АСИНХРОННІ ПОТОКИ

Питання 13.3. (глава 3 з книги)

Асинхронні потоки

- Асинхронні потоки – механізм асинхронного отримання кількох елементів даних.
  - Вони будуються на основі асинхронних перелічуваних об’єктів (IAsyncEnumerable<T> - асинхронна версія перелічуваного об’єкта (enumerable), може асинхронно виробляти елементи за вимогою для споживача).
- Стандартного асинхронного механізму з Task<T> достатньо тільки для асинхронної обробки одного значення даних.
  - Після того як Task<T> завершиться, все закінчено; одна задача Task<T> не може надати своїм споживачам більше одного значення T.
  - Навіть якщо T є колекцією, значення може надаватись лише 1 раз.
- При порівнянні Task<T> з асинхронними потоками бачимо, що асинхронні потоки більш схожі на перелічувані об’єкти.
  - IAsyncEnumerator<T> може надавати довільну кількість значень T, по одному за раз.
  - Як і IEnumerator<T>, IAsyncEnumerator<T> може мати необмежену довжину.

Асинхронні потоки

- Коли ваш код перебирає IEnumerable<T>, то блокує кожний елемент із перелічуваного об’єкта.
  - Якщо IEnumerable <T> представляє деяку операцію, пов'язану з вводом-виводом, то код-споживач у підсумку блокується по вводу-виводу.
  - IAsyncEnumerable <T> працює так само, як і IEnumerable <T>, за винятком того, що асинхронно отримує кожен наступний елемент.
- Ніщо не заважає асинхронно повернути колекцію, яка містить більше одного елемента; типовий приклад - Task<List<T>>.
  - Проте async-методи, які повертають List <T>, можуть виконати лише одну команду return; колекція повинна бути заповнена до повернення.
  - Навіть методи, які повертають Task<IEnumerable<T>>, можуть асинхронно повернути перелічуваний об'єкт, але тоді цей об'єкт обробляється синхронно.
- Уявіть, що LINQ-to-Entities містить метод LINQ ToListAsync, який повертає Task <List<T>>.
  - Коли цей метод виконується провайдером LINQ, він повинен взаємодіяти з БД і отримати всі відповідні відповіді до того, як він завершить заповнення списку і поверне його.

Асинхронні потоки

- Принципове обмеження типу Task<IEnumerable<T>> полягає в тому, що він не може повертати елементи по мірі отримування.
  - Якщо повертається колекція, він повинен завантажити всі свої елементи в пам’ять, заповнити колекцію, а потім повернути всю колекцію відразу.
  - Навіть якщо повертається LINQ-запит, він може асинхронно побудувати цей запит, проте після повернення запиту отримання елементів з нього буде відбуватись синхронно.
  - IAsyncEnumerable<T> теж повертає кілька елементів асинхронно, проте відмінність у тому, що IAsyncEnumerable<T> може асинхронно діяти з кожним поверненим елементом.
  - Це справжній асинхронний потік елементів.

Асинхронні потоки та IObservable<T>

- Спостережувані об’єкти є істинним втіленням асинхронних потоків; вони генерують свої сповіщення по одному з повноцінною підтримкою асинхронного генерування (без блокування).
  - Проте паттерн споживання об’єктів для IObservable<T> повністю відрізняється від IAsyncEnumerable<T>.
- Щоб споживати IObservable<T>, код повинен визначити LINQ-подібний запит, через який будуть проходити спостережувані сповіщення, після чого підписатись на спостережуваний об’єкт для запуску потока.
  - При роботі зі спостережуваними об’єктами код спочатку визначає, як буде реагувати на вхідні сповіщення, а потім включає їх (звідси і «реактивність»).
  - З іншого боку, споживання IAsyncEnumerable<T> дуже схоже на споживання IEnumerable<T>, крім асинхронності.

Асинхронні потоки та IObservable<T>

- Також виникає проблема зворотного тиску: всі сповіщення в System.Reactive синхронні, тому відразу ж після того, як сповіщення одного елемента відправляється підписникам, спостережуваний об’єкт продовжує виконання та отримує наступний елемент для публікації, можливо — з повторним викликом API.
  - Якщо споживаючий код використовує потік асинхронно, то спостережуваний об’єкт випередить споживаючий код.
- Зручно вважати, що IObservable<T> працює за принципом проштовхування (push), а IAsyncEnumerable<T> — витягування (pull).
  - Спостережуваний потік проштовхує уведомления коду, но асинхронный поток пассивно позволяет коду (асинхронно) вытягивать данные.
  - Только когда потребляющий код запросит следующий элемент, наблюдаемый поток возобновит выполнение.

Теоретичний приклад

- Багато API отримують параметри offset і limit для створення сторінкової організації результатів.
  - Нехай потрібно визначити метод, який отримує результати від API з підтримкою сторінкової організації, і метод повинен обробляти сторінки, щоб цим не доводилось займатись нашим низькорівневим методам.
- Якщо метод повертає Task<T>, ви обмежені поверненням лише одного T.
  - Це нормально для одного виклику API, результатом якого є T, проте він не буде погано працювати в якості вихідного типу, якщо бажано, щоб метод викликав API кілька разів.
- Якщо метод повертає IEnumerable<T>, можна створити цикл, що перебирає результати API по сторінках, викликаючи його кілька разів.
  - Кожний раз, коли метод звертається з викликом до API, він використовує yield return з результатами цієї сторінки.
  - Подальші виклики API необхідні лише тоді, коли перелічення продовжується.
  - На жаль, методы, які повертають IEnumerable<T>, не можуть бути асинхронными, тому всі виклики API змушені бути синхронними.

Теоретичний приклад

- Якщо метод повертає Task<List<T>>, можна створити цикл, який по сторінках перебирає результати API та викликає API асинхронно.
  - Тим не менш, код не може повертати кожний елемент при отриманні відгуку; йому доведеться побудувати всі результати і повернути їх одночасно.
- Якщо ваш метод повертає IObservable<T>, ви зможете використати System.Reactive для реалізації спостережуваного потоку, який починає запити при підписці та публікує кожний елемент при отриманні.
  - Абстракція працює за принципом виштовхування; для споживаючого коду все виглядає так, наче результати API проштовхуються ним, що дещо ускладнює обробку.
  - IObservable<T> буде більш доречним для таких сценаріїв, як отримання та реакція на повідомлення WebSocket/SignalR.
- Якщо ваш метод повертає IAsyncEnumerable<T>, можна створити природній цикл, що використовує як await, так і yield return для створення справжнього асинхронного потоку на базі витягування.
  - IAsyncEnumerable<T> чудово підходить для таких сценаріїв.

Зведення різних ролей розповсюджених типів

Тип	Одно или несколько значений	Асинхронно или синхронно	Вытягивание или проталкивание
`T`	Одно	Синхронно	–
`IEnumerable<T>`	Несколько	Синхронно	–
`Task<T>`	Одно	Асинхронно	Вытягивание
`IAsyncEnumerable<T>`	Несколько	Асинхронно	Вытягивание
`IObservable<T>`	Одно или несколько	Асинхронно	Проталкивание

Створення асинхронних потоків

- ***Завдання:*** потрібно повернути кілька значень, при цьому кожне значення може вимагати деякої асинхронної роботи. Задачу можна вирішити одним з двох способів:
  - Існує кілька вихідних значень (наприклад, IEnumerable<T>), а потім слід виконати певну асинхронну роботу.
  - Існує одне асинхронне повернення (як Task<T>), після якого додаються інші вихідні значення.
- ***Вирішення:*** повернення кількох значень із методу може здійснюватись командою yield return, а асинхронні методи використовують async-await.
  - З асинхронними потоками можна об'єднати ці 2 підходи; використовуйте вихідний тип IAsyncEnumerable<T>:

```
async IAsyncEnumerable<int> GetValuesAsync()
{
    await Task.Delay(1000); // some asynchronous work
    yield return 10;
    await Task.Delay(1000); // more asynchronous work
    yield return 13;
}
```

  - Приклад демонструє, як await може використовуватись разом з yield return для створення асинхронного потоку.

Створення асинхронних потоків

```
async IAsyncEnumerable<string> GetValuesAsync(HttpClient client)
{
    int offset = 0;
    const int limit = 10;
    while (true)
    {
        // Get the current page of results and parse them
        string result = await client.GetStringAsync(
            $"https://example.com/api/values?offset={offset}&limit={limit}");
        string[] valuesOnThisPage = result.Split('\n');

        // Produce the results for this page
        foreach (string value in valuesOnThisPage)
            yield return value;

        // If this is the last page, we're done
        if (valuesOnThisPage.Length != limit)
            break;

        // Otherwise, proceed to the next page
        offset += limit;
    }
}
```

- В іншому, більш реалістичному прикладі асинхронно перебираються результати API, який використовує параметри для сторінкової організації результатів.
- Коли метод GetValuesAsync починає роботу, він видає асинхронний запит першої сторінки даних, після чого виробляє перший елемент.
  - Коли буде запитано другий елемент, GetValuesAsync видає його негайно, тому що він міститься на тій же першій сторінці даних.
  - Наступний елемент також знаходиться на цій сторінці… і т. д. до 10 елементів.
  - Потім при запиті 11-го елемента були утворені всі значення в valuesOnThisPage, а на першій сторінці елементів уже не лишилось.
  - GetValuesAsync продовжить виконання свого циклу while, перейде до наступної сторінки, виконає асинхронний запит 2ї сторінки даних, отримає назад нову групу значень, після чого виробить 11-й елемент.

Створення асинхронних потоків

- З моменту появи async-await задаються питання, як їх використовувати з yield return.
  - Довго це було неможливо, проте асинхронні потоки ввели цю можливість у C# і сучасні версії .NET.
  - У більш реалістичному прикладі варто звернути увагу на одну особливість: асинхронна робота потрібна не для всіх результатів.
  - З довжиною сторінки 10 лише приблизно кожному 10му елементу потрібна асинхронна робота.
- Це звичайний паттерн з асинхронними потоками.
  - Для багатьох потоків більшість операцій асинхронного перебору насправді синхронна; асинхронні потоки тільки дозволяють асинхронно отримати довільний наступний елемент.
  - Асинхронні потоки проєктувались з урахуванням як асинхронного, так і синхронного коду; тому асинхронні потоки будуються на основі ValueTask<T>.
  - Використовуючи ValueTask<T> у внутрішній реалізації, асинхронні потоки максимізують свою ефективність як при синхронному, так і при асинхронному отриманні елементів.

Споживання асинхронних потоків

- ***Завдання:*** потрібно обробити результати асинхронного потоку, який також називають асинхронним перелічуваним об'єктом.
- ***Вирішення:*** споживання асинхронної операції здійснюється ключовим словом await, а для споживання перелічуваного об'єкта зазвичай використовується foreach.
  - Споживання асинхронного перелічуваного об'єкта базується на об'єднанні цих конструкцій у await foreach.
  - Наприклад, для асинхронного перелічуваного об'єкта, який видає відповіді API по сторінках, можна організувати споживання та вивести кожний елемент на консоль:

```
IAsyncEnumerable<string> GetValuesAsync(HttpClient client);

public async Task ProcessValueAsync(HttpClient client)
{
    await foreach (string value in GetValuesAsync(client)) {
        Console.WriteLine(value);
    }
}
```

  - На концептуальному рівні викликається метод GetValuesAsync(), який повертає IAsyncEnumerable<T>.
  - Потім цикл foreach створює асинхронний перелічувач на основі асинхронного перелічуваного об'єкта.

Споживання асинхронних потоків

- Асинхронні нумератори на логічному рівні схожі на звичайні нумератори, не враховуючи того, що операція «отримати наступний елемент» може бути асинхронною.
  - Таким чином, await foreach чекатиме надходження наступного елемента або завершення асинхронного нумератора.
  - Якщо елемент надійшов, то await foreach виконає своє тіло циклу; якщо асинхронний нумератор завершено, відбувається вихід з циклу.

- Також можна виконати асинхронну обробку кожного елемента:

```
IAsyncEnumerable<string> GetValuesAsync(HttpClient client);

public async Task ProcessValueAsync(HttpClient client) {
    await foreach (string value in GetValuesAsync(client)) {
        await Task.Delay(100); // asynchronous work
        Console.WriteLine(value);
    }
}
```

  - Тут await foreach не переходить до наступного елемента до завершення тіла циклу.
  - Таким чином, await foreach асинхронно отримає перший елемент, після чого асинхронно виконує тіло циклу для першого елемента, потім асинхронно отримує перший елемент, асинхронно виконує тіло циклу для наступного елемента і т. д.

Споживання асинхронних потоків

- В await foreach прихована команда await: до операції «отримати наступний елемент» застосовується await.
  - Зі звичайною командою await можна обійти неявно збережений контекст за допомогою ConfigureAwait(false).
  - Асинхронні потоки також підтримують ConfigureAwait(false), які передаються прихованим командам await:

```
IAsyncEnumerable<string> GetValuesAsync(HttpClient client);

public async Task ProcessValueAsync(HttpClient client)
{
    await foreach (string value in GetValuesAsync(client).ConfigureAwait(false))
    {
        await Task.Delay(100).ConfigureAwait(false); // asynchronous work
        Console.WriteLine(value);
    }
}
```

  - await foreach – найбільш логічний спосіб споживання асинхронних потоків: мова підтримує ConfigureAwait(false) для запобігання контексту в await foreach.
  - Також можливий варіант з передачею маркерів скасування; цей варіант трохи складніше через складність асинхронних потоків.
  - Хоч природно використовувати await foreach для споживання асинхронних потоків, є велика бібліотека асинхронних операторів LINQ

Споживання асинхронних потоків

- Тіло await foreach може бути як синхронним, так і асинхронним.
  - Для асинхронного випадку правильно реалізувати його набагато складніше, ніж з іншими потоковими абстракціями (наприклад, IObservable<T>).
  - Це пояснюється тим, що спостережувані підписки повинні бути синхронними, але await foreach допускає природну асинхронну обробку.
- Конструкція await foreach генерує команду await, що використовується для операції «отримати наступний елемент»; вона також генерує команду await, яка застосовується для асинхронного звільнення перелічуваного об'єкта.

Використання LINQ з асинхронними потоками

- ***Завдання:*** потрібно обробити асинхронний потік, використовуючи чітко визначені та добре відтестовані оператори.
- ***Вирішення:*** IEnumerable<T> підтримує LINQ to Objects, а IObservable<T> підтримує LINQ to Events.
  - Обидва типи підтримують бібліотеки методів розширення, які визначають оператори, які використовуються для побудови запитів.
  - IAsyncEnumerable <T> також включає підтримку LINQ, що надається спільнотою .NET в NuGet-пакеті System.Linq.Async.
- Наприклад, одне з найпоширеніших питань про LINQ полягає в тому, як використовувати оператор Where, якщо предикат Where є асинхронним.
  - Ви хочете відфільтрувати послідовність на підставі деякої асинхронної умови – наприклад, необхідно провести пошук кожного елемента в БД або API, щоб дізнатися, чи повинен він бути включений в підсумкову послідовність.
  - Where не працює з асинхронними умовами, тому що оператор Where вимагає, щоб його делегат повертав негайну синхронну відповідь.

У асинхронних потоков є допоміжна бібліотека з багатьма корисними операціями

```
IAsyncEnumerable<int> values = SlowRange().WhereAwait(
    async value => {
          // do some asynchronous work to determine
          //  if this element should be included
          await Task.Delay(10);
              return value % 2 == 0;
    }
);

await foreach (int result in values)
{
    Console.WriteLine(result);
}

// Produce sequence that slows down as it progresses
async IAsyncEnumerable<int> SlowRange()
{
    for (int i = 0; i != 10; ++i)
    {
        await Task.Delay(i * 100);
        yield return i;
    }
}
```

- У прикладі WhereAwait є правильним рішенням.
- Оператори LINQ для асинхронних потоків також включають синхронні версії; має сенс застосувати синхронну операцію Where (Select і т. д.) для асинхронного потоку.
  - Результат все-одно є асинхронним потоком:

```
IAsyncEnumerable<int> values = SlowRange().Where(
            value => value % 2 == 0);

await foreach (int result in values)
{
    Console.WriteLine(result);
}
```

  - Багато операторів LINQ тепер можуть отримувати асинхронні делегати

Використання LINQ з асинхронними потоками

- Асинхронні потоки працюють за принципом витягування, тому тут немає операторів, пов'язаних з часом (як для спостережуваних об'єктів).
  - Throttle і Sample тут не мать смислу, оскільки елементи витягуються з асинхронного потоку за вимогою.
- Методи LINQ для асинхронних потоків також можуть принести користь для звичайних перелічуваних об'єктів.
  - Опинившись в цій ситуації, можна викликати ToAsyncEnumerable () для будь-якого IEnumerable <T>;
  - тоді ви отримаєте інтерфейс асинхронного потоку, який можна використовувати з WhereAwait, SelectAwait і іншими операторами, які підтримують асинхронних делегатів.
- Кілька слів щодо назв.
  - Приклад в цьому рецепті використовує WhereAwait як асинхронний еквівалент Where.
  - При вивченні операторів LINQ для асинхронних потоків ви побачите, що одні з них закінчуються суфіксом Async, а інші - суфіксом Await.
  - Оператори, що закінчуються суфіксом Async, повертають об'єкт, що допускає очікування; вони представляють звичайне значення, а не асинхронну послідовність.
  - Оператори з суфіксом Await отримують асинхронний делегат; Await в імені має на увазі, що вони фактично виконують await з переданим їм делегатом.

Використання LINQ з асинхронними потоками

- Суфікс Async застосовується тільки до операторів термінації (termination operators) - операторам, які витягують деяке значення або виконують деякі обчислення і повертають асинхронне скалярне значення замість асинхронної послідовності.
  - Приклад такого оператора - CountAsync, версія Count для асинхронного потоку, яка може підрахувати кількість елементів, відповідна деякому предикату:

```
int count = await SlowRange().CountAsync(
            value => value % 2 == 0);
```

  - Предикат може також бути асинхронним; тоді використовується оператор CountAwaitAsync, оскільки він отримує асинхронний делегат (який буде використовуватися з await) і виробляє одне термінальне значення:

```
int count = await SlowRange().CountAwaitAsync(
    async value =>
    {
        await Task.Delay(10);
        return value % 2 == 0;
    });
```

  - Оператори, які можуть отримувати делегати, існують в двох іменах: з суфіксом Await і без нього.
  - Крім того, оператори, які повертають термінальне значення замість асинхронного потоку, завершуються суфіксом Async.
  - Якщо оператор отримує асинхронного делегата і повертає термінальне значення, то має обидва суфікси.

Асинхронні потоки та скасування

```
async Task Test()
{
    await foreach (int result in SlowRange())
    {
        Console.WriteLine(result);
        if (result >= 8)
            break;
    }

    // Produce sequence that slows down as it progresses
    async IAsyncEnumerable<int> SlowRange()
    {
        for (int i = 0; i != 10; ++i)
        {
            await Task.Delay(i * 100);
            yield return i;
        }
    }
}
```

- ***Завдання:*** потрібний механізм скасування асинхронних потоків.
- ***Вирішення:*** не всім асинхронним потокам необхідне скасування.
  - Перелічення може бути просто зупинене при виконанні умови.
  - Якщо це єдиний реально необхідний різновид «скасування» в програмі, то повноцінне скасування не потрібне, як показує приклад.
- Часто корисно скасовувати асинхронні потоки, оскільки деякі оператори передають маркери скасування своїм потокам-джерелам.
  - У такому випадку слід використовувати CancellationToken для зупинки await foreach із зовнішнього коду

Асинхронні потоки та скасування

```
async Task Test2()
{
    using var cts = new CancellationTokenSource(500);
    CancellationToken token = cts.Token;
    await foreach (int result in SlowRange(token))
    {
        Console.WriteLine(result);
    }
}

// Produce sequence that slows down as it progresses
async IAsyncEnumerable<int> SlowRange(
    [EnumeratorCancellation] CancellationToken token = default)
{
    for (int i = 0; i != 10; ++i)
    {
        await Task.Delay(i * 100, token);
        yield return i;
    }
}
```

- async-метод, що повертає IAsyncEnumerable<T>, може отримати маркер скасування, для чого визначається параметр, відмічений атрибутом EnumeratorCancellation.
  - Після цього маркер можна використати природним чином, для чого він зазвичай передається іншим API, що отримують маркери скасування.
  - У прикладі CancellationToken передається безпосередньо методу, який повертає асинхронний перелічувач.
  - Це найбільш розповсюджений варіант використання.

Асинхронні потоки та скасування

```
async Task ConsumeSequence(IAsyncEnumerable<int> items)
{
    using var cts = new CancellationTokenSource(500);
    CancellationToken token = cts.Token;
    await foreach (int result in items.WithCancellation(token))
    {
        Console.WriteLine(result);
    }
}

// Produce sequence that slows down as it progresses
async IAsyncEnumerable<int> SlowRange(
    [EnumeratorCancellation] CancellationToken token = default)
{
    for (int i = 0; i != 10; ++i)
    {
        await Task.Delay(i * 100, token);
        yield return i;
    }
}

async Task Test() => await ConsumeSequence(SlowRange());
```

- Можливі інші сценарії, коли код отримує асинхроний перелічувач і хоче застосувати CancellationToken до перелічувачів, які він використовує.
  - Маркери скасування використовуються призапуску нового перелічення для перелічуваного об’єкта, тому є сенс саме так використовувати CancellationToken.
  - Сам перелічуваний об'єкт визначається методом SlowRange(), але він не запускається до моменту споживання.
- Бувають навіть ситуації, в яких різні маркери скасування повинні передаватися різним переліченням перелічуваного об'єкта.
  - Коротко, скасовуватися може не перелічуваний об'єкт, а перелічувач, створений ним.
  - Це нетиповий, але важливий сценарій використання; саме з цієї причини асинхронні потоки підтримують метод розширення WithCancellation, який може використовуватися для приєднання маркера CancellationToken до конкретної ітерації асинхронного потоку.
  - При наявності атрибута EnumeratorCancellation компілятор забезпечить передачу маркера з WithCancellation параметру token, позначеного EnumeratorCancellation, і запит скасування тепер змусить await foreach видати виняток OperationCanceledException після того, як він обробить кілька перших елементів.

Асинхронні потоки та скасування

- Метод розширення WithCancellation не перешкоджає ConfigureAwait(false).
  - Обидва методи розширення можуть бути об’єднані в ланцюг:

```
async Task ConsumeSequence(IAsyncEnumerable<int> items)
{
    using var cts = new CancellationTokenSource(500);
    CancellationToken token = cts.Token;
    await foreach (int result in items
        .WithCancellation(token).ConfigureAwait(false))
    {
        Console.WriteLine(result);
    }
}
```

ДЯКУЮ ЗА УВАГУ!

Наступна тема: Конструювання графічного інтерфейсу користувача на базі технології Windows Forms