ГЛАВА 17

Процессы, домены приложений и контексты объектов

В предыдущих двух главах рассматривались шаги, которые CLR-среда предпринимает для выяснения местонахождения внешних сборок, на которые имеются ссылки, а также роль, которую в .NET играют метаданные. В этой главе мы более детально посмотрим, как сборки обслуживаются в CLR-среде и какие отношения существуют между процессами, доменами приложений и контекстами объектов.

По сути, *доменом приложения* (Application Domain — AppDomain) называются логические подразделы внутри отдельного процесса, в которых обслуживается ряд связанных между собой сборок .NET. Как будет показано в этой главе, каждый домен приложения, в свою очередь, может делиться на отдельные *контекстные границы* (context boundaries), которые применяются для группирования вместе подобных .NET-объектов. За счет использования понятия контекста, CLR-среде удается гарантировать, что объекты с особыми требованиями будут во время выполнения обрабатываться надлежащим образом.

Повторный обзор традиционных процессов Win32

Понятие "процесс" существовало в операционных системах Windows еще задолго до появления платформы .NET. Попросту говоря, термин *процесс* используется для описания ряда ресурсов (наподобие внешних библиотек кода и главного потока) и необходимой выполняющемуся приложению выделяемой памяти. Для каждого загружаемого в память файла `*.exe` операционная система создает отдельный и изолированный процесс, который потом применяет на протяжении всей его жизни. Благодаря такому подходу к изолированию приложений, среда выполнения в результате получается гораздо более надежной и стабильной, поскольку выход из строя одного процесса никак не сказывается на работе других процессов.

Каждый процесс Win32 получает уникальный идентификатор PID (Process ID — идентификатор процесса) и может независимым образом загружаться и выгружаться операционной системой по мере необходимости (а также программным образом вызовами API-интерфейса Win32). Как уже, возможно, известно, в окне утилиты Windows Task Manager (Диспетчер задач) предлагается вкладка Processes (Процессы), на которой

можно просматривать различные статические данные о выполняющихся на данной машине процессах, в том числе их PID-идентификаторы и имена образов (рис. 17.1).

На заметку! Отображать в окне диспетчера задач дополнительные столбцы (вроде PID (Идентификатор процесса), User Name (Имя пользователя) и т.д.) можно, выбирая в меню View (Вид) пункт Select Columns (Выбрать столбцы).

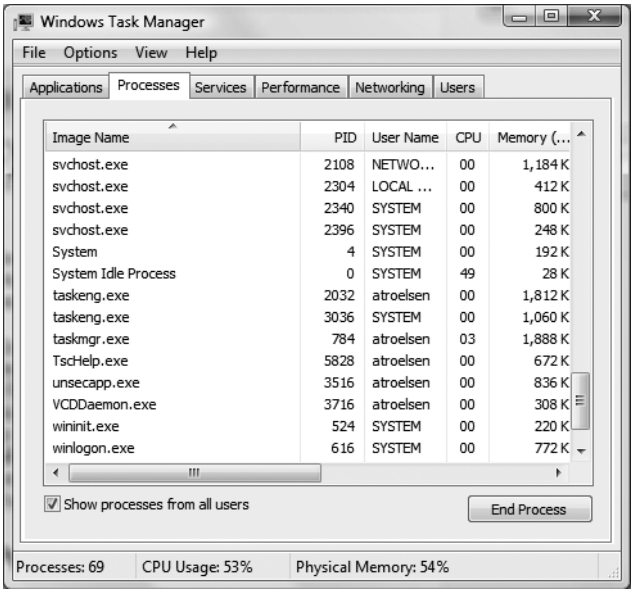

Рис. 17.1. Вкладка Processes в окне диспетчера задач Windows

Общая информация о потоках

Каждый процесс Win32 имеет один главный "поток", который является входной точкой приложения. В следующей главе подробно рассказывается о том, как создавать потоки в рамках платформы .NET с помощью пространства имен `System.Threading` но пока, чтобы облегчить понимание излагаемых здесь тем, достаточно представить несколько рабочих определений. *Потоком* называется используемый в процессе путь выполнения. Формально поток, который во входной точке процесса создается первым, называется *главным потоком* (primary thread). В настольных приложениях с графическим пользовательским интерфейсом, создаваемых с использованием API-интерфейса Win32, роль входной точки приложения исполняет метод `WinMain()`, а в программах с консольным интерфейсом — метод `Main()`.

Процессы, которые содержат единственный главный поток выполнения, уже внутреннее являются *безопасными к потокам* (thread safe), поскольку в каждый отдельный момент времени доступ к данным приложения может получать только один поток. Такие однопоточные процессы, однако, (особенно те, что обладают графическим пользовательским интерфейсом) часто немного замедленно реагируют на действия пользователя, когда их один поток выполняет какую-то сложную операцию (например, печатает длинный текстовый файл, производит сложные математические вычисления или пытается подключиться к удаленному серверу, который находится за тысячу миль).

Из-за такого возможного недостатка однопоточных приложений, API-интерфейс Win32 (равно как и платформа .NET) разрешает главному потоку порождать дополнительные вторичные потоки (также называемые *рабочими потоками*) с помощью удобных функций наподобие `CreateThread()`. Каждый поток (как первичный, так и вторичный) превращается в процессе в уникальный путь выполнения и может параллельно получать доступ ко всем открытым для совместного доступа элементам данных.

Как не трудно догадаться, разработчики обычно создают дополнительные потоки с целью улучшения общей степени восприимчивости программы к действиям пользователя. Многопоточные процессы обеспечивают иллюзию того, что выполнение многочисленных действий происходит примерно в одно и то же время. Например, дополнительный рабочий поток может порождаться в приложении для решения какой-нибудь трудоемкой задачи (такой как печать большого текстового файла). После начала выполнения задачи вторничным потоком основной поток все равно не утрачивает свою способность реагировать на действия пользователя, что дает всему процессу возможность сопровождаться куда более высокой производительностью. Однако такого может и не происходить: в случае использования слишком большого количества потоков в одном процессе его производительность может даже *ухудшаться* из-за возникновения у ЦП необходимости переключаться между активными потоками в процессе (что отнимает определенное время).

В реальности следует всегда помнить о том, что многопоточность по большей части представляет собой не более чем просто обеспечиваемую операционной системой иллюзию. Машины с одним (не поддерживающим гиперпоточность) процессором буквально не имеют возможности обрабатывать множество потоков в точности в одно и то же время. Вместо этого они выполняют по одному потоку за одну единицу времени (называемую *квантом*), исходя отчасти из приоритета потока. По истечении выделенного кванта времени выполнение существующего потока приостанавливается для предоставления другому потоку возможности выполнить свою задачу. Чтобы поток не забывал, на чем остановился перед тем, как его выполнение было приостановлено, каждому потоку предоставляется возможность записи данных в локальное хранилище потоков (Thread Local Storage — TLS) и выделяется отдельный стек вызовов, как показано на рис. 17.2.

Рис. 17.2. Отношения между потоками и процессами Win32

Если тема потоков является новой, не стоит сильно беспокоиться по поводу понимания деталей. На данном этапе главное запомнить, что любой поток представляет собой просто уникальный путь выполнения внутри процесса Win32, и что каждый процесс обязательно обладает главным потоком (который создается в точке входа в приложение) и может содержать дополнительные потоки, создаваемые программным образом.

Взаимодействие с процессами в рамках платформы .NET

Хотя в самих процессах и потоках нет ничего нового, способ взаимодействия с ними в рамках платформы .NET довольно прилично изменился (в лучшую сторону). Чтобы создать основу для понимания приемов построения многопоточных сборок (см. главу 18), давайте посмотрим, как взаимодействовать с процессами с помощью библиотек базовых классов .NET.

В пространстве имен `System.Diagnostics` поставляется ряд типов, которые позволяют программно взаимодействовать с процессами и различными связанными с диагностикой средствами наподобие системного журнала событий и счетчиков производительности. В настоящей главе нас интересуют только те типы, которые позволяют взаимодействовать с процессами. Некоторые наиболее важные из них перечислены в табл. 17.1.

Таблица 17.1. Некоторые члены пространства имен `System.Diagnostics`

Типы в `System.Diagnostics`, которые позволяют взаимодействовать с процессами	Описание
`Process`	Предоставляет доступ к локальным и удаленным процессам, а также позволяет запускать и останавливать процессы программным образом.
`ProcessModule`	Позволяет представлять модуль (`*.dll` или `*.exe`), который должен загружаться в определенный процесс. Важно понимать, что этот тип может применяться для представления *любого* модуля: на базе COM, на базе .NET или традиционного двоичного модуля на базе C.
`ProcessModuleCollection`	Позволяет создавать строго типизированную коллекцию объектов `ProcessModule`.
`ProcessStartInfo`	Позволяет указывать ряд значений, которые должны использоваться при запуске процесса посредством метода `Process.Start()`.
`ProcessThread`	Позволяет представлять поток внутри определенного процесса. Следует иметь в виду, что этот тип применяется для осуществления диагностики ряда потоков в процессе, но не для ответвления внутри него новых потоков.
`ProcessThreadCollection`	Позволяет создавать строго типизованную коллекцию объектов `ProcessThread`.

Тип `System.Diagnostics.Process` позволяет анализировать процессы, выполняющиеся на определенной машине (как локальной, так и удаленной). Класс `Process` обладает членами, которые позволяют запускать и останавливать процессы программным образом, просматривать приоритет процесса, а также получать список активных потоков и/или загруженных модулей, которые выполняются в рамках конкретного процесса. В табл. 17.2 перечислены некоторые наиболее важные члены `System.Diagnostics.Process`.

Таблица 17.2. Некоторые члены типа `Process`

Член	Описание
`ExitCode`	Представляет собой свойство и позволяет извлекать значение, указываемое соответствующим процессом при завершении его работы. Важно обратить внимание на то, что для получения этого значения необходимо либо обрабатывать событие `Exited` (и тем самым обеспечивать асинхронное уведомление), либо вызывать метод `WaitForExit()` (и тем самым обеспечивать синхронное уведомление).
`ExitTime`	Представляет собой свойство и позволяет получать значение даты и времени, ассоциируемое с процессом, которые завершил свою работу (в виде типа `DateTime`).
`Handle`	Представляет собой свойство и возвращает дескриптор, который был присвоен процессу операционной системой.
`HandleCount`	Представляет собой свойство и возвращает информацию о количестве дескрипторов, которые были открыты процессом.
`Id`	Представляет собой свойство и позволяет получать идентификатор (PID) соответствующего процесса.
`MachineName`	Представляет собой свойство и позволяет получать имя компьютера, на котором выполняется соответствующий процесс.
`MainModule`	Представляет собой свойство и позволяет получать тип `ProcessModule`, представляющий главный модуль в текущем процессе.
`MainWindowTitle` `MainWindowHandle`	Первый представляет собой свойство и позволяет получать заголовок главного окна процесса (если у процесса нет такового, возвращается пустая строка). Второй тоже представляет собой свойство, но позволяет получать (представленный в виде типа `System.IntPtr`) дескриптор соответствующего окна (если у процесса нет главного окна, типу `IntPtr` присваивается значение `System.IntPtr.Zero`).
`Modules`	Представляет собой свойство и предоставляет доступ к строго типизованной коллекции `ProcessModuleCollection`, содержащей ряд модулей (`*.dll` или `*.exe`), которые были загружены в рамках текущего процесса.
`PriorityBoostEnabled`	Представляет собой свойство и позволяет указывать, должна ли операционная система временно ускорять выполнение процесса в случае наведения фокуса на его главное окно.
`PriorityClass`	Представляет собой свойство и позволяет считывать или изменять приоритет соответствующего процесса.
`ProcessName`	Представляет собой свойство и позволяет получать имя процесса (которое, как не трудно догадаться, совпадает с именем самого приложения).
`Responding`	Представляет собой свойство и позволяет получать значение, показывающее, реагирует ли пользовательский интерфейс соответствующего процесса на действия пользователя (и не находится ли в текущий момент в “зависшем” состоянии).
`StartTime`	Представляет собой свойство и позволяет получать информацию о времени, когда был запущен соответствующий процесс (в виде типа `DateTime`).
`Threads`	Представляет собой свойство и позволяет получать информацию о потоках, которые выполняются в рамках соответствующего процесса (в виде массива типов `ProcessThread`).
`CloseMainWindow()`	Представляет собой метод и позволяет завершать процесс, обладающий пользовательским интерфейсом, за счет отправки в его главное окно сообщения о закрытии.

Окончание табл. 17.2

Член	Описание
`GetCurrentProcess()`	Представляет собой статический метод и возвращает новый тип `Process`, представляющий процесс, который в текущий момент является активным.
`GetProcesses()`	Представляет собой статический метод и возвращает массив новых компонентов `Process`, которые выполняются на текущей машине.
`Kill()`	Представляет собой метод и позволяет немедленно останавливать соответствующий процесс.
`Start()`	Представляет собой метод и позволяет запускать процесс.

Перечисление выполняющихся процессов

Чтобы посмотреть, как манипулировать типами `Process`, создадим новый проект типа C# Console Application (Консольное приложение на C#) по имени `ProcessManipulator` и определим в нем внутри класса `Program` следующий вспомогательный статический метод (не забыв импортировать пространство имен `System.Diagnostics`):

```
public static void ListAllRunningProcesses()
{
    // Получение списка всех процессов, которые выполняются на текущей машине.
    Process[] runningProcs = Process.GetProcesses(".");
    // Отображение идентификатора и имени каждого из процессов.
    foreach(Process p in runningProcs)
    {
      string info = string.Format("-> PID: {0}\tName: {1}", p.Id, p.ProcessName);
      Console.WriteLine(info);
    }
    Console.WriteLine("*************************************\n");
}
```

Обратите внимание на то, что статический метод `Process.GetProcesses()` возвращает массив типов `Process`, которые представляют выполняющиеся процессы на целевой машине (используемое здесь обозначение в виде точки указывает на то, что в данном случае целевой машиной является локальный компьютер). После получения массива типов `Process` можно вызывать любой из описанных в табл. 17.2 член. Здесь мы просто отображаем идентификатор (PID) и имя каждого процесса. В случае изменения метода `Main()` так, чтобы он вызывал этот вспомогательный метод `ListAllRunningProcesses()`, можно увидеть примерно такой вывод, как показан на рис. 17.3.

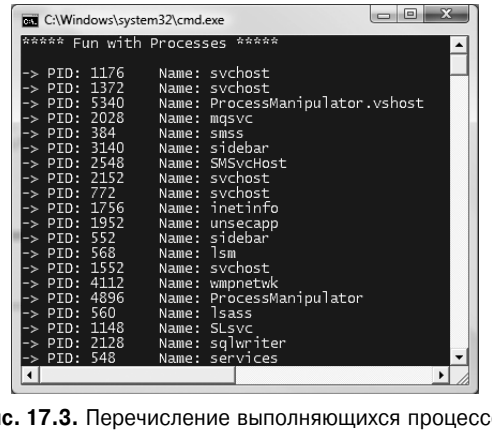

Рис. 17.3. Перечисление выполняющихся процессов

Изучение конкретного процесса

Помимо полного перечня всех выполняющихся на конкретной машине процессов статический метод `Process.GetProcessById()` также позволяет получать информацию о конкретном типе `Process` с помощью соответствующего идентификатора (PID). В случае запроса доступа к несуществующему PID генерируется исключение `ArgumentException`. Например, если необходимо получить информацию об объекте `Process`, представляющем процесс с PID-идентификатором `987`, можно написать следующий код:

```
// В случае отсутствия процесса с PID-идентификатором 987
// во время выполнения будет генерироваться соответствующее исключение.
static void GetSpecificProcess()
{
   Process theProc = null;
   try
   {
      theProc = Process.GetProcessById(987);
   }
   catch    // Общий блок catch используется для простоты.
   {
      Console.WriteLine("-> Sorry...bad PID!"); // неверный PID
   }
}
```

Изучение ряда потоков внутри процесса

Тип класса `Process` позволяет программно изучать набор потоков, которые в текущий момент используются в конкретном процессе. Набор потоков представляется в виде строго типизованной коллекции `ProcessThreadCollection`, в которой содержится некоторое количество отдельных типов `ProcessThread`. Для примера предположим, что в текущее приложение добавлена следующая вспомогательная статическая функция:

```
static void EnumThreadsForPid(int pID)
{
   Process theProc = null;
   try
   {
      theProc = Process.GetProcessById(pID);
   }
   catch
   {
      Console.WriteLine("-> Sorry...bad PID!");
      Console.WriteLine("***********************************\n");
      return;
   }
   // Отображение статистических данных 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 =
       string.Format("-> Thread ID: {0}\tStart Time {1}\tPriority {2}",
       pt.Id , pt.StartTime.ToShortTimeString(), pt.PriorityLevel);
     Console.WriteLine(info);
   }
   Console.WriteLine("***********************************\n");
}
```

Нетрудно заметить, что свойство `Threads` в типе `System.Diagnostics.Process` предоставляет доступ к классу `ProcessThreadCollection`. В этом коде предусмотрено отображение идентификатора, времени запуска и уровня приоритета каждого потока, который используется в указанном клиентом процессе. Следовательно, в случае обновления метода `Main()` внутри класса `Program` так, чтобы он приглашал пользователя предоставлять PID-идентификатор интересующего процесса, показанным ниже образом:

```
static void Main(string[] args)
{
...
    // Отображение приглашения пользователю на ввод PID-идентификатора
    // и вывод информации о соответствующих активных потоках.
    Console.WriteLine("***** Enter PID of process to investigate *****");
    Console.Write("PID:  ");
    string pID = Console.ReadLine();
    int theProcID = int.Parse(pID);
    EnumThreadsForPid(theProcID);
    Console.ReadLine();
}
```

можно будет увидеть примерно такой вывод, как показан на рис. 17.4.

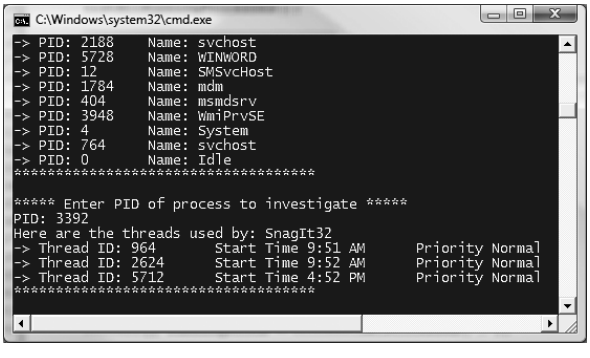

Рис. 17.4. Перечисление потоков, используемых в рамках выполняющегося процесса

Помимо `Id`, `StartTime` и `PriorityLevel`, тип `ProcessThread` имеет еще и дополнительные члены. Некоторые наиболее интересные из них перечислены в табл. 17.3.

Таблица 17.3. Некоторые дополнительные члены типа `ProcessThread`

Член	Описание
`BasePriority`	Позволяет получать информацию о базовом приоритете потока.
`CurrentPriority`	Позволяет получать информацию о текущем приоритете потока.
`Id`	Позволяет получать уникальный идентификатор потока.
`IdealProcessor`	Позволяет указывать предпочтительный процессор, на котором должен выполняться данный поток.
`PriorityLevel`	Позволяет получать или устанавливать уровень приоритета потока.
`ProcessorAffinity`	Позволяет указывать процессоры, на которых может выполняться соответствующий поток.
`StartAddress`	Позволяет получать адрес в памяти, по которому операционная система вызывала функцию, приведшую к запуску данного потока.

Окончание табл. 17.3

Член	Описание
`StartTime`	Позволяет получать информацию о времени, когда операционная система запустила поток.
`ThreadState`	Позволяет получать информацию о текущем состоянии потока.
`TotalProcessorTime`	Позволяет получать информацию об общем количестве времени, в течение которого данный поток использовал процессор.
`WaitReason`	Позволяет узнавать причину, по которой поток находится в состоянии ожидания.

Прежде чем двигаться дальше, необходимо четко уяснить, что тип `ProcessThread` *не является* сущностью, применяемой для создания, приостановки и уничтожения потоков в .NET. Вместо этого он скорее представляет собой средство, служащее для получения диагностической информации по активным потокам Win32 внутри выполняющегося процесса. В главе 18 подробно рассматривается создание многопоточных приложений с использованием пространства имен `System.Threading`.

Изучение модулей внутри процесса

Теперь давайте посмотрим, как проходить по загруженным модулям, которые обслуживаются в рамках конкретного процесса. Напоминаем, что *модуль* представляет собой общий термин, используемый для описания определенного файла `*.dll` (или даже `*.exe`), который обслуживается в определенном процессе. При получении доступа к `ProcessModuleCollection` через свойство `Process.Module` можно извлекать перечень всех модулей, которые обслуживаются внутри процесса: NET-, COM- или традиционных основанных на C библиотек. Для примера создадим следующую дополнительную вспомогательную функцию, способную перечислять модули в конкретном процессе на основании предоставляемого PID-идентификатора:

```
static void EnumModsForPid(int pID)
{
   Process theProc = null;
   try
   {
      theProc = Process.GetProcessById(pID);
   }
   catch
   {
      Console.WriteLine("-> Sorry...bad PID!"); // неверный PID
      Console.WriteLine("************************************\n");
      return;
   }
   Console.WriteLine("Here are the loaded modules for: {0}", theProc.ProcessName);
   try
   {
      ProcessModuleCollection theMods = theProc.Modules;
      foreach(ProcessModule pm in theMods)
      {
         string info = string.Format("-> Mod Name: {0}", pm.ModuleName);
         Console.WriteLine(info);
      }
      Console.WriteLine("************************************\n");
   }
```

```
    catch
    {
        Console.WriteLine("No mods!"); // модули не обнаружены
    }
}
```

Чтобы увидеть, как может выглядеть вывод, изучим загруженные модули в процессе, обслуживающем текущую демонстрационную программу (`ProcessManipulator`). Для этого понадобится запустить приложение, выяснить, какой PID-идентификатор был присвоен `ProcessManipulator.exe` (при помощи окна Task Manager (Диспетчер задач)), и передать это значение методу `EnumModsForPid()` (разумеется, соответствующим образом модифицировав метод `Main()`). После этого можно будет увидеть список всех модулей `*.dll`, которые используются в нашем простом консольном приложении (`GDI32.dll`, `USER32.dll`, `ole32.dll` и т.д.); их количество вполне может удивить.

На рис. 17.5 показан возможный вывод.

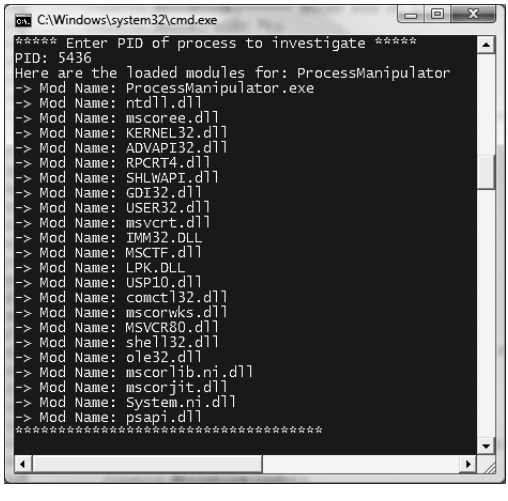

Рис. 17.5. Перечисление загруженных модулей внутри выполняющегося процесса

Запуск и остановка процессов программным образом

И, наконец, последними членами типа `System.Diagnostics.Process`, которые осталось здесь рассмотреть, являются методы `Start()` и `Kill()`. Эти методы, как не трудно догадаться по их именам, предоставляют возможность, соответственно, программного запуска и завершения процесса. Например, создадим следующий вспомогательный статический метод `StartAndKillProcess()`:

```
static void StartAndKillProcess()
{
    // Запуск Internet Explorer.
    Process ieProc = Process.Start("IExplore.exe", "www.intertech.com");
    Console.Write("--> Hit enter to kill {0}...", ieProc.ProcessName);
    Console.ReadLine();
    // Уничтожение процесса iexplore.exe.
    try
    {
        ieProc.Kill();
    }
    catch{}    // На случай, если пользователь уже завершил этот процесс...
}
```

Статический метод `Process.Start()` имеет несколько перегруженных версий. Как минимум, понадобится указывать дружественное имя запускаемого процесса (такого как `iexplore.exe` для Microsoft Internet Explorer). В этом примере используется версия метода `Start()`, которая позволяет указывать любое количество дополнительных аргументов, подлежащих передаче в точку входа программы (т.е. методу `Main()`).

Также метод `Start()` позволяет передавать тип `System.Diagnostics.ProcessStartInfo` и тем самым предоставлять дополнительные фрагменты информации касательно того, как должен запускаться определенный процесс. Ниже показано, как выглядит формальное определение `ProcessStartInfo` (все остальные подробности можно найти в документации по .NET Framework 3.5 SDK):

```
public sealed class System.Diagnostics.ProcessStartInfo : object
{
   public ProcessStartInfo();
   public ProcessStartInfo(string fileName);
   public ProcessStartInfo(string fileName, string arguments);
   public string Arguments { get; set; }
   public bool CreateNoWindow { get; set; }
   public StringDictionary EnvironmentVariables { get; }
   public bool ErrorDialog { get; set; }
   public IntPtr ErrorDialogParentHandle { get;  set; }
   public string FileName { get; set;  }
   public bool RedirectStandardError { get; set; }
   public bool RedirectStandardInput { get; set; }
   public bool RedirectStandardOutput { get; set; }
   public bool UseShellExecute { get; set; }
   public string Verb { get; set; }
   public string[] Verbs { get; }
   public ProcessWindowStyle WindowStyle { get; set; }
   public string WorkingDirectory { get; set; }
   public virtual bool Equals(object obj);
   public virtual int GetHashCode();
   public Type GetType();
   public virtual string ToString();
}
```

Важно обратить внимание на то, что какая бы версия метода `Process.Start()` не вызывалась, в ответ всегда возвращается ссылка на новый активизированный процесс. Если необходимо завершить этот процесс, достаточно вызвать на уровне экземпляра метод `Kill()`.

Исходный код. Проект `ProcessManipulator` доступен в подкаталоге `Chapter 17`.

Домены приложений .NET

Теперь, когда известно, какую роль играют процессы Win32 и как с ними взаимодействовать из управляемого кода, необходимо разобраться с тем, что собой представляет понятие домена приложения в .NET. В .NET исполняемые файлы не обслуживаются прямо внутри процесса (как это происходит в традиционных приложениях Win32). Вместо этого исполняемый файл .NET обслуживается в рамках отдельного логического раздела внутри процесса, который и называется *доменом приложения* (Application Domain — AppDomain). Как будет показано позже, в единственном процессе может содержаться несколько доменов приложений, обслуживающих свои исполняемые файлы .NET. Такое дополнительное разделение традиционного процесса Win32 обеспечивает ряд преимуществ, часть из которых перечислено ниже.

- Домены приложений играют ключевую роль в обеспечении нейтральности платформы .NET к операционной системе, поскольку такое логическое деление стирает отличия касательно того, каким образом базовая операционная система представляет загружаемый исполняемый файл.
- Домены приложений являются гораздо менее дорогостоящими в плане потребления вычислительных ресурсов и памяти по сравнению с полноценными процессами. Благодаря этому CLR-среде удается загружать и выгружать домены приложений намного быстрее, чем формальные процессы.
- Домены приложений обеспечивают более глубокий уровень изоляции для обслуживания загружаемого приложения. В случае выхода из строя какого-то одного домена приложения внутри процесса, остальные домены приложений остаются работоспособными.

Из приведенного выше списка основных преимуществ становится понятно, что один процесс может обслуживать любое количество доменов, каждый из которых совершенно и полностью изолируется от всех остальных доменов приложения внутри самого этого процесса (и внутри любого другого процесса). Из-за этого следует очень четко понимать, что приложение, выполняющееся в одном домене приложения, не может получать данные никакого рода (глобальные переменные или статические поля) из другого домена приложения, кроме как за счет использования какого-нибудь протокола распределенного программирования (такого как Windows Communication Foundation).

Хотя один процесс и *может* обслуживать множество доменов приложений, обычно такого не происходит. Как минимум, процесс ОС всегда обслуживает так называемый *домен приложения по умолчанию* (default application domain). Этот специальный домен приложения создается автоматически самой CLR-средой во время запуска процесса. После этого CLR-среда создает все остальные дополнительные домены приложений по мере необходимости. В случае возникновения такой потребности (которая *вряд ли* будет часто возникать при разработке .NET-приложений), домены приложений можно создавать программно во время выполнения внутри определенного процесса с помощью статических методов класса `System.AppDomain`. Этот класс также очень полезен для низкоуровневого управления доменами приложений. Наиболее важные члены этого класса перечислены в табл. 17.4.

Таблица 17.4. Некоторые члены класса `AppDomain`

Член	Описание
`CreateDomain()`	Этот статический метод позволяет создавать в текущем процессе новый домен приложения. Важно понимать, что CLR-среда будет сама создавать новые домены приложений по мере необходимости, поэтому вероятность возникновения реальной потребности в вызове этого члена близка к нулю.
`GetCurrentThreadId()`	Этот статический метод возвращает идентификатор активного потока в текущем домене приложения.
`Unload()`	Этот статический метод позволяет выгружать определенный домен приложения из конкретного процесса.
`BaseDirectory`	Это свойство возвращает информацию о базовом каталоге, который используется для выполнения зондирования на предмет наличия зависимых сборок.
`CreateInstance()`	Этот метод позволяет создавать экземпляр конкретного типа, определенного в конкретном файле сборки.

Окончание табл. 17.4

Член	Описание
`ExecuteAssembly()`	Этот метод позволяет выполнять сборку в рамках домена приложения за счет указания имени ее файла.
`GetAssemblies()`	Этот метод позволяет получать информацию о ряде .NET-сборок, которые были загружены в данный домен приложения (двоичные файлы на базе COM и C игнорируются).
`Load()`	Этот метод применяется для загрузки сборки в текущий домен приложения динамическим образом.

Вдобавок тип `AppDomain` поддерживает небольшой набор событий, которые отражают различные аспекты жизненного цикла домена приложения; все они перечислены в табл. 17.5.

Таблица 17.5. События типа `AppDomain`

Событие	Описание
`AssemblyLoad`	Возникает при загрузке сборки.
`AssemblyResolve`	Возникает, когда не удается установить местонахождение сборки.
`DomainUnload`	Возникает перед началом выгрузки домена приложения.
`ProcessExit`	Возникает в используемом по умолчанию домене приложения, когда его родительский процесс завершает свою работу.
`ResourceResolve`	Возникает, когда не удается установить местонахождение ресурса.
`TypeResolve`	Возникает, когда не удается установить местонахождение типа.
`UnhandledException`	Возникает в случае, если исключение не было перехвачено никаким обработчиком событий.

Перечисление доменов приложений процесса

Чтобы посмотреть, как программно взаимодействовать с доменами приложений .NET, создадим новый проект типа C# Console Application по имени `AppDomainManipulator` и определим в нем внутри типа `Program` статический метод `PrintAllAssembliesInAppDomain()`, использующий `AppDomain.GetAssemblies()` для получения списка всех двоичных файлов .NET, которые обслуживаются в интересующем домене приложения.

Этот список должен иметь вид массива типов `System.Reflection.Assembly`, а это значит, что необходимо использовать пространство имен `System.Reflection` (см. главу 16). После получения массива сборок по нему должен осуществляться проход с отображением дружественного имени и версии каждой сборки:

```
public static void PrintAllAssembliesInAppDomain(AppDomain ad)
{
   Assembly[] loadedAssemblies = ad.GetAssemblies();
   Console.WriteLine("***** Here are the assemblies loaded in {0} *****\n",
      ad.FriendlyName);
   foreach(Assembly a in loadedAssemblies)
   {
      Console.WriteLine("-> Name: {0}", a.GetName().Name);           // имя
      Console.WriteLine("-> Version: {0}\n", a.GetName().Version);   // версия
   }
}
```

Теперь модифицируем метод `Main()` так, чтобы перед вызовом `PrintAllAssembliesInAppDomain()` он получал ссылку на текущий домен приложения через свойство `AppDomain.CurrentDomain`.

Чтобы пример стал еще немного интереснее, давайте сделаем так, чтобы метод `Main()` отображал окно сообщения Windows Forms и вынуждал CLR-среду загружать сборки `System.Windows.Forms.dll`, `System.Drawing.dll` и `System.dll` (обязательно добавьте ссылки на эти сборки и соответствующим образом обновите набор операторов `using`):

```
static void Main(string[] args)
{
   Console.WriteLine("***** Fun with AppDomains *****\n");
   // Получение информации о текущем домене приложения.
   AppDomain defaultAD= AppDomain.CurrentDomain;
   // Этот вызов делается просто для выполнения загрузки
   // дополнительных сборок в данный домен приложения.
   MessageBox.Show("Hello");
   PrintAllAssembliesInAppDomain(defaultAD);
   Console.ReadLine();
}
```

На рис. 17.6 показано, как будет выглядеть вывод.

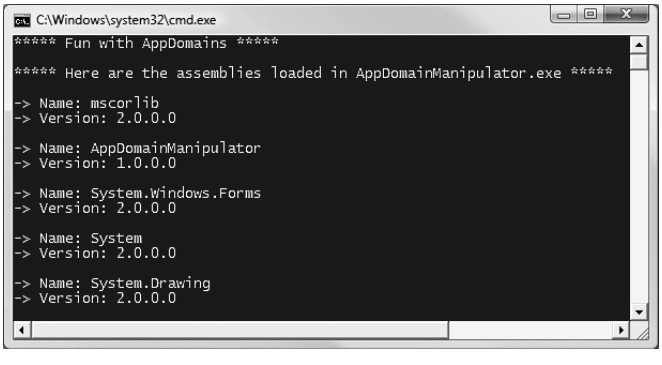

Рис. 17.6. Перечисление сборок, используемых в рамках текущего домена приложения

Создание новых доменов приложений программным образом

Напоминаем, что один процесс может обслуживать множество доменов приложений. Хотя необходимость создавать домены приложений вручную в коде действительно возникает крайне редко (а то и почти никогда), все-таки возможность делать это существует и заключается она в использовании статического метода `CreateDomain()`. Как не трудно догадаться, метод `AppDomain.CreateDomain()` имеет несколько перегруженных версий. Как минимум, ему требуется указывать дружественное имя, которым должен обладать новый домен приложения, как показано ниже:

```
static void Main(string[] args)
{
...
   // Создание нового домена приложения в текущем процессе.
   AppDomain anotherAD = AppDomain.CreateDomain("SecondAppDomain");
   PrintAllAssembliesInAppDomain(anotherAD);
   Console.ReadLine();
}
```

Если теперь запустить приложение (рис. 17.7), можно увидеть, что сборки `System.Windows.Forms.dll`, `System.Drawing.dll` и `System.dll` будут загружаться только в используемый по умолчанию домен приложения. Это может показаться нелогичным для тех, кто привык иметь дело с традиционными приложениями Win32 (потому что по идее оба домена приложений должны иметь доступ к одному и тому же набору сборок). Напоминаем, однако, что каждая сборка загружается в *домен приложения*, а не прямо в сам процесс.

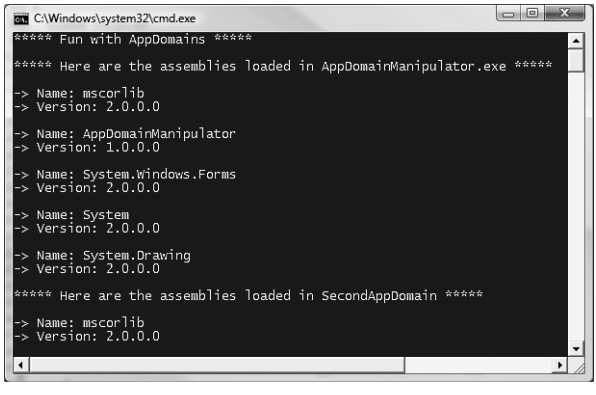

Рис. 17.7. Один процесс с двумя доменами приложений

Далее важно обратить внимание на то, в домен приложения `SecondAppDomain` будет автоматически помещаться своя собственная копия `mscorlib.dll`, поскольку эта важная сборка автоматически загружается CLR-средой для каждого домена приложения. Отсюда возникает вопрос: как тогда программно загружать сборку в домен приложения? Очень просто: с помощью метода `AppDomain.Load()` (или, в качестве альтернативного варианта, посредством метода `AppDomain.executeAssembly()`, позволяющего загружать и выполнять код метода `Main()` сборки `*.exe`).

Например, если скопировать сборку `CarLibrary.dll` в каталог приложения `AppDomainManipulator.exe`, тогда можно загрузить ее в домен приложения `SecondAppDomain` следующим образом:

```
static void Main(string[] args)
{
...
   // Загрузка CarLibrary.dll в новый домен приложения.
   AppDomain anotherAD = AppDomain.CreateDomain("SecondAppDomain");
   try
   {
     anotherAD.Load("CarLibrary");
     PrintAllAssembliesInAppDomain(anotherAD);
   }
   catch(Exception ex)
   {
     Console.WriteLine(ex.Message);
   }
   Console.ReadLine();
}
```

Чтобы еще четче проиллюстрировать отношения между процессами, доменами приложения и сборками, на рис. 17.8 приведена схема внутреннего устройства только что созданного процесса `AppDomainManipulator.exe`.

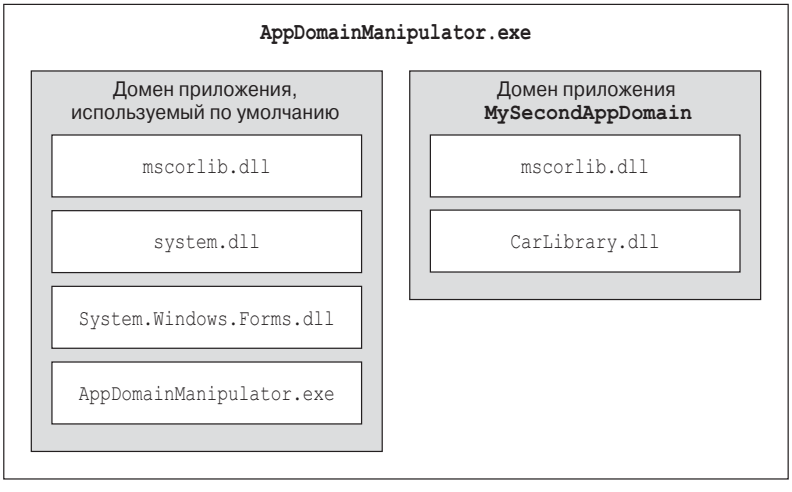

Рис. 17.8. Внутреннее устройство процесса `AppDomainManipulator.exe`

На заметку! Во время отладки данного проекта (нажатием <F5>) в каждый домен приложения будет загружаться много дополнительных сборок, необходимых Visual Studio для выполнения процесса отладки. В случае просто запуска этого проекта (нажатием <Ctrl+F5>) будут отображаться только сборки, загружаемые конкретно в каждый домен приложения.

Выгрузка доменов приложений программным образом

Важно отметить, что CLR-среда не позволяет выгружать отдельные сборки .NET. Однако с помощью метода `AppDomain.Unload()` все-таки можно производить выборочную выгрузку определенного домена приложения из обслуживающего его процесса. В таком случае будет происходить поочередная выгрузка из этого домена каждой содержащейся в нем сборки.

Напоминаем, что тип `AppDomain` поддерживает небольшой набор событий, одним из которых является `DomainUnload`. Это событие срабатывает тогда, когда домен приложения (не тот, что используется по умолчанию) выгружается из процесса, в котором он содержится. Другим интересным событием является событие `ProcessExit`, которое срабатывает тогда, когда из процесса выгружается используемый по умолчанию домен (что, очевидно, влечет за собой завершение самого процесса). Следовательно, если необходимо программно выгрузить домен `anotherAD` из процесса `AppDomainManipulator.exe` и получить уведомление о его уничтожении, можно реализовать следующую логику обработки событий:

```
static void Main(string[] args)
{
...
    // Привязка к событию DomainUnload.
    anotherAD.DomainUnload += new EventHandler(anotherAD_DomainUnload);
    // Выгрузка домена приложения anotherAD.
    AppDomain.Unload(anotherAD);
    Console.ReadLine();
}
```

Важно обратить внимание, что событие `DomainUnload` работает совместно с делегатом `System.EventHandler`, и потому метод `anotherAD_DomainUnload()` должен принимать следующие аргументы:

```
static void anotherAD_DomainUnload(object sender, EventArgs e)
{
   Console.WriteLine("***** Unloaded anotherAD! *****\n");
}
```

Чтобы получать уведомление о выгрузке используемого по умолчанию домена приложения, достаточно изменить метод `Main()` так, чтобы он обрабатывал генерируемое этим доменом событие `ProcessEvent`:

```
static void Main(string[] args)
{
   Console.WriteLine("***** Fun with AppDomains *****\n");
   AppDomain defaultAD = AppDomain.CurrentDomain;
   defaultAD.ProcessExit += new EventHandler(defaultAD_ProcessExit);
...
}
```

и определить соответствующий обработчик событий:

```
static void defaultAD_ProcessExit(object sender, EventArgs e)
{
   Console.WriteLine("***** Unloaded defaultAD! *****\n");
}
```

Исходный код. Проект `AppDomainManipulator` доступен в подкаталоге `Chapter 17`.

Границы контекстов объектов

Как было только что показано, домены приложений представляют собой логические разделы внутри процесса, которые используются для обслуживания сборок .NET. Однако на этом дело не заканчивается, потому что каждый домен приложения может делиться дальше на многочисленные контексты. Вкратце, контекст в .NET позволяет домену приложения обеспечивать "отдельное место" для каждого объекта.

В случае применения контекста CLR-среда гарантирует обработку объектов со специальными требованиями времени выполнения надлежащим и соответствующим образом путем перехвата вызовов методов, производимых в и за пределами конкретного контекста. Этот слой перехвата позволяет CLR-среде корректировать текущий вызов метода так, чтобы он соответствовал контекстным настройкам конкретного объекта. Например, в случае определения типа класса C#, требующего автоматического обеспечения безопасности в отношении потоков (с помощью атрибута `[Synchronization]`), CLR-среда будет создавать во время его размещения "синхронизированный контекст".

Точно так же, как для каждого процесса создается свой используемый по умолчанию домен приложения, для каждого домена приложения создается свой используемый по умолчанию контекст. Этот контекст (иногда еще называемый *нулевым*, потому он всегда создается в любом домене приложения первым) применяется для группирования вместе объектов .NET, которые не имеют никаких ни специфических, ни уникальных контекстных потребностей. Как не трудно догадаться, подавляющее большинство объектов .NET загружается именно в нулевой контекст. В случае если CLR-среда определяет, что у создаваемого нового объекта имеются специальные потребности, она создает внутри отвечающего за его обслуживание домена приложения новые границы контекста. На рис. 17.9 схематично показаны отношения между процессом, доменами и контекстами.

Контекстно-свободные и контекстно-зависимые типы

Типы .NET, которые не требуют никакого особого контекстного сопровождения, называются *контекстно-свободными* (context-agile) объектами. К таким объектам доступ может получаться из любого места внутри обслуживающего домена приложения без нарушения их требований времени выполнения.

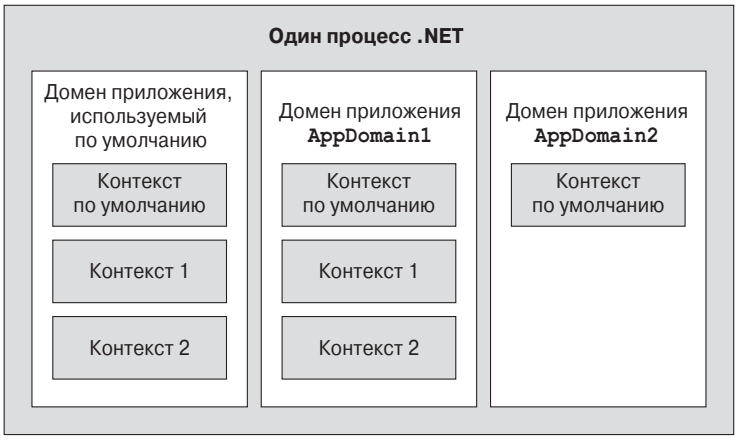

Рис. 17.9. Процессы, домены приложений и границы контекста

Создание контекстно-свободных объектов не требует приложения никаких особых усилий, поскольку для их создания просто не нужно ничего делать (ни снабжать их какими-либо контекстными атрибутами, ни порождать от базового класса `System.ContextBoundObject`):

```
// Контекстно-свободный объект загружается в нулевой контекст.
public class SportsCar{}
```

С другой стороны, объекты, которые действительно требуют выделения отдельного контекста, называются *контекстно-зависимыми* (context-bound) объектами и должны *обязательно* наследоваться от базового класса `System.ContextBoundObject`. Этот базовый класс отражает тот факт, что соответствующий объект может правильно функционировать только в рамках контекста, в котором он был создан. Зная о роли контекста .NET должно быть совершенно ясно, что в случае попадания контекстно-зависимого объекта каким-то образом в несоответствующий контекст, будет обязательно случаться что-то плохое, причем в самый неподходящий момент.

Помимо наследования от `System.ContextBoundObject`, контекстно-зависимый тип должен сопровождаться специальными атрибутами .NET, называемыми *контекстными атрибутами*. Все контекстные атрибуты наследуются от базового класса `ContextAttribute`, который является членом пространства имен `System.Runtime.Remoting.Contexts`:

```
public class ContextAttribute :
    Attribute, IContextAttribute, IContextProperty
{
    public ContextAttribute(string name);
    public string Name { virtual get; }
    public object TypeId { virtual get; }
    public virtual bool Equals(object o);
    public virtual void Freeze(
        System.Runtime.Remoting.Contexts.Context newContext);
    public virtual int GetHashCode();
    public virtual void GetPropertiesForNewContext(
        System.Runtime.Remoting.Activation.IConstructionCallMessage ctorMsg);
    public Type GetType(); public virtual bool IsContextOK(
        System.Runtime.Remoting.Contexts.Context ctx,
        System.Runtime.Remoting.Activation.IConstructionCallMessage ctorMsg);
    public virtual bool IsDefaultAttribute();
```

```
    public virtual bool IsNewContextOK(
        System.Runtime.Remoting.Contexts.Context newCtx);
    public virtual bool Match(object obj);
    public virtual string ToString();
}
```

Благодаря тому, что класс `ContextAttribute` не является герметизированным, допускается создавать и свои собственные специальные контекстные атрибуты (унаследовав от `ContextAttribute` и переопределив необходимые виртуальные методов). После этого, конечно же, можно создавать и специальные программные единицы, способные реагировать на контекстные настройки.

На заметку! В настоящей книге о том, как создавать специальные контексты для объектов, подробно рассказываться не будет; всю необходимую информацию можно найти в книге *Applied .NETAttributes* Джейсона Бока (Jason Bock) и Тома Барнаби (Tom Barnaby), которая была опубликована издательством Apress в 2003 г.

Определение контекстно-зависимых объектов

Давайте предположим, что требуется определить класс (`SportsCarTS`) так, чтобы он автоматически являлся безопасным к потокам по своей природе, даже без размещения внутри реализации его членов жестко закодированной логики синхронизации потоков. Чтобы получить такой класс, достаточно унаследовать его от `ContextBoundObject` и применить к нему атрибут `[Synchronization]`, как показано ниже.

```
using System.Runtime.Remoting.Contexts;
// Этот контекстно-зависимый тип будет загружаться только
// в синхронизированный (т.е. безопасный к потокам) контекст.
[Synchronization]
public class SportsCarTS : ContextBoundObject
{}
```

Типы, которые сопровождаются атрибутом `[Synchronization]`, всегда загружаются в безопасный к потокам контекст. Зная о специальных контекстных потребностях класса `MyThreadSafeObject`, давайте представим, какие проблемы будут возникать в случае перемещения соответствующего объекта из синхронизированного контекста в не синхронизированный. Объект тут же перестанет быть безопасным к потокам и, следовательно, станет кандидатом на массовое повреждение данных, поскольку тогда к нему будет пытаться получить доступ одновременно множество потоков. За счет наследования `SportsCarTS` от `ContextBoundObject` обеспечивается гарантия того, что CLR-среда никогда не будет пытаться перемещать объекты `SportsCarTS` за пределы синхронизированного контекста.

Инспектирование контекста объекта

Хотя необходимость программного взаимодействия с контекстом при написании приложений возникает очень редко, один пример того, как это делается, рассмотреть не помешает. Давайте создадим новый проект типа Console Application по имени `ContextManipulator` и определим в нем один контекстно-свободный класс (`SportsCar`) и один контекстно-зависимый (`SportsCarTS`):

```
using System.Runtime.Remoting.Contexts;       // Для типа Context.
using System.Threading;                       // Для типа Thread.

// SportsCar не имеет никаких специальных контекстных потребностей и будет
// загружаться в создаваемый по умолчанию контекст внутри домена приложения.
```

```
class SportsCar
{
    public SportsCar()
    {
        // Получение информации о контексте и отображение его идентификатора.
        Context ctx = Thread.CurrentContext;
        Console.WriteLine("{0} object in context {1}", this.ToString(), ctx.ContextID);
        foreach(IContextProperty itfCtxProp in ctx.ContextProperties)
          Console.WriteLine("-> Ctx Prop: {0}", itfCtxProp.Name);
    }
}

// Тип SportsCarTS требует загрузки в синхронизированный контекст.
[Synchronization]
class SportsCarTS : ContextBoundObject
{
    public SportsCarTS()
    {
        // Получение информации о контексте и отображение его идентификатора.
        Context ctx = Thread.CurrentContext;
        Console.WriteLine("{0} object in context {1}", this.ToString(), ctx.ContextID);
        foreach(IContextProperty itfCtxProp in ctx.ContextProperties)
            Console.WriteLine("-> Ctx Prop: {0}", itfCtxProp.Name);
    }
}
```

Здесь важно обратить внимание, что каждый конструктор получает тип `Context` из текущего потока выполнения через статическое свойство `Thread.CurrentContext`. За счет использования объекта `Context` можно легко отображать статистические данные о контексте, такие как присвоенный ему идентификатор, а также ряд дескрипторов, получаемых через `Context.ContextProperties`. Это свойство возвращает объект, реализующий интерфейс `IContextProperty`, который предоставляет доступ к каждому дескриптору через свойство `Name`. Теперь модифицируем метод `Main()` так, чтобы он размещал экземпляр каждого из этих классов:

```
static void Main(string[] args)
{
    Console.WriteLine("***** Fun with Object Context *****\n");
    // При создании этих объектов будет отображаться информация об их контексте.
    SportsCar sport = new SportsCar();
    Console.WriteLine();

    SportsCar sport2 = new SportsCar();
    Console.WriteLine();

    SportsCarTS synchroSport = new SportsCarTS();
    Console.ReadLine();
}
```

По мере создания объектов конструкторы классов будут отображать различные фрагменты касающейся их контекста информации, как показано на рис. 17.10.

Из-за того, что класс `SportsCar` не был снабжен атрибутом контекста, CLR-среда разместила объекты `sport` и `sport2` в контексте с идентификатором 0 (т.е. в контексте по умолчанию). Объект `SportsCarTS`, однако, был загружен в уникальный контекст (с идентификатором 1), поскольку его контекстно-зависимый класс был снабжен атрибутом `[Synchronization]`.

Исходный код. Проект `ContextManipulator` доступен в подкаталоге `Chapter 17`.

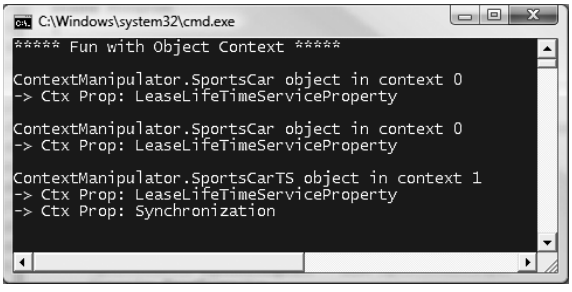

Рис. 17.10. Изучение информации о контексте объекта

Подведение итогов по процессам доменам приложений и контекстам

К этому моменту должно было сложиться четкое представление о том, каким образом происходит обслуживание .NET-сборки в CLR-среде. Ниже перечислены некоторые ключевые моменты, которые следует запомнить.

- Любой процесс .NET может обслуживать один и более доменов приложений. Каждый домен приложения, в свою очередь, может обслуживать любое количество взаимосвязанных сборок .NET. Все домены приложений могут по отдельности загружаться и выгружаться CLR-средой (или программно с помощью типа `System.AppDomain`).
- Любой отдельно взятый домен приложения может включать в себя один и более контекстов. Посредством контекста CLR-среде удается помещать информацию об "особых потребностях" объекта в логический контейнер и тем самым гарантировать принятие их во внимание на этапе выполнения.

Возможно, кому-то приведенный выше материал показался слишком сложным. Однако по большей части исполняющая среда .NET автоматически разбирается с деталями процессов, доменов приложений и контекстов без участия программиста. Представленная ранее информация обеспечивает солидную базу для изучения приемов программирования многопоточных приложений в рамках платформы .NET.

Резюме

Главной задачей этой главы было продемонстрировать, каким образом происходит обслуживание исполняемой сборки .NET на платформе .NET. Как здесь было показано, старое понятие процесса Win32 было внутренне изменено и адаптировано под потребности CLR. Любой отдельно взятый процесс (которым на программном уровне можно управлять с использованием типа `System.Diagnostics.Process`) теперь состоит из множества доменов приложений, которые представляют собой изолированные и независимые границы внутри данного процесса.

Как было показано, каждый процесс может обслуживать несколько доменов приложений, а каждый из этих доменов приложений, в свою очередь, может обслуживать и выполнять любое количество связанных между собой сборок. Более того, еще каждый домен приложения может содержать и любое число контекстов. За счет использования такого дополнительного слоя изоляции CLR-среде удается гарантировать корректную обработку специфических потребностей объектов на этапе выполнения.