Analiza mechanizmów asynchroniczności w C#

Na podstawie dostarczonego kodu

12 grudnia 2025

1 Wstęp

Poniższy dokument analizuje dostarczone fragmenty kodu (`FlightScanner` oraz generatory fraktali) pod kątem wykorzystania wzorców asynchronicznych, wielowątkowości oraz obsługi zadań.

2 Użycie Task oraz Async/Await

Podstawowym wzorcem w nowoczesnym C# jest użycie słów kluczowych `async` i `await` do nieblokowania wątku głównego oraz `Task.Run` do zrzucania obliczeń na pulę wątków.

2.1 Definicja i wywołanie metod asynchronicznych

W klasie `Program` (FlightScanner) widzimy wejście do aplikacji asynchronicznej oraz oczekiwanie na zakończenie metody głównej.

```
public static async Task Main(string[] args)
{
    // ...
    try
    {
        // Oczekiwanie na zakonczenie metody asynchronicznej
        await RunFlightScannerAsync();
    }
    // ...
}
```

Listing 1: Punkt wejścia aplikacji asynchronicznej

2.2 Uruchamianie zadań w tle (CPU-bound)

W klasie `TasksGenerator` wykorzystano `Task.Run`, aby ręcznie zlecić obliczenia (generowanie fraktala) do puli wątków. Jest to klasyczny przykład zrównoleglania zadań obliczeniowych.

```
tasks[t] = Task.Run(() =>
{
    for (var y = startRow; y < endRow; y++)
    {
        // Obliczenia dla konkretnego fragmentu obrazu
        // ...
        var iterations = Calculate(a, b);
        image[x, y] = GetColor(iterations);
    }
});
```

Listing 2: Uzycie Task.Run do zadan obliczeniowych

3 Sekwencje asynchroniczne i współbieżność

Aby efektywnie pobierać dane z wielu źródeł jednocześnie (I/O-bound), w `FlightScanner` zastosowano wzorzec "Scatter-Gather"przy użyciu `Task.WhenAll`.

3.1 Tworzenie listy zadań

Zamiast oczekiwać na każde zapytanie po kolei (co byłoby wolne), kod tworzy listę zadań (`List<Task>`), które są uruchamiane niemal natychmiastowo.

```
var tasks = new List<Task<ProviderResponseDto?>>();

foreach (var provider in providers)
{
    // Dodawanie "goracego" taska do listy - zapytanie juz leci
    tasks.Add(GetFlightsFromProviderAsync(provider, cts.Token,
        progress));
}
```

Listing 3: Inicjalizacja listy zadan asynchronicznych

3.2 Oczekiwanie na wszystkie wyniki (Materializacja)

Metoda `Task.WhenAll` pozwala poczekać, aż wszystkie zapytania HTTP wrócą, i zebrać ich wyniki w jednej tablicy.

```
// Oczekiwanie az wszystkie linie lotnicze odpowiedza
var results = await Task.WhenAll(tasks);

// Przetwarzanie wynikow (LINQ) po zakonczeniu wszystkich zadan
var top10CheapestFlights = results
    .Where(response => response != null && response.Flights != null)
    // ...
```

Listing 4: Agregacja wynikow za pomoca WhenAll

4 Raportowanie postępu (IProgress)

Do informowania interfejsu użytkownika (w tym przypadku konsoli) o postępach wewnątrz metod asynchronicznych użyto interfejsu `IProgress<T>`.

4.1 Inicjalizacja

Obiekt `Progress` jest tworzony w metodzie nadrzędnej i przekazywany w dół.

```
// Definicja akcji, ktora ma sie wykonac przy raporcie (tu: wypisanie
    na konsole)
IProgress<string> progress =
    new Progress<string>(Console.WriteLine);
```

Listing 5: Utworzenie obiektu Progress

4.2 Zgłaszanie postępu

Metody podrzędne wywołują `.Report()`, co powoduje wykonanie delegatu zdefiniowanego wyżej.

```
// Wewnatrz GetFlightsFromProviderAsync
progress.Report($"\t[START] Querying {provider.Name}
    ({provider.Endpoint})");

// ... po wykonaniu zapytania ...

progress.Report($"\t[SUCCESS] {provider.Name} returned
    {providerResponse?.Flights.Count ?? 0} flights.");
```

Listing 6: Raportowanie statusu z glebi metody asynchronicznej

5 Obsługa anulowania (CancellationToken)

Mechanizm `CancellationToken` pozwala na przerwanie długotrwałych operacji (np. gdy użytkownik zrezygnuje lub minie czas oczekiwania).

5.1 Źródło tokenu (CancellationTokenSource)

W `FlightScanner` ustawiono sztywny limit czasu (timeout) dla całej operacji.

```
public const int TimeoutMs = 3000;

// Token zostanie anulowany automatycznie po 3000ms
using var cts = new CancellationTokenSource(TimeoutMs);
```

Listing 7: Utworzenie tokenu z timeoutem

5.2 Przekazywanie tokenu

Token jest przekazywany do metod asynchronicznych, a ostatecznie do bibliotek systemowych (np. `HttpClient`).

```
// Przekazanie tokenu do metody
tasks.Add(GetFlightsFromProviderAsync(provider, cts.Token, progress));

// Wewnatrz metody - przekazanie do HttpClient
// To wlasnie tutaj zostanie rzucony wyjatek, jesli czas minie
var response = await httpClient.GetAsync(provider.Endpoint, ct);
```

Listing 8: Przekazanie tokenu do HttpClient

5.3 Obsługa przerwania (Exception Handling)

Anulowanie zadania manifestuje się rzuceniem wyjątku `OperationCanceledException`, który należy obsłużyć.

```
catch (OperationCanceledException)
{
    progress.Report($"[TIMEOUT] Failed to fetch providers within
        {TimeoutMs}ms.");
    return;
}
```

Listing 9: Obsluga wyjatku anulowania