02

Reportando progresso ao usuário

Transcrição

Queremos deixar de nos preocuparmos com o `TaskScheduler` e suas ações, ou com interface gráfica, consolidando as contas por meio de *tasks*, de forma paralela. Em `Task.Factory.StartNew` , a tarefa principal consiste em reportar o progresso, e para isto pode-se criar um objeto responsável por receber uma notificação de progresso.

O problema em relação à recuperação de `TaskScheduler` , ao *report* de um progresso, é que tudo isso já existia no momento em que a Microsoft atualizou o .NET com a biblioteca de *tasks*. Nela, incluiu-se também uma interface chamada `IProgress` , genérica, que recebe por parâmetro o objeto que indica o progresso. Neste caso, o tipo de parâmetro genérico do progresso é uma `string` , considerando que o progresso obtido a partir de uma consolidação é deste tipo. Receberemos por parâmetro este `IProgress` denominado `reportadorDeProgresso` .

```
private async Task<string[]> ConsolidarContas(IEnumerable<ContaCliente> contas, IProgress<string> re

//...

// Não utilizaremos atualização do PgsProgresso na Thread de Trabalho
// PgsProgresso.Value++;

reportadorDeProgresso.Report(resultadoConsolidacao);

return  resultadoConsolidacao;
```

`Report()` é o método definido para gerar o *report* do progresso. Podemos apagar os códigos referentes a `Task.Factory.StartNew` , e também a `taskSchedulerGui` , pois a tarefa de recuperação do `TaskScheduler` podemos deixar implementada em `IProgress` . Os comentários podem ser deletados também. Ficaremos com o código assim:

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

      reportadorDeProgresso.Report(resultadoConsolidacao);

      return resultadoConsolidacao;
      })
);
```

Agora que mudamos a assinatura de `ConsolidarContas` (já que estamos delegando a função de atualização de *view* a outro objeto), criaremos a implementação permitindo que ela seja utilizada em vários lugares. Depois, vamos gerar uma pasta clicando com o lado direito do mouse em cima de `ByteBank.View` e selecionando a opção "Add > New Folder", que chamaremos de "Utils".

Clicando com o lado direito do mouse em cima da recém criada pasta, selecionaremos "*Add> Class*", para criarmos uma nova classe, aqui chamada `ByteBankProgress`. Queremos que a implementação seja o mais reutilizável possível. Para isto, vamos criar uma implementação genérica ao `IProgress`.

O Visual Studio traz uma mensagem de erro pois não implementamos esta interface ("*`ByteBankProgress<T>` does not implement interface member `IProgress<T>.Report(T)`*"). Pode-se colocar o cursor em cima de `IProgress`, apertando-se "Ctrl + .", e o programa nos mostra uma lista de dicas, dentre as quais se encontra a criação de um escopo de implementação quando não se tem uma interface implementada. É esta a opção que iremos utilizar. No arquivo `ByteBankProgress.cs`, então, temos:

```
namespace ByteBank.View.Utils
{
  public class ByteBankProgress<T> : IProgress<T>
  {
    public void Report(T value)
    {
      throw new NotImplementedException();
    }
  }
}
```

Como queremos algo bem genérico, vamos fazê-lo reportar uma ação qualquer definida pelo usuário da classe `ByteBankProgress`. Faremos isto no método do clique do botão (`MainWindow.xaml.cs`), recebendo no construtor por parâmetro o `ctor`, *code snippet* do Visual Studio. No momento em que o digitamos, aparece uma lista do IntelliSense com uma dica que aponta que temos um *code snippet* de classe `tor`. Assim que apertamos a tecla "Tab", ele completa com um construtor *default* desta classe.

```
namespace ByteBank.View.Utils
{
  public class ByteBankProgress<T> : IProgress<T>
  {
    public ByteBankProgress(Action)
    {

    }
    public void Report(T value)
    {
      throw new NotImplementedException();
    }
  }
}
```

Como parâmetro, receberemos uma `Action`, muito semelhante (senão a mesma), que utilizamos no `Task.Factory.StartNew()`. Vejam que o primeiro parâmetro recebido é uma função, pois estamos utilizando uma sobrecarga com retorno. Porém, se utilizássemos a criação de tarefas sem retorno, teríamos uma `Action`. Se não nos preocuparmos com o retorno, obteremos `Action`. Caso contrário, `Function`. Elas possuem o mesmo comportamento, gerando um *delegate*, com a diferença de que uma possui `return` e a outra não.

```
namespace ByteBank.View.Utils
{
  public class ByteBankProgress<T> : IProgress<T>
```

```
    {
      public ByteBankProgress(Action<T> handler)
      {

      }
      public void Report(T value)
      {
        throw new NotImplementedException();
      }
    }
  }
```

Utilizaremos o mesmo tipo de objeto, que por convenção se chama `handler`, pois vem de "manipulador". Vamos criar a propriedade privada cujo `handler` será passado por parâmetro no construtor. No momento do *report*, vamos usar o `handler`, *delegate* do tipo `Action` que se comporta como uma função. Desse modo, chamaremos o manipulador passando por parâmetro o valor:

```
namespace ByteBank.View.Utils
{
  public class ByteBankProgress<T> : IProgress<T>
  {
    private readonly Action<T> _handler;
    private readonly TaskScheduler _taskScheduler;

    public ByteBankProgress(Action<T> handler)
    {
        _taskScheduler = TaskScheduler.FromCurrentSynchronizationContext();
        _handler = handler;
    }
    public void Report(T value)
    {
      _handler(value);
    }
  }
}
```

Qual foi a mudança realizada, uma vez que apenas criamos um objeto que fará o mesmo que escrever *hard-coded* de outro lugar? Realmente, queremos recuperar o `TaskScheduler` no momento em que estivermos na interface gráfica, para utilizarmos o `TaskScheduler` no `Report()`. Para isto, vamos aproveitar o construtor de classe `ByteBankProgress`, recuperando o `TaskScheduler` da *thread* que está executando este código.

Como executaremos a construção deste objeto na *thread* principal, o `TaskScheduler` será o mesmo da interface gráfica. Criaremos uma propriedade, um *field*, cujo `_taskScheduler` será responsável pela criação do *report*. Portanto, criaremos também uma tarefa em `Report` a ser executada no `TaskScheduler`. Assim, tiramos a responsabilidade de manipular e gerenciar a localização da execução da *view*, do código que precisa consolidar a conta. Não teremos que nos preocupar com isto.

No `Report`, portanto, a implementação chamará aquela sobrecarga do `Task.Factory.StartNew()`, o qual recebe vários parâmetros. O primeiro deles é uma `Action` a ser executada - neste caso, ela será uma chamada do manipulador, para o qual passaremos, por parâmetro, o valor. O segundo parâmetro é o `CancellationToken`, ainda não visto anteriormente.

O terceiro parâmetro é o `TaskCreationOptions`, também sem nenhuma configuração adicional. Vamos utilizar o *default* do .NET. Por fim, o quarto parâmetro indica ao `_taskScheduler` a execução da tarefa. Feito isto, deletaremos a linha `_handler(value);` que provavelmente causaria uma exceção, afinal, o método altera a interface gráfica, e não podemos fazer isto.

```
namespace ByteBank.View.Utils
{
  public class ByteBankProgress<T> : IProgress<T>
  {
    private readonly Action<T> _handler;
    private readonly TaskScheduler _taskScheduler;

    public ByteBankProgress(Action<T> handler)
    {
        _taskScheduler = TaskScheduler.FromCurrentSynchronizationContext();
        _handler = handler;
    }
    public void Report(T value)
    {
      Task.Factory.StartNew(
        () => _handler(value),
        System.threading.CancellationToken.None,
        TaskCreationOptions.None,
        _taskScheduler
        );
    }
  }
}
```

Neste momento, com um objeto `ByteBankProgress` que implementa o `IProgress` que é bem reutilizável, vamos usá-lo. No arquivo `MainWindow.xaml.cs`, antes de `ConsolidarContas()`, criaremos a variável `byteBankProgress`:

```
var inicio = DateTime.Now;

var byteBankProgress = new ByteBankProgress()
var resultado = await ConsolidarContas(contas);
```

Aqui temos como objetivo atualizarmos o `PgsProgresso`. O `ByteBankProgress()` não está encontrando a classe (e consequentemente, está gerando um erro) pois ela foi definida dentro do `namespace` "*Utils*". No começo do código, acrescentaremos `using ByteBank.View.Utils`, tornando a classe identificável para o Visual Studio. Usaremos um parâmetro de função, `PgsProgresso`, seu valor, e o incrementaremos.

```
var inicio = DateTime.Now;

var byteBankProgress = new ByteBankProgress<String>(() =>
  PgsProgresso.Value++);
var resultado = await ConsolidarContas(contas, byteBankProgress);
```

A `Action` acima não possui definição de tipo, e o compilador aponta este erro. No entanto, o parâmetro que recebemos é uma `Action<T>`, em que `T` é uma `string`. Não estamos utilizando o valor de progresso (a `string` da consolidação), porém

somos obrigados a indicá-lo a partir da criação de um *delegate* que faz uso de seu valor. Só o faremos recebendo por parâmetro, sem colocá-lo em nenhuma parte do código.

Vamos executar a aplicação, fazendo o processamento. A barra de progresso está atualizando seu valor, toda a consolidação foi feita, a aplicação está responsiva, e o código, mais elegante, pois abstraímos o controle do `TaskScheduler` da tela, que só faz a consolidação dos clientes.

Ainda podemos utilizar este `ByteBankProgress` outras vezes neste código, uma vez que ele é genérico, recebendo um manipulador genérico por parâmetro, também. Podemos atualizar a lista usando o valor que está sendo recebido, caso queiramos. É possível também usá-lo em outra aplicação. O próximo passo é simplificar ainda mais o código.