Содержание

Часть VI. Коммуникации (на веб-сайте)

13

Асинхронное программирование

В ЭТОЙ ГЛАВЕ...

- Важность асинхронного программирования
- Асинхронные шаблоны
- Ключевые слова `async` и `await`
- Создание и использование асинхронных методов
- Обработка ошибок с помощью асинхронных методов

Загружаемый код для этой главы

Загружаемый код для этой главы содержит следующие основные примеры:

- `AsyncPatterns`
- `Foundations`
- `ErrorHandling`

Важность асинхронного программирования

Наиболее важным изменением версии C# 5 является прогресс в отношении асинхронного программирования. В C# 5 добавлены всего два ключевых слова: `async` и `await`. На этих ключевых словах и сосредоточено основное внимание в этой главе.

При *асинхронном программировании* вызванный метод выполняется в фоновом режиме (обычно с помощью потока или задачи), а вызывающий поток не блокируется.

В этой главе вы ознакомитесь с различными шаблонами асинхронного программирования, такими как *асинхронный шаблон*, *асинхронный шаблон, основанный на событиях*, и новый *асинхронный шаблон, основанный на задачах* (task-based asynchronous pattern — TAP). Шаблон TAP использует ключевые слова `async` и `await`. Сравнивая указанные шаблоны, можно оценить реальные преимущества нового стиля асинхронного программирования.

После обсуждения шаблонов будет продемонстрирована основа асинхронного программирования на примере создания задач и вызова асинхронных методов. Вы узнаете, что находится "за кулисами" задач продолжения и контекста синхронизации.

Обработка ошибок заслуживает особого внимания; как и в случае асинхронных задач, некоторые сценарии требуют разных подходов к обработке ошибок.

В конце этой главы обсуждаются способы обеспечения отмены. Фоновые задачи могут требовать некоторого времени на свое выполнение, поэтому иногда возникает необходимость в отмене задачи, пока она еще выполняется. В настоящей главе будет показано, как это делать.

Другие сведения о параллельном программировании приведены в главе 21.

Пользователей обычно раздражает, если приложение не реагирует немедленно на запросы. Работая с мышью, мы уже привыкли к возникновению задержек, т.к. сталкивались с подобным поведением в течение нескольких десятилетий. Однако при наличии сенсорного пользовательского интерфейса приложение должно реагировать на запросы немедленно. В противном случае пользователь попытается повторить действие.

Поскольку в старых версиях .NET Framework асинхронное программирование было затруднено, оно не всегда делалось, когда это было необходимо. Примером приложения, которое довольно часто блокировало поток пользовательского интерфейса, может служить Visual Studio 2010. В этой версии открытие решения, содержащего сотни проектов, приводило к длительной паузе, в течение которой вполне можно было успеть выпить чашку кофе. В Visual Studio 2012 подобное не происходит, т.к. проекты загружаются в фоновом режиме, с загрузкой первым выбранного проекта. Такое поведение загрузки — лишь один пример важных изменений, связанных с асинхронным программированием, которые были внесены в версию Visual Studio 2012. Аналогичным образом пользователи Visual Studio 2010 очень хорошо знакомы с ситуацией, когда какое-то диалоговое окно не реагирует на действия. В Visual Studio 2012 это менее вероятно.

Многие API-интерфейсы в .NET Framework предлагают синхронные и асинхронные версии. Так как синхронную версию API-интерфейса намного легче использовать, ее часто применяли там, где это совершенно не подходило. Благодаря новой исполняющей среде Windows Runtime (WinRT), если ожидается, что вызов API-интерфейса займет более 40 миллисекунд, будет доступна только асинхронная версия этого API-интерфейса. С появлением .NET 4.5 асинхронное программирование стало таким же простым, как синхронное, поэтому никаких препятствий к использованию асинхронных API-интерфейсов возникать не должно.

Асинхронные шаблоны

Перед тем, как перейти к рассмотрению новых ключевых слов `async` и `await`, имеет смысл разобраться в асинхронных шаблонах .NET Framework. Асинхронные возможности были доступны, начиная еще с версии .NET 1.0, и многие классы в .NET Framework реализуют один или несколько таких шаблонов. Асинхронный шаблон также доступен с типом делегата.

Из-за того, что обновление пользовательского интерфейса с помощью как Windows Forms, так и WPF с асинхронным шаблоном было довольно сложным, в версии .NET 2.0 появился *асинхронный шаблон, основанный на событиях*. В данном шаблоне обработчик событий вызывается из потока, который владеет контекстом синхронизации, поэтому код обновления пользовательского интерфейса здесь прост. Ранее этот шаблон был известен также под названием *шаблона асинхронных компонентов*.

В .NET 4.5 введен новый подход к асинхронному программированию — *асинхронный шаблон, основанный на задачах* (TAP). Этот шаблон базируется на типе `Task`, появившемся в .NET 4, и применении ключевых слов `async` и `await`.

Чтобы оценить преимущество использования ключевых слов `async` и `await`, в первом примере приложения для предоставления обзора асинхронного программирования применяется инфраструктура Windows Presentation Foundation (WPF) и работа с сетью. Если вы не имеете опыта использования WPF и программирования для сети, переживать не следует. Вы все равно сможете уловить суть и понять принципы асинхронного программирования.

В последующих примерах демонстрируются отличия между асинхронными шаблонами. Затем с помощью простых консольных приложений будут проиллюстрированы основы асинхронного программирования.

> **На заметку!** Инфраструктура WPF подробно рассматривается в главах 35 и 36, а программирование для сети — в главе 26.

Для отражения отличий между асинхронными шаблонами создается WPF-приложение, в котором используются типы из библиотеки классов. Это приложение позволяет находить изображения в веб-сети с применением служб Bing и Flickr. Для нахождения изображений пользователь вводит критерий поиска, который отправляется службам Bing и Flickr в виде простого HTTP-запроса.

Внешний вид пользовательского интерфейса, построенного в Visual Studio, можно видеть на рис. 13.1. В верхней части расположено поле ввода, за которым следуют несколько кнопок, предназначенных для запуска поиска и очистки списка результатов. Слева под управляющей областью находится элемент управления `ListBox`, в котором будут отображаться все найденные изображения. Справа помещен элемент управления `Image`, который позволяет показывать изображение, выбранное в `ListBox`, с высоким разрешением.

Мы начнем с библиотеки классов `AsyncLib`, содержащей множество вспомогательных классов. Эти классы используются WPF-приложением.

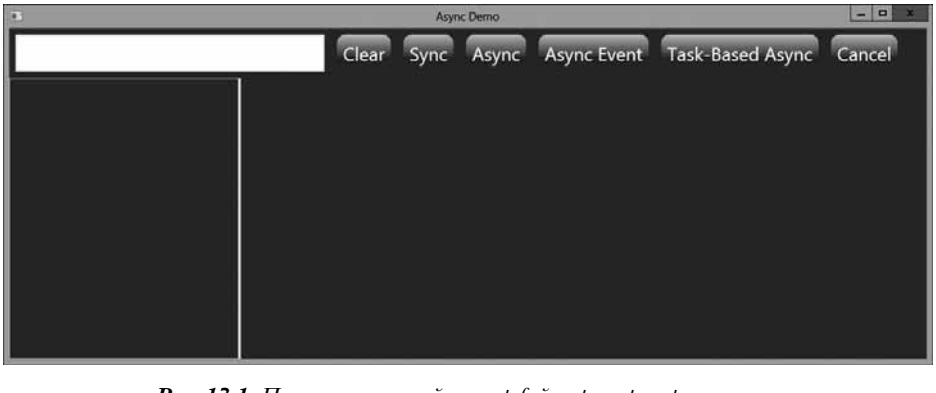

Рис. 13.1. *Пользовательский интерфейс примера приложения*

Класс SearchItemResult представляет одиночный элемент из результирующей коллекции, который используется для вывода изображения вместе с заголовком и источником, откуда было получено изображение. В этом классе просто определены простые свойства: Title, Url, ThumbnailUrl и Source. Свойство ThumbnailIUrl служит для ссылки на изображение-миниатюру, свойство Url содержит ссылку на изображение крупного размера. Свойство Title хранит текст, описывающий изображение. Базовым классом для SearchItemResult является BindableBase. Этот базовый класс всего лишь поддерживает механизм уведомлений за счет реализации интерфейса INotifyPropertyChanged, который применяется инфраструктурой WPF для организации обновлений через привязку данных (файл AsyncLib/SearchItemResult.cs):

```
namespace Wrox.ProCSharp.Async
{
  public class SearchItemResult : BindableBase
  {
    private string title;
    public string Title
    {
      get { return title; }
      set { SetProperty(ref title, value); }
    }
    private string url;
    public string Url
    {
      get { return url; }
      set { SetProperty(ref url, value); }
    }
    private string thumbnailUrl;
    public string ThumbnailUrl
    {
      get { return thumbnailUrl; }
      set { SetProperty(ref thumbnailUrl, value); }
    }
    private string source;
    public string Source
    {
      get { return source; }
      set { SetProperty(ref source, value); }
    }
  }
}
```

Еще одним классом, используемым с привязкой данных, является SearchInfo. Его свойство SearchTerm содержит пользовательский ввод для поиска изображений указанного типа. Свойство List возвращает список всех найденных изображений, представленных с помощью типа SearchItemResult (файл AsyncLib/SearchInfo.cs):

```
using System.Collections.ObjectModel;
namespace Wrox.ProCSharp.Async
{
  public class SearchInfo : BindableBase
  {
    public SearchInfo()
    {
      list = new ObservableCollection<SearchItemResult>();
      list.CollectionChanged += delegate { OnPropertyChanged("List"); };
    }
    private string searchTerm;
    public string SearchTerm
    {
      get { return searchTerm; }
      set { SetProperty(ref searchTerm, value); }
    }
```

```
      private ObservableCollection<SearchItemResult> list;
      public ObservableCollection<SearchItemResult> List
      {
         get
         {
            return list;
         }
      }
   }
}
```

В коде XAML для ввода критерия поиска применяется `TextBox`. Этот элемент управления привязан к свойству `SearchTerm` типа `SearchInfo`. Элементы управления `Button` запускают обработчики событий, например, кнопка Sync (Синхронизировать) вызывает метод `OnSearchSync()` (файл `AsyncPatterns/MainWindow.xaml`):

```
<StackPanel Orientation="Horizontal" Grid.Row="0">
  <StackPanel.LayoutTransform>
    <ScaleTransform ScaleX="2" ScaleY="2" />
  </StackPanel.LayoutTransform>
  <TextBox Text="{Binding SearchTerm}" Width="200" Margin="4" />
  <Button Click="OnClear">Clear</Button>
  <Button Click="OnSearchSync">Sync</Button>
  <Button Click="OnSearchAsyncPattern">Async</Button>
  <Button Click="OnAsyncEventPattern">Async Event</Button>
  <Button Click="OnTaskBasedAsyncPattern">Task Based Async</Button>
</StackPanel>
```

Вторая часть кода XAML содержит элемент управления `ListBox`. Для создания специального представления элементов в `ListBox` применяется шаблон `ItemTemplate`. Каждый элемент списка представлен с помощью двух элементов управления `TextBlock` и одного `Image`. Элемент управления `ListBox` привязан к свойству `List` класса `SearchInfo`, а свойства элементов управления для элемента списка привязаны к свойствам типа `SearchItemResult`:

```
<Grid Grid.Row="1">
  <Grid.ColumnDefinitions>
    <ColumnDefinition Width="*" />
    <ColumnDefinition Width="3*" />
  </Grid.ColumnDefinitions>
  <ListBox Grid.IsSharedSizeScope="True" ItemsSource="{Binding List}"
           Grid.Column="0" IsSynchronizedWithCurrentItem="True"
           Background="Black">
    <ListBox.ItemTemplate>
      <DataTemplate>
        <Grid>
          <Grid.ColumnDefinitions>
            <ColumnDefinition SharedSizeGroup="ItemTemplateGroup" />
          </Grid.ColumnDefinitions>
          <StackPanel HorizontalAlignment="Stretch" Orientation="Vertical"
                      Background="{StaticResource linearBackgroundBrush}">
            <TextBlock Text="{Binding Source}" Foreground="White" />
            <TextBlock Text="{Binding Title}" Foreground="White" />
            <Image HorizontalAlignment="Center"
                   Source="{Binding ThumbnailUrl}" Width="100" />
          </StackPanel>
        </Grid>
      </DataTemplate>
    </ListBox.ItemTemplate>
  </ListBox>
  <GridSplitter Grid.Column="1" Width="3" HorizontalAlignment="Left" />
  <Image Grid.Column="1" Source="{Binding List/Url}" />
</Grid>
```

А теперь давайте займемся анализом класса `BingRequest`. Этот класс содержит некоторую информацию о том, как отправлять запрос службе Bing. Свойство `Url` данного класса возвращает строку URL, которая может применяться для запрашивания изображений. Запрос состоит из критерия поиска, количества изображений, которые должны быть запрошены (`Count`), и количества изображений, которые должны быть пропущены (`Offset`). Для службы Bing требуется аутентификация. Идентификатор пользователя определен в `AppId`, и он применяется со свойством `Credentials`, которое возвращает объект `NetworkCredential`. Для успешного запуска приложения необходимо зарегистрироваться в магазине Windows Azure Marketplace и подписаться на Bing Search API. На момент написания этой книги первые 5000 транзакций в месяц были бесплатными — этого должно хватить для работы с примером приложения. Каждый поиск представляет собой одну транзакцию. Подписка на Bing Search API производится по адресу `https://datamarket.azure.com/dataset/bing/search`. После регистрации понадобится скопировать идентификатор приложения. Этот идентификатор приложения необходимо добавить в класс `BingRequest`.

В результате отправки запроса к Bing с использованием созданного URL служба Bing возвращает разметку XML. Метод `Parse()` класса `BingRequest` позволяет выполнить разбор этой разметки XML и возвращает коллекцию объектов `SearchItemResult` (файл `AsyncLib/BingRequest.cs`):

> **На заметку!** Методы `Parse()` в классах `BingRequest` и `FlickrRequest` пользуются LINQ to XML. Технология LINQ to XML описана в главе 34.

```
using System.Collections.Generic;
using System.Linq;
using System.Net;
using System.Xml.Linq;

namespace Wrox.ProCSharp.Async
{
  public class BingRequest : IImageRequest
  {
    private const string AppId = "enter your Bing AppId here";

    public BingRequest()
    {
      Count = 50;
      Offset = 0;
    }

    private string searchTerm;
    public string SearchTerm
    {
      get { return searchTerm; }
      set { searchTerm = value; }
    }

    public ICredentials Credentials
    {
      get
      {
        return new NetworkCredentials(AppId, AppId);
      }
    }

    public string Url
    {
      get
      {
```

```
            return string.Format("https://api.datamarket.azure.com/" +
               "Data.ashx/Bing/Search/v1/Image?Query=%27{0}%27&" +
               "$top={1}&$skip={2}&$format=Atom",
               SearchTerm, Count, Offset);
         }
      }

      public int Count { get; set; }
      public int Offset { get; set; }

      public IEnumerable<SearchItemResult> Parse(string xml)
      {
         XElement respXml = XElement.Parse(xml);
         // XNamespace atom = XNamespace.Get("http://www.w3.org/2005/Atom");
         XNamespace d = XNamespace.Get(
            "http://schemas.microsoft.com/ado/2007/08/dataservices");
         XNamespace m = XNamespace.Get(
            "http://schemas.microsoft.com/ado/2007/08/dataservices/metadata");

         return (from item in respXml.Descendants(m + "properties")
                 select new SearchItemResult
                 {
                    Title = new string(item.Element(d +
                       "Title").Value.Take(50).ToArray()),
                    Url = item.Element(d + "MediaUrl").Value,
                    ThumbnailUrl = item.Element(d + "Thumbnail").
                    Element(d + "MediaUrl").Value,
                       Source = "Bing"
                 }).ToList();
      }
   }
}
```

Классы `BingRequest` и `FlickrRequest` реализуют интерфейс `IImageRequest`. В этом интерфейсе определены свойства `SearchTerm` и `Url`, а также метод `Parse()`, который позволяет организовать простую итерацию по обоим поставщикам служб изображений (файл `AsyncLib/IImageRequest.cs`):

```
using System;
using System.Collections.Generic;
using System.Net;

namespace Wrox.ProCSharp.Async
{
   public interface IImageRequest
   {
      string SearchTerm { get; set; }
      string Url { get; }
      IEnumerable<SearchItemResult> Parse(string xml);
      ICredentials Credentials { get; }
   }
}
```

Класс `FlickrRequest` очень похож на `BingRequest`. Он только создает другой URL для запрашивания изображений по критерию поиска и содержит собственную реализацию метода `Parse()`, поскольку разметка XML, возвращаемая службой Flickr, отличается от разметки, которую возвращает Bing. Как и в случае Bing, чтобы создать идентификатор приложения для Flickr, необходимо зарегистрироваться в службе Flickr и запросить его: `http://www.flickr.com/services/apps/create/apply/`.

```
using System.Collections.Generic;
using System.Linq;
using System.Xml.Linq;
```

```
namespace Wrox.ProCSharp.Async
{
   public class FlickrRequest : IImageRequest
   {
      private const string AppId = "Enter your Flickr AppId here";
      public FlickrRequest()
      {
         Count = 50;
         Page = 1;
      }

      private string searchTerm;
      public string SearchTerm
      {
         get { return searchTerm; }
         set { searchTerm = value; }
      }

      public string Url
      {
         get
         {
            return string.Format("http://api.flickr.com/services/rest?" +
               "api_key={0}&method=flickr.photos.search&content_type=1&" +
               "text={1}&per_page={2}&page={3}", AppId, SearchTerm, Count, Page);
         }
      }

      public ICredentials Credentials
      {
         get { return null; }
      }

      public int Count { get; set; }
      public int Page { get; set; }
      public IEnumerable<SearchItemResult> Parse(string xml)
      {
         XElement respXml = XElement.Parse(xml);
         return (from item in respXml.Descendants("photo")
                 select new SearchItemResult
                 {
                    Title = new string(item.Attribute("title").Value.
                       Take(50).ToArray()),
                    Url = string.Format("http://farm{0}.staticflickr.com/" +
                       "{1}/{2}_{3}_z.jpg",
                       item.Attribute("farm").Value, item.Attribute("server").Value,
                       item.Attribute("id").Value, item.Attribute("secret").Value),
                    ThumbnailUrl = string.Format("http://farm{0}." +
                       "staticflickr.com/{1}/{2}_{3}_t.jpg",
                       item.Attribute("farm").Value,
                       item.Attribute("server").Value,
                       item.Attribute("id").Value,
                       item.Attribute("secret").Value),
                    Source = "Flickr"
                 }).ToList();
      }
   }
}
```

Теперь нужно просто соединить типы из библиотеки и WPF-приложение. В конструкторе класса `MainWindow` создается экземпляр `SearchInfo` и затем присваивается свойству `DataContext` окна. Привязка данных, показанная ранее в коде XAML, начинает действовать (файл `AsyncPatterns/MainWindow.xaml.cs`):

```
public partial class MainWindow : Window
{
   private SearchInfo searchInfo;
   public MainWindow()
   {
      InitializeComponent();
      searchInfo = new SearchInfo();
      this.DataContext = searchInfo;
   }
```

Класс `MainWindow` также содержит вспомогательный метод `GetSearchRequests()`, который возвращает коллекцию объектов `IImageRequest` в форме типов `BingRequest` и `FlickrRequest`. В случае если вы зарегистрировались только в одной из этих служб, измените соответствующим образом код. Разумеется, можно также создать типы `IImageRequest` для других служб, например, Google или Yahoo. Добавьте типы запросов в возвращаемую коллекцию:

```
private IEnumerable<IImageRequest> GetSearchRequests()
{
   return new List<IImageRequest>
   {
      new BingRequest { SearchTerm = searchInfo.SearchTerm },
      new FlickrRequest { SearchTerm = searchInfo.SearchTerm}
   };
}
```

Синхронный вызов

Теперь, когда все должным образом настроено, давайте начнем с синхронного вызова указанных служб. В обработчике событий щелчка на кнопке Sync, `OnSearchSync()`, производится проход по всем поисковым запросам, возвращенным из метода `GetSearchRequests()`, и с помощью класса `WebClient` делается HTTP-запрос с применением свойства `Url`. Метод `DownloadString()` блокируется до тех пор, пока не будет получен результат. Результирующая разметка XML присваивается переменной `resp`. Эта разметка XML анализируется с помощью метода `Parse()`, который возвращает коллекцию объектов `SearchItemResult`. Элементы этой коллекции затем добавляются в список, содержащийся внутри `searchInfo` (файл `AsyncPatterns/MainWindow.xaml.cs`):

```
private void OnSearchSync(object sender, RoutedEventArgs e)
{
   foreach (var req in GetSearchRequests())
   {
      var client = new WebClient();
      client.Credentials = req.Credentials;
      string resp = client.DownloadString(req.Url);
      IEnumerable<SearchItemResult> images = req.Parse(resp);
      foreach (var image in images)
      {
         searchInfo.List.Add(image);
      }
   }
}
```

В функционирующем приложении (рис. 13.2) пользовательский интерфейс блокируется до тех пор, пока метод `OnSearchSync()` не завершит обращения через сеть к службам Bing и Flickr, а также анализ результатов. Промежуток времени, необходимый для завершения этих вызовов, варьируется в зависимости от пропускной способности сети и текущей рабочей нагрузке служб Bing и Flickr. Тем не менее, в любом случае ожидание неприятно для пользователя.

Рис. 13.2. *Пример приложения во время выполнения*

Итак, давайте реализуем вызов асинхронным образом.

Асинхронный шаблон

Один из способов сделать вызов асинхронным предусматривает использование асинхронного шаблона. При асинхронном шаблоне определяются методы `Begin`*`XXX`*`()` и `End`*`XXX`*`()`. Например, если предлагается синхронный метод `DownloadString()`, то асинхронными вариантами будут `BeginDownloadString()` и `EndDownloadString()`. Метод `Begin`*`XXX`*`()` принимает все входные аргументы синхронного метода, а `End`*`XXX`*`()` — все выходные аргументы и возвращаемый тип, чтобы вернуть результат. В асинхронном шаблоне метод `Begin`*`XXX`*`()` также определяет параметр `AsyncCallback`, который принимает делегат, вызываемый по завершении асинхронного метода. Метод `Begin`*`XXX`*`()` возвращает объект `IAsyncResult`, который может применяться для опроса с целью проверки, завершился ли вызов, и ожидания, пока метод закончится.

Класс `WebClient` не предоставляет реализацию асинхронного шаблона. Вместо него мог бы использоваться класс `HttpWebRequest`, который поддерживает этот шаблон посредством методов `BeginGetResponse()` и `EndGetResponse()`. В следующем примере это не делается, а применяется делегат. Тип делегата определяет метод `Invoke()` для выполнения синхронного вызова метода и методы `BeginInvoke()` и `EndInvoke()` для его использования в асинхронном шаблоне. В примере объявляется делегат `downloadString` типа `Func<string, string>` для ссылки на метод, который принимает параметр `string` и возвращает тип `string`. Метод, на который ссылается переменная `downloadString`, реализован в виде лямбда-выражения и вызывает синхронный метод `DownloadString()` типа `WebClient`. Делегат вызывается асинхронно путем вызова метода `BeginInvoke()`. Для выполнения асинхронного вызова этот метод применяет поток из пула потоков.

Первый параметр метода `BeginInvoke()` — это первый обобщенный параметр `string` делегата `Func`, в котором можно передавать URL. Второй параметр имеет тип `AsyncCallback`. Тип `AsyncCallback` представляет собой делегат, который требует `IAsyncResult` в качестве параметра. Метод, на который ссылаются с помощью этого делегата, вызывается по завершении асинхронного метода. Когда это происходит, вызывается метод `downloadString.EndInvoke()` для извлечения результата, который обрабатывается тем же самым способом, что и описанный ранее — анализ разметки XML и получение коллекции элементов. Однако здесь обратиться напрямую к пользовательскому интерфейсу не удастся, т.к. пользовательский интерфейс привязан к единственному потоку, а метод обратного вызова выполняется внутри фонового потока. Следовательно, необходимо переключиться на поток пользовательского интерфейса с применением свойства `Dispatcher` окна. Метод `Invoke()` класса `Dispatcher` требует в качестве параметра делегат; именно

поэтому указан делегат Action<SearchItemResult>, который добавляет элемент в коллекцию, привязанную к пользовательскому интерфейсу (файл AsyncPatterns/MainWindow.xaml.cs):

```
private void OnSearchAsyncPattern(object sender, RoutedEventArgs e)
{
   Func<string, ICredentials, string> downloadString = (address, cred) =>
   {
      var client = new WebClient();
      client.Credentials = cred;
      return client.DownloadString(address);
   };
   Action<SearchItemResult> addItem = item => searchInfo.List.Add(item);
   foreach (var req in GetSearchRequests())
   {
      downloadString.BeginInvoke(req.Url, req.Credentials, ar =>
      {
         string resp = downloadString.EndInvoke(ar);
         IEnumerable<SearchItemResult> images = req.Parse(resp);
         foreach (var image in images)
         {
            this.Dispatcher.Invoke(addItem, image);
         }
      }, null);
   }
}
```

Преимущество асинхронного шаблона заключается в том, что он может быть легко реализован с использованием функциональности делегатов. Теперь приложение ведет себя так, как должно; пользовательский интерфейс больше не блокируется. Однако применение асинхронного шаблона сопряжено и с трудностями. К счастью, в версии .NET 2.0 появился асинхронный шаблон, основанный на событиях, который упрощает реализацию обновлений пользовательского интерфейса. Этот шаблон обсуждается в следующем разделе.

На заметку! Типы делегатов и лямбда-выражения рассматривались в главе 8. Потоки и пулы потоков будут описаны в главе 21.

Асинхронный шаблон, основанный на событиях

Асинхронный шаблон, основанный на событиях, используется в методе OnAsyncEventPattern(). Данный шаблон реализован классом WebClient, следовательно, им можно пользоваться напрямую. Согласно этому шаблону, определяется метод, имя которого заканчивается на Async. Таким образом, например, для синхронного метода DownloadString() класс WebClient предлагает асинхронный вариант DownloadStringAsync(). Вместо объявления делегата, вызываемого по завершении асинхронного метода, определяется событие. Событие DownloadStringCompleted инициируется при завершении асинхронного метода DownloadStringAsync(). Метод, назначенный в качестве обработчика событий, реализован в виде лямбда-выражения. Реализация очень похожа на предыдущую, но теперь возможно напрямую обращаться к элементам пользовательского интерфейса, т.к. обработчик событий вызывается из потока, имеющего контекст синхронизации, а в случае приложений Windows Forms и WPF это будет сам поток пользовательского интерфейса (файл AsyncPatterns/MainWindow.xaml.cs):

```
private void OnAsyncEventPattern(object sender, RoutedEventArgs e)
{
   foreach (var req in GetSearchRequests())
   {
      var client = new WebClient();
```

```
      client.Credentials = req.Credentials;
      client.DownloadStringCompleted += (sender1, e1) =>
         {
            string resp = e1.Result;
            IEnumerable<SearchItemResult> images = req.Parse(resp);
            foreach (var image in images)
            {
               searchInfo.List.Add(image);
            }
         };
      client.DownloadStringAsync(new Uri(req.Url));
   }
}
```

Преимущество асинхронного шаблона, основанного на событиях, связано с простотой его использования. Однако следует отметить, что этот шаблон не так легко реализовать в специальном классе. Класс `BackgroundWorker` позволяет воспользоваться существующей реализацией этого шаблона для превращения синхронных методов в асинхронные. Упомянутый класс реализует асинхронный шаблон, основанный на событиях.

Это намного упрощает код. Тем не менее, по сравнению с вызовами синхронных методов порядок является обратным. То есть определять, что произойдет, когда вызов метода завершится, нужно перед вызовом асинхронного метода. В следующем разделе мы погрузимся в новый мир асинхронного программирования с применением ключевых слов `async` и `await`.

Асинхронный шаблон, основанный на задачах

В версии .NET 4.5 класс `WebClient` был обновлен, чтобы поддерживать также и асинхронный шаблон, основанный на задачах (TAP). Этот шаблон предполагает определение метода, имя которого заканчивается на `Async`, возвращающего тип `Task`. Поскольку класс `WebClient` уже предлагает метод с суффиксом `Async` для реализации асинхронного шаблона, основанного на событиях, новому методу назначено имя `DownloadStringTaskAsync()`.

Метод `DownloadStringTaskAsync()` объявлен как возвращающий `Task<string>`. Нет никакой необходимости в объявлении переменной типа `Task<string>` для присваивания результата, возвращаемого методом `DownloadStringTaskAsync()`; вместо этого можно объявить переменную типа `string` и применить ключевое слово `await`. Ключевое слово `await` разблокирует поток (в случае потока пользовательского интерфейса) для работы других задач. После того, как метод `DownloadStringTaskAsync()` завершит свою фоновую обработку, поток пользовательского интерфейса сможет продолжить, получив результат из фоновой задачи в строковую переменную `resp`. Выполнение будет продолжено с кода, следующего за этой строкой (файл `AsyncPatterns/MainWindow.xaml.cs`):

```
private async void OnTaskBasedAsyncPattern(object sender,
                                           RoutedEventArgs e)
{
   foreach (var req in GetSearchRequests())
   {
      var client = new WebClient();
      client.Credentials = req.Credentials;
      string resp = await client.DownloadStringTaskAsync(req.Url);

      IEnumerable<SearchItemResult> images = req.Parse(resp);
      foreach (var image in images)
      {
         searchInfo.List.Add(image);
      }
   }
}
```

На заметку! Ключевое слово `async` создает конечный автомат подобно оператору `yield return`, который обсуждался в главе 6.

Код теперь стал намного проще. Отсутствует блокирование, и нет ручного переключения на поток пользовательского интерфейса, т.к. это делается автоматически; к тому же код записывается в том же порядке, что и при синхронном программировании.

Далее код изменяется для использования класса, отличного от `WebClient` — одного из тех, где асинхронный шаблон, основанный на задачах, реализован непосредственно, а синхронные методы вообще не предлагаются. Такой класс появился в .NET 4.5 и называется он `HttpClient`. Асинхронный запрос `GET` выполняется с помощью метода `GetAsync()`. Для чтения содержимого необходим еще один асинхронный метод. Метод `ReadAsStringAsync()` возвращает содержимое в виде строки.

```
private async void OnTaskBasedAsyncPattern(object sender, RoutedEventArgs e)
{
   foreach (var req in GetSearchRequests())
   {
      var clientHandler = new HttpClientHandler
      {
         Credentials = req.Credentials
      };
      var client = new HttpClient(clientHandler);
      var response = await client.GetAsync(req.Url);
      string resp = await response.Content.ReadAsStringAsync();
      IEnumerable<SearchItemResult> images = req.Parse(resp);
      foreach (var image in images)
      {
         searchInfo.List.Add(image);
      }
   }
}
```

Разбор строки XML может занять некоторое время. Поскольку код разбора выполняется в потоке пользовательского интерфейса, в это время данный поток не может реагировать на запросы, поступающие от пользователя. Чтобы создать из синхронной функциональности фоновую задачу, можно применить метод `Task.Run()`. В показанном ниже примере внутри `Task.Run()` производится разбор строки XML для возврата коллекции `SearchItemResult`:

```
private async void OnTaskBasedAsyncPattern(object sender, RoutedEventArgs e)
{
   foreach (var req in GetSearchRequests())
   {
      var clientHandler = new HttpClientHandler
      {
         Credentials = req.Credentials
      };
      var client = new HttpClient(clientHandler);
      var response = await client.GetAsync(req.Url, cts.Token);
      string resp = await response.Content.ReadAsStringAsync();
      await Task.Run(() =>
      {
         IEnumerable<SearchItemResult> images = req.Parse(resp);
         foreach (var image in images)
         {
            searchInfo.List.Add(image);
         }
      }
   }
}
```

Из-за того, что метод, передаваемый `Task.Run()`, выполняется в фоновом потоке, возникает та же самая проблема с обращением к коду пользовательского интерфейса, которая была описана ранее. Решение могло бы заключаться в вызове `req.Parse()` внутри метода `Task.Run()` и организации цикла `foreach` за пределами задачи для добавления результата к списку в потоке пользовательского интерфейса.

Однако инфраструктура WPF в .NET 4.5 предлагает лучшее решение, которое позволяет заполнять в фоновом потоке коллекции, привязанные к элементам управления пользовательского интерфейса. Это расширение требует только включения синхронизации для коллекции с помощью метода `BindingOperations.EnableCollectionSynchronization()`, как показано в следующем фрагменте кода:

```
public partial class MainWindow : Window
{
   private SearchInfo searchInfo;
   private object lockList = new object();

   public MainWindow()
   {
      InitializeComponent();
      searchInfo = new SearchInfo();
      this.DataContext = searchInfo;
      BindingOperations.EnableCollectionSynchronization(
         searchInfo.List, lockList);
   }
```

После демонстрации преимуществ применения ключевых слов `async` и `await` в следующем разделе рассматривается внутренняя поддержка, лежащая в основе этих слов.

Основы асинхронного программирования

Ключевые слова `async` и `await` являются просто средством компилятора. Компилятор создает код, использующий класс `Task`. Вместо применения этих новых ключевых слов ту же функциональность можно получить с помощью C# 4 и методов класса `Task`, но не настолько удобно.

В этом разделе приведена информация о том, что делает компилятор с ключевыми словами `async` и `await`, предложен простой способ создания асинхронного метода, а также показано, как вызвать несколько асинхронных методов параллельно, и каким образом изменить класс, поддерживающий только асинхронный шаблон, для использования новых ключевых слов.

Создание задач

Давайте начнем с синхронного метода `Greeting()`, который делает паузу и затем возвращает строку (файл `Foundations/Program.cs`):

```
static string Greeting(string name)
{
   Thread.Sleep(3000);
   return string.Format("Hello, {0}", name);
}
```

Чтобы сделать метод такого рода асинхронным, определим метод `GreetingAsync()`. Согласно асинхронному шаблону, основанному на задачах, асинхронный метод имеет имя, заканчивающееся на `Async`, и возвращает задачу. Метод `GreetingAsync()` определен с тем же входным параметром, что и `Greeting()`, но с возвращаемым типом `Task<string>`. Тип `Task<string>` определяет задачу, которая возвращает строку. Проще всего возвратить задачу с применением метода `Task.Run()`. Обобщенная версия метода `Task.Run<string>()` создает задачу, возвращающую строку:

```
static Task<string> GreetingAsync(string name)
{
   return Task.Run<string>(() =>
      {
         return Greeting(name);
      });
}
```

Вызов асинхронного метода

Вызвать асинхронный метод `GreetingAsync()` можно с использованием ключевого слова `await` для возвращенной задачи. Ключевое слово `await` требует, чтобы метод был объявлен с модификатором `async`. Код внутри этого метода не будет продолжать выполнение, пока не завершится метод `GreetingAsync()`. Тем не менее, поток, запустивший метод `CallerWithAsync()`, можно использовать повторно. Этот поток не блокируется:

```
private async static void CallerWithAsync()
{
   string result = await GreetingAsync("Stephanie");
   Console.WriteLine(result);
}
```

Вместо передачи результата из асинхронного метода в переменную можно также применять ключевое слово `await` внутри параметров. В следующем коде результат из метода `GreetingAsync()` ожидается посредством `await`, как и в предыдущем фрагменте кода, но на этот раз результат передается прямо методу `Console.WriteLine()`:

```
private async static void CallerWithAsync2()
{
   Console.WriteLine(await GreetingAsync("Stephanie"));
}
```

На заметку! Модификатор `async` может использоваться только с методами, возвращающими `Task` или `void`. Его нельзя применять для точки входа в программу (для метода `Main()`). Ключевое слово `await` может использоваться только с методами, возвращающими `Task`.

В следующем разделе будет показано, что является движущей силой ключевого слова `await`. На самом деле "за кулисами" применяются задачи продолжения.

Задачи продолжения

Метод `GreetingAsync()` возвращает объект `Task<string>`. Объект `Task` содержит информацию о созданной задаче и позволяет организовать ожидание ее завершения. Метод `ContinueWith()` класса `Task` определяет код, который должен быть вызван, когда задача завершается. Делегат, заданный для метода `ContinueWith()`, принимает завершенную задачу в своем аргументе, что позволяет получить доступ к результату задачи с помощью свойства `Result`:

```
private static void CallerWithContinuationTask()
{
   Task<string> t1 = GreetingAsync("Stephanie");
   t1.ContinueWith(t =>
      {
         string result = t.Result;
         Console.WriteLine(result);
      });
}
```

Компилятор преобразует ключевое слово `await`, помещая весь следующий за ним код внутрь блока метода `ContinueWith()`.

Контекст синхронизации

На протяжении времени существования внутри методов `CallerWithAsync()` и `CallerWithContinuationTask()` используются разные потоки. Один поток применяется для вызова метода `GreetingAsync()`, а другой поток начинает действовать после ключевого слова `await` или внутри блока кода в методе `ContinueWith()`.

В консольном приложении обычно это не является проблемой. Однако необходимо обеспечить, чтобы, по меньшей мере, один поток переднего плана функционировал, прежде чем все фоновые задачи будут завершены. В примере приложения вызывается метод `Console.ReadLine()` для удержания главного потока в режиме выполнения вплоть до нажатия клавиши <Enter>.

С другой стороны, в приложениях, которые привязаны к конкретному потоку для выполнения определенных действий (например, в WPF-приложениях элементы пользовательского интерфейса могут быть доступны только из потока пользовательского интерфейса), это будет проблемой.

Применяя ключевые слова `async` и `await`, не потребуется предпринимать никаких специальных действий для доступа к потоку пользовательского интерфейса после завершения `await`. По умолчанию сгенерированный код переключается на поток, имеющий контекст синхронизации. Приложение WPF устанавливает объект `DispatcherSynchronizationContext`, а приложение Windows Forms — объект `WindowsFormsSynchronizationContext`. Если вызывающему потоку асинхронного метода назначен контекст синхронизации, то при продолжении выполнения после `await` по умолчанию используется тот же самый контекст синхронизации. Если тот же самый контекст синхронизации не должен применяться, потребуется вызвать метод `ConfigureAwait( continueOnCapturedContext: false)` класса `Task`. Примером может служить WPF-приложение, в котором код, следующий за `await`, не обращается к каким-либо элементам пользовательского интерфейса. В этом случае эффективнее избежать переключения контекста синхронизации.

Использование множества асинхронных методов

Внутри асинхронного метода можно вызывать не только какой-то один, но и несколько асинхронных методов. Способ реализации зависит от того, должны ли результаты, возвращаемые одним асинхронным методом, передаваться другому такому методу.

Последовательный вызов асинхронных методов

Для вызова каждого асинхронного метода можно использовать ключевое слово `await`. В случае, когда один метод зависит от результатов выполнения другого метода, это очень удобно. В приведенном ниже коде второй вызов `GreetingAsync()` совершенно не зависит от результата первого вызова `GreetingAsync()`. Следовательно, метод `MultipleAsyncMethods()` мог бы возвратить результат быстрее, если не применять `await` с каждым отдельным методом:

```
private async static void MultipleAsyncMethods()
{
   string s1 = await GreetingAsync("Stephanie");
   string s2 = await GreetingAsync("Matthias");
   Console.WriteLine("Finished both methods.\n " +
      "Result 1: {0}\n Result 2: {1}", s1, s2);
}
```

Использование комбинаторов

Если асинхронные методы не зависят друг от друга, намного быстрее не указывать `await` для каждого метода по отдельности, а вместо этого присвоить результаты работы асинхронных методов переменным типа `Task`.

Метод GreetingAsync() возвращает Task<string>. Оба метода теперь могут выполняться параллельно. Помогут в этом *комбинаторы*. Комбинатор принимает множество параметров одного типа и возвращает значение того же самого типа. Передаваемые параметры "комбинируются" в один. Комбинаторы Task принимают несколько объектов Task в качестве параметров и возвращают объект Task.

В следующем примере кода вызывается метод комбинатора Task.WhenAll(), к которому можно применить await для ожидания завершения обеих задач:

```
private async static void MultipleAsyncMethodsWithCombinators1()
{
   Task<string> t1 = GreetingAsync("Stephanie");
   Task<string> t2 = GreetingAsync("Matthias");
   await Task.WhenAll(t1, t2);
   Console.WriteLine("Finished both methods.\n " +
      "Result 1: {0}\n Result 2: {1}", t1.Result, t2.Result);
}
```

В классе Task определены комбинаторы WhenAll() и WhenAny(). Задача, возвращаемая из метода WhenAll(), завершается, когда завершены все задачи, переданные методу; задача, возвращаемая из метода WhenAny(), завершается, когда завершена одна из задач, переданных методу.

Для метода WhenAll() в классе Task предусмотрено несколько перегруженных версий. Если все задачи возвращают один и тот же тип, await можно применить к массиву этого типа. Метод GreetingAsync() возвращает Task<string>, и результатом await для этого метода будет string. Следовательно, метод Task.WhenAll() можно использовать для возврата массива string:

```
private async static void MultipleAsyncMethodsWithCombinators2()
{
   Task<string> t1 = GreetingAsync("Stephanie");
   Task<string> t2 = GreetingAsync("Matthias");
   string[] result = await Task.WhenAll(t1, t2);
   Console.WriteLine("Finished both methods.\n " +
      "Result 1: {0}\n Result 2: {1}", result[0], result[1]);
}
```

Преобразование асинхронного шаблона

Не все классы в .NET Framework поддерживают новый стиль асинхронных методов, появившийся в версии .NET 4.5. Многие классы по-прежнему предлагают реализацию асинхронного шаблона с методами Begin*XXX*() и End*XXX*(), а не асинхронные методы, основанные на задачах.

Для начала давайте создадим асинхронный метод из ранее определенного синхронного метода Greeting() с помощью делегата. Метод Greeting() получает строку в качестве параметра и возвращает строку, поэтому для ссылки на данный метод применяется переменная типа делегата Func<string, string>. Согласно асинхронному шаблону, в дополнение к параметру AsyncCallback метод BeginGreeting() принимает параметр типа string и возвращает IAsyncResult. Метод EndGreeting() возвращает результат, полученный из метода Greeting() — объект string — и принимает параметр типа IAsyncResult. Для асинхронной реализации используется только упомянутый делегат.

```
private static Func<string, string> greetingInvoker = Greeting;

static IAsyncResult BeginGreeting(string name, AsyncCallback callback,
                                  object state)
{
   return greetingInvoker.BeginInvoke(name, callback, state);
}
```

```
static string EndGreeting(IAsyncResult ar)
{
   return greetingInvoker.EndInvoke(ar);
}
```

Теперь методы BeginGreeting() и EndGreeting() доступны, и можно приступать к преобразованию кода с целью применения ключевых слов async и await для получения результата. В классе TaskFactory определен метод FromAsync(), который позволяет преобразовать методы, использующие асинхронный шаблон, в шаблон TAP.

В приведенном ниже примере кода первый обобщенный параметр типа Task, Task<string>, определяет возвращаемое значение вызываемого метода. Обобщенный параметр метода FromAsync() определяет входной тип метода. В этом случае входным типом снова является string. Первые два параметра метода FromAsync() имеют типы делегатов для передачи адресов методов BeginGreeting() и EndGreeting(). За ними следуют входные параметры и параметр состояния объекта. Состояние объекта не используется, так что для него указывается null. Поскольку метод FromAsync() возвращает тип Task, в примере кода Task<string> ключевое слово await может применяться следующим образом:

```
private static async void ConvertingAsyncPattern()
{
   string s = await Task<string>.Factory.FromAsync<string>(
      BeginGreeting, EndGreeting, "Angela", null);
   Console.WriteLine(s);
}
```

Обработка ошибок

Подробное описание обработки ошибок и исключений представлено в главе 16. Тем не менее, в контексте асинхронных методов вы должны быть осведомлены о некоторых специальных способах обработки ошибок. Начнем с простого метода, который генерирует исключение после заданной паузы (файл ErrorHandling/Program.cs):

```
static async Task ThrowAfter(int ms, string message)
{
   await Task.Delay(ms);
   throw new Exception(message);
}
```

Если асинхронный метод вызывается без await, вызов можно поместить внутрь блока try/catch, однако исключение не будет перехвачено. Причина в том, что метод DontHandle() завершится до того, как сгенерируется исключение в ThrowAfter(). К вызову метода ThrowAfter() понадобится применить ключевое слово await.

```
private static void DontHandle()
{
   try
   {
      ThrowAfter(200, "first");
      // Исключение не будет перехвачено, т.к. этот метод
      // завершится до его генерации.
   }
   catch (Exception ex)
   {
      Console.WriteLine(ex.Message);
   }
}
```

Внимание! К асинхронным методам, возвращающим `void`, ключевое слово `await` не может быть применено. Проблема в том, что исключения, генерируемые внутри методов `async void`, не могут быть перехвачены. Именно поэтому лучше возвращать из асинхронного метода тип `Task`. Данное правило не распространяется на методы обработчиков или переопределенные методы базовых классов.

Обработка исключений, возникающих в асинхронных методах

Удобный способ обработки исключений, возникающих в асинхронных методах, заключается в применении ключевого слова `await` и помещении вызовов этих методов внутрь оператора `try/catch`, как показано в приведенном фрагменте кода. Метод `HandleOneError()` освобождает поток после вызова метода `ThrowAfter()` асинхронным образом, но сохраняет объект `Task` для продолжения работы после того, как задача завершается. Когда это происходит (в данном случае, когда исключение генерируется через две секунды), условие в `catch` дает совпадение и код внутри блока `catch` выполняется:

```
private static async void HandleOneError()
{
   try
   {
      await ThrowAfter(2000, "first");
   }
   catch (Exception ex)
   {
      Console.WriteLine("handled {0}", ex.Message);
   }
}
```

Исключения, возникающие в нескольких асинхронных методах

Что делать в ситуации, когда оба асинхронных метода, которые были вызваны, генерируют исключения? В следующем примере вызывается первый метод `ThrowAfter()`, который через две секунды генерирует исключение с сообщением `"first"`. По его завершении этого вызова метод `ThrowAfter()` вызывается еще раз для того, чтобы спустя одну секунду сгенерировать исключение с сообщением `"second"`. Так как первый вызов метода `ThrowAfter()` уже сгенерировал исключение, код внутри блока `try` выполняться больше не будет и до второго вызова дело не дойдет, а управление будет передано в блок `catch` для обработки первого исключения.

```
private static async void StartTwoTasks()
{
   try
   {
      await ThrowAfter(2000, "first");
      await ThrowAfter(1000, "second");    // Второй вызов не будет выполнен,
                     // поскольку первый вызов метода генерирует исключение.
   }
   catch (Exception ex)
   {
      Console.WriteLine("handled {0}", ex.Message);
   }
}
```

Теперь давайте обеспечим параллельную работу двух вызовов `ThrowAfter()`. Первый вызов генерирует исключение через две секунды, а второй — через одну секунду. За счет применения метода `Task.WhenAll()` организуется ожидание завершения обеих задач, независимо от того, возникло исключение или нет. Следовательно, спустя две секунды вызов

Task.WhenAll() завершится, а исключение будет перехвачено оператором catch. Однако вы увидите только информацию об исключении из первого вызова ThrowAfter(), которая была передана методу WhenAll(). Это не та задача, которая сгенерировала исключение первой (на самом деле это сделала вторая задача), но это первая задача в списке.

```
private async static void StartTwoTasksParallel()
{
   try
   {
      Task t1 = ThrowAfter(2000, "first");
      Task t2 = ThrowAfter(1000, "second");
      await Task.WhenAll(t1, t2);
   }
   catch (Exception ex)
   {
      // Просто отображает информацию об исключении, возникшем
      // в первой задаче, ожидаемой с помощью метода WhenAll()
      Console.WriteLine("handled {0}", ex.Message);
   }
}
```

Один из способов получения информации об исключениях из всех задач предусматривает объявление переменных t1 и t2 для представления задач за пределами блока try, что позволит обращаться к ним в блоке catch. Здесь с помощью свойства IsFaulted можно проверить состояние задачи для определения состояния сбоя. Если было сгенерировано исключение, свойство IsFaulted возвратит true. Информацию об исключении можно получить с использованием свойства Exception.InnerException класса Task. Другой, обычно более эффективный способ извлечения сведений об исключениях из всех задач демонстрируется в следующем разделе.

Использование информации типа AggregateException

Чтобы получить сведения об исключении из всех потерпевших отказ задач, результат выполнения метода Task.WhenAll() может быть сохранен в переменной типа Task. Затем к этой переменной применяется await для ожидания до тех пор, пока все задачи не будут завершены. В противном случае исключение по-прежнему будет утеряно. Как будет показано в последнем разделе, с помощью оператора catch может быть перехвачено только исключение, возникшее в первой задаче. Однако теперь имеется доступ к свойству Exception внешней задачи. Свойство Exception имеет тип AggregateException. В данном типе определено свойство InnerExceptions (помимо InnerException), которое содержит список всех исключений, сгенерированных в ожидаемых задачах. Это позволяет легко проходить по всем исключениям:

```
private static async void ShowAggregatedException()
{
   Task taskResult = null;
   try
   {
      Task t1 = ThrowAfter(2000, "first");
      Task t2 = ThrowAfter(1000, "second");
      await (taskResult = Task.WhenAll(t1, t2));
   }
   catch (Exception ex)
   {
      Console.WriteLine("handled {0}", ex.Message);
      foreach (var ex1 in taskResult.Exception.InnerExceptions)
      {
         Console.WriteLine("inner exception {0}", ex1.Message);
      }
   }
}
```

Отмена

При наличии фоновых задач, которые в определенных ситуациях могут выполняться слишком долго, полезно иметь возможность отмены задач. Для таких целей в .NET предлагается стандартный механизм, который появился в версии .NET 4. Этот механизм может применяться вместе с асинхронным шаблоном, основанным на задачах.

Инфраструктура отмены основана на коллективном поведении. Длительно выполняющаяся задача нуждается в самостоятельной проверке, отменена ли она, и в этом случае задача отвечает за очистку всех открытых ресурсов и завершение своей работы.

Отмена основана на классе `CancellationTokenSource`, который применяется для отправки запросов на отмену. Запросы отправляются задачам, которые ссылаются на значение `CancellationToken`, связанное с `CancellationTokenSource`. В следующем разделе представлен пример добавления поддержки отмены к ранее построенному примеру `AsyncPatterns`.

Начало отмены

Сначала определяется переменная `cts` типа `CancellationTokenSource` как закрытое поле-член класса `MainWindow`. Этот член будет использоваться для отмены задач и передачи маркеров методам, которые должны быть отменены (файл `AsyncPatterns/MainWindow.xaml.cs`):

```
public partial class MainWindow : Window
{
   private SearchInfo searchInfo;
   private object lockList = new object();
   private CancellationTokenSource cts;
```

Затем добавляется обработчик событий `OnCancel` для новой кнопки, щелчок на которой приведет к отмене выполняющейся задачи. Внутри этого метода переменная `cts` применяется для отмены задачи с помощью метода `Cancel()`:

```
private void OnCancel(object sender, RoutedEventArgs e)
{
   if (cts != null)
      cts.Cancel();
}
```

Класс `CancellationTokenSource` также поддерживает отмену по истечении указанного промежутка времени. Метод `CancelAfter()` позволяет передавать значение времени в миллисекундах, по прошествии которого задача должна быть отменена.

Отмена выполнения средств платформы

Теперь давайте передадим `CancellationToken` асинхронному методу. Несколько асинхронных методов платформы поддерживают отмену, предлагая перегруженные версии, которым можно передавать `CancellationToken`. Примером может служить метод `GetAsync()` класса `HttpClient`. Перегруженная версия метода `GetAsync()` принимает `CancellationToken` вдобавок к строке URI. Маркер может быть извлечен из `CancellationTokenSource` с использованием свойства `Token`.

Реализация метода `GetAsync()` периодически проверяет, должна ли операция быть отменена. Если это так, то перед генерацией исключения `OperationCanceledException` производится очистка ресурсов. Данное исключение перехватывается оператором `catch`, как показано в следующем фрагменте кода:

```
private async void OnTaskBasedAsyncPattern(object sender,
                                           RoutedEventArgs e)
{
```

```
    cts = new CancellationTokenSource();
    try
    {
       foreach (var req in GetSearchRequests())
       {
          var client = new HttpClient();
          var response = await client.GetAsync(req.Url, cts.Token);
          string resp = await response.Content.ReadAsStringAsync();
          //...
       }
    }
    catch (OperationCanceledException ex)
    {
       MessageBox.Show(ex.Message);
    }
}
```

Отмена выполнения специальных задач

А как быть с отменой выполнения специальных задач? Метод `Run()` класса `Task` предлагает перегруженную версию, которая также принимает `CancellationToken`. Тем не менее, специальные задачи должны проверять, была ли запрошена отмена. В показанном ниже примере это реализовано внутри цикла `foreach`. Маркер может быть проверен с помощью свойства `IsCancellationRequsted`. Если перед генерацией исключения требуется определенная очистка, то лучше предварительно проверить, действительно ли запрошена отмена. Если же очистка не нужна, исключение может быть сгенерировано немедленно после проверки, что и делается с помощью метода `ThrowIfCancellationRequested()`.

```
await Task.Run(() =>
{
   var images = req.Parse(resp);
   foreach (var image in images)
   {
      cts.Token.ThrowIfCancellationRequested();
      searchInfo.List.Add(image);
   }
}, cts.Token);
```

Таким образом, пользователи получили возможность отменять длительно выполняющиеся задачи.

Резюме

В этой главе были представлены ключевые слова `async` и `await`, появившиеся в версии C# 5. На нескольких примерах были продемонстрированы преимущества асинхронного шаблона, основанного на задачах, по сравнению с обычным асинхронным шаблоном и асинхронным шаблоном, основанным на событиях, которые доступны в более ранних версиях .NET.

Вы также увидели, насколько легко создавать асинхронные методы с помощью класса `Task`, и узнали, как использовать ключевые слова `async` и `await` для ожидания завершения этих методов, не блокируя потоки. Наконец, вы ознакомились с особенностями обработки ошибок в асинхронных методах.

За дополнительными сведениями о параллельном программировании, потоках и задачах обращайтесь в главу 21.

В следующей главе мы продолжим рассматривать основные средства C# и .NET, обратившись к вопросам управления памятью и ресурсами.