Асинхронные шаблоны

Отмена

Часто важно иметь возможность отмены параллельной операции после ее запуска, скажем, в ответ на пользовательский запрос. Реализовать это проще всего с помощью флага отмены, который можно было бы инкапсулировать в классе следующего вида:

```
class CancellationToken
{
  public bool IsCancellationRequested { get; private set; }
  public void Cancel() { IsCancellationRequested = true; }
  public void ThrowIfCancellationRequested()
  {
    if (IsCancellationRequested)
      throw new OperationCanceledException();
  }
}
```

Затем можно было бы написать асинхронный метод с возможностью отмены:

```
async Task Foo(CancellationToken cancellationToken)
{
  for (int i = 0; i < 10; i++)
  {
    Console.WriteLine(i);
    await Task.Delay(1000);
    cancellationToken.ThrowIfCancellationRequested();
  }
}
```

Когда вызывающий код желает отменить операцию, он обращается к методу *Cancel* признака отмены, который передается в метод *Foo*. В результате *IsCancellationRequested* устанавливается в *true*, что через короткий промежуток времени приводит к отказу метода *Foo* с генерацией исключения *OperationCanceledException* (предопределенный в пространстве имен *System* класс, который предназначен для данной цели).

Если оставить в стороне безопасность к потокам (мы должны блокировать чтение/запись в *IsCancellationRequested*), то такой шаблон вполне эффективен, и среда CLR предлагает тип по имени *CancellationToken*, который очень похож на только что рассмотренный тип. Тем не менее, в нем отсутствует метод *Cancel*; этот метод открыт в другом типе – *CancellationTokenSource*. Подобное разделение обеспечивает определенную безопасность: метод, который имеет доступ только к объекту *CancellationToken*, может проверять, но не *инициировать* отмену.

Чтобы получить признак отмены, сначала необходимо создать экземпляр *CancellationTokenSource*:

```
var cancelSource = new CancellationTokenSource();
```

После этого станет доступным свойство *Token*, которое возвращает объект *CancellationToken*. В итоге вызвать наш метод *Foo* можно было бы следующим образом:

```
var cancelSource = new CancellationTokenSource();
Task foo = Foo(cancelSource.Token);
...
... (в какой - то момент позже)
cancelSource.Cancel();
```

Признаки отмены поддерживает большинство асинхронных методов в CLR, включая *Delay*. Если модифицировать метод *Foo* так, чтобы он передавал свой признак отмены методу *Delay*, то задача будет завершаться немедленно по запросу (а не секунду спустя):

```
async Task Foo(CancellationToken cancellationToken)
{
  for (int i = 0; i < 10; i++)
  {
    Console.WriteLine(i);
    await Task.Delay(1000, cancellationToken);
  }
}
```

Обратите внимание, что нам больше не понадобится вызывать метод *ThrowIfCancellationRequested*, поскольку это делает *Task.Delay*. Признаки отмены нормально распространяются вниз по стеку вызовов (так же как запросы отмены каскадным образом продвигаются *вверх* по стеку вызовов посредством исключений).

- ❖ UWP полагается на типы WinRT, чьи асинхронные методы при отмене следуют низкоуровневому протоколу, согласно которому вместо принятия *CancellationToken* тип *IAsyncInfo* открывает доступ к методу *Cancel*. Однако метод *AsTaskextension* перегружен для приема признака отмены, ликвидируя данный разрыв.

Синхронные методы также могут поддерживать отмену (как делает метод *Wait* класса *Task*). В таких случаях инструкция для отмены должна будет поступать асинхронно (скажем, из другой задачи). Например:

```
var cancelSource = new CancellationTokenSource();
Task.Delay(5000).ContinueWith(ant => cancelSource.Cancel());
...
```

В действительности при конструировании *CancellationTokenSource* можно указывать временной интервал, чтобы инициировать отмену по его прошествии (как только что было продемонстрировано). Прием удобен для реализации тайм-аутов, как синхронных, так и асинхронных:

```
var cancelSource = new CancellationTokenSource(5000);
try { await Foo(cancelSource.Token); }
catch (OperationCanceledException ex) { Console.WriteLine("Cancelled"); }
```

Структура *CancellationToken* предоставляет метод *Register*, позволяющий зарегистрировать делегат обратного вызова, который будет запущен при отмене; он возвращает объект, который можно освободить с целью отмены регистрации.

Задачи, генерируемые асинхронными функциями компилятора, автоматически входят в состояние отмены при появлении необработанного исключения *OperationCanceledException* (свойство *IsCanceled* возвращает *true*, а свойство *IsFaulted* – *false*). То же самое происходит и в случае задач, созданных с помощью метода *Task.Run*, конструктору которых передается (тот же признак) *CancellationToken*. Отличие между отказавшей и отмененной задачей в асинхронных сценариях не является важным, т.к. обе они генерируют исключение *OperationCanceledException* во время ожидания; это играет роль в расширенных сценариях параллельного программирования (особенно при условном продолжении).

Сообщение о ходе работ

Временами желательно, чтобы асинхронная операция во время выполнения сообщала о ходе работ. Простое решение заключается в передаче асинхронному методу делегата *Action*, который запускается всякий раз, когда состояние хода работ меняется:

```
Task Foo(Action<int> onProgressPercentChanged)
{
  return Task.Run(() =>
  {
    for (int i = 0; i < 1000; i++)
    {
      if (i % 10 == 0) onProgressPercentChanged(i / 10);
      // Делать что-нибудь, требующее интенсивных вычислений...
    }
  });
}
```

Вот как его можно вызывать:

```
Action progress = i => Console.WriteLine(i + " %");
await Foo(progress);
```

Хотя такой прием нормально работает в консольном приложении, он не идеален в сценариях обогащенных клиентов, поскольку сообщает о ходе работ из рабочего потока, вызывая потенциальные проблемы с безопасностью к потокам у потребителя. (Фактически мы позволяем побочному эффекту от параллелизма “просочиться” во внешний мир, что нежелательно, т.к. в противном случае метод изолируется, если он вызван в потоке пользовательского интерфейса.)

IProgress<T> и Progress<T>

Для решения описанной выше проблемы среда *CLR* предлагает пару типов: интерфейс *IProgress<T>* и класс *Progress<T>*, который реализует этот интерфейс. В действительности они предназначены для того, чтобы служить оболочкой делегата, позволяя приложениям с пользовательским интерфейсом безопасно сообщать о ходе работ через контекст синхронизации.

Интерфейс *IProgress<T>* определяет только один метод:

```
public interface IProgress
{
  void Report(T value);
}
```

Интерфейс *IProgress<T>* определяет только один метод:

```
Task Foo(IProgress<int> onProgressPercentChanged)
{
  return Task.Run(() =>
  {
    for (int i = 0; i < 1000; i++)
    {
      if (i % 10 == 0)
        onProgressPercentChanged.Report(i / 10);
      // Делать что-нибудь, требующее интенсивных вычислений...
    }
  });
}
```

Класс *Progress<T>* имеет конструктор, принимающий делегат типа *Action<T>*, который помещается в оболочку:

```
var progress = new Progress<int>(i => Console.WriteLine(i + " %"));
await Foo(progress);
```

(В классе *Progress<T>* также определено событие *ProgressChanged*, на которое можно подписаться вместо передачи делегата Action конструктору (или в дополнение к ней).) После создания экземпляра *Progress<int>* захватывается контекст синхронизации, если он существует. Когда метод *Foo* затем обращается к *Report*, делегат вызывается через упомянутый контекст.

Асинхронные методы могут реализовать более сложное сообщение о ходе работ путем замены *int* специальным типом, открывающим доступ к набору свойств.

- Если вы знакомы с библиотекой Reactive Extensions, то заметите, что интерфейс *IProgress<T>* вместе с типом задачи, возвращаемым асинхронной функцией, предоставляют набор средств, который подобен такому набору, предлагаемому интерфейсом *IObserver<T>*. Отличие в том, что тип задачи может открывать доступ к “финальному” возвращаемому значению в дополнение к значениям (других типов), выдаваемым интерфейсом *IProgress<T>*.

  Значения, выдаваемые *IProgress<T>*, обычно являются “одноразовыми” (скажем, процент выполненной работы или количество загруженных байтов), тогда как значения, возвращаемые методом *MoveNext* интерфейса *IObserver<T>*, обычно содержат в себе сам результат и поэтому существует веская причина для его вызова.

Асинхронные методы в *WinRT* также поддерживают возможность сообщения о ходе работ, хотя применяемый протокол сложнее из-за (относительно) слабой системы типов COM. Взамен приема объекта, реализующего *IProgress<T>*, асинхронные методы *WinRT*, которые сообщают о ходе работ, возвращают на месте *IAsyncAction* и *IAsyncOperation<TResult>* один из следующих интерфейсов:

```
IAsyncActionWithProgress<TProgress>
IAsyncOperationWithProgress<TResult, TProgress>
```

Интересно отметить, что оба интерфейса основаны на IAsyncInfo (не на IAsyncAction и IAsyncOperation<TResult>).

Хорошая новость в том, что расширяющий метод *AsTask* также перегружен, чтобы принимать *IProgress<T>* для вышеупомянутых интерфейсов, поэтому потребители .NET могут игнорировать интерфейсы COM и поступать так, как показано ниже:

```
var progress = new Progress<int>(i => Console.WriteLine(i + " %"));
CancellationToken cancelToken = ...
var task = someWinRTobject.FooAsync().AsTask(cancelToken, progress);
```

Асинхронный шаблон, основанный на задачах

Временами В .NET доступны сотни асинхронных методов, возвращающих задачи, к которым можно применять *await* (они относятся главным образом к вводу-выводу). Большинство таких методов (по крайней мере, частично) следуют шаблону, который называется *асинхронным шаблоном, основанным на задачах* (Task-Based Asynchronous Pattern – TAP), и представляет собой практичную формализацию всего того, что было описано до настоящего момента. Метод TAP обладает следующими характеристиками:

- возвращает “горячий” (выполняющийся) экземпляр *Task* или *Task<TResult>*;
- имеет суффикс *Async* (за исключением специальных случаев, таких как комбинаторы задач);
- перегружен для приема признака отмены и/или *IProgress<T>*, если он поддерживает отмену и/или сообщение о ходе работ;
- быстро возвращает управление вызывающему коду (имеет только небольшую начальную синхронную фазу);
- не связывает поток, если является интенсивным в плане ввода-вывода.

Как видите, методы TAP легко писать с использованием асинхронных функций C#.

Комбинаторы задач

Важным последствием наличия согласованного протокола для асинхронных функций (в соответствии с которым они возвращают объекты задач) является возможность применения и написания *комбинаторов задач* – функций, которые удобно объединяют задачи, не принимая во внимание то, что конкретно делает та или иная задача.

Среда *CLR* включает два комбинатора задач: *Task.WhenAny* и *Task.WhenAll*. При их описании мы будем предполагать, что определены следующие методы:

```
async Task Delay1() { await Task.Delay(1000); return 1; }
async Task Delay2() { await Task.Delay(2000); return 2; }
async Task Delay3() { await Task.Delay(3000); return 3; }
```

WhenAny

Метод *Task.WhenAny* возвращает объект задачи, которая завершается при завершении любой задачи из набора. В следующем примере задача завершается через одну секунду:

```
Task winningTask = await Task.WhenAny(Delay1(), Delay2(), Delay3());
Console.WriteLine("Done");
Console.WriteLine(winningTask.Result); // 1
```

Поскольку метод *Task.WhenAny* сам возвращает объект задачи, мы применяем к его вызову *await*, что дает в итоге задачу, завершающуюся первой. Приведенный пример является полностью неблокирующим – включая последнюю строку, где производится доступ к свойству *Result* (т.к. задача *winningTask* уже будет завершена). Несмотря на это, обычно лучше применять *await* и к *winningTask*:

```
Console.WriteLine (await winningTask);
```

потому что тогда любое исключение генерируется повторно без помещения в оболочку *AggregateException*. На самом деле оба *await* могут находиться в одном операторе:

```
int answer = await await Task.WhenAny (Delay1() , Delay2(), Delay3());
```

Если какая-то из задач кроме завершившейся первой впоследствии откажет, то исключение станет необнаруженным, если только для объекта задачи не будет организовано ожидание посредством *await* (или не будет произведен доступ к его свойству *Exception*).

Метод *WhenAny* удобен для применения тайм-аутов или отмены к операциям, которые иначе подобное не поддерживают:

```
Task task = SomeAsyncFunc();
Task winner = await(Task.WhenAny(task, Task.Delay(5000)));
if (winner != task) throw new TimeoutException();
string result = await task;  // Извлечь результат или повторно
                             // сгенерировать исключение
```

Обратите внимание, что поскольку в данном случае метод *WhenAny* вызывается с задачами разных типов, выигравшая задача возвращается как простой объект типа *Task* (а не *Task<string>*).

WhenAll

Метод *Task.WhenAll* возвращает объект задачи, которая завершается, когда завершены *все* переданные ему задачи. В следующем примере задача завершается через три секунды (и демонстрируется шаблон *ветвления/присоединения* (*fork/join*)):

```
await Task.WhenAll (Delay1() , Delay2(), Delay3());
```

Похожий результат можно было бы получить без использования *WhenAll*, организовав ожидание *task1*, *task2* и *task3* по очереди:

```
Task task1 = Delay1(), task2 = Delay2(), task3 = Delay3();
await task1; await task2; await task3;
```

Отличие такого подхода (помимо меньшей эффективности из-за требования трех ожиданий вместо одного) связано с тем, что в случае отказа *task1* мы никогда не перейдем к ожиданию задач *task2*/*task3*, и любые их исключения останутся необнаруженными.

Напротив, метод *Task.WhenAll* не завершается до тех пор, пока не будут завершены все задачи – даже когда возникает отказ. При появлении нескольких отказов их исключения объединяются в экземпляр *AggregateException* задачи (именно здесь класс *AggregateException* становится действительно полезным, потому что вы должны быть заинтересованы в получении всех исключений). Тем не менее, ожидание комбинированной задачи обеспечивает генерацию только первого исключения, так что для просмотра всех исключений понадобится поступить следующим образом:

```
Task task1 = Task.Run(() => { throw null; });
Task task2 = Task.Run(() => { throw null; });
Task all = Task.WhenAll(task1, task2);
try { await all; }
catch
{
  Console.WriteLine(all.Exception.InnerExceptions.Count); // 2
}
```

Вызов *WhenAll* с задачами типа *Task<TResult>* возвращает *Task<Result[]>*, предоставляя объединенные результаты всех задач. При ожидании все сводится к *TResult[]*:

```
Task task1 = Task.Run(() => 1);
Task task2 = Task.Run(() => 2);
int[] results = await Task.WhenAll(task1, task2); // {1, 2}
```

В качестве практического примера рассмотрим параллельную загрузку веб-страниц по нескольким URI с подсчетом их суммарной длины:

```
async Task<int> GetTotalSize(string[] uris)
{
  IEnumerable<Task<byte[]>> downloadTasks = uris.Select(uri =>
    new WebClient().DownloadDataTaskAsync(uri));
  byte[][] contents = await Task.WhenAll(downloadTasks);
  return contents.Sum(c => c.Length);
}
```

Однако здесь присутствует некоторая неэффективность, связанная с тем, что во время загрузки мы излишне удерживаем байтовый массив до тех пор, пока не будет завершена каждая задача. Было бы более эффективно сразу же после загрузки сворачивать байтовые массивы в их длины. Для этого очень удобно применять асинхронные лямбда-выражения, потому что нам необходимо передавать выражение *await* в операцию запроса *Select* из LINQ:

```
async Task<int> GetTotalSize(string[] uris)
{
  IEnumerable<Task<int>> downloadTasks = uris.Select(async uri =>
    (await new WebClient().DownloadDataTaskAsync(uri)).Length);
  int[] contentLengths = await Task.WhenAll(downloadTasks);
  return contentLengths.Sum();
}
```

Специальные комбинаторы

Временами удобно создавать собственные комбинаторы задач. Простейший “комбинатор” принимает одиночную задачу вроде приведенной ниже, что позволяет организовать ожидание любой задачи с использованием тайм-аута:

```
async static Task<TResult> WithTimeout(this Task task, TimeSpan timeout)
{
  Task winner = await Task.WhenAny(task, Task.Delay(timeout))
                       .ConfigureAwait(false);
  if (winner != task) throw new TimeoutException();
  return await task.ConfigureAwait(false); // Извлечь результат или
                                  // повторно сгенерировать исключение
}
```

Поскольку это в значительной степени “библиотечный метод”, который не имеет доступа к внешнему разделяемому состоянию, при ожидании мы используем *ConfigureAwait*(*false*), чтобы избежать потенциального возврата в контекст синхронизации пользовательского интерфейса. Мы можем еще больше повысить эффективность, отменяя *Task.Delay*, когда задача завершается вовремя (что позволяет устранить небольшие накладные расходы, связанные с таймером):

```
async static Task<TResult> WithTimeout(this Task task, TimeSpan timeout)
{
  var cancelSource = new CancellationTokenSource();
  var delay = Task.Delay(timeout, cancelSource.Token);
  Task winner = await Task.WhenAny(task, delay).ConfigureAwait(false);
  if (winner == task)
    cancelSource.Cancel();
  else
    throw new TimeoutException();
  return await task.ConfigureAwait(false);   // Извлечь результат или
                                   // повторно сгенерировать исключение
}
```

Следующий комбинатор позволяет “отменить” задачу посредством *CancellationToken*:

```
static Task<TResult> WithCancellation(this Task task,
                        CancellationToken cancelToken)
{
  var tcs = new TaskCompletionSource();
  var reg = cancelToken.Register(() => tcs.TrySetCanceled());
  task.ContinueWith(ant =>
  {
    reg.Dispose();
    if (ant.IsCanceled)
      tcs.TrySetCanceled();
    else if (ant.IsFaulted)
      tcs.TrySetException(ant.Exception.InnerException);
    else
      tcs.TrySetResult(ant.Result);
  });
  return tcs.Task;
}
```

Комбинаторы задач могут оказаться сложными в написании, иногда требуя применения сигнализирующих конструкций. На самом деле это хорошо, т.к. способствует вынесению сложности, связанной с параллелизмом, за пределы бизнес-логики и ее помещению в многократно используемые методы, которые могут быть протестированы в изоляции.

Следующий комбинатор работает подобно *WhenAll* за исключением того, что если любая из задач отказывает, то результирующая задача откажет незамедлительно:

```
async Task<TResult[]> WhenAllOrError(params Task[] tasks)
{
  var killJoy = new TaskCompletionSource<TResult[]>();
  foreach (var task in tasks)
    task.ContinueWith(ant =>
    {
      if (ant.IsCanceled)
        killJoy.TrySetCanceled();
      else if (ant.IsFaulted)
        killJoy.TrySetException(ant.Exception.InnerException);
    });
  return await await Task.WhenAny(killJoy.Task,
    Task.WhenAll(tasks)).ConfigureAwait(false);
}
```

Мы начинаем с создания экземпляра *TaskCompletionSource*, единственной работой которого является завершение всего в случае, если какая-то задача отказывает. Таким образом, мы никогда не вызываем его метод *SetResult*, а только методы *TrySetCanceled* и *TrySetException*. В данном случае метод *ContinueWith* более удобен, чем *GetAwaiter*().*OnCompleted*, потому что мы не обращаемся к результатам задач и в этой точке не хотим возврата в поток пользовательского интерфейса.