Устаревшие шаблоны

Введение

В .NET задействованы и другие шаблоны асинхронности, которые применялись до появления задач и асинхронных функций. Теперь они редко востребованы, поскольку асинхронность на основе задач стала доминирующим шаблоном.

Модель асинхронного программирования

Самый старый шаблон назывался *моделью асинхронного программирования* (Asynchronous Programming Model — APM) и использовал пару методов, имена которых начинаются с *Begin* и *End*, а также интерфейс по имени *IAsyncResult*. В целях иллюстрации мы возьмем класс *Stream* из пространства имен *System.IO* и рассмотрим его метод *Read*. Вначале взглянем на синхронную версию:

```
public int Read (byte[] buffer, int offset, int size);
```

Вероятно, вы уже в состоянии предугадать, каким образом выглядит асинхронная версия на основе *задач*:

```
public Task<int> ReadAsync (byte[] buffer, int offset, int size);
```

Теперь давайте посмотрим на версию APM:

```
public IAsyncResult BeginRead (byte[] buffer, int offset, int size
                    AsyncCallback callback, object state);
public int EndRead (IAsyncResult asyncResult);
```

Вызов метода *Begin** инициирует операцию, возвращая объект *IAsyncResult*, который действует в качестве признака для асинхронной операции. Когда операция завершается (или отказывает), запускается делегат *AsyncCallback*:

```
public delegate void AsyncCallback (IAsyncResult ar);
```

Компонент, поддерживающий этот делегат, затем вызывает метод *End**, который предоставляет возвращаемое значение операции, а также повторно генерирует исключение, если операция потерпела неудачу.

Шаблон APM не только неудобен в применении, но также неожиданно сложен в плане корректной реализации. Проще всего иметь дело с методами APM, вызывая метод адаптера *Task.Factory.FromAsync*, который преобразует пару методов APM в объект *Task*. Внутренне он использует *TaskCompletionSource*, чтобы предоставить объект задачи, которой отправляется сигнал, когда операция APM завершается или отказывает.

Метод *FromAsync* требует передачи следующих параметров:

- делегат, указывающий метод *BeginXXX*;
- делегат, указывающий метод *EndXXX*;
- дополнительные аргументы, которые будут передаваться данным методам.

Метод *FromAsync* перегружен для приема типов делегатов и аргументов, которые соответствуют практически всем сигнатурам асинхронных методов, определенным в .NET. Например, исходя из предположения, что *stream* имеет тип *Stream*, а *buffer* – тип *byte*[ ] , мы можем записать так:

```
Task<int> readChunk = Task<int>.Factory.FromAsync (stream.BeginRead,
  stream.EndRead, buffer, 0, 1000, null);
```

Асинхронный шаблон на основе событий

Асинхронный шаблон на основе событий (Event-Based Asynchronous Pattern – EAP) появился в 2005 году с целью предоставления более простой альтернативы шаблону APM, особенно в сценариях с пользовательским интерфейсом. Тем не менее, он был реализован лишь в небольшом количестве типов, наиболее примечательным из которых является *WebClient* в пространстве имен *System.Net*. Следует отметить, что EAP – это просто шаблон; никаких специальных типов для его поддержки не предусмотрено. По существу шаблон выглядит так: класс предлагает семейство членов, которые внутренне управляют параллелизмом, примерно как в показанном далее коде.

```
// Это члены класса WebClient:
public byte[] DownloadData (Uri address);  // Синхронная версия
public void DownloadDataAsync (Uri address);
public void DownloadDataAsync (Uri address, object userToken);
public event DownloadDataCompletedEventHandler
DownloadDataCompleted;

public void CancelAsync (object userState);  // Отменяет операцию
public bool IsBusy { get; }  // Указывает, выполняется ли операция
```

Методы **Async* инициируют выполнение операции асинхронным образом. Когда операция завершается, генерируется событие **Completed* (с автоматической отправкой захваченному контексту синхронизации, если он имеется). Такое событие передает объект аргументов события, содержащий перечисленные ниже элементы:

- флаг, который указывает, была ли операция отменена (за счет вызова потребителем метода *CancelAsync*);
- объект *Error*, указывающий исключение, которое было сгенерировано (если было);
- объект *userToken*, если он предоставлялся при вызове метода **Async*.

Типы EAP могут также определять событие сообщения о ходе работ, которое инициируется всякий раз, когда состояние хода работ изменяется (и вдобавок отправляется в контекст синхронизации):

```
public event DownloadProgressChangedEventHandler DownloadProgressChanged;
```

Реализация шаблона EAP требует написания большого объема стереотипного кода, делая этот шаблон неудобным с композиционной точки зрения.

BackgroundWorker

Универсальной реализацией шаблона EAP является класс *BackgroundWorker* из пространства имен *System.ComponentModel*. Он позволяет обогащенным клиентским приложениям запускать рабочий поток и сообщать о проценте выполненной работы без необходимости в явном захвате контекста синхронизации. Вот пример:

```
var worker = new BackgroundWorker {WorkerSupportsCancellation = true };
worker.DoWork += (sender, args) =>
{ // Выполняется в рабочем потоке
  if (args.Cancel) return;
  Thread.Sleep(1000);
  args.Result = 123;
} ;
worker.RunWorkerCompleted += (sender, args) =>
{ // Выполняется в потоке пользовательского интерфейса
  // Здесь можно безопасно обновлять элементы управления
  // пользовательского интерфейса...
  if (args.Cancelled)
    Console.WriteLine ("Cancelled");  // Отменено
  else if (args.Error != null)
    Console.WriteLine ("Error: " + args.Error.Message);       // Ошибка
  else
    Console.WriteLine ("Result is: " + args.Result);   // Результат
};
worker.RunWorkerAsync();  // Захватывает контекст синхронизации
                          // и запускает операцию
```

Метод *RunWorkerAsync* запускает операцию, инициируя событие *DoWork* в рабочем потоке из пула. Он также захватывает контекст синхронизации, и когда операция завершается (или отказывает), через данный контекст генерируется событие *RunWorkerCompleted* (подобно признаку продолжения).

Класс *BackgroundWorker* порождает крупномодульный параллелизм, при котором событие *DoWork* инициируется полностью в рабочем потоке. Если в этом обработчике событий нужно обновлять элементы управления пользовательского интерфейса (помимо отправки сообщения о проценте выполненных работ), тогда придется использовать *Dispatcher.BeginInvoke* или похожий метод.