
Hydrogen Sessions

#13 – Tasks

© 2020 PRIMAVERA

Tasks, async/sync, await

Este é seguramente o segundo tema mais “exotérico” do .NET atualmente, logo a seguir ao GC.

Isso resulta em parte da sua natureza, porque afinal threading sempre foi um tema extremamente complexo por causa da concorrência e por impactar imenso na performance das aplicações.

Por outro lado, sempre que se adiciona uma abstração para um tema complexo como este, torna-se o assunto mais acessível ao “comum dos mortais”, mas também se criam condições para a abstração ser utilizada sem a mínima noção do seu funcionamento, dos cenários mais complexos, das implicações de cada abordagem, de cada método, de cada parâmetro, etc.

Há inúmeros artigos escritos sobre este tema e documentação muito detalhada. Muitos são contraditórios, a maior parte é incompleta porque se foca em determinado cenário ou determinado comportamento. Nesta área não há regras unanimemente aceites. Não há receitas infalíveis.

No entanto, há cada vez mais código async. Vale a pena pelo menos conhecer um pouco melhor alguns aspetos.

System.Threading.Tasks

O namespace System.Threading.Tasks foi acrescentado ao .NET 4.5 e ao C# 5.0 (Visual Studio 2012).

<https://docs.microsoft.com/en-us/dotnet/api/system.threading.tasks>

Os tipos mais importantes são: Task e Task<TResult>

De forma simplista, uma Task representa uma operação assíncrona pela qual se pode esperar (await) e que pode ser cancelada. Uma task representa um valor (o resultado) que ficará disponível no futuro.

Podendo ser executada assincronamente, uma task é particularmente útil para realizar operações de IO (acesso a ficheiros, acesso a base de dados, acesso à rede, etc.). Nas operações de computação pura (CPU) não existem vantagens tão evidentes em utilizar tasks (a documentação refere as diferenças entre os dois cenários).

Em C# uma task é, normalmente, definida como um método que tem como resultado uma Task ou Task<TResult> e usa a keyword async. É de esperar que tenha pelo menos uma instrução awaited (operador await):

```
2 references | Hugo Lourenço | 1 author, 1 change | 1 incoming change
protected virtual async Task<DiscoveryDocumentResponse> GetDiscoveryDocumentAsync(DiscoveryDocumentRequest discoveryRequest, CancellationToken cancellationToken)
{
    // Execution time

    using (new ExecutionTimer($"{nameof(TokenClient)}.{nameof(TokenClient.GetDiscoveryDocumentAsync)}"))
    {
        // Run discovery client

        using (DiscoveryClient client = new DiscoveryClient(this.HttpMessageHandler, false))
        {
            return await client.GetDocumentAsync(discoveryRequest, cancellationToken).ConfigureAwait(false);
        }
    }
}
```

<https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/keywords/async>

<https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/operators/await>

Referência

Como foi referido antes, há N artigos que podem servir de referência sobre programação assíncrona em .NET e C#.
Estes, pelo menos, são de leitura obrigatória:

- <https://docs.microsoft.com/en-us/dotnet/standard/asynchronous-programming-patterns/task-based-asynchronous-pattern-tap>
- <https://docs.microsoft.com/en-us/dotnet/csharp/async>
- <https://devblogs.microsoft.com/dotnet/configureawait-faq/>

Funcionamento

Este artigo explica o funcionamento do padrão em detalhe:

<https://docs.microsoft.com/en-us/dotnet/standard/async-in-depth>

Vale a pena salientar as vantagens nos cenários I/O bound:

What does this mean for a server scenario?

This model works well with a typical server scenario workload. Because there are no threads dedicated to blocking on unfinished tasks, the server threadpool can service a much higher volume of web requests.

Consider two servers: one that runs async code, and one that does not. For the purpose of this example, each server only has 5 threads available to service requests. Note that these numbers are imaginarily small and serve only in a demonstrative context.

Assume both servers receive 6 concurrent requests. Each request performs an I/O operation. The server *without* async code has to queue up the 6th request until one of the 5 threads have finished the I/O-bound work and written a response. At the point that the 20th request comes in, the server might start to slow down, because the queue is getting too long.

The server *with* async code running on it still queues up the 6th request, but because it uses `async` and `await`, each of its threads are freed up when the I/O-bound work starts, rather than when it finishes. By the time the 20th request comes in, the queue for incoming requests will be far smaller (if it has anything in it at all), and the server won't slow down.

Although this is a contrived example, it works in a very similar fashion in the real world. In fact, you can expect a server to be able to handle an order of magnitude more requests using `async` and `await` than if it were dedicating a thread for each request it receives.

What does this mean for client scenario?

The biggest gain for using `async` and `await` for a client app is an increase in responsiveness. Although you can make an app responsive by spawning threads manually, the act of spawning a thread is an expensive operation relative to just using `async` and `await`. Especially for something like a mobile game, impacting the UI thread as little as possible where I/O is concerned is crucial.

More importantly, because I/O-bound work spends virtually no time on the CPU, dedicating an entire CPU thread to perform barely any useful work would be a poor use of resources.

Additionally, dispatching work to the UI thread (such as updating a UI) is very simple with `async` methods, and does not require extra work (such as calling a thread-safe delegate).

Sync to async

Um dos primeiros problemas que surge neste contexto é do que como executar código async a partir de métodos sync.

Um exemplo clássico é usar o HttpClient – que só tem métodos async – a partir de um método sync.

Há muita teoria sobre este assunto e abordagens distintas: Task.Run(), AsyncHelper, etc.

Como é visível na documentação anterior, o método Task.Run existe essencialmente para suportar o cenário CPU-bound, não para resolver este problema em concreto.

No Hydrogen segue-se sempre a mesma abordagem, que é a seguinte e que, acreditamos, minimiza os problemas de deadlocks (mas não os elimina completamente).

```
/// <summary>
/// Retrieves the current time (UTC) from the NTP server.
/// </summary>
/// <returns>
/// The current time (UTC) retrieved from the NTP server.
/// </returns>
0 references | Hugo Ribeiro, 54 days ago | 1 author, 1 change
public DateTime RetrieveUtcTime()
{
    // Result

    return this.RetrieveUtcTimeAsync()
        .ConfigureAwait(false)
        .GetAwaiter()
        .GetResult();
}
```

async Task Method() vs Task Method()

Outra confusão comum está relacionada com a utilização da keyword `async`.

Os dois métodos seguintes são formalmente semelhantes e estão ambos corretos. Como se pode verificar também na documentação referida antes, a diferença está no momento em que o `await` é executado (e o processamento é “devolvido” ao caller).

```
U references
public async Task<string> RunSomething1()
{
    // (...)

    return await this.RetrieveValueAsync("id").ConfigureAwait(false);
}
```

```
U references
public Task<string> RunSomething2()
{
    // (...)

    return this.RetrieveValueAsync("id");
}
```

ConfigureAwait(false)

Outra questão pertinente tem a ver com a utilização ou não da chamada a `ConfigureAwait(false)`.

Este método controla (ou melhor, permite desligar) a sincronização de contexto entre threads, o que tem inúmeras implicações enumeradas num dos artigos referidos antes.

Importa reter desse artigo 2 dados muito relevantes:


- Em bibliotecas (como o Hydrogen) é seguro assumir sempre `ConfigureAwait(false)` uma vez que isso remete a decisão relevante para o código cliente.
- O `HttpContext` no .NET Core não é estático, o que minimiza imenso os cenários em que a sincronização de contexto (não é o mesmo contexto, claro!) é crucial para manter os dados associados às requests.

Code analysis

O Visual Studio 2019 valida algumas das boas práticas associadas a tasks. Devem ser estritamente observadas!!!


CS1998

```
0 references
public async Task AsyncKeywordRequiresAwait()
{
    // (...)
    if (this.Value != null)
    {
    }
}
```

 (awaitable) Task AsyncSamples.AsyncKeywordRequiresAwait()
Usage:
await AsyncKeywordRequiresAwait();
This async method lacks 'await' operators and will run synchronously. Consider using the 'await' operator to await non-blocking API calls, or 'await Task.Run(...)' to do CPU-bound work on a background thread.
[Show potential fixes \(Alt+Enter or Ctrl+.\)](#)

CA2007

```
0 references
public async Task ConfigureAwaitIsRecommendedByDefault()
{
    string value = await this.RetrieveValueAsync("id");
    if (value != null)
    {
        // (...)
    }
}
```

 (awaitable) Task<string> AsyncSamples.RetrieveValueAsync(string key)
Usage:
string x = await RetrieveValueAsync(...);
Consider calling ConfigureAwait on the awaited task
[Show potential fixes \(Alt+Enter or Ctrl+.\)](#)

Code analysis

As seguintes são adicionadas por analyzers adicionais, configurados por Primavera.Hydrogen.DesignTime.Configuration:

VSTHRD200

```
0 references
public Task AsyncMethodShouldHaveAsyncSuffix()
{
    if (this.Value != null)
    {
    }

    return Task.CompletedTask;
}
```

Usage:
await AsyncMethodShouldHaveAsyncSuffix();
Use "Async" suffix in names of methods that return an awaitable type.
Show potential fixes (Alt+Enter or Ctrl+.)

VSTHRD100

```
0 references
public async void AsyncMethodShouldReturnTask()
{
    string value = await Task.FromResult("value");
    if (value != null)
    {
        // (...)
    }
}
```

Usage:
void AsyncMethodShouldReturnTask();
Avoid "async void" methods, because any exceptions not handled by the method will crash the process.
Show potential fixes (Alt+Enter or Ctrl+.)

VSTHRD110

```
0 references
public void AsyncMethodsShouldAwaited()
{
    this.RetrieveValueAsync("key");
}
```

Usage:
string x = await RetrieveValueAsync(...);
Observe the awaitable result of this method call by awaiting it, assigning to a variable, or passing it to another method.
Show potential fixes (Alt+Enter or Ctrl+.)

CancellationToken

Uma task deve poder ser cancelada. Isso implica que receba um parâmetro para que possa ser cancelada e que reaja devidamente (avaliando o cancellation token).

Os background services dos micro serviços são bons exemplos deste padrão:

```
/// <inheritdoc />
[SuppressMessage("Design", "CA1031:Do not catch general exception types", Justification = "Required by design.")]
0 references
public override async Task ExecuteAsync(CancellationToken cancellationToken)
{
    // Logging (alive)

    this.Logger.LogInformation("The notifications sender worker is alive.");

    // Canceled?

    if (cancellationToken.IsCancellationRequested)
    {
        return;
    }

    // Execute

    try{...
}
}
```

Async + sync methods

Finalmente, é boa prática disponibilizar as duas versões de determinado método, para suportar os dois cenários, quando o caller pode executar código async e quando não pode.

No Hydrogen implementamos na maioria das vezes o caso de disponibilizar um método síncrono para um assíncrono. Mas também pode ser necessário, em alguns casos, o contrário (em benefício da coerência das API).

```
/// <summary>
/// Retrieves the current time (UTC) from the NTP server.
/// </summary>
/// <returns>
/// The current time (UTC) retrieved from the NTP server.
/// </returns>
0 references | Hugo Ribeiro, 54 days ago | 1 author, 1 change
public DateTime RetrieveUtcTime()
{
    // Result

    return this.RetrieveUtcTimeAsync()
        .ConfigureAwait(false)
        .GetAwaiter()
        .GetResult();
}

/// <summary>
/// Retrieves the current time (UTC) from the NTP server.
/// </summary>
/// <returns>
/// The <see cref="Task{TResult}"> that represents the asynchronous operation.
/// The current time (UTC) retrieved from the NTP server.
/// </returns>
6 references | Hugo Ribeiro, 54 days ago | 1 author, 1 change
public async Task<DateTime> RetrieveUtcTimeAsync()
{
    // Retrieve offset

    TimeSpan offset = await this.RetrieveCorrectionOffsetAsync().ConfigureAwait(false);

    // Result

    return DateTime.UtcNow + offset;
}
```

Locking

A livreria `System.Threading.Tasks` não faz qualquer assunção sobre os cenários de concorrência entre threads. O tema de thread-safety continua a ser um assunto próprio, que deve ser devidamente implementado caso a caso.

A principal implicação nesse tema é que não é possível usar o locking tradicional (com a instrução `lock`) dentro de métodos `async`. Há várias formas de atingir o mesmo fim, dependendo do cenário. A mais simples é a seguinte:

```
private static readonly SemaphoreSlim semaphoreSlim = new SemaphoreSlim(1, 1);

public async Task WithThreadSafetyAsync()
{
    await semaphoreSlim.WaitAsync().ConfigureAwait(false);

    try
    {
        string value = await this.RetrieveValueAsync("key").ConfigureAwait(false);
        if (value != null) {...}
    }
    finally
    {
        semaphoreSlim.Release();
    }
}
```

Boas práticas

Eis um resumo das boas práticas:

- Evitar o uso de `Task.Run()`, exceto em cenários muito específicos (CPU-bound), nunca para executar código async a partir de código sync.
- Usar `Task.ConfigurationAwait(false).GetAwaiter().GetResult()` para executar código async a partir de código sync.
- Adicionar sempre a chamada `.ConfigureAwait(false)` (quando a task é awaited).
- Nunca utilizar métodos async com resultado void.
- Nunca deixar métodos com a keyword `async` se não tiverem nenhum `await`.
- Adicionar sempre um parâmetro `CancellationToken` para cancelar a task (exceto nos casos em que esta apenas chama outros métodos async que não aceitem esse parâmetro).
- Adicionar sempre o sufixo “Async” aos métodos async (os que devolvem `Task`, tenham ou não a keyword `async`) (ex: `Get()` => `GetAsync()`).