Высокоуровневая многопоточность

Юрий Литвинов
yurii.litvinov@gmail.com

27.09.2019г

Правда про многопоточность

Если в вашей программе есть new Thread(), ваша программа уже устарела

- Проектируйте не в терминах потоков, а в терминах задач, которые могут исполняться параллельно
- Доверяйте управление потоками библиотекам
- Используйте высокоуровневые языковые средства
  - Проще
  - Надёжнее
  - Гораздо меньше кода писать
- Помните, что внутри всё равно потоки, продумывание синхронизации никто не отменял
  - Иногда низкоуровневые потоки внезапно наносят ответный удар

Foreground- и Background-потоки

- Когда все Foreground-потоки завершили работу, рантайм останавливает все Background-потоки и заканчивает работу приложения
- Thread по умолчанию создаётся как Foreground
  - Способ прострелить себе ногу №1: создать foreground-поток и забыть о нём, приложение будет висеть в списке задач и не завершится

Способ прострелить себе ногу №2: создать background-поток и не дать ему доработать:

```
var t = new Thread(Worker);
t.IsBackground = true;
t.Start();
Console.WriteLine("Returning from Main");
```

Пул потоков

- Содержит набор заранее созданных потоков, которые могут исполнять задачи
- Управляется рантаймом
  - Новые потоки создаются при необходимости
  - Потоки автоматически удаляются, если они долго не используются и потоков больше, чем надо
  - “Сколько надо” рантайм определяет по количеству доступных ядер процессора
- Используется в .NET практически повсеместно
  - Идеологически многопоточное приложение оперирует не потоками, а задачами и асинхронными операциями
- Все потоки из пула — Background

Пример

```
public static void Main() {
   Console.WriteLine("Главный поток: ставим задачу в очередь");
   ThreadPool.QueueUserWorkItem(ComputeBoundOp, 5);
   Console.WriteLine("Главный поток: занимаемся своими делами...");
   Thread.Sleep(10000);  // Симуляция работы в главном потоке
   Console.WriteLine("Нажмите <Enter>, чтобы закрыть программу...");
   Console.ReadLine();
}

private static void ComputeBoundOp(Object state) {
   Console.WriteLine($"Внутри ComputeBoundOp: state={state}");
   Thread.Sleep(1000);  // Симуляция работы в потоке из пула
}
```

Отмена операций

- CancellationToken — отдаётся потоку, он должен сам проверять состояние токена и прерваться, если запрошена отмена
  - Может прерваться не мгновенно, проверка возможна только время от времени
  - Называется “коллаборативная отмена”
- CancellationTokenSource — возвращает связанный с ним CancellationToken, может выставлять для него флаг отмены, остаётся в основном потоке

Пример

```
public static void Main() {
    var cts = new CancellationTokenSource();
    ThreadPool.QueueUserWorkItem(o => Count(cts.Token, 1000));
    Console.ReadLine();
    cts.Cancel();
    Console.ReadLine();
}

private static void Count(CancellationToken token, int countTo) {
    for (int count = 0; count < countTo; count++) {
        if (token.IsCancellationRequested) {
            break;
        }
        Thread.Sleep(200);
    }
}
```

Task

- Абстракция задачи, которая может быть выполнена в отдельном потоке
- Эквивалентные строки кода:

```
ThreadPool.QueueUserWorkItem(ComputeBoundOp, 5);

new Task(ComputeBoundOp, 5).Start();

Task.Run(() => ComputeBoundOp(5));
```

- Позволяет ждать окончание задачи и получать результат
- Тоже важен для реализации некоторых вещей в C#, но часто используется и независимо

Пример

```
private static int Sum(int n) {
    int sum = 0;
    for (; n > 0; n--)
        sum += n;
    return sum;
}
...
Task<int> t = new Task<int>(n => Sum((int)n), 1000000000);
t.Start();
// Тут занимаемся своими делами, Sum считается в отдельном потоке из пула
t.Wait();  // t.Result сам делает Wait(), так что тут это только для иллюстрации
Console.WriteLine("Сумма: " + t.Result);
```

Отмена Task-а

```
private static int Sum(CancellationToken ct, int n) {
    int sum = 0;
    for (; n > 0; n--) {
        ct.ThrowIfCancellationRequested();
        sum += n;
    }
    return sum;
}
```

Кидает OperationCanceledException в основной поток при обращении к результату (на самом деле, AggregateException с OperationCanceledException)

- Это идеологически правильнее, чем проверять IsCancellationRequested вручную

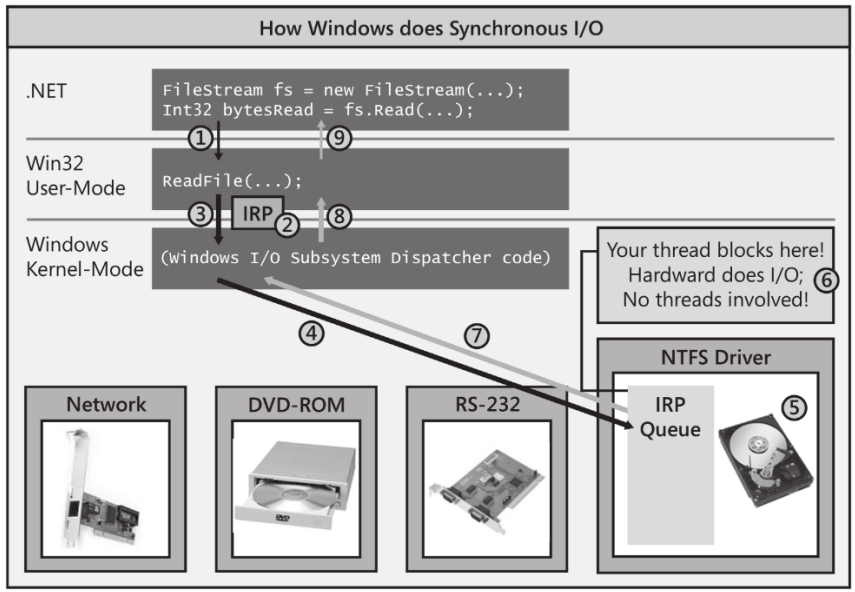
TaskScheduler

- Класс, позволяющий управлять тем, как Task-и обрабатываются пулом потоков (и пулом потоков ли вообще)
  - По умолчанию Task-и ставятся в очередь в пуле потоков
- Бывает полезно, например, чтобы задача могла модифицировать элементы GUI
  - Это можно делать только из главного потока (который создал GUI)

```
Task<int> t = Task.Run(() => Sum(cts.Token, 20000), cts.Token);
t.ContinueWith(task => Text = "Result: " + task.Result,
    CancellationToken.None,
    TaskContinuationOptions.OnlyOnRanToCompletion,
    TaskScheduler.FromCurrentSynchronizationContext());
```

async/await

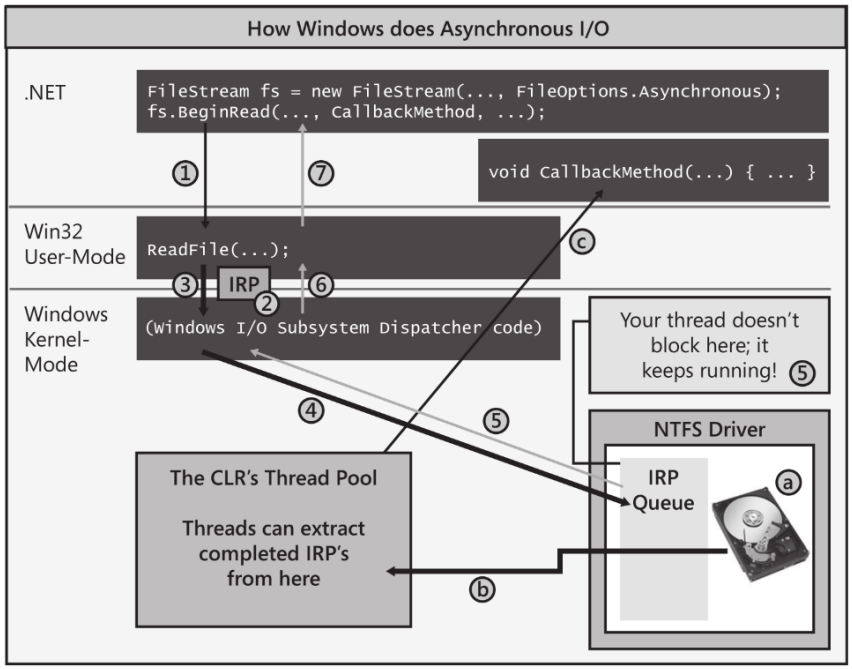
- Task и пул потоков хороши для дорогих по времени операций
- Чаще поток ждёт окончания операции ввода-вывода
- Блокирующий ввод-вывод “вешает” поток, заставляя пул потоков создавать новые



(Рисунок из Jeffrey Richter. CLR via C#)

async/await (2)

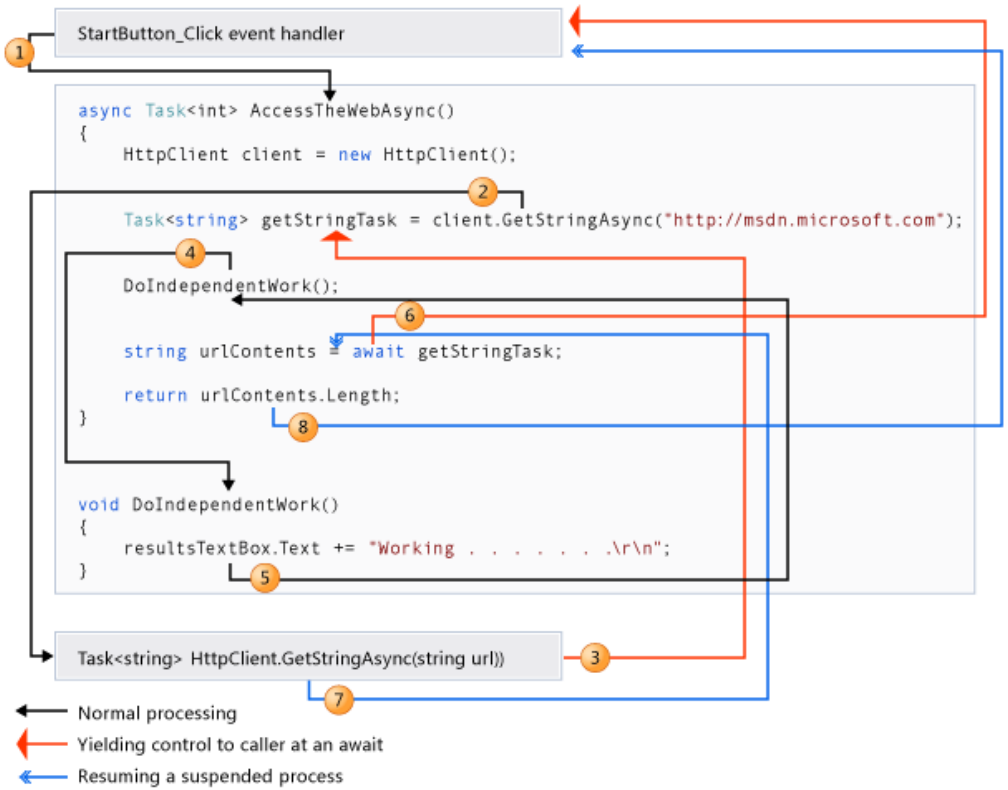
- Асинхронные операции ввода-вывода не блокируют поток, возвращая управление тут же
  - Данные, естественно, не готовы
- Старая модель в .NET — Begin...() и End...()
  - Begin...() инициирует операцию, принимая колбэк, где можно использовать End...(), чтобы забрать результат



(Рисунок из Jeffrey Richter. CLR via C#)

async/await (3)

- Новая модель: async/await
- Требуется поддержка компилятора
- Можно понимать как сопрограмму
- На самом деле, генерируется конечный автомат
  - Запоминает, на каком await сейчас мы находимся
  - Следит за исключениями



(Рисунок из MSDN)

Пример

```
private static async Task<int> Method1Async() { ... }
private static async Task<string> Method2Async() { ... }

private static async Task<string> MyMethodAsync(int argument) {
    int local = argument;
    try {
        int result1 = await Method1Async();
        for (int x = 0; x < 3; x++) {
            string result2 = await Method2Async();
        }
    }
    catch (Exception) { Console.WriteLine("Catch"); }
    return "Done";
}
```

Особенности

- Может возвращать только Task, Task<Result> или **void**
  - **void** используется для асинхронных обработчиков событий
- Любой Task можно ждать await-ом, любой async можно не ждать
  - Вызов Result у результата async-метода заставляет его исполниться синхронно
  - Если забыть await у асинхронного метода, он вернёт не T, а Task<T>
- Работает только с .NET 4.5 и C# 5
  - Microsoft.BCL, если надо поддержать более старый рантайм

async и исключения

```
static async Task ThrowExceptionAsync() {
    await Task.Delay(TimeSpan.FromSeconds(1));
    throw new InvalidOperationException("Всё плохо");
}

static async Task TestAsync() {
    // Исключение не бросается, а сохраняется в Task-е
    Task task = ThrowExceptionAsync();
    try {
        // Исключение бросается при выполнении await (или обращении к Result)
        await task;
    } catch (InvalidOperationException) {
        // Исключение ловится здесь как обычно
    }
}
```

Async-методы в стандартной библиотеке

- System.IO.Stream и потомки: ReadAsync, WriteAsync, FlushAsync, CopyToAsync
- System.IO.TextReader и потомки: ReadAsync, ReadLineAsync, ReadToEndAsync, ReadBlockAsync
- System.IO.TextWriter и потомки: WriteAsync, WriteLineAsync, FlushAsync
- System.Net.Http.HttpClient: GetAsync, GetStreamAsync, GetByteArrayAsync, PostAsync, PutAsync, DeleteAsync и т.д.
- System.Net.WebRequest и потомки: GetRequestStreamAsync и GetResponseAsync
- System.Data.SqlClient.SqlCommand: ExecuteDbDataReaderAsync

Ещё один способ прострелить себе ногу

Вызвать асинхронный метод из GUI-потока и заблокироваться

```
private sealed class MyWpfWindow : Window {
   public MyWpfWindow() { Title = "WPF Window"; }

   protected override void OnActivated(EventArgs e) {
      string http = GetHttp().Result; // Синхронно вызываемся
      base.OnActivated(e);
   }

   private async Task<String> GetHttp() {
      HttpResponseMessage msg =
            await new HttpClient().GetAsync("http://google.com/");
      return await msg.Content.ReadAsStringAsync();  // Никогда не дойдём сюда
   }
}
```

async и Task, полезные вещи

- Подождать сколько-то миллисекунд:
  await Task.Delay(delay);
- Сразу же вернуть результат:
  // Возвращает Task, который исполняется синхронно
  Task.FromResult(13);
- Подождать несколько Task-ов:
  int[] results = **await** Task.WhenAll(task1, task2, task3);

Task.WhenAny

Простой способ организовать таймаут:

```
static async Task<string> DownloadStringWithTimeout(string uri)
{
    using (var client = new HttpClient())
    {
        var downloadTask = client.GetStringAsync(uri);
        var timeoutTask = Task.Delay(3000);
        var completedTask = await Task.WhenAny(downloadTask, timeoutTask);
        if (completedTask == timeoutTask)
            return null;
        return await downloadTask;
    }
}
```

Task Parallel Library

- Набор классов для выполнения вычислительно сложных параллельных операций (в отличие от async/await, ориентированных на ввод-вывод)
  - В стандартной поставке начиная с .NET 4.0
- “Распараллеливание в одну строчку”, берёт на себя управление потоками и ядрами

Пример

```
for (int i = 0; i < 1000; i++) DoWork(i);
```

```
Parallel.For(0, 1000, i => DoWork(i));
```

Или:

- Parallel.ForEach(collection, item => DoWork(item));
- Parallel.Invoke(
  () => Method1(),
  () => Method2(),
  () => Method3());

Прерывание операции

- Изнутри:

```
void InvertMatrices(IEnumerable<Matrix> matrices)
{
    Parallel.ForEach(matrices, (matrix, state) => {
        if (!matrix.IsInvertible)
            state.Stop();
        else
            matrix.Invert();
    });
}
```

- Извне:

```
void RotateMatrices(IEnumerable<Matrix> matrices, float degrees,
    CancellationToken token)
{
    Parallel.ForEach(matrices,
            new ParallelOptions { CancellationToken = token },
            matrix => matrix.Rotate(degrees));
}
```

Синхронизация нужна

```
int InvertMatrices(IEnumerable<Matrix> matrices)
{
    object mutex = new object();
    int nonInvertibleCount = 0;
    Parallel.ForEach(matrices, matrix =>
    {
        if (matrix.IsInvertible) {
            matrix.Invert();
        } else {
            // Не лучший способ увеличить счётчик, но как умею :(
            lock (mutex) {
                ++nonInvertibleCount;
            }
        }
    });
    return nonInvertibleCount;
}
```

Потокобезопасные коллекции

- Одновременная модификация данных из разных потоков (включая потоки пула, которые используются TPL, PLINQ и т.д.) может привести к гонке
- Потокобезопасность — свойство кода не иметь гонок при обращении из нескольких потоков
  - Thread-safe — возможность вызывать код из нескольких потоков
  - Reentrant — возможность вызывать из нескольких потоков, если каждый вызов использует свои данные
  - Весь thread-safe код reentrant, но не наоборот
- Все стандартные коллекции (List<T>, Dictionary<K, V> и т.д.) **не потокобезопасны**

Немутабельные коллекции

Нет изменяемого состояния — нет гонок

- ImmutableStack<T>
- ImmutableQueue<T>
- ImmutableList<T>
- ImmutableHashSet<T>
- ImmutableSortedSet<T>
- ImmutableDictionary<K, V>
- ImmutableSortedDictionary<K, V>

Но как?

```
var list = ImmutableList<int>.Empty;
list = list.Insert(0, 13);
list = list.Insert(0, 7);

// Пишет "7", затем "13".
foreach (var item in list)
    Console.WriteLine(item);

list = list.RemoveAt(1);
```

А что со скоростью и памятью?

Операция	List<T>	ImmutableList<T>
Add	амортизированная O(1)	O(log N)
Insert	O(N)	O(log N)
RemoveAt	O(N)	O(log N)
Item[index]	O(1)	O(log N)

Используются сбалансированные двоичные деревья, разделяемые между коллекциями

Потокобезопасные структуры данных

Мутабельны, используют примитивы синхронизации для избегания гонок

- Медленны, заметно медленнее немутабельных и “обычных” коллекций
- Позволяют общаться разным потокам

Что есть в стандартной библиотеке:

- ConcurrentDictionary<K, V>
- BlockingCollection<T>
- ConcurrentBag<T>
- ConcurrentQueue<T>
- ConcurrentStack<T>

Пример

```
var cq = new ConcurrentQueue<int>();

for (int i = 0; i < 10000; i++)
    cq.Enqueue(i);

if (!cq.TryPeek(out int result))
    Console.WriteLine("С TryPeek тут всё должно было быть ок");
else if (result != 0)
    Console.WriteLine($"Ожидалось 0, получили {result}");

int outerSum = 0;
Action action = () => {
    int localSum = 0;
    while (cq.TryDequeue(out int localValue))
        localSum += localValue;
    // Более правильный способ прибавлять, о котором потом
    Interlocked.Add(ref outerSum, localSum);
};

Parallel.Invoke(action, action, action, action);

Console.WriteLine($"outerSum = {outerSum}, should be 49995000");
```

Литература

Stephen Cleary, Concurrency in C# Cookbook, O'Reilly Media, 2014. 207PP.

Многие примеры взяты оттуда