06

Passando o CancellationToken adiante

Transcrição

Ainda existem alguns fatores que nos incomodam na aplicação. Um deles ocorre quando o usuário começa o processamento cancelando-o logo em seguida. O intervalo de espera até que o cancelamento seja efetuado e a aplicação volte a responder é muito grande. Outro fator tem a ver com o *break* do Visual Studio sempre que uma exceção é lançada.

Sabemos e queremos que a exceção seja lançada pois estamos seguindo um padrão do .NET quando uma operação é cancelada. Então, para melhorarmos a experiência de desenvolvimento, desmarcaremos este *flag* de "*Break when this exception type is thrown*", clicando em "*Continue*" logo em seguida. As outras *threads* lançam esta exceção, e não estamos mais nos preocupando com isto, o Visual Studio não está travando. Vamos parar de debugar para melhorarmos o tempo que a aplicação leva até parar quando é cancelada pelo usuário.

Isto acontece porque a função "bloqueante" `ConsolidarMovimentacao` não aproveita o recurso do `CancellationToken`. Deve-se sempre passar adiante o `CancellationToken` quando houver oportunidade, porque ao verificarmos se a operação foi ou não cancelada, iremos liberar o uso da CPU também, o que neste caso só ocorre após a execução de `r_Servico.ConsolidarMovimentacao()`, com alguma demora.

Como ele se encontra sob nosso domínio, no `ByteBank.Core`, podemos modificá-lo. Por mais que não tenhamos que alterar nada nele, identificar padrões facilmente paralelizáveis é mais fácil utilizando-se a biblioteca de *tasks*. Clicaremos em `F12` para vermos a implementação do código referente a `ConsolidarMovimentacao`.

Geraremos uma nova sobrecarga deste método, que recebe por parâmetro um `CancellationToken` chamado `ct`. No entanto, para mantermos a API retrocompatível, como quando temos aplicações antigas utilizando esta mesma DLL, atualizadas posteriormente.

Para isto, manteremos a sobrecarga que só possui como parâmetro uma `ContaCliente`. Recriaremos a sobrecarga `ConsolidarMovimentacao`, que recebe como parâmetro apenas `ContaCliente` e, dentro dela, para não repetirmos código, chamaremos a implementação "rica" que recebe por parâmetro um `CancellationToken` passando a mesma conta. Este parâmetro não é associado a nenhum `CancellationTokenSource`, sendo disponível com a propriedade estática `.None` do `CancellationToken`. Por conta disto, ele nunca possui requisição, a propriedade `IsCancellationRequested = true;`. Também é necessário retornar seu valor.

```
namespace ByteBank.Core.Service
{
  public class ContaClienteService
  {
    public string ConsolidarMovimentacao(ContaCliente conta)
    {
      return ConsolidarMovimentacao(conta, CancellationToken.None);
    }
    public string ConsolidarMovimentacao(ContaCliente conta, CancellationToken ct)
    {
      ...
    }
  }
}
```

O código original será alterado, nos lugares que identificamos o *gap* que pode estar segurando recursos da CPU. O laço de repetição abaixo faz uso intenso dela, tratando-se de um bom local de identificação de um cancelamento. Portanto o alteramos:

```
foreach (var movimento in conta.Movimentacoes)
  soma += movimento.Valor * FatorDeMultiplicacao(movimento.Data);
```

Colocamos este bloco entre chaves e, antes do cálculo de soma, e a cada iteração, vamos verificar se houve cancelamento, podendo-se utilizar tanto `if` quanto simplesmente lançar uma exceção para o caso de ter ocorrido requisição de cancelamento. Como não nos preocupamos em desfazermos nenhuma ação, usaremos o método `ThrowIfCancellationRequested()` do `CancellationToken`.

Outro local que vemos como potencialmente pesado em se tratando de recursos de CPU é o `AtualizarInvestimentos`. Sendo assim, antes de chamar o `ThrowIfCancellationRequested()`, vamos lançar uma exceção também, se houver cancelamento. Ou seja, sempre que possível, optaremos por liberar recursos da CPU. Assim, não gastaremos recursos de nossa máquina com trabalho inútil, afinal ele será descartado, já que o usuário cancelou a aplicação, desejando-se liberar uso para a realização de qualquer outra tarefa.

Agora que criamos uma nova sobrecarga, podemos utilizá-la em nossa tarefa, para a qual passaremos adiante o `Token` de cancelamento:

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

    var resultadoConsolidacao = r_Servico.ConsolidarMovimentacao(conta, ct);

    reportadorDeProgresso.Report(resultadoConsolidacao);

    ct.ThrowIfCancellationRequested();
    return resultadoConsolidacao;
    }, ct)
  );
```

Feito isto, executamos novamente a aplicação pressionando "*Start*", "Fazer Processamento" e, imediatamente, "Cancelar". Vejam que agora a resposta da aplicação é muito mais rápida. Começamos o processamento, cancelamos e, quase imediatadamente, a app voltará a ficar disponível para a realização de qualquer outra tarefa.