Внутри .NET Remoting

Реализация чуда

Автор: Игорь Ткачев
The RSDN Group
Источник: RSDN Magazine #1-2003

Опубликовано: *11.07.2003*
Исправлено: *13.03.2005*
Версия текста: *1.1*

Врага нужно знать в лицо.

Распределённые приложения уже давно перестали быть чем-то особенным и необычным. Сегодня практически любой разработчик имеет в своём арсенале одну, а то и две технологии для разработки распределённых систем. До сих пор лидерами распределённых вычислений были DCOM, RMI и CORBA. Сегодня же мы наблюдаем выход на сцену ещё одного игрока – .NET Remoting.

Как говорится, врага нужно знать в лицо. Хочется добавить, что для успешной борьбы с ним необходимо знать как его сильные, так и слабые стороны, а ещё лучше – его внутреннее устройство и механизмы работы. Идеальный вариант – наличие документации по его применению, исходных текстов и отладчика. К счастью, всё это у нас есть и нам ничто не мешает заглянуть внутрь .NET Remoting и немножко его попотрошить.

Как раз этим мы с вами сейчас и займёмся.

Простейшее распределённое приложение

Напишем простейшее приложение, состоящее, как и положено, из сервера, клиента и испытуемого объекта.

В качестве сервера вполне подойдёт обычное консольное .NET-приложение, исходный текст которого приведён ниже:

```
// Server.cs

using System;
using System.Runtime.Remoting;

namespace Server
{
  class Server
  {
    [STAThread]
    static void Main(string[] args)
    {
      RemotingConfiguration.Configure("Server.exe.config");

      Console.WriteLine("Press Enter to exit");
      Console.ReadLine();
    }
  }
}
```

Сервером эту программу делает всего лишь одна выделенная в листинге строчка, которая вызывает процедуру конфигурирования подсистемы Remoting. Для этого она использует следующий конфигурационный файл:

```
<!-- App.config -->

<configuration>
  <system.runtime.remoting>
    <application name="RemotingTest">
      <service>
        <wellknown
          mode="SingleCall"
          type="TestObject.Test, TestObject"
          objectUri="Test.rem" />
      </service>
      <channels>
        <channel ref="tcp" port="8008" />
      </channels>
    </application>
  </system.runtime.remoting>
</configuration>
```

Обратите внимание: имена того конфигурационного файла, который мы используем в программе, и того, который мы создаём в Visual Studio, различаются. Во время сборки проекта Visual Studio копирует файл App.config из каталога проекта в целевой каталог программы и переименовывает его по правилам ИмяПрограммы.exe.config. В нашем случае это будет Server.exe.config.

Также нам будет просто необходим сам тестируемый объект. Вот он:

```
// TestObject.cs

using System;

namespace TestObject
{
  public class Test: MarshalByRefObject
  {
    public string GetAppName()
    {
      return AppDomain.CurrentDomain.FriendlyName;
    }
```

```
    }
  }
```

Здесь ключевым моментом является наследование от класса MarshalByRefObject. Этого достаточно для возможности удаленного использования экземпляра такого класса.

Наш объект является dll-модулем и будет возвращать название той программы, в которой он загружен. Это позволит легко определять, где в данный момент работает объект – на клиенте или на сервере.

И, наконец, клиент:

```
// Client.cs

using System;
using System.Runtime.Remoting;
using TestObject;

namespace Client
{
  class Client
  {
    [STAThread]
    static void Main(string[] args)
    {
      RemotingConfiguration.Configure("Client.exe.config");

      Test test = new Test();
      Console.WriteLine(test.GetAppName());
    }
  }
}
```

Так же, как и в случае с сервером, для конфигурирования мы будем использовать файл:

```
<!-- App.config -->

<configuration>
  <system.runtime.remoting>
    <application>
      <client>
        <wellknown
          type="TestObject.Test, TestObject"
          url="tcp://localhost:8008/RemotingTest/Test.rem" />
      </client>
    </application>
  </system.runtime.remoting>
</configuration>
```

После сборки всех модулей запустите сервер, а затем клиента. Если вы всё сделали правильно, клиент должен вывести на экран следующую строчку:

```
Server.exe
```

Это говорит о том, что домен, в котором выполняется объект – сервер.

Теперь изменим одну строчку в файле Client.exe.config:

```
<!-- Client.exe.config -->

<configuration>
  <system.runtime.remoting>
    <application>
```

```
        <client>
          <wellknown
            type="TestObject.Test1, TestObject"
            url="tcp://localhost:8008/RemotingTest/Test.rem" />
        </client>
      </application>
    </system.runtime.remoting>
  </configuration>
```

Запустите клиентскую программу ещё раз. Результат будет другим:

```
Client.exe
```

То есть теперь объект запускается на стороне клиента, хотя код программы мы не меняли.

Что это, чудо?

Нет, конечно, не чудо. Мы-то с вами знаем, что чудес на свете не бывает, и сделали это не какие-нибудь добрые феи из сказки, а, скорее всего, злые дядьки из Microsoft. А раз уж это сделали не сказочные феи, ничто не мешает и нам разобраться в данной ситуации.

Proxy

Итак, давайте ещё раз проанализируем наш пример и полученный результат.

Всё что связывает наш объект и сервер – это следующая строчка в config-файле:

```
<wellknown
    mode="SingleCall"
    type="TestObject.Test, TestObject"
    objectUri="Test.rem" />
```

Когда на сервер приходит запрос от клиента, то сервер ищет сборку с именем TestObject (часть параметра после запятой), и в ней – класс TestObject.Test (первая часть параметра). В данном случае TestObject – это пространство имён.

> **ПРИМЕЧАНИЕ**
>
> Существует также и программный способ регистрации удалённых объектов, как для сервера, так и для клиента. В частности, мы можем воспользоваться функциями RegisterWellKnownServiceType и RegisterWellKnownClientType из пространства имён System.Runtime.Remoting.RemotingConfiguration.

Для регистрации удалённого объекта на клиенте используется похожий механизм:

```
<wellknown
    type="TestObject.Test, TestObject"
    url="tcp://localhost:8008/RemotingTest/Test.rem" />
```

При регистрации объекта клиентское приложение конфигурирует систему ремоутинга таким образом, что, создавая объект обычными штатными средствами языка через оператор new, мы, тем не менее, получаем не сам объект, а некий заменитель, который переадресует все вызовы на сервер.

В этом легко убедиться, если прибегнуть к помощи нашего лучшего друга и помощника – отладчика. Запустим программу под отладчиком и проанализируем объект test после его создания. На рисунке 1 показано примерно то, что мы должны увидеть в окне Watch.

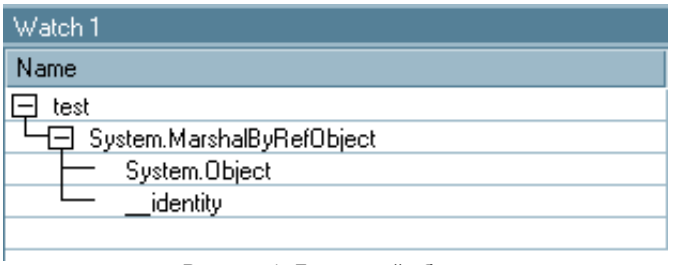

Рисунок 1. Локальный объект.

Все, как и ожидалось. Наш объект является наследником System.MarshalByRefObject, который, в свою очередь, происходит от System.Object.

Теперь восстановим Client.exe.config и выполним эту процедуру ещё раз. На рисунке 2 показан результат.

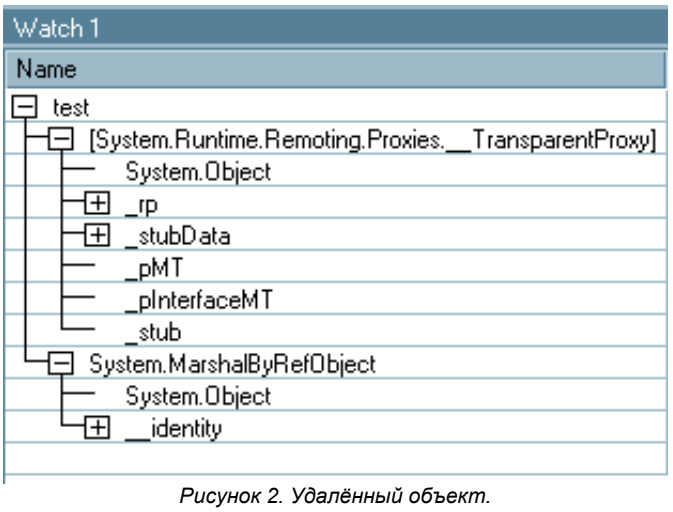

Рисунок 2. Удалённый объект.

Наследование от MBR сохранилось, но теперь у нас есть ещё один базовый класс – __TransparentProxy.

> Здесь и далее я буду использовать сокращение **MBR**, которое, как вы уже догадались, означает **marshal-by-reference**.

Собственно говоря, этот самый transparent proxy мы с вами и получаем вместо объекта, когда создаём его удалённо. Разберём подробнее, что это такое.

Proxy представляет собой объект-заменитель, который создаётся внутри клиентского процесса и предоставляет клиенту те же интерфейсы, что и сам объект, с которым клиент имеет дело.

В .NET Remoting имеются два вида прокси – transparent proxy и real proxy. Честно говоря, второй вид не очень соответствует приведённому выше определению, но это стандартная .NET терминология, которой мы и будем придерживаться.

Первый прокси создаётся на лету во время выполнения программы и предназначен для перехвата клиентских вызовов объекта. Получив управление, transparent proxy упаковывает параметры метода в специальную структуру и передаёт её далее real proxy. Последний обрабатывает вызов по своему усмотрению и возвращает структуру, содержащую выходные параметры. Transparent proxy распаковывает эти параметры и возвращает управление клиенту.

На рисунке 3 схематически показан механизм вызова объекта клиентом.

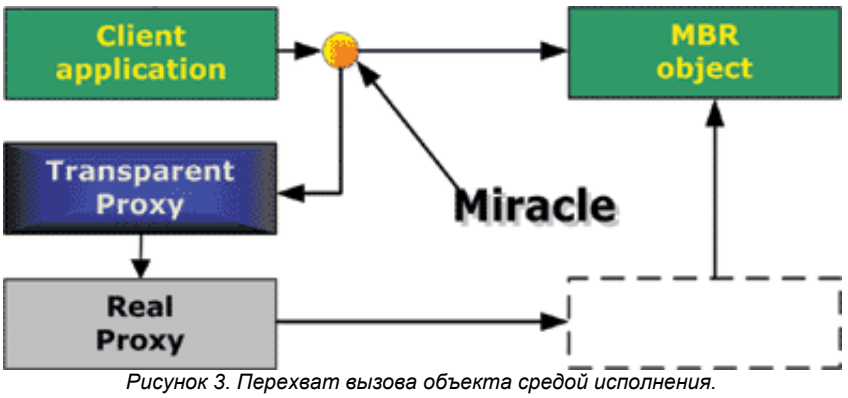

Рисунок 3. Перехват вызова объекта средой исполнения.

Таким образом, роль transparent proxy заключается исключительно в перехвате вызовов клиента и передачи управления real proxy. Transparent proxy создаётся средой исполнения непосредственно как unmanaged-код, и мы никак не можем вмешаться в этот процесс и заменить его своей реализацией. Но это и ни к чему. Вместо этого мы можем создать такой transparent proxy, который будет вызывать наш собственный real proxy.

Следующий пример демонстрирует такую возможность:

```
using System;
using System.Runtime.Remoting;
using System.Runtime.Remoting.Proxies;
using System.Runtime.Remoting.Messaging;
using System.Runtime.Remoting.Activation;
using TestObject;

namespace Client
{
  class TestProxy : RealProxy
  {
    MarshalByRefObject obj;

    public TestProxy(Type type,MarshalByRefObject o)
    : base(type)
    {
      obj = o;
    }

    // Этот метод будет вызываться из transparent proxy
    // при каждом вызове любого метода объекта клиентом.
    public override IMessage Invoke(IMessage msg)
    {
      IMethodCallMessage msgCall = (IMethodCallMessage)msg;
      return RemotingServices.ExecuteMessage(obj,msgCall);
    }

    static public Test CreateInstance()
    {
      Test test = new Test();

      RealProxy rp = new TestProxy(typeof(Test),test);

      // Создание transparent proxy.
      return (Test)rp.GetTransparentProxy();
    }
  }

  class Client
  {
    [STAThread]
    static void Main(string[] args)
    {
      Test test = TestProxy.CreateInstance();

      Console.WriteLine(test.GetAppName());
    }
```

```
    }
}
```

Результат выполнения программы:

```
Client.exe
```

Transparent proxy в данном примере создаётся вызовом метода GetTransparentProxy класса RealProxy. Далее клиентское приложение уже работает только с ним – рисунок 4.

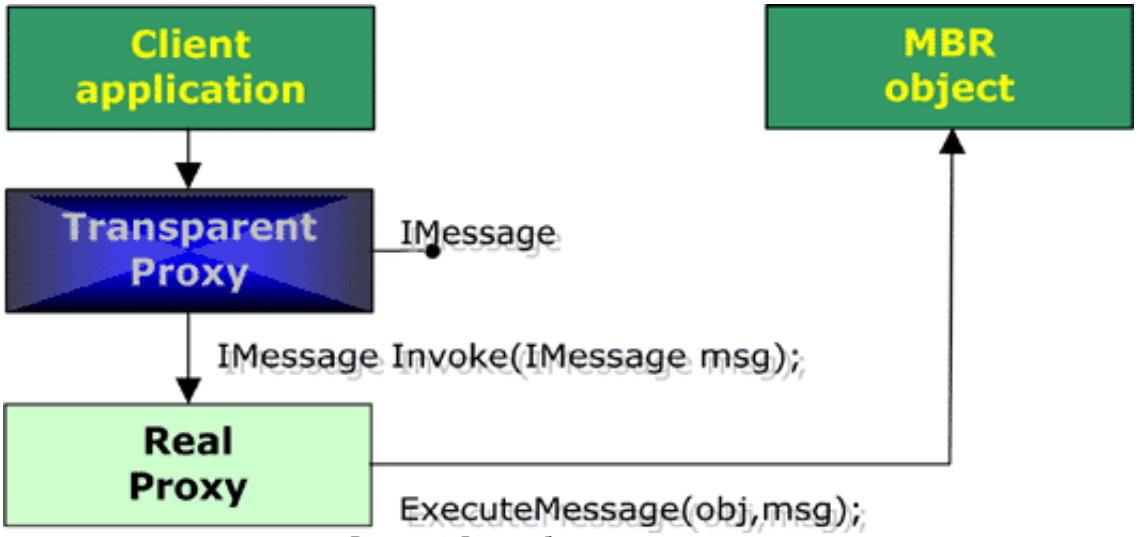

Рисунок 4. Вызов объекта через наш proxy.

Интерфейс IMessage, который вы можете видеть на рисунке, как раз и является той самой специальной структурой, в которую упаковываются параметры вызываемого метода. Существует несколько расширений этого интерфейса – IMethodCallMessage, IConstructionCallMessage, IMethodReturnMessage. Как видно из названий, они предназначены для разных целей. Transparent proxy создаёт экземпляр объекта, поддерживающего интерфейс IMessage, заполняет его единственное свойство Properties данными, содержащими всю необходимую информацию для вызова удалённого метода, после чего вызывает метод Invoke объекта real proxy. Таким образом, производится обратный переход в managed-код.

В предыдущем примере мы вызывали метод объекта локально, используя для этого функцию RemotingServices.ExecuteMessage. Но ничто не мешает нам вызвать объект удалённо на сервере. Следующий пример как раз это и демонстрирует:

```
using System;
using System.Runtime.Remoting;
using System.Runtime.Remoting.Proxies;
using System.Runtime.Remoting.Messaging;
using System.Runtime.Remoting.Activation;
using TestObject;

namespace Client
{
  class TestProxy : RealProxy
  {
    MarshalByRefObject obj;

    public TestProxy(Type type,MarshalByRefObject o)
    : base(type)
    {
      obj = o;
    }

    public override IMessage Invoke(IMessage msg)
    {
      MethodCallMessageWrapper mcw =
        new MethodCallMessageWrapper((IMethodCallMessage)msg);
```

```
      mcw.Uri = RemotingServices.GetObjectUri(obj);

      IMessageSink sink = RemotingServices.GetEnvoyChainForProxy(obj);
      return sink.SyncProcessMessage(msg);
    }

    static public Test CreateInstance()
    {
      //Test test = new Test();

      // Если закоментировать следующие строки создания объекта
      // и раскомментировать предыдущую и регистрацию в процедуре Main,
      // то результат будет таким же.
      Test test =
        (Test)Activator.GetObject(typeof(Test),
        "tcp://localhost:8008/RemotingTest/Test.rem");

      RealProxy rp = new TestProxy(typeof(Test),test);

      return (Test)rp.GetTransparentProxy();
    }
  }

  class Client
  {
    [STAThread]
    static void Main(string[] args)
    {
      //RemotingConfiguration.Configure("Client.exe.config");

      Test test = TestProxy.CreateInstance();

      Console.WriteLine(test.GetAppName());
    }
  }
}
```

Результат работы программы:

```
Server.exe
```

Реализация чуда

Всё это здорово. Мы уже знаем, как работает перехватчик, и даже можем заставить его передавать управление в наш код. Но единственное, чем мы пока не можем управлять, – это прозрачным для клиента конструированием объекта, т.е. тем самым "чудом". Поэтому давайте вернёмся к нашим злобным феям и разберём подробно процесс создания MBR-объекта в .NET Framework. На этот раз нам понадобится не только отладчик, но и исходные коды SS CLI.

Начнём с дисассемблера. Вот как выглядит код создания нашего объекта на MSIL:

```
          ; Test test = new Test();
IL_0000:  newobj instance void [TestObject]TestObject.Test::.ctor()
```

Команда newobj создаёт объект и вызывает конструктор, который передаётся ей в качестве параметра.

А вот как выглядит ассемблер после обработки MSIL-кода JIT-компилятором:

```
          ; Test test = new Test();
00000025  mov   ecx,3755C4h
0000002a  call  76297C23 ; JIT_NewCrossContext, создание "сырого" объекта
0000002f  mov   esi,eax
```

```
00000031  mov   ecx,esi
00000033  call  dword ptr ds:[00375608h] ; вызов конструктора Test()
00000039  mov   edi,esi
```

Здесь мы видим, что объект создаётся в два этапа: создание row-объекта и затем уже вызов конструктора.

ПРИМЕЧАНИЕ

JIT (Just in Time) компиляция – процесс преобразования p-кода в машинные команды во время исполнения программы. JIT-компилятор в .NET Framework запускается каждый раз при первом вызове любой managed-процедуры.

Первый этап полностью контролируется JIT. При обработке команды "newobj" в недрах JIT-компилятора вызывается следующий метод (взято из исходных текстов CLI 1.0 Release):

```
// fjit.cpp

FJitResult FJit::compileCEE_NEWOBJ();
```

Этот метод проверяет тип создаваемого объекта, на основании чего решает, какая из семейства функций JIT_New будет вызвана для создания данного объекта, и помещает вызов этой функции в результирующий код. Существует множество функций JIT_New: одна – для массивов, другая – для строк, еще одна – для простых объектов, а также отдельная для объектов-наследников класса MarshalByRefObject. Для проверки на принадлежность к иерархии MBR JIT-компилятор вызывает следующий метод:

```
// remoting.cpp

BOOL CRemotingServices::IsRemoteActivationRequired(EEClass* pClass)
{
  BOOL fRequiresNewContext = pClass->IsMarshaledByRef();
  return fRequiresNewContext;
}
```

То есть всё что нужно – это чтобы класс был MBR-объектом или его наследником. Если это условие соблюдается, то в выходной код подставляется вызов следующей функции, которая и отвечает за создание MBR row-объекта:

```
// jitinterface.cpp

Object* JIT_NewCrossContext(CORINFO_CLASS_HANDLE typeHnd_);
```

ПРИМЕЧАНИЕ

В файле jitinterface.cpp можно также найти методы new и для других типов объектов.

После того, как функция откомпилирована, она может быть вызвана исполняющей средой.

Далее приведены методы, участвующие в создании row-объекта. Текст программ максимально сокращён. Из них удалены фрагменты, которые, как я думаю, не относятся непосредственно к исследуемой теме. По ходу текста вы найдёте мои комментарии, в том числе и комментарии к комментариям. Надеюсь, вы легко отличите их от оригинальных :o)

Как уже упоминалось, следующий метод вызывается непосредственно из managed-кода для конструирования row-объекта. Здесь проверяется необходимость управляемой (managed) активации и если такая активация необходима, то вызывается метод CreateProxyOrObject.

```
// jitinterface.cpp

Object* JIT_NewCrossContext(CORINFO_CLASS_HANDLE typeHnd_)
{
  MethodTable *pMT = (MethodTable *)typeHnd_;
  ManagedActivationType ActType;
  ActType = CRemotingServices::RequiresManagedActivation(pMT->GetClass());

  // IT: Вообще-то здесь должна быть проверка
  // на ActType != NoManagedActivation
  if (ActType) {
    // IT: Вызываем следующую функцию, если объект
    // нуждается в особой активации.
    OBJECTREF newobj = CRemotingServices::CreateProxyOrObject(pMT);
    return(OBJECTREFToObject(newobj));
  }

  return JIT_NewFast(typeHnd_);
}
```

Название метода RequiresManagedActivation нужно понимать буквально, то есть как необходимость активации управляемой средой, так как в отличие от других вариантов JIT_New*, JIT_NewCrossContext для создания MBR-объекта осуществляет переход в управляемый код.

Функция RequiresManagedActivation как раз и принимает решение о необходимости управляемой активации:

```
// remoting.cpp

ManagedActivationType
CRemotingServices::RequiresManagedActivation(EEClass* pClass)
{
  if (!pClass->IsMarshaledByRef())
    return NoManagedActivation;

  ManagedActivationType bManaged = NoManagedActivation;

  if (pClass->IsConfigChecked())
  {
    // IT: Возможно тип уже сконфигурирован.

    // We have done work to figure this out in the past ...
    // use the cached result
    bManaged = pClass->IsRemoteActivated()?
            ManagedActivation : NoManagedActivation;
  }
  else if (pClass->IsContextful() || pClass->HasRemotingProxyAttribute())
  {
    // Contextful and classes that have a remoting proxy attribute
    // (whether they are MarshalByRef or ContextFul) always take
    // the slow path of managed activation
    bManaged = ManagedActivation;

    // IT: Т.е. если мы поставили классу Proxy-атрибут,
    // о котором речь пойдёт ниже, то managed-активация
    // будет вызываться всегда.
  }
  else
  {
    // If we have parsed a config file that might have configured
    // this Type to be activated remotely
    if (GetAppDomain()->IsRemotingConfigured())
    {
      bManaged = ManagedActivation;
    }
    // IT: Это относится к конфигурированию Remoting через файл.
```

```
    }

    return bManaged;
  }
```

Первый "if" содержит вызов метода IsConfigChecked. Эта функция даёт нам знать, был ли уже сконфигурирован данный тип, или это только предстоит сделать. Если объекты данного типа уже создавались, мы просто используем полученный ранее результат.

Давайте рассмотрим подробнее код в последнем "else". Получается, что если мы используем для конфигурации ремотинга config-файл, то первый объект любого MBR-типа будет создаваться через более медленную managed-активацию, даже если он в этом не нуждается. Таким образом, будут создаваться только первые объекты каждого типа, остальные уже пойдут как сконфигурированные и подпадут под первый "if".

Конфигурирование типа на самом деле состоит из пары флажков, которые говорят среде о том, какой тип активации следует вызывать. После того, как тип определён, исполняемая среда заносит результат в эти флажки и использует их в дальнейшем. Всё это здорово, но ведь мы можем создать объект до вызова функции, конфигурирующей систему Remoting, и в этом случае данный тип также будет сконфигурирован ранее. В результате IsConfigChecked() будет всегда возвращать "true" и конфигурирование системы Remoting на такие объекты уже не будет распространяться. Это легко проверить с помощью следующего примера:

```
using System;
using System.Runtime.Remoting;
using TestObject;

namespace Client
{
  class Client
  {
    [STAThread]
    static void Main(string[] args)
    {
      Test test1 = new Test();
      Console.WriteLine(test1.GetAppName());

      RemotingConfiguration.Configure("Client.exe.config");

      Test test2 = new Test();
      Console.WriteLine(test2.GetAppName());
    }
  }
}
```

После выполнения этого теста на экране будет:

```
Client.exe
Client.exe
```

Таким образом, как мы и предположили, регистрация создаваемых ранее объектов с помощью config-файлов не работает. Ну что же, будем внимательнее, поскольку теперь нам известно, что конфигурировать remoting нужно ещё до того, как в программе начнутся какие-либо активные действия.

Но вернёмся в дебри исполняемой среды и рассмотрим следующий метод, отвечающий за создание и активацию MBR-объектов.

```
// remoting.cpp

OBJECTREF CRemotingServices::CreateProxyOrObject(MethodTable* pMT,BOOL fIsCom)
{
  // Get the address of IsCurrentContextOK in managed code
  void* pTarget =
```

```
    (void*)CRemotingServices::MDofIsCurrentContextOK()->GetMethodEntryPoint();

  EEClass* pClass = pMT->GetClass();

  // Get the type seen by reflection
  LPVOID pvType = pClass->GetExposedClassObject();

  // This will return either an uninitialized object or a proxy
  Object *pServer =
    (Object*)CTPMethodTable::CallTarget(pTarget,pvType,NULL);

  return ObjectToOBJECTREF(pServer);
}
```

Эта процедура всего лишь получает указатель на метод IsCurrentContextOK в managed-коде и передаёт туда управление. Дальше начинает работать managed-код. Ниже приведены некоторые выдержки из него. Метод ActivationService.IsCurrentContextOK является точкой входа в управляемый код. Название этого метода выбрано, скорее всего, не очень удачно. Гораздо лучше, думаю, было бы придумать что-нибудь наподобие предыдущего CreateProxyOrObject.

```
// activationservices.cs

namespace System.Runtime.Remoting.Activation
{
  internal sealed class ActivationServices
  {
    private static MarshalByRefObject IsCurrentContextOK(
        Type serverType,object[] props)
    {
      MarshalByRefObject retObj = null;

      // Obtain the method info which will create an instance
      // of type RealProxy
      ProxyAttribute pa = GetProxyAttribute(serverType);

      if (object.ReferenceEquals(pa, DefaultProxyAttribute))
        retObj = pa.CreateInstanceInternal(serverType);
      else
        retObj = pa.CreateInstance(serverType);

      return retObj;
    }

    private static
    ProxyAttribute DefaultProxyAttribute = new ProxyAttribute();

    internal static ProxyAttribute GetProxyAttribute(Type serverType)
    {
      if (!serverType.HasProxyAttribute)
        return DefaultProxyAttribute;

      object[] ca = Attribute.GetCustomAttributes(
              serverType,
              System.Runtime.Remoting.Proxies.ProxyAttribute,
              true);

      return ca[0] as ProxyAttribute;
    }
  }
}
```

Здесь производится проверка наличия атрибута ProxyAttribute у класса. Если такой атрибут имеется, то вызывается его виртуальный метод CreateInstance. В противном случае вызывается реализация этого метода по умолчанию, которая, в конце концов, приводит к вызову следующего метода:

```
namespace System.Runtime.Remoting.Activation
{
  internal sealed class ActivationServices
  {
    // Creates either an uninitialized object or a proxy depending
    // on whether the current context is OK or not.
    internal static MarshalByRefObject CreateInstance(Type serverType)
    {
      // Здесь вы найдёте много интересного.
    }
  }
}
```

В зависимости от того, нуждается ли объект в удалённой активации, этот метод создаёт real proxy или обычный объект в куче. Загляните на досуге в реализацию этого метода, уверяю вас, вы найдёте там много интересного.

В общем-то, в этом и заключается чудо. Фактически решение о возможности управляемой активации принимается ещё JIT на этапе компиляции. Если наш объект является MBR-объектом, компилятор вставляет в код вызов метода, который при соблюдении некоторых условий вызывает активацию объекта из управляемого кода.

Перехват создания объекта

Теперь можно заняться рассмотрением механизма перехвата создания объекта. Но сначала давайте подробнее рассмотрим ряд моментов, обнаруженных нами в процессе исследования исходных текстов CLI. Прежде всего, это контекст объекта и proxy-атрибут.

Контекст объекта представляет собой окружение, в котором исполняется данный объект. Контекст создаётся в процессе активизации объекта, для работы которого необходимы те или иные сервисы, такие как синхронизация, поддержка распределённых транзакций, безопасность, журналирование и т.п. Проще всего понять природу и принцип работы контекстов на примере. Наш пример будет записывать в журнал исключения, возникающие в процессе работы программы.

```
using System;
using System.Runtime.Remoting.Activation;
using System.Runtime.Remoting.Contexts;
using System.Threading;

namespace Client
{
  [AttributeUsage(AttributeTargets.Class)]
  public class ExceptionLogAttribute : ContextAttribute
  {
    private string fileName;

    public ExceptionLogAttribute(string fName)
      : base("ExceptionLog")
    {
      fileName = fName;
    }

    public override bool IsContextOK(
      Context ctx,
      IConstructionCallMessage msg)
    {
      ExceptionLogContextProperty prop =
        ctx.GetProperty(base.AttributeName) as
          ExceptionLogContextProperty;

      return prop != null && prop.FileName == fileName;
    }

    public override void GetPropertiesForNewContext(
```

```
      IConstructionCallMessage msg)
    {
      msg.ContextProperties.Add(
        new ExceptionLogContextProperty(fileName));
    }
  }

  public class ExceptionLogContextProperty : IContextProperty
  {
    public string FileName;

    public ExceptionLogContextProperty(string fName)
    {
      FileName = fName;
    }

    public string Name
    {
      get { return "ExceptionLog"; }
    }

    public void Freeze(Context ctx)
    {
    }

    public bool IsNewContextOK(Context ctx)
    {
      return true;
    }
  }

  [ExceptionLog("test.log")]
  public class Test : ContextBoundObject
  {
    public void TestException()
    {
      try
      {
        throw new Exception("test exception");
      }
      catch (Exception ex)
      {
        ExceptionLogContextProperty prop =
          Thread.CurrentContext.GetProperty("ExceptionLog") as
            ExceptionLogContextProperty;

        System.Console.WriteLine(
          "{0} << '{1}'",prop.FileName,ex.Message);
      }
    }
  }

  class Client
  {
    [STAThread]
    static void Main(string[] args)
    {
      Test test = new Test();
      test.TestException();
    }
  }
}
```

Каждый раз, когда исполняемая среда создаёт объект, требующий управляемой активации, она проверяет наличие у этого объекта атрибутов контекста. Для каждого такого атрибута вызывается метод *IsContextOK*, которому в качестве одного из параметров передаётся текущий контекст. Если хотя бы для одного из атрибутов данный метод возвращает *false*, для объекта создаётся новый контекст. В процессе создания нового контекста CLR вызывает метод

GetPropertiesForNewContext для атрибутов контекста данного объекта, которые могут добавить в новый контекст свойства по своему усмотрению.

В нашем примере метод *IsContextOK* проверяет наличие у заданного контекста свойства *ExceptionLogContextProperty* и сравнивает содержимое его переменной *FileName* со имеющейся у него копией. Таким образом, если в контексте нет данного атрибута, или его свойство отличается, то для создаваемого объекта будет создан новый контекст. Далее в процессе создания контекста будет вызван метод *GetPropertiesForNewContext* нашего атрибута, который добавит в контекст свойство, содержащее необходимый параметр.

Но, тем не менее, мы ещё не достигли цели, да и продемонстрированная возможность не очень-то впечатляет, т.к. вся работа по перехвату исключений и формированию лога всё равно возложена на само приложение.

Мы уже кое-что знаем о *ProxyAttribute*, в частности, нам известно, что CLR вызывает его метод *CreateInstance* для создания экземпляра объекта. Теперь самое время попробовать этот атрибут в действии.

```
using System;
using System.Runtime.Remoting;
using System.Runtime.Remoting.Activation;
using System.Runtime.Remoting.Proxies;
using System.Runtime.Remoting.Messaging;
using System.Runtime.Remoting.Services;
using System.Threading;

namespace Client
{
  [AttributeUsage(AttributeTargets.Class)]
  public class ExceptionLogAttribute : Attribute
  {
    public string FileName;

    public ExceptionLogAttribute(string fName)
    {
      FileName = fName;
    }
  }

  class TestProxy : RealProxy
  {
    MarshalByRefObject obj = null;

    public TestProxy(Type type) : base(type)
    {
    }

    public override IMessage Invoke(IMessage msg)
    {
      if (msg is IConstructionCallMessage)
      {
        if (obj == null) // 2
        {
          TestProxy tp = new TestProxy(GetProxiedType());
          tp.InitializeServerObject(null);
          obj = tp.GetUnwrappedServer();
        }

        return EnterpriseServicesHelper.CreateConstructionReturnMessage(
          (IConstructionCallMessage)msg,
          (MarshalByRefObject)GetTransparentProxy());
      }
      else
      {
        IMethodCallMessage  mcm = (IMethodCallMessage) msg;
        IMethodReturnMessage rm = RemotingServices.ExecuteMessage(obj,mcm);

        if (rm.Exception != null)
```

```
            {
              ExceptionLogAttribute log =
                Attribute.GetCustomAttribute(
                  GetProxiedType(),typeof(ExceptionLogAttribute)) as
                    ExceptionLogAttribute;

              if (log != null)
              {
                System.Console.WriteLine(
                  "{0} << '{1}'",log.FileName,rm.Exception.Message);
              }
            }

            return rm;
          }
        }
      }

      [AttributeUsage(AttributeTargets.Class)]
      public class TestProxyAttribute: ProxyAttribute
      {
        public override MarshalByRefObject CreateInstance(Type type)
        {
          TestProxy rp = new TestProxy(type);
          return (MarshalByRefObject)rp.GetTransparentProxy(); // 1
        }
      }

      [TestProxy]
      [ExceptionLog("test.log")]
      public class Test : ContextBoundObject
      {
        public void TestException()
        {
          throw new Exception("test exception");
        }
      }

      class Client
      {
        [STAThread]
        static void Main(string[] args)
        {
          try
          {
            Test test = new Test();
            test.TestException();
          }
          catch
          {
          }
        }
      }
    }
```

Здесь мы снова видим уже знакомый класс *RealProxy*, но, в отличие от предыдущих примеров, мы не используем его явно. Вместо этого созданием экземпляра этого класса занимается *ProxyAttribute*, метод *CreateInstance* которого вызывается исполняемой средой при наличии такого атрибута у объекта.

Обратите также внимание на то, что метод *Invoke* нашего proxy пополнился дополнительной обработкой сообщения, инициализирующего экземпляр класса. Помните, мы выяснили, что создание объекта в CLR производится в два этапа: создание сырого объекта и вызов конструктора?

В нашем случае первый этап ограничивается созданием прокси (1), а сам объект создаётся позже при вызове конструктора (2). В принципе, этот порядок не так важен, главное – что для клиентской программы это совершенно прозрачно.

Конечно, данный пример является лишь демонстрацией мощи, заложенной во внутренние механизмы работы .NET Framework. Здесь лишь продемонстрированы способы перехвата управления при вызове объектов и некоторые методы применения атрибутов.

Надо отметить, что механизмы перехвата не являются чем-то новым в среде .NET, они давно используются в COM. Да и атрибуты как средство контроля над поведением объектов, с успехом применяются в COM+. Но сегодня эти средства стали доступны конечным разработчикам, т.е. нам. Чем мы с вами, конечно же, не замедлим воспользоваться. :o)

Эта статья опубликована в журнале RSDN Magazine #1-2003. Информацию о журнале можно найти здесь