Параллелизм и асинхронность в C#

Параллелизм – выполнение программой нескольких активностей одновременно

Сценарии применения параллелизма

- Написание отзывчивых пользовательских интерфейсов.
- Обеспечение одновременной обработки запросов.
- Параллельное программирование.
- Упреждающее выполнение.

Разделяемое состояние

В **многопоточной** программе внутри единственного процесса запускается множество потоков, разделяя одну и ту же среду выполнения (скажем, память).

Например, один поток может извлекать данные в фоновом режиме, в то время как другой поток — отображать их по мере поступления. Такие данные называются **разделяемым состоянием** (shared state)

Клиентская программа (консольная, WPF, UWP или Windows Forms) запускается в единственном потоке, который создается автоматически операционной системой («главный» поток). Здесь он и будет существовать как **однопоточное** приложение.

Но такое приложение – однопоточное только условно, т.к. «за кулисами» CLR создает другие потоки, предназначенные для сборки мусора и финализации.

Примечание: в приложениях UWP нельзя создавать и запускать потоки напрямую; взамен это должно делаться через Task.

Создание потока (Thread)

```
using System;
using System.Threading;

class ThreadTest
{
   static void Main()
   {
      Thread t = new Thread(WriteY);          // Начать новый поток,
      t.Start();                              // выполняющий WriteY.
      // Одновременно делать что-то в главном потоке,
      for (int i = 0; i < 1000; i++)
       Console.Write ("x");
   }

   static void WriteY
   {
      for (int i = 0; i < 1000; i++)
            Console.Write("y");
   }
}
```

// Типичный вывод:

xxxxxxxxxxxxxxxxyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyy

xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxyyyyyyyyyyyyy

yyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyxxxxxxxxxxxxxxxxxxxxxxyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyy

yyyyyyyyyyyyyxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx

Ожидание окончания выполнения потока

С помощью вызова метода **Join** можно организовать ожидание окончания другого потока:

```
static void Main()
{
    Thread t = new Thread(Go);
    t.Start();
    t.Join();         // Можно задать тайм-аут
    Console.WriteLine("Thread t has ended!");    // Поток t завершен!
}

static void Go()
{
    for (int i = 0; i < 1000; i++)
        Console.Write("y");
}
```

Метод Thread.Sleep приостанавливает текущий поток на заданный период:

```
static void Main()
{
    Thread.Sleep(500);                                  // Ожидать 500 миллисекунд
    Console.WriteLine("500 ms elapsed!");
}
```

На время ожидания Sleep или Join поток **блокируется**.

Локальное или разделяемое состояние

```
class ThreadTest
{
    static bool _done;                          // Есть вероятность вывода "Done” дважды
    static void Main()
    {
        new Thread(Go).Start();
        Go();
    }

    static void Go()
    {
        if (!_done)
        {
                Console.WriteLine("Done");
                _done = true;
        }
    }
}
```

Монопольная блокировка на период чтения и записи разделяемого поля

```
class ThreadSafe
{
    static bool _done;
    static readonly object _locker = new object();

    static void Main()
    {
        new Thread(Go).Start();
        Go();
    }

    static void Go()
    {
        lock (_locker)
        {
            if (!_done) { Console.WriteLine ("Done"); _done = true; }
        }
    }
}
```

Потоки переднего плана и фоновые потоки

Потоки **переднего плана** удерживают приложение в активном состоянии до тех пор, пока хотя бы один из них выполняется, но **фоновые потоки** этого не делают.

Как только все потоки переднего плана завершают свою работу, заканчивается и приложение, а любые все еще выполняющиеся фоновые потоки будут принудительно прекращены.

По умолчанию потоки (Thread), создаваемые явно, являются потоками переднего плана.

Выяснить либо изменить фоновое состояние потока можно с использованием его свойства IsBackground:

```
static void Main (string[] args)
{
    Thread worker = new Thread ( () => Console .ReadLine () );
    if (args.Length > 0)
      worker.IsBackground = true;
    worker.Start();
}
```

Состояние переднего плана или фоновое состояние потока не имеет никакого отношения к его приоритету (выделению времени на выполнение).

Задачи (Task)

Минусы потоков Thread:

- Не существует простого способа получить “возвращаемое значение” обратно из потока.
- После завершения потоку нельзя сообщить о том, что необходимо запустить что-нибудь еще.
- Высокая зависимость от ручной синхронизации (блокировки, выдачи сигналов и т.д.)
- Сложности с отловом исключений, выброшенных в потоках

Класс **Task** помогает решить все упомянутые проблемы.

В сравнении с потоком тип Task — абстракция более высокого уровня, т.к. он представляет параллельную операцию, которая может быть или не быть подкреплена потоком.

Плюсы Task:

- Задачи поддерживают возможность композиции (их можно соединять вместе с использованием продолжения)
- Они могут работать с пулом потоков в целях снижения задержки во время запуска
- Можем получить возвращаемое значение через подкласс Task<TResult>
- Исключение автоматически повторно сгенерируется при вызове метода Wait или доступе к свойству Result

По умолчанию задачи используют **потоки из пула**, которые являются фоновыми потоками.

Это означает, что когда главный поток завершается, то завершаются и любые созданные вами задачи.

Запуск задачи похож на запуск потока:

```
Task.Run (() => Console.WriteLine (“Hello"));
```

Вызов метода Wait на объекте задачи приводит к блокированию до тех пор, пока
она не будет завершена, и эквивалентен вызову метода Join на объекте потока:

```
Task task = Task.Run ( () =>
{
    Thread.Sleep (2000);
    Console.WriteLine ("Foo");
});
Console.WriteLine (task.IsCompleted);        // False
task.Wait();                                 // Блокируется вплоть до завершения задачи
```

Асинхронное программирование

Принцип асинхронного программирования состоит в том, что длительно выполняющиеся (или потенциально длительно выполняющиеся) функции запускаются **асинхронно**.

Применение Task позволяет минимизировать количество кода, запущенного асинхронно. Мы можем точечно запускать асинхронно методы нижней части графа вызовов, которые непосредственно выполняют вычисления. Это повышает безопасность потоков.

В результате получается **мелкомодульный параллелизм** — последовательность небольших параллельных операций, между которыми выполнение возвращается в поток пользовательского интерфейса.

Ключевое слово await

Приведенные ниже строки:

```
var результат = await выражение;
оператор(ы) ;
```

компилятор развернет в следующий функциональный эквивалент:

```
var awaiter = выражение.GetAwaiter();
awaiter.OnCompleted(() =>
{
    var результат = awaiter.GetResult();
    оператор(ы);
) ;
```

Использование на практике

Синхронная реализация:

```
void Go()
{
    for (int i = 1; i < 5; i++)
        _results.Text += GetPrimesCount (i * 1000000, 1000000) +
                " primes between " + (i*1000000) + " and " + ( (i+1) *1000000-1) +
                 Environment.NewLine;
}

int GetPrimesCount (int start, int count)
{
      return ParallelEnumerable.Range (start, count).Count (n =>
           Enumerable .Range (2, (int) Math. Sqrt (n)-1) .All (i => n % i > 0) ) ;
}
```

Асинхронная реализация:

```
Task<int> GetPrimesCountAsync (int start, int count)
{
    return Task.Run (() =>
        ParallelEnumerable.Range (start, count).Count (n =>
        Enumerable.Range (2, (int)Math. Sqrt (n)-1).All (i => n % i > 0) )) ;
}

async void Go ()
{
    _button.IsEnabled = false;
    for (int i = 1; i < 5; i++)
          _results.Text += await GetPrimesCountAsync (i * 1000000, 1000000) +
             " primes between " + (i*1000000) + " and " + ((i + 1) *1000000-1) +
             Environment.NewLine;
     _button.IsEnabled = true;
}
```