Принципы асинхронности

Введение

В данном разделе мы объясним, что собой представляют асинхронные операции, и покажем, как они приводят к асинхронному программированию.

Сравнение синхронных и асинхронных операций

Синхронная операция выполняет свою работу *перед* возвратом управления вызывающему коду. *Асинхронная операция* может выполнять большую часть или всю свою работу *после* возврата управления вызывающему коду.

Большинство создаваемых и вызываемых вами методов будут синхронными. Примерами могут служить *List<T>.Add*, *Console.WriteLine* и *Thread.Sleep*. Асинхронные методы менее распространены и инициируют *параллелизм*, т.к. они продолжают работать параллельно с вызывающим кодом. Асинхронные методы обычно быстро (или немедленно) возвращают управление вызывающему компоненту, потому их также называют *неблокирующими методами*.

Большинство асинхронных методов могут быть описаны как универсальные методы:

- *Thread.Start*;
- *Task.Run*;
- методы, которые присоединяют признаки продолжения к задачам.

Что собой представляет асинхронное программирование

Принцип асинхронного программирования состоит в том, что длительно выполняющиеся (или потенциально длительно выполняющиеся) функции реализуются асинхронным образом. Он отличается от традиционного подхода синхронной реализации

длительно выполняющихся функций с последующим их вызовом в новом потоке или в задаче для введения параллелизма по мере необходимости.

Отличие от синхронного подхода заключается в том, что параллелизм инициируется *внутри* длительно выполняющей функции, а не *за ее пределами*. В результате появляются два преимущества.

- Параллельное выполнение с интенсивным вводом-выводом может быть реализовано без связывания потоков, улучшая показатели масштабируемости и эффективности.
- Обогащенные клиентские приложения в итоге содержат меньше кода в рабочих потоках, что упрощает достижение безопасности в отношении потоков.

В свою очередь это приводит к двум различающимся сценариям использования асинхронного программирования. Первый из них связан с написанием (обычно серверных) приложений, которые эффективно обрабатывают большой объем параллельных операций ввода-вывода. Проблемой здесь является не обеспечение *безопасности* к потокам (т.к. разделяемое состояние обычно минимально), а достижение *эффективности* потоков; в частности, отсутствие потребления одного потока на сетевой запрос. Следовательно, в таком контексте выигрыш от асинхронности получают только операции с интенсивным вводом-выводом.

Второй сценарий применения касается упрощения поддержки безопасности в отношении потоков внутри обогащенных клиентских приложений. Он особенно актуален с ростом размера программы, поскольку для борьбы со сложностью мы обычно проводим рефакторинг крупных методов в методы меньших размеров, получая в результате цепочки методов, которые вызывают друг друга (*графы вызовов*).

Если любая операция внутри традиционного графа *синхронных* вызовов является длительно выполняющейся, тогда мы должны запускать целый граф вызовов в рабочем потоке, чтобы обеспечить отзывчивость пользовательского интерфейса. Таким образом, мы в конечном итоге получаем единственную параллельную операцию, которая охватывает множество методов (*крупномодульный параллелизм*), что требует учета безопасности к потокам для каждого метода в графе.

В случае графа *асинхронных* вызовов мы не должны запускать поток до тех пор, пока это не станет действительно необходимым – как правило, в нижней части графа (или вообще не запускать поток для операций с интенсивным вводом-выводом). Все остальные методы могут выполняться полностью в потоке пользовательского интерфейса со значительно упрощенной поддержкой безопасности в отношении потоков. В результате получается *мелкомодульный параллелизм* – последовательность небольших параллельных операций, между которыми выполнение возвращается в поток пользовательского интерфейса.

- Чтобы извлечь из этого выгоду, операции с интенсивным вводом-выводом и интенсивными вычислениями должны быть реализованы асинхронным образом; хорошее эмпирическое правило предусматривает асинхронную реализацию любой операции, выполнение которой может занять более 50 мс.

  (Оборотная сторона заключается в том, что *чрезмерно* мелкомодульная асинхронность может нанести ущерб производительности, потому что с асинхронными операциями связаны определенные накладные расходы)

- Инфраструктура UWP поддерживает асинхронное программирование до момента, когда синхронные версии некоторых длительно выполняющихся методов либо недоступны, либо генерируют исключения. Взамен потребуется вызывать асинхронные методы, возвращающие объекты задач (или объекты, которые могут быть преобразованы в задачи посредством расширяющего метода *AsTask*).

Асинхронное программирование и продолжение

Задачи идеально подходят для асинхронного программирования, т.к. они поддерживают признаки продолжения, которые являются жизненно важными в реализации асинхронности. При написании метода *Delay* мы использовали класс *TaskCompletionSource*, который предлагает стандартный способ реализации асинхронных методов с интенсивным вводом-выводом “нижнего уровня”.

В случае методов с интенсивными вычислениями для инициирования параллелизма, связанного с потоками, мы применяем метод *Task.Run*. Асинхронный метод создается просто за счет возвращения вызывающему компоненту объекта задачи. Асинхронное программирование отличается тем, что мы стремимся поступать подобным образом на как можно более низком уровне графа вызовов. Тогда высокоуровневые методы в обогащенных клиентских приложениях могут быть оставлены в потоке пользовательского интерфейса и получать доступ к элементам управления и разделяемому состоянию, не порождая проблем с безопасностью к потокам. В целях иллюстрации рассмотрим показанный ниже метод, который вычисляет и подсчитывает простые числа, используя все доступные ядра:

```
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));
}
```

Детали того, как работает алгоритм, не особенно важны; имеет значение лишь то, что метод требует некоторого времени на выполнение. Мы можем продемонстрировать это, написав другой метод, который вызывает *GetPrimesCount*:

```
void DisplayPrimeCounts()
{
  for (int i = 0; i < 10; i++)
    Console.WriteLine(GetPrimesCount(i * 1000000 + 2, 1000000) +
      " primes between " + (i * 1000000) + " and " + ((i + 1) *
```

```
        1000000 - 1));
  // " простые числа между " + (i*1000000) + " и " +
  // ((i+1)*1000000-1));
  Console.WriteLine("Done!");
}
```

Вот как выглядит вывод:

```
78498 primes between 0 and 999999
70435 primes between 1000000 and 1999999
67883 primes between 2000000 and 2999999
66330 primes between 3000000 and 3999999
65367 primes between 4000000 and 4999999
64336 primes between 5000000 and 5999999
63799 primes between 6000000 and 6999999
63129 primes between 7000000 and 7999999
62712 primes between 8000000 and 8999999
62090 primes between 9000000 and 9999999
```

Теперь у нас есть *граф вызовов* с методом *DisplayPrimeCounts*, обращающимся к методу *GetPrimesCount*. Для простоты внутри *DisplayPrimeCounts* применяется метод *Console.WriteLine*, хотя в реальности, скорее всего, будут обновляться элементы управления пользовательского интерфейса в обогащенном клиентском приложении, что демонстрируется позже. Крупномодульный параллелизм для такого графа вызовов можно инициировать следующим образом:

```
Task.Run(() => DisplayPrimeCounts());
```

В случае асинхронного подхода с мелкомодульным параллелизмом мы начинаем с написания асинхронной версии метода *GetPrimesCount*:

```
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)));
}
```

Важность языковой поддержки

Теперь мы должны модифицировать метод *DisplayPrimeCounts* так, чтобы он вызывал *GetPrimesCount**Async***. Именно здесь в игру вступают ключевые слова *await* и *async* языка C#, поскольку поступить по-другому намного сложнее, чем может показаться. Если мы просто изменим цикл, как показано ниже:

```
for (int i = 0; i < 10; i++)
{
  var awaiter = GetPrimesCountAsync(i * 1000000 + 2, 1000000).
    GetAwaiter();
  awaiter.OnCompleted(() =>
    Console.WriteLine(awaiter.GetResult() + " primes between... "));
}
Console.WriteLine("Done");
```

то цикл быстро пройдет через 10 итераций (методы не являются блокирующими) и все 10 операций будут выполняться параллельно (с ранним выводом строки “Done”).

- Выполнять приведенные задачи параллельно в данном случае нежелательно, т.к. их внутренние реализации уже распараллелены; это приведет лишь к более длительному ожиданию первых результатов (и нарушению упорядочения).

  Однако существует намного более распространенная причина для последовательного выполнения задач – ситуация, когда задача Б зависит от результатов выполнения задачи А. Например, при выборке веб-страницы DNS-поиск должен предшествовать HTTP-запросу.

Для обеспечения последовательного выполнения следующую итерацию цикла нужно запускать из самого продолжения, что означает устранение цикла *for* и реализацию рекурсивного вызова в продолжении:

```
void DisplayPrimeCounts()
{
  DisplayPrimeCountsFrom(0);
}

void DisplayPrimeCountsFrom(int i)
{
  var awaiter = GetPrimesCountAsync(i * 1000000 + 2, 1000000).
    GetAwaiter();
```

```
  awaiter.OnCompleted(() =>
  {
    Console.WriteLine(awaiter.GetResult() + " primes between...");
    if (i++ < 10) DisplayPrimeCountsFrom(i);
    else Console.WriteLine("Done");
  });
}
```

Все становится еще хуже, если необходимо сделать асинхронным *сам* метод *DisplayPrimesCount*, возвращая объект задачи, которая отправляет сигнал о своем завершении. Достижение такой цели требует создания объекта *TaskCompletionSource*:

```
Task DisplayPrimeCountsAsync()
{
  var machine = new PrimesStateMachine();
  machine.DisplayPrimeCountsFrom(0);
  return machine.Task;
}

class PrimesStateMachine
{
  TaskCompletionSource<object> _tcs = new TaskCompletionSource<object>();
  public Task Task { get { return _tcs.Task; } }
    public void DisplayPrimeCountsFrom(int i)
  {
    var awaiter = GetPrimesCountAsync(i * 1000000 + 2, 1000000).
      GetAwaiter();
    awaiter.OnCompleted(() =>
    {
      Console.WriteLine(awaiter.GetResult());
      if (i++ < 10) DisplayPrimeCountsFrom(i);
      else { Console.WriteLine("Done"); _tcs.SetResult(null); }
    });
  }
}
```

К счастью, *асинхронные функции* C# делают всю работу подобного рода. Благодаря новым ключевым словам *async* и *await* нам придется написать только следующий код:

```
async Task DisplayPrimeCountsAsync()
{
  for (int i = 0; i < 10; i++)
    Console.WriteLine(await GetPrimesCountAsync(i * 1000000 +
      2, 1000000) + " primes between " + (i * 1000000) + " and " +
      ((i + 1) * 1000000 - 1));
  Console.WriteLine("Done!");
}
```

Таким образом, ключевые слова *async* и *await* очень важны для реализации асинхронности без чрезмерной сложности. Давайте посмотрим, как они работают.

- Взглянуть на данную проблему можно и по-другому: императивные конструкции циклов (*for, foreach* и т.д.) не очень хорошо сочетаются с признаками продолжения, поскольку они полагаются на *текущее локальное состояние* метода (т.е. сколько раз цикл планирует выполняться).

  Хотя ключевые слова *async* и *await* предлагают одно решение, иногда решить проблему удается другим способом, заменяя императивные конструкции циклов их *функциональными* эквивалентами (другими словами, запросами LINQ). Это является основой библиотеки *Reactive Extensions* (Rx) и может оказаться удачным вариантом, когда в отношении результата нужно выполнить операции запросов или скомбинировать несколько последовательностей. Недостаток связан с тем, что во избежание блокировки инфраструктура Rx оперирует на последовательностях с *активным* источником, которые могут оказаться концептуально сложными.