exceção), podemos usar um método do `CancellationToken` chamado `ThrowIfCancellationIsRequested`, que justamente verifica se a propriedade de `IsCancellationRequested` é *true*, lançando-se uma exceção se este for o caso.

```
var tasks = contas.Select(conta =>
  Task.Factory.StartNew(() =>
  {
    ct.ThrowIfCancellationIsRequested();

    var resultadoConsolidacao = r_Servico.ConsolidarMovimentacao(conta);

    reportadorDeProgresso.Report(resultadoConsolidacao);

    ct.ThrowIfCancellationIsRequested();

    return resultadoConsolidacao;
    }, ct)
  );
```

Utilizaremos portanto este método substituindo as duas linhas de `if` para tornar o código mais limpo. Feito isto, executaremos a app por meio de "*Start*", observando-se que o comportamento permanece igual após os cliques em "Fazer Processamento" e em "Cancelar". Aguardaremos um pouco, pois o processamento está sendo feito na linha referente a `ConsolidarMovimentacao`. Repetimos o procedimento de apertar o "*Continue*" durante os lançamentos das exceções nas *threads*.

Conforme vamos clicando em "*Continue*", após as linhas de `CancellationToken`, o Visual Studio pausava logo no início da tarefa na linha de `ct.ThrowIfCancellationIsRequested();`. Agora isto deixa de ocorrer por estarmos passando o `CancellationToken` como parâmetro, na sobrecarga do `StartNew`. Desta forma, o `TaskScheduler` deixa de iniciar uma tarefa previamente cancelada.