ОБРАЗОВАТЕЛЬНЫЙ ЦЕНТР
ПРОГРАММИРОВАНИЯ И ВЫСОКИХ ТЕХНОЛОГИЙ

2020г.

Основы асинхронного программирования

AGENDA

- ✓ Понятия асинхронности и параллельности
- ✓ Паттерн EAP
- ✓ Паттерн TAP
- ✓ Пул потоков
- ✓ Конструкция async-await
- ✓ Контекст синхронизации

Понятия асинхронности и параллельности

Есть несколько разных понятий, связанных с областью параллельных вычислений.

Конкурентность (concurrency) – это общий термин, который говорит, что одновременно выполняется более одной задачи.

Конкурентное исполнение не говорит о том, каким образом эта конкурентность будет получена: путем приостановки некоторых вычислительных элементов и их переключение на другую задачу, путем действительно одновременного исполнения, путем делегации работы другим устройствам или еще как-либо.

Конкурентное исполнение говорит о том, что за определенный промежуток времени будет решена более, чем одна задача.

Параллельное исполнение (parallel computing) – подразумевает наличие более одного вычислительного устройства (например, процессора), которые будут одновременно выполнять несколько задач.

Понятия асинхронности и параллельности

Многопоточное исполнение (multithreading) – это один из способов реализации конкурентного исполнения путем выделения абстракции **«потока» (thread).**

Потоки «абстрагируют» от пользователя низкоуровневые детали и позволяют выполнять более чем одну работу «параллельно». Операционная система, среда исполнения или библиотека прячет подробности того, будет многопоточное исполнение конкурентным (когда потоков больше чем физических процессоров), или параллельным (когда число потоков меньше или равно числу процессоров и несколько задач физически выполняются одновременно).

Асинхронное исполнение (asynchrony) – подразумевает инициацию некоторой операцию, об окончании которой главный поток узнает спустя некоторое время. Такая операция может быть выполнена кем-то на стороне: удаленным веб-узлом, сервером или другим устройством за пределами текущего вычислительного устройства.

Основное свойство таких операций в том, что начало такой операции требует значительно меньшего времени, чем основная работа. Что позволяет выполнять множество асинхронных операций одновременно даже на устройстве с небольшим числом вычислительных устройств.

Понятия асинхронности и параллельности

Асинхронность говорит о порядке исполнения кода. Если вызываемая функция не возвращает значение сразу, а отдаёт управление вызывающему, то эта функция асинхронная. При этом нет никаких предположений о том, как эта операция будет считаться: параллельно или нет.

Параллельность говорит о том, что в машине физически происходит несколько процессов одновременно.

Другими словами: **Асинхронность** – логическая оптимизация выполнения, которая может работать и в одном, и во многих потоках.

На практике, эти понятия пересекаются очень часто, потому что асинхронные вызовы делаются как раз для того, чтобы исполнять функции параллельно с основным кодом. То есть, асинхронные вызовы – самый распространённый способ управления распараллеливанием на уровне кода.

Паттерны асинхронного программирования

Выделают следующие паттерны асинхронного программирования:

- **Asynchronous Programming Model (APM)** – асинхронная модель программирования. Основана на 2 методах. Метод **BeginMethodName** возвращает интерфейс **IAsyncResult**. Метод **EndMethodName** принимает **IAsyncResult** (если к моменту вызова **EndMethodName** операция не завершена, поток блокируется).

- **Event-based Asynchronous Pattern (EAP)** – асинхронный шаблон, этот поход основан на событиях, которые срабатывают по завершении операции и обычного метода, вызывающего эту операцию.

- **Task-based Asynchronous Pattern (TAP)** – асинхронный шаблон, этот поход основан на задачах, использующих типы **Task** и **Task<TResult>**. В настоящий момент характеризуется конструкцией **async-await.**

Asynchronous Programming Model (APM)

Модель асинхронного программирования зависит от следующего:

- Делегат, представляющий адрес вызываемой функции.
- Разделение функции вызова на 2 части или пару начало/конец.

```
private delegate int DelegateHandler();
private DelegateHandler myDelegate;

public IAsyncResult BeginMethod()
{
   myDelegate = this.Method;
   return myDelegate.BeginInvoke();
}
public int EndMethod (IAsyncResult result)
{
   return myDelegate.EndInvoke(result);
}
```

Event-based Asynchronous Pattern (EAP)

Идея **EAP** состоит в том, чтобы зарегистрироваться для события и дождаться обратного вызова от обработки/асинхронного кода.

- Метод должен выполняться асинхронно.
- Он должен вызывать событие, когда выполнение подходит к концу.

```
public class EAP_Pattern
{
    public event Action<int> OnTriggerExecuted;

    public void Start()
    {
        // Блок кода
        OnTriggerExecuted?.Invoke(result);
    }
}
```

Task-based Asynchronous Pattern (TAP)

Идея **TAP** основана на типах **Task** и **Task<T>,** которые используются для представления произвольных асинхронных операций.

- Чтобы возвращаемое значение было **Task**, **Task<T>** или **void** .
- Чтобы метод был помечен ключевым словом **async**, а внутри содержал **await**.
- Для приличия соблюдать конвенцию о суффиксе **Async**.

```
public async Task<int> StartAsync()
{
   var task = Task.Run<int>(() =>
      {
         // Блок кода
      });
   // Блок кода
   return await task;
}
```

Пулл потоков

Класс **System.Threading.ThreadPool** обеспечивает приложение пулом рабочих потоков, управляемых системой, позволяя пользователю сосредоточиться на выполнении задач приложения, а не на управлении потоками. Если имеются небольшие задачи, которые требуют фоновой обработки, пул управляемых потоков – это самый простой способ воспользоваться преимуществами нескольких потоков.

Чтобы запросить поток из пула для обработки вызова метода, можно использовать метод **QueueUserWorkItem().**

- Пул потоков управляет потоками эффективно, уменьшая количество создаваемых, запускаемых и останавливаемых потоков.
- Используя пул потоков, можно сосредоточиться на решении задачи, а не на инфраструктуре потоков приложения.

Конструкция async-await

Ключевыми для работы с асинхронными вызовами в **C#** являются два ключевых слова: **async** и **await**, цель которых – упростить написание асинхронного кода. Они используются вместе для создания асинхронного метода, обладающего следующими признаками:

- В заголовке метода используется модификатор **async.**
- Метод содержит одно или несколько выражений **await.**

В качестве возвращаемого типа используется один из следующих:

- **void** – асинхронный метод ничего не возвращает.
- **Task** – возвращает объект типа **Task**.
- **Task<T>** – возвращает некоторое значение, которое оборачивается в объект **Task**.
- **ValueTask<T>** – аналогично **Task<T>**, за исключением различий в работе с памятью, поскольку**ValueTask** – структура, а **Task** – класс.

Конструкция async-await

Асинхронный метод, как и обычный, может использовать любое количество параметров или не использовать их вообще. Однако асинхронный метод не может определять параметры с модификаторами **out** и **ref**.

Также стоит отметить, что слово **async**, которое указывается в определении метода, не делает автоматически метод асинхронным. Оно лишь указывает, что данный метод может содержать одно или несколько выражений **await**.

```
async void WriteAsync(string s)
{
   using (var writer = new StreamWriter(“text.txt", false))
   {
      await writer.WriteLineAsync(s);
   }
}
```

Конструкция async-await

Рекомендуемый способ	Нерекомендуемый способ	Задача
await	Task.Wait или Task.Result	Получение результата фоновой задачи
await Task.WhenAny	Task.WaitAny	Ожидание завершения выполнения любой задачи
await Task.WhenAll	Task.WaitAll	Ожидание завершения выполнения всех задач
await Task.Delay	Thread.Sleep	Ожидание в течение заданного времени

Контекст синхронизации

Контекст синхронизации **SynchronizationContext** представляет собой абстракцию, определяющую расположение, в котором выполняется код вашего приложения.

В каждом потоке есть своя сущность **SynchronizationContext** ассоциированная с этим потоком. Она находится в поле **SynchronizationContext.Current**.

Отключение контекста синхранизации:

Метод **ConfigureAwait(),** настраивает задачу так, чтобы продолжение после **await** не должно было выполняться в контексте вызывающего объекта.

Задание

1. Создайте Console Application. Создайте метод, который будет принимать string и int -> смысл в том, что метод будет выводить на консоль строку столько раз, какое значение имеет int. После каждого вывода на консоль, поток должен засыпать на 100ms. Из Main параллельно вызовете данный метод 2 раза.
2. Сделайте так, чтобы методы вызывались асинхронно, но последующий метод должен жожидаться выполнения предыдущего.