WhallaLabs

Async & await – best practices

Karol Szmaj
CTO
Whalla Labs

karol.szmaj@whallalabs.com

Agenda - Podstawy

- Jak to było kiedyś
- Podstawowe informacje dotyczące TPL & TAP
- Async vs Sync
- Garbage Collection
- SynchronizationContext & Deadlocks
- Deser
- Q&A

Poprzednie implementacje asynchroniczności

Poprzednie implementacje AP

Asynchronous Programming Model (APM) – BeginMethod + EndMethod

- Znane ze streamów czy socketów

Event-Based Asynchronous Pattern (EAP)

- Windows Forms, stary WebClient (DownloadCompleted)

Powyższe modele nie są rekomendowane w czasach TPL ☺

Task-based Asynchronous Pattern (TAP)

Zbudowane na podstawie Task Parallel Library (TPL)

Dostępny dla każdego projektu, który wykorzystuje .Net 4.5 (Web, Desktop, WINRT, WP, …)

Async&Await

Task -> async & await

Czym jest Task - I

- Opakowanie wobec danego zadania, które zostanie zakończone w przyszłości.
  - obliczenia, żądania HTTP, IO, ...
- Udostępnia status operacji:
  - cancelled, running, completed, faulted, ...
- Dwa warianty: Task oraz Task<T>
- Task<T> posiada **właściwość** reprezentującą rezultat (operacja jest blokująca)

Czym jest Task - II

- Task: fast path
- Dostępne API do zarządzania zadaniami:
  - Task.WhenAll / WhenAny
  - Task.FromResult
  - Task.Run
  - Task.WaitAny/WaitAll (blokujące)

Słowo kluczowe await

- Najczęściej wykorzystywany, aby poczekać za rezultatem danego Taska.
- Zatrzymuje działanie danej funkcji do momentu, kiedy Task zostanie zakończony.
- Kiedy Task zostanie zakończony, to może:
  - rzucić wyjątkiem, jeżeli Task ma status faulted,
  - zwrócić wartość, jeżeli został użyty Task<T>,
  - nic nie zwraca, jeżeli użyliśmy Task.

Słowo kluczowe async

Dodanie **async** do funkcji umożliwia:

- wykorzystanie słowa kluczowego ***await,***
- umożliwia kompilatorowi wygenerowanie maszyny stanowej.

Podstawy

async **void** Example1_ContextSwitching();

- Wykorzystywany w EventHandlerach
- Nie możemy poczekać za rezultatem
- Nie możemy złapać wyjątków

Podstawy

async Task<T> Example1_ContextSwitching();

- Możemy poczekać za rezultatem
- Umożliwia łapanie wyjątków
- Fire&Forget

Wyjątki

Async void:

- Rzucane w wątku dispatcher lub innym jeżeli dispatcher nie istnieje.
- AppDomain.UnhandledException.

Async Task/Task<T>:

- TaskScheduler.UnobservedTaskException.
- Nigdy nie powoduje crasha aplikacji.

Demo

Async vs Sync

Async vs Sync

Wiemy, że metody synchroniczne są „tanie”.

```
0 references
private void ConsoleTest()
{
    Console.WriteLine("coooooooś");
}
```

```
.maxstack  8
IL_0000:  nop
IL_0001:  ldstr      bytearray (63 00 6F 00 6F 00 6F 00 6F 00 6F 00 6F 00 6F 00   // c.o.o.o.o.o.o.o.
                                5B 01 )                                          // [.
IL_0006:  call       void [mscorlib]System.Console::WriteLine(string)
IL_000b:  nop
IL_000c:  ret
```

Async vs Sync

Przykład metody z użyciem async Task.

```
0 references
private async Task ConsoleTestAsync()
{
    Console.WriteLine("coooooooś");
}
```

```
// Code size       62 (0x3e)
.maxstack  2
.locals init ([0] valuetype ContextSwitching.MainWindow/'<
         [1] class [mscorlib]System.Threading.Tasks.Task
         [2] valuetype [mscorlib]System.Runtime.CompilerS
IL_0000:  ldloca.s   V_0
IL_0002:  ldarg.0
IL_0003:  stfld      class ContextSwitching.MainWindow Co
IL_0008:  ldloca.s   V_0
IL_000a:  call       valuetype [mscorlib]System.Runtime.C
IL_000f:  stfld      valuetype [mscorlib]System.Runtime.C
IL_0014:  ldloca.s   V_0
IL_0016:  ldc.i4.m1
IL_0017:  stfld      int32 ContextSwitching.MainWindow/'<
IL_001c:  ldloca.s   V_0
IL_001e:  ldfld      valuetype [mscorlib]System.Runtime.C
IL_0023:  stloc.2
IL_0024:  ldloca.s   V_2
IL_0026:  ldloca.s   V_0
IL_0028:  call       instance void [mscorlib]System.Runti
IL_002d:  ldloca.s   V_0
IL_002f:  ldflda     valuetype [mscorlib]System.Runtime.C
IL_0034:  call       instance class [mscorlib]System.Threa
IL_0039:  stloc.1
IL_003a:  br.s       IL_003c
IL_003c:  ldloc.1
IL_003d:  ret
// end of method MainWindow::ConsoleTestAsync
```

Async vs Sync

```
.try
{
  IL_0000:  ldc.i4.1
  IL_0001:  stloc.0
  IL_0002:  ldarg.0
  IL_0003:  ldfld      int32 ContextSwitching.MainWindow/'<ConsoleTestAsync>d__0'::'<>1__state'
  IL_0008:  stloc.2
  IL_0009:  ldloc.2
  IL_000a:  ldc.i4.s   -3
  IL_000c:  beq.s      IL_0010
  IL_000e:  br.s       IL_0012
  IL_0010:  br.s       IL_0020
  IL_0012:  br.s       IL_0014
  IL_0014:  nop
  IL_0015:  ldstr      bytearray (63 00 6F 00 6F 00 6F 00 6F 00 6F 00 6F 00 6F 00   // c.o.o.o.o.o.o.o.
                                  5B 01 )                                          // [.
  IL_001a:  call       void [mscorlib]System.Console::WriteLine(string)
  IL_001f:  nop
  IL_0020:  leave.s    IL_003a
}  // end .try
catch [mscorlib]System.Exception
{
  IL_0022:  stloc.1
  IL_0023:  ldarg.0
  IL_0024:  ldc.i4.s   -2
  IL_0026:  stfld      int32 ContextSwitching.MainWindow/'<ConsoleTestAsync>d__0'::'<>1__state'
  IL_002b:  ldarg.0
  IL_002c:  ldflda     valuetype [mscorlib]System.Runtime.CompilerServices.AsyncTaskMethodBuilder ContextSwi
  IL_0031:  ldloc.1
  IL_0032:  call       instance void [mscorlib]System.Runtime.CompilerServices.AsyncTaskMethodBuilder::SetEx
  IL_0037:  nop
```

Async overhead

- Blok try/catch
- Wykorzystywanie wielu klas pomocniczych, np. AsyncTaskMethodBuilder, ...
- Dostęp do pól zamiast zmiennych lokalnych (w CLR minimalnie wolniejsze)

Demo

Garbage Collection

Garbage Collection

- .Net jest środowiskiem zarządzalnym
- Alokacja obiektów nie jest kosztowna
- Płacimy za GC ☹
- GC skanuje nasze obiekty kiedy potrzebujemy więcej pamięci
- Więcej obiektów/duże obiekiekty -> częstsze GC
- Cel: unikanie niepotrzebnych alokacji/ dużych obiektów

Alokacje i metody asynchroniczne

```
using ...
[CompilerGenerated]
[StructLayout(LayoutKind.Auto)]
private struct <ConsoleTestAsync>d__0 : IAsyncStateMachine
{
    public int <>1__state;
    public AsyncTaskMethodBuilder <>t__builder;
    public MainWindow <>4__this;
    void IAsyncStateMachine.MoveNext()
    {
        try
        {
            int num = this.<>1__state;
            if (num != -3)
            {
                Console.WriteLine("coooooooś");
            }
        }
        catch (Exception exception)
        {
            this.<>1__state = -2;
            this.<>t__builder.SetException(exception);
            return;
        }
        this.<>1__state = -2;
        this.<>t__builder.SetResult();
    }
    [DebuggerHidden]
    void IAsyncStateMachine.SetStateMachine(IAsyncStateMachine param0)
    {
        this.<>t__builder.SetStateMachine(param0);
    }
}
```

1. Alokacja na stercie
2. Delegat zakończenia

<>t__awaiter1.OnCompleted
(<>t__MoveNextDelegate);

Alokacje i metody asynchroniczne

Metoda asynchroniczna startuje jako synchroniczna… WTF?

Metoda asynchroniczna może działać do końca jako synchroniczna – „await fast path”

```
TResult result = await FooAsync();
```

```
var $awaiter = FooAsync().GetAwaiter();
if (!$awaiter.IsCompleted) {
    SAVE_STATE;
    $awaiter.OnCompleted(CompletionDelegate);
    return;
    Label:
    RESTORE_STATE;
}
TResult result = $awaiter.GetResult();
```

Alokacje i metody asynchroniczne

Wiele metod asynchronicznych zakończy się w sposób synchroniczny.

Pewnych alokacji tasków nie da się uniknąć.

Niektóre z nich możemy „cache'ować".

Task caching

Task cache

MemoryStream.ReadAsync

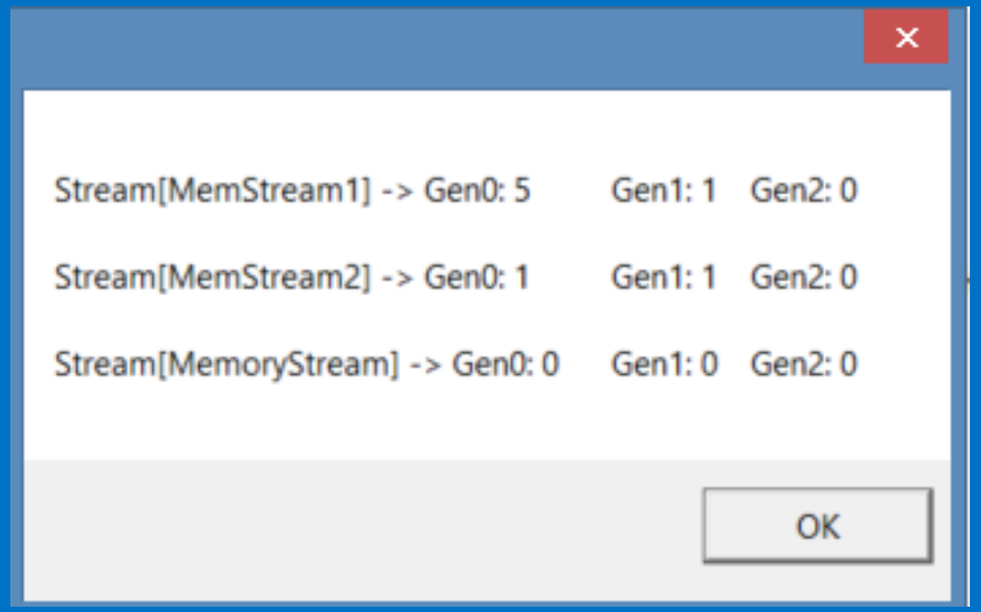
```
class PGNETStream : MemoryStream
{
    public override async Task<int> ReadAsync(byte[] buffer, int offset, int count, CancellationToken cancellationToken)
    {
        cancellationToken.ThrowIfCancellationRequested();
        return Read(buffer, offset, count);
    }
}
```

```
public static async Task CopyStreamAsync(Stream input, Stream output)
{
    var buffer = new byte[1024];
    int readBytes;

    while ((readBytes = await input.ReadAsync(buffer, 0, buffer.Length)) > 0)
    {
        await output.WriteAsync(buffer, 0, readBytes);
    }
}
```

Task cache

```
public override Task<int> ReadAsync(byte[] buffer,
    int offset, int count, CancellationToken cancellationToken)
{
    if (cancellationToken.IsCancellationRequested)
    {
        var tcs = new TaskCompletionSource<int>();
        tcs.SetCanceled();
        return tcs.Task;
    }

    try
    {
        int readBytes = this.Read(buffer, offset, count);
        return _lastTask != null && readBytes == _lastTask.Result
            ? _lastTask
            : (_lastTask = Task.FromResult(readBytes));
    }
    catch (Exception ex)
    {
        var tcs = new TaskCompletionSource<int>();
        tcs.SetException(ex);
        return tcs.Task;
    }
}
```

Cache dla ostatniego taska



Task cache

```
private static ConcurrentDictionary<string,string> s_urlToContents;

public static async Task<string> GetContentsAsync(string url)
{
  string contents;
  if (!s_urlToContents.TryGetValue(url, out contents))
  {
      var response = await new HttpClient().GetAsync(url);
      contents = response.EnsureSuccessStatusCode().Content.ReadAsString();
      s_urlToContents.TryAdd(url, contents);
  }
  return contents;
}
```

Cache dla ostatniego taska

Synchronization Context

SynchronizationContext

Zapewnia powrót do odpowiedniego wątku po wykonaniu operacji.

- WindowsFormsSynchronizationContext
- DispatcherSynchronizationContext
- AspNetSynchronizationContext

SynchronizationContext

„await task;” -> wznawiany jest:

- SynchronizationContext
- TaskScheduler

Kod aplikacji

- Prawie zawsze chcemy wracać do danego kontekstu.

Kod biblioteki

- Prawie nigdy nie chcemy wracać do danego kontekstu ☺

SynchronizationContext

Task.ConfigureAwait(bool);

- Domyślnie ustawiony na true (wracamy do oryginalnego kontekstu)
- W przypadku false nie wracamy do oryginalnego kontekstu

Deadlocks

Kod klienta

```
async void button1_Click(…)
{
    await DoWorkAsync();
}
```

Biblioteka

```
async Task DoWorkAsync()
{
    await Task.Run(…);
    Console.WriteLine("Done task");
}
```

Deadlocks

Kod klienta

```
async void button1_Click(…)
{
    await DoWorkAsync();
}
```

Biblioteka

```
async Task DoWorkAsync()
{
    await Task.Run(…);
    Console.WriteLine("Done task");
}
```

```
async void button1_Click(…)
{
    DoWorkAsync().Wait();
}
```

```
async Task DoWorkAsync()
{
    await Task.Run(…);
    Console.WriteLine("Done task");
}
```

Async Method Builder

Awaiter

Dziękuję za uwagę