ГЛАВА 14

Процессы, домены приложений и контексты загрузки

В настоящей главе будут представлены детали обслуживания сборки исполняющей средой, а также отношения между процессами, доменами приложений и контекстами загрузки.

Выражаясь кратко, *домены приложений* (Application Domain или просто AppDomain) представляют собой логические подразделы внутри заданного процесса, обслуживающего набор связанных сборок .NET Core. Как вы увидите, каждый домен приложения в дальнейшем подразделяется на *контекстные границы*, которые используются для группирования вместе похожих по смыслу объектов .NET Core. Благодаря понятию контекста исполняющая среда способна обеспечивать надлежащую обработку объектов со специальными требованиями.

Хотя вполне справедливо утверждать, что многие повседневные задачи программирования не предусматривают работу с процессами, доменами приложений или контекстами загрузки напрямую, их понимание важно при взаимодействии с многочисленными API-интерфейсами .NET Core, включая многопоточную и параллельную обработку, а также сериализацию объектов.

Роль процесса Windows

Концепция "процесса" существовала в операционных системах Windows задолго до выпуска платформы .NET/.NET Core. Пользуясь простыми терминами, *процесс* — это выполняющаяся программа. Тем не менее, формально процесс является концепцией уровня операционной системы, которая применяется для описания набора ресурсов (таких как внешние библиотеки кода и главный поток) и необходимых распределений памяти, используемой функционирующим приложением. Для каждого загруженного в память приложения .NET Core операционная система создает отдельный изолированный процесс для применения на протяжении всего времени его существования.

При использовании такого подхода к изоляции приложений в результате получается намного более надежная и устойчивая исполняющая среда, поскольку отказ одного процесса не влияет на работу других процессов. Более того, данные в одном процессе не доступны напрямую другим процессам, если только не применяются специфичные инструменты вроде пространства имен `System.IO.Pipes` или класса `MemoryMappedFile`.

Каждый процесс Windows получает уникальный идентификатор процесса (process identifier — PID) и может по мере необходимости независимо загружаться и выгружаться операционной системой (а также программно). Как вам возможно известно, в окне диспетчера задач Windows (открываемом по нажатию комбинации клавиш <Ctrl+Shift+Esc>) имеется вкладка Processes (Процессы), на которой можно просматривать разнообразные статические данные о процессах, функционирующих на машине. На вкладке Details (Подробности) можно видеть назначенный идентификатор PID и имя образа (рис. 14.1).

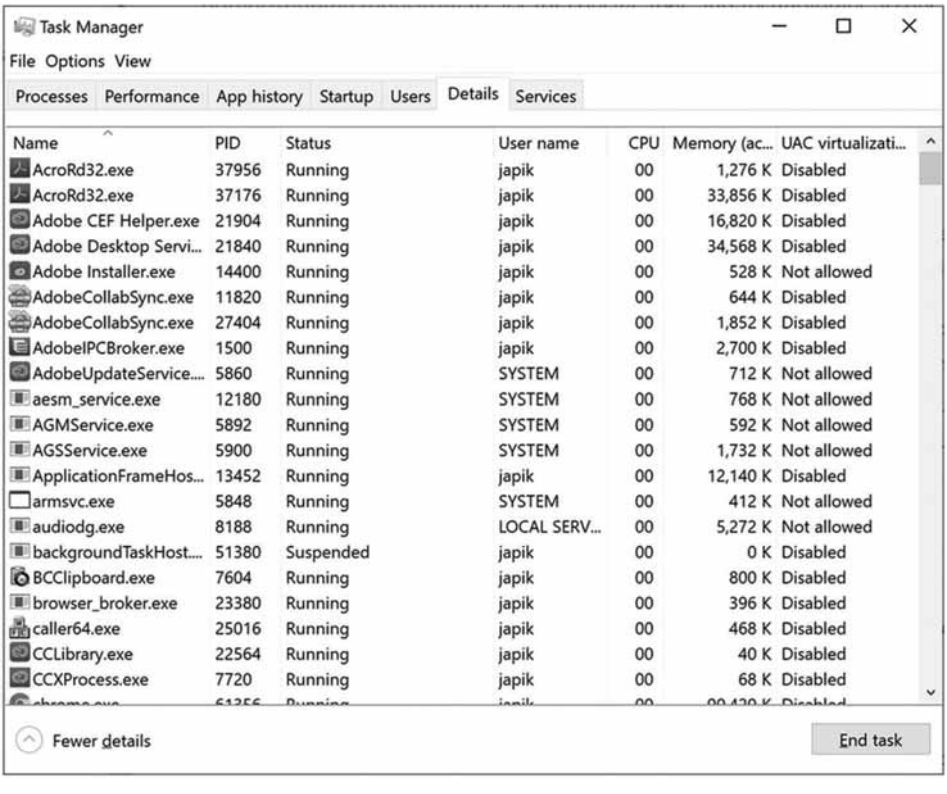

Рис. 14.1. Диспетчер задач Windows

Роль потоков

Каждый процесс Windows содержит начальный “поток”, который действует как точка входа для приложения. Особенности построения многопоточных приложений на платформе .NET Core рассматриваются в главе 15; однако для понимания материала настоящей главы необходимо ознакомиться с несколькими рабочими определениями. *Поток* представляет собой путь выполнения внутри процесса. Выражаясь формально, первый поток, созданный точкой входа процесса, называется *главным потоком*. В любой программе .NET Core (консольном приложении, Windows-службе, приложении WPF и т.д.) точка входа помечается с помощью метода `Main()` или файла, содержащего операторы верхнего уровня. При обращении к этому коду автоматически создается главный поток.

Процессы, которые содержат единственный главный поток выполнения, по своей сути *безопасны в отношении потоков*, т.к. в каждый момент времени доступ к данным приложения может получать только один поток. Тем не менее, однопоточный процесс (особенно с графическим пользовательским интерфейсом) часто замедленно реагирует на действия пользователя, когда его единственный поток выполняет сложную операцию (наподобие печати длинного текстового файла, сложных математических вычислений или попытки подключения к удаленному серверу, находящемуся на расстоянии тысяч километров).

Учитывая такой потенциальный недостаток однопоточных приложений, операционные системы, которые поддерживаются .NET Core, и сама платформа .NET Core предоставляют главному потоку возможность порождения дополнительных вторичных потоков (называемых *рабочими потоками*) с использованием нескольких функций из API-интерфейса Windows, таких как `CreateThread()`. Каждый поток (первичный или вторичный) становится уникальным путем выполнения в процессе и имеет параллельный доступ ко всем совместно используемым элементам данных внутри этого процесса.

Нетрудно догадаться, что разработчики обычно создают дополнительные потоки для улучшения общей степени отзывчивости программы. Многопоточные процессы обеспечивают иллюзию того, что выполнение многочисленных действий происходит более или менее одновременно. Например, приложение может порождать дополнительный рабочий поток для выполнения трудоемкой единицы работы (вроде вывода на печать крупного текстового файла). После запуска вторичного потока главный поток продолжает реагировать на пользовательский ввод, что дает всему процессу возможность достигать более высокой производительности. Однако на самом деле так происходит не всегда: применение слишком большого количества потоков в одном процессе может приводить к *ухудшению* производительности из-за того, что центральный процессор должен переключаться между активными потоками внутри процесса (а это отнимает время).

На некоторых машинах многопоточность по большей части является иллюзией, обеспечиваемой операционной системой. Машины с единственным (не поддерживающим гиперпотоки) центральным процессором не обладают возможностью обработки множества потоков в одно и то же время. Взамен один центральный процессор выполняет по одному потоку за единицу времени (называемую *квантом времени*), частично основываясь на приоритете потока. По истечении выделенного кванта времени выполнение существующего потока приостанавливается, позволяя выполнять работу другому потоку. Чтобы поток не "забывал", что происходило до того, как его выполнение было приостановлено, ему предоставляется возможность записывать данные в локальное хранилище потоков (Thread Local Storage — TLS) и выделяется отдельный стек вызовов (рис. 14.2).

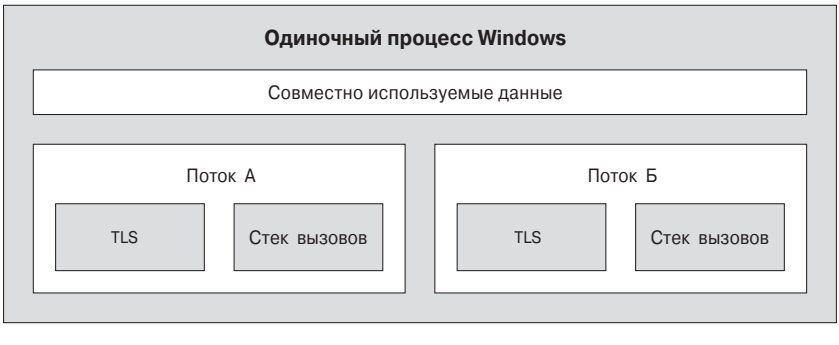

Рис. 14.2. Отношения между потоками и процессами в Windows

Если тема потоков для вас нова, то не стоит беспокоиться о деталях. На данном этапе просто запомните, что любой поток представляет собой уникальный путь выполнения внутри процесса Windows. Каждый процесс имеет главный поток (созданный посредством точки входа исполняемого файла) и может содержать дополнительные потоки, которые создаются программно.

Взаимодействие с процессами, используя платформу .NET Core

Несмотря на то что с процессами и потоками не связано ничего нового, способ взаимодействия с ними в рамках платформы .NET Core значительно изменился (в лучшую сторону). Чтобы подготовить почву для понимания области построения многопоточных сборок (см. главу 15), давайте начнем с выяснения способов взаимодействия с процессами, используя библиотеки базовых классов .NET Core.

В пространстве имен `System.Diagnostics` определено несколько типов, которые позволяют программно взаимодействовать с процессами и разнообразными типами, связанными с диагностикой, такими как журнал событий системы и счетчики производительности. В текущей главе нас интересуют только типы, связанные с процессами, которые описаны в табл. 14.1.

Таблица 14.1. Избранные типы пространства имен `System.Diagnostics`

Тип, связанный с процессами	Описание
`Process`	Предоставляет доступ к локальным и удаленным процессам, а также позволяет программно запускать и останавливать процессы
`ProcessModule`	Представляет модуль (`*.dll` или `*.exe`), загруженный в определенный процесс. Важно понимать, что тип `ProcessModule` может представлять *любой* модуль, т.е. двоичные сборки, основанные на COM, .NET или традиционном языке C
`ProcessModuleCollection`	Предоставляет строго типизированную коллекцию объектов `ProcessModule`
`ProcessStartInfo`	Указывает набор значений, применяемых при запуске процесса с помощью метода `Process.Start()`
`ProcessThread`	Представляет поток внутри заданного процесса. Имейте в виду, что тип `ProcessThread` используется для диагностирования набора потоков процесса, но не для порождения новых потоков выполнения в рамках процесса
`ProcessThreadCollection`	Предоставляет строго типизованную коллекцию объектов `ProcessThread`

Класс `System.Diagnostics.Process` позволяет анализировать процессы, выполняющиеся на заданной машине (локальные или удаленные). В классе `Process` также определены члены, предназначенные для программного запуска и завершения процессов, просмотра (или модификации) уровня приоритета процесса и получения списка активных потоков и/или загруженных модулей внутри указанного процесса. В табл. 14.2 перечислены некоторые основные свойства класса `System.Diagnostics.Process`.

Таблица 14.2. Избранные свойства класса `Process`

Свойство	Описание
`ExitTime`	Позволяет извлекать отметку времени, ассоциированную с процессом, который был завершен (представленную с помощью типа `DateTime`)
`Handle`	Возвращает дескриптор (представленный типом `IntPtr`), который был назначен процессу операционной системой. Это может быть полезно при построении приложений .NET, нуждающихся во взаимодействии с неуправляемым кодом
`Id`	Позволяет получать идентификатор PID связанного процесса
`MachineName`	Позволяет получать имя компьютера, на котором выполняется связанный процесс
`MainWindowTitle`	Позволяет получать заголовок главного окна процесса (если у процесса нет главного окна, то возвращается пустая строка)
`Modules`	Предоставляет доступ к строго типизованной коллекции `ProcessModuleCollection`, представляющей набор модулей (`*.dll` или `*.exe`), которые были загружены внутри текущего процесса
`ProcessName`	Позволяет получать имя процесса (которое, как и можно было предполагать, представляет собой имя самого приложения)
`Responding`	Позволяет получать значение, которое указывает, реагирует ли пользовательский интерфейс процесса на пользовательский ввод (или в текущий момент находится в “зависшем” состоянии)
`StartTime`	Позволяет получать значение времени, когда был запущен процесс (представленное с помощью типа `DateTime`)
`Threads`	Позволяет получать набор потоков, выполняемых в связанном процессе (представленный посредством коллекции объектов `ProcessThread`)

Кроме перечисленных выше свойств в классе `System.Diagnostics.Process` определено несколько полезных методов (табл. 14.3).

Таблица 14.3. Избранные методы класса `Process`

Метод	Описание
`CloseMainWindow()`	Этот метод закрывает процесс, который содержит пользовательский интерфейс, отправляя его главному окну сообщение о закрытии
`GetCurrentProcess()`	Этот статический метод возвращает новый объект `Process`, который представляет процесс, активный в текущий момент
`GetProcesses()`	Этот статический метод возвращает массив объектов `Process`, представляющих процессы, которые выполняются на заданной машине
`Kill()`	Этот метод немедленно останавливает связанный процесс
`Start()`	Этот метод запускает процесс

Перечисление выполняющихся процессов

Для иллюстрации способа манипулирования объектами `Process` создайте новый проект консольного приложения C# по имени `ProcessManipulator` и определите в классе `Program` следующий вспомогательный статический метод (не забудьте импортировать в файл кода пространства имен `System.Diagnostics` и `System.Linq`):

```
static void ListAllRunningProcesses()
{
  // Получить все процессы на локальной машине, упорядоченные по PID.
  var runningProcs = from proc in Process.GetProcesses(".")
                     orderby proc.Id select proc;

  // Вывести для каждого процесса идентификатор PID и имя.
  foreach(var p in runningProcs)
  {
    string info = $"-> PID: {p.Id}\tName: {p.ProcessName}";
    Console.WriteLine(info);
  }
  Console.WriteLine("*************************************\n");
}
```

Статический метод `Process.GetProcesses()` возвращает массив объектов `Process`, которые представляют выполняющиеся процессы на целевой машине (передаваемая методу строка `"."` обозначает локальный компьютер). После получения массива объектов `Process` можно обращаться к любым членам, описанным в табл. 14.2 и 14.3. Здесь просто для каждого процесса выводятся идентификатор PID и имя с упорядочением по PID. Модифицируйте операторы верхнего уровня, как показано ниже:

```
using System;
using System.Diagnostics;
using System.Linq;

Console.WriteLine("***** Fun with Processes *****\n");
ListAllRunningProcesses();
Console.ReadLine();
```

Запустив приложение, вы увидите список имен и идентификаторов PID для всех процессов на локальной машине. Ниже показана часть вывода (ваш вывод наверняка будет отличаться):

```
***** Fun with Processes *****

-> PID: 0 Name: Idle
-> PID: 4 Name: System
-> PID: 104 Name: Secure System
-> PID: 176 Name: Registry
-> PID: 908 Name: svchost
-> PID: 920 Name: smss
-> PID: 1016 Name: csrss
-> PID: 1020 Name: NVDisplay.Container
-> PID: 1104 Name: wininit
-> PID: 1112 Name: csrss
*************************************
```

Исследование конкретного процесса

В дополнение к полному списку всех выполняющихся процессов на заданной машине статический метод `Process.GetProcessById()` позволяет получать одиночный объект `Process` по ассоциированному с ним идентификатору PID. В случае запроса несуществующего PID генерируется исключение `ArgumentException`. Например, чтобы получить объект `Process`, который представляет процесс с PID, равным 30592, можно написать следующий код:

```
// Если процесс с PID, равным 30592, не существует,
// то сгенерируется исключение во время выполнения.
static void GetSpecificProcess()
{
  Process theProc = null;
  try
  {
    theProc = Process.GetProcessById(30592);
  }
  catch(ArgumentException ex)
  {
    Console.WriteLine(ex.Message);
  }
}
```

К настоящему моменту вы уже знаете, как получить список всех процессов, а также специфический процесс на машине посредством поиска по PID. Наряду с выяснением идентификаторов PID и имен процессов класс `Process` позволяет просматривать набор текущих потоков и библиотек, применяемых внутри заданного процесса. Давайте посмотрим, как это делается.

Исследование набора потоков процесса

Набор потоков представлен в виде строго типизованной коллекции `ProcessThreadCollection`, которая содержит определенное количество отдельных объектов `ProcessThread`. В целях иллюстрации добавьте к текущему приложению приведенный далее вспомогательный статический метод:

```
static void EnumThreadsForPid(int pID)
{
  Process theProc = null;
  try
  {
    theProc = Process.GetProcessById(pID);
  }
  catch(ArgumentException ex)
  {
    Console.WriteLine(ex.Message);
    return;
  }
  // Вывести статистические сведения по каждому потоку в указанном процессе
  Console.WriteLine("Here are the threads used by: {0}",
    theProc.ProcessName);
  ProcessThreadCollection theThreads = theProc.Threads;
```

```
  foreach(ProcessThread pt in theThreads)
  {
    string info =
     $"-> Thread ID: {pt.Id}\tStart Time: {pt.StartTime.ToShortTimeString()}\
tPriority:{pt.PriorityLevel}";
    Console.WriteLine(info);
  }
  Console.WriteLine("*********************************\n");
}
```

Как видите, свойство `Threads` в типе `System.Diagnostics.Process` обеспечивает доступ к классу `ProcessThreadCollection`. Здесь для каждого потока внутри указанного клиентом процесса выводится назначенный идентификатор потока, время запуска и уровень приоритета. Обновите операторы верхнего уровня в своей программе, чтобы запрашивать у пользователя идентификатор PID процесса, подлежащего исследованию:

```
static void Main(string[] args)
{
  ...
  // Запросить у пользователя PID и вывести набор активных потоков.
  Console.WriteLine("***** Enter PID of process to investigate *****");
  Console.Write("PID: ");
  string pID = Console.ReadLine();
  int theProcID = int.Parse(pID);
  EnumThreadsForPid(theProcID);
  Console.ReadLine();
}
```

После запуска приложения можно вводить PID любого процесса на машине и просматривать имеющиеся внутри него потоки. В следующем выводе показан неполный список потоков, используемых процессом с PID 3804, который (так случилось) обслуживает браузер Edge:

```
***** Enter PID of process to investigate *****
PID: 3804
Here are the threads used by: msedge
-> Thread ID: 3464 Start Time: 01:20 PM Priority: Normal
-> Thread ID: 19420 Start Time: 01:20 PM Priority: Normal
-> Thread ID: 17780 Start Time: 01:20 PM Priority: Normal
-> Thread ID: 22380 Start Time: 01:20 PM Priority: Normal
-> Thread ID: 27580 Start Time: 01:20 PM Priority: -4
…
*********************************
```

Помимо `Id`, `StartTime` и `PriorityLevel` тип `ProcessThread` содержит дополнительные члены, наиболее интересные из которых перечислены в табл. 14.4.

Прежде чем двигаться дальше, необходимо уяснить, что тип `ProcessThread` *не является* сущностью, применяемой для создания, приостановки или уничтожения потоков на платформе .NET Core. Тип `ProcessThread` скорее представляет собой средство, позволяющее получать диагностическую информацию по активным потокам Windows внутри выполняющегося процесса. Более подробные сведения о том, как создавать многопоточные приложения с использованием пространства имен `System.Threading`, приводятся в главе 15.

Таблица 14.4. Избранные члены типа `ProcessThread`

Член	Описание
`CurrentPriority`	Получает текущий приоритет потока
`Id`	Получает уникальный идентификатор потока
`IdealProcessor`	Устанавливает предпочитаемый процессор для выполнения заданного потока
`PriorityLevel`	Получает или устанавливает уровень приоритета потока
`ProcessorAffinity`	Устанавливает процессоры, на которых может выполняться связанный поток
`StartAddress`	Получает адрес в памяти функции, вызванной операционной системой, которая запустила данный поток
`StartTime`	Получает время, когда операционная система запустила поток
`ThreadState`	Получает текущее состояние потока
`TotalProcessorTime`	Получает общее время использования процессора данным потоком
`WaitReason`	Получает причину, по которой поток находится в состоянии ожидания

Исследование набора модулей процесса

Теперь давайте посмотрим, как реализовать проход по загруженным модулям, которые размещены внутри конкретного процесса. Когда речь идет о процессах, *модуль* —это общий термин, применяемый для описания заданной сборки `*.dll` (или самого файла `*.exe`), которая обслуживается специфичным процессом. Когда производится доступ к коллекции `ProcessModuleCollection` через свойство `Process.Modules`, появляется возможность перечисления *всех модулей*, размещенных внутри процесса: библиотек на основе .NET Core, COM и традиционного языка C. Взгляните на показанный ниже дополнительный вспомогательный метод, который будет перечислять модули в процессе с указанным идентификатором PID:

```
static void EnumModsForPid(int pID)
{
  Process theProc = null;
  try
  {
    theProc = Process.GetProcessById(pID);
  }
  catch(ArgumentException ex)
  {
    Console.WriteLine(ex.Message);
    return;
  }
  Console.WriteLine("Here are the loaded modules for: {0}",
                    theProc.ProcessName);
  ProcessModuleCollection theMods = theProc.Modules;
```

```
    foreach(ProcessModule pm in theMods)
    {
      string info = $"-> Mod Name: {pm.ModuleName}";
      Console.WriteLine(info);
    }
    Console.WriteLine("************************************\n");
  }
```

Чтобы получить какой-то вывод, давайте просмотрим загружаемые модули для процесса, обслуживающего программу текущего примера (`ProcessManipulator`). Для этого нужно запустить приложение, выяснить идентификатор PID, назначенный `ProcessManipulator.exe` (посредством диспетчера задач), и передать значение PID методу `EnumModsForPid()`. Вас может удивить, что с простым консольным приложением связан настолько внушительный список библиотек `*.dll` (`GDI32.dll`, `USER32.dll`, `ole32.dll` и т.д.). Ниже показан частичный список загруженных модулей (ради краткости отредактированный):

```
Here are (some of) the loaded modules for: ProcessManipulator
Here are the loaded modules for: ProcessManipulator
-> Mod Name: ProcessManipulator.exe
-> Mod Name: ntdll.dll
-> Mod Name: KERNEL32.DLL
-> Mod Name: KERNELBASE.dll
-> Mod Name: USER32.dll
-> Mod Name: win32u.dll
-> Mod Name: GDI32.dll
-> Mod Name: gdi32full.dll
-> Mod Name: msvcp_win.dll
-> Mod Name: ucrtbase.dll
-> Mod Name: SHELL32.dll
-> Mod Name: ADVAPI32.dll
-> Mod Name: msvcrt.dll
-> Mod Name: sechost.dll
-> Mod Name: RPCRT4.dll
-> Mod Name: IMM32.DLL
-> Mod Name: hostfxr.dll
-> Mod Name: hostpolicy.dll
-> Mod Name: coreclr.dll
-> Mod Name: ole32.dll
-> Mod Name: combase.dll
-> Mod Name: OLEAUT32.dll
-> Mod Name: bcryptPrimitives.dll
-> Mod Name: System.Private.CoreLib.dll
...
**********************************
```

Запуск и останов процессов программным образом

Финальными аспектами класса `System.Diagnostics.Process`, которые мы здесь исследуем, являются методы `Start()` и `Kill()`. Они позволяют программно запускать и завершать процесс. В качестве примера создадим вспомогательный статический метод `StartAndKillProcess()` с приведенным ниже кодом.

На заметку! В зависимости от настроек операционной системы, касающихся безопасности, для запуска новых процессов могут требоваться права администратора.

```
static void StartAndKillProcess()
{
  Process proc = null;

  // Запустить Edge и перейти на Facebook!
  try
  {
    proc = Process.Start(@"C:\Program Files (x86)\Microsoft\Edge\
Application\msedge.exe",
      "www.facebook.com");
  }
  catch (InvalidOperationException ex)
  {
    Console.WriteLine(ex.Message);
  }

  // Уничтожить процесс по нажатию <Enter>.
  Console.Write("--> Hit enter to kill {0}...",
    proc.ProcessName);
  Console.ReadLine();

  // Уничтожить все процессы msedge.exe.
  try
  {
    foreach (var p in Process.GetProcessesByName("MsEdge"))
    {
      p.Kill(true);
    }
  }
  catch (InvalidOperationException ex)
  {
    Console.WriteLine(ex.Message);
  }
}
```

Статический метод `Process.Start()` имеет несколько перегруженных версий. Как минимум, понадобится указать путь и имя файла запускаемого процесса. В рассматриваемом примере используется версия метода `Start()`, которая позволяет задавать любые дополнительные аргументы, подлежащие передаче в точку входа программы, в данном случае веб-страницу для загрузки.

В результате вызова метода `Start()` возвращается ссылка на новый активизированный процесс. Чтобы завершить данный процесс, потребуется просто вызвать метод `Kill()` уровня экземпляра. Поскольку Microsoft Edge запускает множество процессов, для их уничтожения организован цикл. Вызовы `Start()` и `Kill()` помещены внутрь блока `try/catch` с целью обработки исключений `InvalidOperationException`. Это особенно важно при вызове метода `Kill()`, потому что такое исключение генерируется, если процесс был завершен до вызова `Kill()`.