ДОПОЛНИТЕЛЬНАЯ ГЛАВА 1

Слой удаленного взаимодействия .NET Remoting

Разработчики, являющиеся новичками в .NET, часто предполагают, что платформа .NET предназначена для разработки Интернет-приложений (учитывая, что ".NET" часто напоминает им "interNET"). Однако на самом деле это не так. Фактически конструирование веб-ориентированных программ — всего лишь один крошечный (но навязчиво рекламируемый) аспект платформы .NET. В той же мере многие разработчики заблуждаются, думая, что веб-службы XML — единственный способ взаимодействия с удаленными объектами. Опять-таки, на самом деле это неправда. Используя слой .NET Remoting, можно строить одноранговые распределенные приложения, не имеющие никакого отношения к HTTP или XML.

Первая цель этой главы — исследовать низкоуровневые механизмы, используемые CLR для передачи информации через границы приложений. По пути вы познакомитесь с множеством терминов, применяемых в обсуждении .NET Remoting, такими как прокси, каналы, маршализация по ссылке (в противоположность по значению), активизируемые сервером (в противоположность активизируемым клиентом) объекты, и т.д. После раскрытия темы этих базовых элементов в оставшейся части главы предлагаются многочисленные примеры кода, иллюстрирующие процесс построения распределенных систем на платформе .NET.

Определение .NET Remoting

Как вы должны помнить, домен приложения (AppDomain) — это логическая область сборки .NET, которая сама существует внутри процесса Win32. Понимание этой концепции критично при обсуждении распределенных вычислений под .NET, учитывая, что *Remoting* (удаленное взаимодействие) — это не что иное, как акт взаимодействия двух объектов между доменами приложений. Эти два домена приложений могут быть сконфигурированы физически в любой из следующих форм:

- два домена приложений в одном процессе (и потому на одной машине);
- два домена приложений в отдельных процессах на одной машине;
- два домена приложений в отдельных процессах на разных машинах.

Исходя их этих трех возможностей, вы можете видеть, что Remoting не обязательно подразумевает участие двух сетевых компьютеров. Фактически, каждый из примеров, представленных в этой главе, может с успехом выполняться на одной отдельной машине. Независимо от расстояния между двумя объектами, принято называть их "клиентом" и "сервером". Просто говоря, *клиент* — это сущность, которая пытается взаимодействовать с удаленными объектами. *Сервер* — программный агент, размещающий в себе удаленные объекты.

Пространства имен .NET Remoting

Прежде чем погрузиться глубоко в детали устройства слоя .NET Remoting, следует ознакомиться с функциональностью, предлагаемой соответствующими пространствами имен. Библиотеки базовых классов .NET предоставляют многочисленные пространства имен, которые позволяют строить распределенные приложения. Большая часть этих типов находится внутри пространств имен, собранных в сборке `mscorlib.dll`, а сборка `System.Runtime.Remoting.dll` дополняет и расширяет пространства имен ядра. В табл. 1.1 кратко описаны роли пространств имен .NET 2.0, связанных с организацией удаленного взаимодействия.

Таблица 1.1. Пространства имен .NET Remoting

Пространство имен	Назначение
`System.Runtime.Remoting`	Это основное пространство имен, которое должно использоваться при построении распределенных приложений .NET любого рода.
`System.Runtime.Remoting.Activating`	Это относительно маленькое пространство имен определяет несколько типов, позволяющих тонко настраивать процесс активизации удаленного объекта.
`System.Runtime.Remoting.Channels`	Это пространство имен содержит типы, представляющие каналы и их группы.
`System.Runtime.Remoting.Channels.Http`	Это пространство имен содержит типы, использующие протокол HTTP для передачи сообщений и объектов между удаленными местоположениями.
`System.Runtime.Remoting.Channels.Ipc`	Это пространство имен (новое в .NET 2.0) содержит типы, использующие архитектуру межпроцессных коммуникаций Win32 (IPC) между доменами приложений на одной и той же физической машине.
`System.Runtime.Remoting.Channels.Tcp`	Это пространство имен содержит типы, использующие протокол TCP для передачи сообщений и объектов между удаленными местоположениями.
`System.Runtime.Remoting.Context`	Это пространство имен позволяет конфигурировать детали контекста объектов.
`System.Runtime.Remoting.Lifetime`	Это пространство имен содержит типы для управления жизненным циклом удаленных объектов.

Окончание табл. 1.1

Пространство имен	Назначение
`System.Runtime.Remoting.Messaging`	Это пространство имен содержит типы, используемые для создания и передачи объектов сообщений.
`System.Runtime.Remoting.Metadata`	Это пространство имен содержит типы, используемые для настройки генерации и обработки форматирования SOAP.
`System.Runtime.Remoting.Metadata.W3cXsd2001`	Тесно связанное с предыдущим, это пространство имен содержит типы, представляющие XML Schema Definitions (XSD), определенные консорциумом World Wide Web Consortium (W3C) в 2001 г.
`System.Runtime.Remoting.MetadataServices`	Это пространство имен содержит типы, используемые инструментом командной строки `soapsuds.exe` для преобразования метаданных .NET в схему XML и обратно для инфраструктуры удаленного взаимодействия.
`System.Runtime.Remoting.Proxies`	Это пространство имен содержит типы, представляющие функциональность прокси-объектов.
`System.Runtime.Remoting.Services`	Это пространство имен определяет множество общих базовых классов (и интерфейсов), которые обычно используются только другими встроенными удаленными агентами.

Каркас .NET Remoting

Когда клиенты и серверы передают информацию между границами приложений, CLR использует несколько низкоуровневых примитивов для обеспечения возможности взаимодействия сущностей между собой как можно более прозрачным образом. Это значит, что как программист .NET вы не обязаны писать большие объемы низкоуровневого сетевого кода для вызова метода удаленного объекта. Аналогично, серверный процесс *не* обязан вручную заталкивать сетевой пакет в очередь и переформатировать сообщения в вид, понятный удаленному объекту. Как и можно было надеяться, CLR заботится обо всех деталях автоматически, используя стандартный набор примитивов удаленного взаимодействия (хотя при желании всегда можно вмешаться в процесс).

По сути, слой .NET Remoting занимается тщательной организацией совместной работы следующих четырех ключевых игроков:

- прокси;
- сообщения;
- каналы;
- форматировщики.

Давайте исследуем каждую из этих сущностей по очереди, и посмотрим, как их комбинированная функциональность облегчает вызовы удаленных методов.

Прокси и сообщения

Клиентские и серверные объекты не взаимодействуют по прямым соединениям, а вместо этого используют посредника, именуемого *прокси*. Роль объектов прокси .NET заключается в том, чтобы заставить клиента поверить, что он взаимодействует непосредственно с удаленным объектом в *том же домене приложений*. Чтобы облегчить создание этой иллюзии, прокси имеет идентичный с удаленным типом интерфейс (т.е. методы, свойства, поля и т.п.). С точки зрения клиента, конкретный прокси-объект и является удаленным объектом. "За кулисами", однако, прокси передает все вызовы удаленному объекту.

Формально говоря, прокси, вызываемый непосредственно клиентом, называется *прозрачным прокси*. Эта автоматически генерируемая CLR сущность отвечает за предоставление клиентом правильного количества и типов параметров при вызове удаленного метода. С учетом этого, прозрачный прокси можно рассматривать как фиксированный промежуточный слой, который *не может* быть модифицирован или расширен программно.

Исходя из того, что прозрачный прокси может верифицировать входящие аргументы, эта информация пакуется в другой сгенерированный CLR тип, именуемый *объектом сообщения*. По определению все объекты сообщений реализуют интерфейс `System.Runtime.Remoting.Messaging.IMessage`:

```
public interface IMessage
{
  IDictionary Properties { get; }
}
```

Как видите, интерфейс `IMessage` определяет единственное свойство по имени `Properties`, которое предоставляет доступ к коллекции, используемой для хранения переданных клиентом аргументов. Как только объект сообщения наполнен CLR, он передается тесно связанному типу, именуемому *реальным прокси*.

Реальный прокси — это сущность, которая действительно передает объект сообщения в канал (описанный далее). В отличие от прозрачного, реальный прокси *может* быть расширен программистом и представлен типом базового класса по имени `RealProxy`. Опять-таки, стоит указать, что CLR всегда будет генерировать реализацию по умолчанию реального прокси клиентской стороны, который будет обслуживать ваши нужды большую часть (если не всю) времени. Как бы то ни было, чтобы получить некоторое представление о функциональности абстрактного базового класса `RealProxy`, взгляните на формальное определение типа:

```
public abstract class RealProxy : object
{
  public virtual ObjRef CreateObjRef(Type requestedType);
  public virtual bool Equals(object obj);
  public virtual IntPtr GetCOMIUnknown(bool fIsMarshalled);
  public virtual int GetHashCode();
  public virtual void GetObjectData(SerializationInfo info,
      StreamingContext context);
  public Type GetProxiedType();
  public static object GetStubData(RealProxy rp);
  public virtual object GetTransparentProxy();
  public Type GetType();
  public IConstructionReturnMessage InitializeServerObject(
      IConstructionCallMessage ctorMsg);
  public virtual IMessage Invoke(IMessage msg);
  public virtual void SetCOMIUnknown(IntPtr i);
  public static void SetStubData(RealProxy rp, object stubData);
```

```
    public virtual IntPtr SupportsInterface(ref Guid iid);
    public virtual string ToString();
  }
```

Если только вы не заинтересованы в построении специальной реализации реального прокси клиентской стороны, единственный член, представляющий интерес — это RealProxy.Invoke(). "За кулисами" сгенерированный CLR прозрачный прокси передает объект форматированного сообщения в тип RealProxy через метод Invoke().

Каналы

Как только прокси проверил и сформатировал аргументы клиентской стороны в объект сообщения, этот IMessage-совместимый тип передается от реального прокси объекту канала. *Каналы* — это сущности, отвечающие за транспортировку сообщения удаленному объекту и, если необходимо, гарантирующие, что любое возвращаемое значение члена будет передано от удаленного объекта обратно клиенту. Библиотеки базовых классов .NET 2.0 предлагают три готовые реализации каналов:

- канал TCP;
- канал HTTP;
- канал IPC.

Канал TCP представлен классом TcpChannel и служит для передачи сообщений с использованием сетевого протокола TCP/IP. TcpChannel полезен тем, что сформатированные пакеты относительно легковесны, учитывая, что сообщения преобразуются в компактный двоичный формат с применением связанного BinaryFormatter. Использование TcpChannel имеет тенденцию ускорять удаленный доступ. Отрицательной стороной каналов TCP является то, что они блокируются брандмауэрами и могут потребовать обращения к системному администратору, чтобы он специально разрешил передачу сообщений между машинами.

В отличие от этого, *каналы HTTP*, представленные типом HttpChannel, преобразуют объекты сообщений в формат SOAP, используя соответствующий форматировщик SOAP. Как было показано, SOAP базируется на XML, и потому имеет тенденцию создавать большую нагрузку, чем нагрузка от типа TcpChannel. Учитывая это, применение HttpChannel может несколько замедлить удаленный доступ. Однако положительная сторона заключается в том, что HTTP гораздо меньше страдает от преград брандмауэров, поскольку передает текстовые пакеты через порт 80.

И, наконец, начиная с .NET 2.0, в нашем распоряжении есть *канал IPC*, представленный типом IpcChannel, который определяет коммуникационный канал для удаленного взаимодействия с использованием системы IPC операционной системы Windows. Поскольку IpcChannel минует традиционные сетевые коммуникации для преодоления границ доменов приложений, IpcChannel намного быстрее, чем каналы HTTP и TCP; однако он может применяться только для взаимодействия между доменами приложений в пределах *одного и того же физического компьютера*. С учетом этого, вы никогда не сможете использовать IpcChannel для построения распределенных приложений, охватывающих несколько физических компьютеров. Тем не менее, IpcChannel может быть идеальным выбором, когда нужно, чтобы две локальные программы разделяли между собой информацию самым быстрым их возможных способов.

Независимо от того, какой тип канала выбран, знайте, что типы HttpChannel, TcpChannel и IpcChannel реализуют интерфейсы IChannel, IChannelSender и IChannelReceiver. Интерфейс IChannel (как будет показано ниже) определяет небольшой набор членов, предоставляющих функциональность всех типов каналов.

Роль IChannelSender заключается в определении общего множества членов для каналов, способных передавать информацию определенному приемнику. С другой сторо-

ны, IChannelReceiver определяет множество членов, позволяющих каналу принимать информацию от заданного отправителя.

Чтобы позволить клиентскому и серверному приложениям регистрировать выбранный ими канал, применяется метод ChannelServices.RegisterChannel(), который принимает тип, реализующий IChannel. Чтобы предварить то, что последует дальше, показанный ниже фрагмент кода иллюстрирует то, как домен приложения серверной стороны может регистрировать канал HTTP на порте 32469 (роль клиента вы увидите далее):

```
// Создать и зарегистрировать HttpChannel серверной стороны на порте 32469.
HttpChannel c = new HttpChannel(32469);
ChannelServices.RegisterChannel(c);
```

Еще раз о роли форматировщиков .NET

Последний фрагмент мозаики .NET Remoting — роль объектов *форматировщиков*. Типы TcpChannel и HttpChannel используют внутренний форматировщик, задача которого состоит в трансляции объекта сообщения в специфичный для протокола вид. Как уже упоминалось, тип TcpChannel использует тип BinaryFormatter, в то время как HttpChannel пользуется функциональностью, предоставленной типом SoapFormatter. Вы уже должны иметь некоторое представление о том, как данный канал будет форматировать входящие сообщения.

Как только сформатированное сообщение сгенерировано, оно передается в канал, через который в конечном итоге достигает целевого домена приложений, где это сообщение форматируется из специфичного для протокола вида обратно в вид, специфичный для .NET, и в этот момент сущность по имени *диспетчер* вызывает корректный метод удаленного объекта.

А теперь — все вместе!

Если у вас закружилась голова от чтения предыдущего раздела, не пугайтесь! Прозрачный прокси, реальный прокси, объект сообщения и диспетчер обычно можно полностью игнорировать, если вас устраивает механизм удаленного взаимодействия по умолчанию. Чтобы закрепить последовательность событий, взгляните на рис. 1.1, на котором иллюстрируется базовый процесс взаимодействия двух объектов из двух разных доменов приложений.

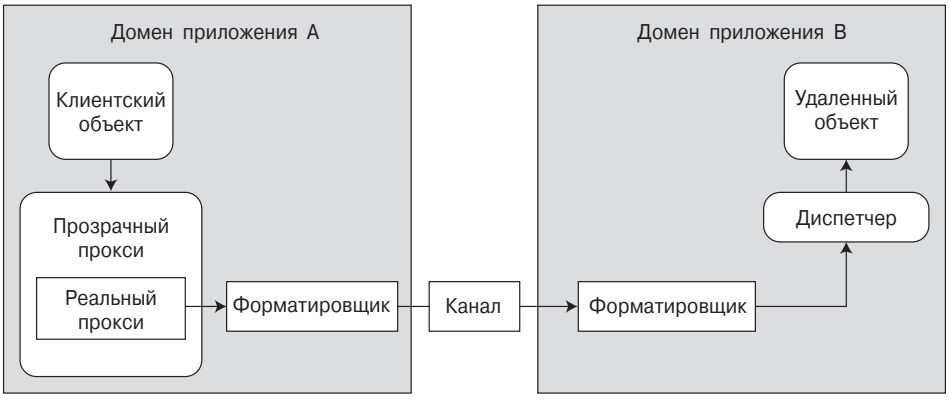

Рис. 1.1. Высокоуровневое представление архитектуры .NET Remoting по умолчанию

Несколько слов о расширении механизмов по умолчанию

Ключевой аспект слоя .NET Remoting состоит в том, что большинство слоев удаленного взаимодействия могут быть расширены или полностью заменены по желанию разработчика. Таким образом, если вы действительно хотите (или, возможно, нуждаетесь) в построении специального диспетчера сообщений, специального форматировщика или специального реального прокси, то можете это сделать. Вы можете также внедрить *дополнительные* уровни посредничества, подключая специальные типы между существующими слоями (например, для выполнения предобработки и постобработки заданного сообщения). Может быть, никогда и не понадобится вмешиваться в работу слоя .NET Remoting подобным образом. Однако имейте в виду, что платформа .NET предоставляет пространства имен типов, которые позволяют это сделать.

На заметку! В этой главе тема расширения стандартного слоя .NET Remoting не рассматривается. Если хотите узнать, как это делается, обратитесь к изданию *Advanced .NET Remoting* (Apress, 2002 г.).

Терминология .NET Remoting

Как любая новая парадигма, .NET Remoting привнесла с собой ряд трехбуквенных аббревиатур. Таким образом, прежде чем вы увидите первый пример кода, нужно определить несколько терминов, используемых при описании композиции приложений .NET Remoting. Как и можно было предположить, эта терминология используется для описания деталей, касающихся общих вопросов конструирования распределенных приложений. Как передать тип через границы домена приложения? Когда именно активизируется удаленный тип? Как управлять жизненным циклом удаленного объекта (и т.д.)? После усвоения соответствующей терминологии акт построения распределенного приложения .NET станет намного проще.

Выбор маршализации объекта: MBR или MBV?

На платформе .NET на выбор доступны два варианта способов маршализации удаленного объекта к клиенту. Просто говоря, *маршализация* (marshaling) описывает передачу удаленного объекта между доменами приложений. При проектировании удаленного объекта можно выбирать между семантикой *маршализации по ссылке* (marshal-by-reference — MBR) и *маршализации по значению* (marshal-by-value — MBV). Разница состоит в следующем:

- *объекты MBR* — вызывающий код получает прокси к удаленному объекту;
- *объекты MBV* — вызывающий код получает копию объекта в своем собственном домене приложения.

При конфигурировании типа объекта MBR среда CLR обеспечивает создание прозрачного и реального прокси в домене приложения клиента, в то время как сам объект MBR остается в домене приложения сервера. Когда клиент вызывает методы удаленного типа, механизм .NET Remoting (описанный ранее) берет дело в свои руки и пакует, передает и возвращает информацию через границы доменов приложений. Объекты MBR имеют множество особенностей, помимо их физического расположения. Как вы увидите, объекты MBR имеют различные опции конфигурации, касающиеся опций активизации и управления жизненным циклом.

Объекты MBV, с другой стороны, являются *локальными копиями* удаленных объектов (использующими протокол сериализации, рассмотренный нами в главе 17). Объекты

MBV имеют намного меньше установок конфигурации, учитывая, что их время жизни напрямую контролируется клиентом. Подобно любому объекту .NET, как только клиент освобождает все ссылки на тип MBV, он становится кандидатом на сборку мусора. Учитывая, что типы MBV являются локальными копиями удаленных объектов, и клиент вызывает члены типа у себя, никакой сетевой активности это уже не требует.

Теперь знайте, что довольно часто принято, что один сервер представляет доступ к множеству типов MBR и MBV. И как можно было бы ожидать, типы MBR имеют тенденцию поддерживать методы, возвращающие различные типы MBV, что дает возможность узнать здесь шаблон фабрики (т.е. объект, который создает и возвращает другие связанные объекты). Следующий вопрос состоит в том, каким образом конфигурировать пользовательские типы классов как сущности MBR или MBV?

Конфигурирование объекта MBV

Процесс конфигурирования объекта как типа MBV идентичен процессу конфигурирования объекта для сериализации. Просто аннотируйте тип атрибутом `[Serializable]`:

```
[Serializable]
public class SportsCar
{...}
```

Конфигурирование объекта MBR

Объекты MBR не помечаются атрибутом .NET, а вместо этого наследуются (напрямую или опосредованно) от базового класса `System.MarshalByRefObject`:

```
public class SportsCarFactory : MarshalByRefObject
{...}
```

Формально тип `MarshalByRefObject` определен так:

```
public abstract class MarshalByRefObject : object
{
  public virtual ObjRef CreateObjRef(Type requestedType);
  public virtual bool Equals(object obj);
  public virtual int GetHashCode();
  public virtual object GetLifetimeService();
  public Type GetType();
  public virtual object InitializeLifetimeService();
  public virtual string ToString();
}
```

Помимо ожидаемой функциональности, предоставленной `System.Object`, в табл. 1.2 описаны роли остальных членов.

Таблица 1.2. Ключевые члены `System.MarshalByRefObject`

Член	Назначение
`CreateObjRef()`	Создает объект, который содержит всю информацию, необходимую для генерации прокси, используемого для взаимодействия с удаленным объектом.
`GetLifetimeServices()`	Извлекает текущие объекты службы жизненного цикла, которые управляют политикой жизненного цикла данного экземпляра.
`InitializeLifetimeServices()`	Получает служебный объект жизненного цикла для управления политикой жизненного цикла данного экземпляра.

Как вы поняли, суть `MarshalByRefObject` состоит в определении членов, которые могут быть переопределены для программного контроля жизненного цикла объекта MBR (подробнее об управлении жизненным циклом рассказывается далее в главе).

На заметку! То, что вы конфигурируете объект как MBV или MBR, не означает, что он будет пригоден для использования только в распределенном приложении. Он просто *может* быть использован в распределенном приложении. Например, тип `System.Windows.Forms.Form` также является наследником `MarshalByRefObject`; поэтому при удаленном доступе он реализуется как тип MBR. Если нет — это просто еще один локальный объект в домене приложения клиента.

На заметку! В качестве вывода из предыдущего замечания: если тип .NET не является сериализуемым и не включает `MarshalByRefObject` в своей цепочке наследования, то такой тип может быть активизирован и использован в исходном домене приложения (т.е. тип *привязан к контексту*).

Теперь, когда вы понимаете разницу между типами MBR и MBV, давайте рассмотрим некоторые проблемы, специфичные для типов MBR (типы MBV в обсуждении не нуждаются).

Выбор активизации типов MBR: WKO или CAO?

Еще один выбор, с которым вы сталкиваетесь как программист .NET, связан с тем, когда следует активизировать объект MBR, и когда он должен стать кандидатом на сборку мусора на сервере. Это может показаться странным, поскольку вполне естественно предположить, что объекты MBR создаются, когда клиент запрашивает их, и исчезают, когда клиент завершает работу с ними. Хотя и верно, что клиент полностью отвечает за инструктирование слоя удаленного взаимодействия о том, как он желает работать с удаленным типом, серверный домен приложений может (или нет) создавать тип именно в тот момент, когда код клиента запрашивает его.

Причина этого на первый взгляд странного поведения состоит в оптимизации. В частности, каждый тип MBR может быть сконфигурирован для активизации с использованием одного из двух приемов:

- как хорошо известный объект (well-known object — WKO);
- как объект, активизируемый клиентом (client-activated object — CAO).

На заметку! Потенциальной причиной путаницы может быть тот факт, что аббревиатура WKO в литературе .NET имеет синоним *активизируемого сервером объекта* (server-activated object — SAO). Действительно, аббревиатуру SAO можно встретить в различных статьях и книгах на тему .NET. Однако ради последовательности, в этой главе будет применяться понятие WKO.

Объекты WKO — это типы MBR, жизненным циклом которых управляет непосредственно домен серверного приложения. Приложение клиентской стороны активизирует удаленный тип, используя дружественное, хорошо известное строковое имя. Домен серверного приложения размещает в памяти объекты WKO, когда клиент осуществляет первый вызов метода на объекте (через прозрачный прокси), а *не* когда код клиентской стороны использует ключевое слово `new`, или когда вызывает статический метод `Activator.GetObject()`. Например:

```
// Получить прокси на удаленный объект. Эта строка не создает экземпляр типа WKO!
object remoteObj = Activator.GetObject( /* параметры будут показаны позже... */
);
```

```
// Вызвать метод удаленного типа WKO. Это создаст объект WKO
// и вызовет его метод ReturnMessage().
RemoteMessageObject simple = (RemoteMessageObject)remoteObj;
Console.WriteLine("Server says: {0}", simple.ReturnMessage());
```

Каково логическое обоснование этого поведения? Такой подход экономит сетевой трафик, исключая обмен, связанный исключительно с созданием нового объекта. Еще одно интересное следствие, вытекающее отсюда: типы WKO *могут создаваться только конструкторами по умолчанию.* Это должно иметь смысл, учитывая, что конструктор удаленного типа вызывается, только когда клиент осуществляет начальный вызов члена. Таким образом, исполняющая система не имеет другого выбора, кроме как вызвать конструктор по умолчанию.

На заметку! Всегда помните: все типы WKO должны поддерживать конструктор по умолчанию!

Если вы хотите позволить клиенту создавать объект MBR с применением специального конструктора, сервер должен конфигурировать объект как CAO. Объекты CAO — это сущности, жизненный цикл которых управляется доменом клиентского приложения. При обращении к типу CAO обмен с сервером происходит в момент использования клиентом ключевого слова `new` (с вызовом любого конструктора типа) или при создании экземпляра посредством типа `Activator`.

Конфигурация типов WKO: одиночный экземпляр или одиночный вызов?

Последний выбор дизайна, связанный с типами MBR состоит в том, каким образом сервер должен обрабатывать множественные запросы к типу WKO. Типов CAO это не касается, учитывая, что там всегда существует отношение “один к одному” между клиентом и удаленным типом CAO (поскольку они являются объектами с состоянием).

Первый вариант состоит в конфигурировании типа WKO для работы в режиме одиночного экземпляра (singleton). CLR создаст единственный экземпляр удаленного типа, который принимает запросы от любого числа клиентов, и это — естественный выбор, если вы хотите поддерживать общую информацию состояния среди всех удаленных клиентов. Учитывая факт, что множество клиентов могут вызывать один и тот же метод в одно и то же время, CLR помещает каждый вызов клиента в новый поток выполнения. Однако на вас возлагается ответственность за обеспечение безопасности к потокам объектов.

В отличие от этого, модель одиночного вызова предполагает, что экземпляр типа существует только на протяжении вызова метода. Таким образом, если 20 клиентов используют тип WKO, сконфигурированный в семантике одиночного вызова, то сервер создаст 20 отдельных объектов (по одному для каждого клиента), и все они становятся кандидатами на сборку мусора сразу после вызова метода. Как можно догадаться, объекты одиночного вызова намного более масштабируемы, чем типы-одиночки, учитывая, что это сущности, неизбежно лишенные состояния.

Сервер — это сущность, отвечающая за определение конфигурации состояния конкретного типа WKO. Программно эти опции выражаются перечислением `System.Runtime.Remoting.WellKnownObjectMode`:

```
public enum WellKnownObjectMode
{
  SingleCall,
  Singleton
}
```

Особенности типов объектов MBR

Как вы видели, конфигурирование объекта MBV несложно: применяйте атрибут `[Serializable]` для того, чтобы позволить возвращать копии объектов типа в домен клиентского приложения. В этой точке все взаимодействие с типом MBR происходит на стороне клиента. Когда клиент завершает использованием типа MBV, он становится кандидатом на сборку мусора, и все хорошо. Однако с типами MBR есть несколько возможных вариантов выбора конфигурации. Как вы видели, конкретный тип MBR может быть сконфигурирован с учетом времени активизации, сохранения состояния и управления жизненным циклом. Чтобы подытожить весь набор возможностей, в табл. 1.3 документировано отношение типов WKO и CAO к рассмотренным особенностям.

Таблица 1.3. Опции конфигурации типов MBR

Особенность объекта MBR	Поведение WKO	Поведение CAO
Опции создания экземпляров	Типы WKO могут активизироваться только конструкторами по умолчанию, вызываемыми при первом вызове метода клиентом.	Типы CAO могут быть активизированы любым конструктором типа. Удаленные объекты создаются в точке, где клиент использует семантику конструктора (или через тип `Activator`).
Управление состоянием	Типы WKO могут быть сконфигурированы как сущности с одиночным экземпляром или сущности с одиночным вызовом. Типы-одиночки могут обслуживать множество клиентов и потому хранят состояние. Типы одиночного вызова живут только на протяжении конкретного клиентского вызова и потому лишены состояния.	Жизненный цикл типа CAO определяется клиентом; поэтому типы CAO — сущности с состоянием.
Управление жизненным циклом	Типы-одиночки WKO используют схему управления на основе “аренды” (будет описана далее в этой главе). Типы WKO с одиночным вызовом становятся кандидатами на сборку мусора сразу после вызова текущего метода.	Типы CAO используют арендную схему управления (описанную далее в этой главе).

Базовое развертывание проекта .NET Remoting

Достаточно аббревиатур! Теперь вы почти готовы к построению первого приложения .NET Remoting. Однако перед этим понадобится рассмотреть последнюю деталь: развертывание. Когда вы строите распределенное приложение .NET Remoting, то имеете дело и с тремя (именно тремя, а не двумя) отдельными сборками .NET, которые составляют приложение. Наверняка вы уже знакомы с первыми двумя из них.

- *Клиент*. Эта сборка представляет собой сущность, заинтересованную в получении доступа к удаленному объекту (такую как приложение Windows Forms или консольное приложение).
- *Сервер*. Эта сборка представляет собой сущность, принимающая запрос от удаленного клиента и размещающая в себе удаленные объекты.

Так что же собой представляет третья сборка? Во многих случаях серверное приложение обычно размещает в себе третью сборку, которая определяет и реализует удаленные объекты. Для удобства назовем ее *общей сборкой* (general assembly). Это отделение сборки, содержащей удаленные объекты, от сборки хоста сервера довольно важно, поскольку обычно клиентская и серверная сборки устанавливают ссылки на общую сборку, чтобы получить метаданные определений удаленных типов.

В простейшем случае общая сборка размещается в каталоге приложения и клиента, и сервера. Единственный минус такого подхода состоит в том, что клиент имеет ссылку на сборку, включающую код CIL, который никогда не используется (что может представлять проблему, если необходимо ограничить возможность для конечного пользователя видеть патентованный код). Единственная причина, по которой клиент нуждается в ссылке на общую сборку, связана с получением метаданных описаний удаленных типов. Это затруднение можно преодолеть несколькими способами.

- Конструировать удаленные объекты с применением приемов программирования на основе интерфейсов. При этом клиент может устанавливать набор ссылок на двоичный код .NET, не содержащий ничего кроме определений интерфейсов.
- Использовать утилиту командной строки `soapsuds.exe`. Применяя этот инструмент, вы получаете возможность генерировать сборку, не содержащую ничего кроме метаданных, описывающих удаленные типы.
- Вручную построить сборку, не содержащую ничего кроме метаданных, описывающих удаленные типы.

В этой главе, чтобы не усложнять, мы будем строить и развертывать общие сборки, содержащие необходимые метаданные вместе с реализациями CLI.

Построение первого распределенного приложения

Нет ничего более приятного, чем построение распределенного приложения с использованием новой платформы. Чтобы проиллюстрировать, как быстро построить и запустить приложение с использованием слоя .NET Remoting, давайте рассмотрим простой пример. Как уже упоминалось, сущность этого примера состоит из трех сборок:

- общая сборка по имени `SimpleRemotingAsm.dll`;
- клиентская сборка по имени `SimpleRemoteObjectClient.exe`;
- серверная сборка по имени `SimpleRemoteObjectServer.exe`.

Построение общей сборки

Сначала давайте создадим общую сборку `SimpleRemotingAsm.dll`, на которую будут ссылаться как сервер, так и клиент. `SimpleRemotingAsm.dll` определяет единственный тип MBR по имени `RemoteMessageObject`, поддерживающий два общедоступных члена. Метод `DisplayMessage()` напечатает присланное клиентом сообщение в консоли сервера, а `ReturnMessage()` вернет сообщение клиенту. Вот полный код этой новой библиотеки классов C#:

```
namespace SimpleRemotingAsm
{
  // Это тип, который будет маршализироваться
  // по ссылке (MBR) при удаленном обращении.
  public class RemoteMessageObject: MarshalByRefObject
  {
```

```
  public RemoteMessageObject()
  { Console.WriteLine("Constructing RemoteMessageObject!"); }

  // Метод принимает входную строку от клиента.
  public void DisplayMessage(string msg)
  { Console.WriteLine("Message is: {0}", msg);}

  // Этот метод вернет значение клиенту.
  public string ReturnMessage()
  { return "Hello from the server!"; }
  }
}
```

Основной момент, представляющий интерес здесь — это факт наследования от базового класса `System.MarshalByRefObject`, что гарантирует доступность производного класса через прокси клиентской стороны. Также обратите внимание на специальный конструктор по умолчанию, который будет печатать сообщение при создании экземпляра этого типа. Вот и все. Теперь соберите первую сборку `SimpleRemotingAsm.dll`.

Построение серверной сборки

Напомним, что серверные сборки, по сути, содержат в себе удаленные объекты. Создайте консольную программу по имени `SimpleRemoteObjectServer`. Назначение этой сборки будет открывать канал для входящих запросов и регистрировать `RemoteMessageObject` как WKO. Для начала установите ссылки на сборки `System.Runtime.Remoting.dll` и `SimpleRemotingAsm.dll` и модифицируйте `Main()` следующим образом:

```
using System;
using System.Runtime.Remoting;
using System.Runtime.Remoting.Channels;
using System.Runtime.Remoting.Channels.Http;
using SimpleRemotingAsm;
namespace SimpleRemoteObjectServer
{
  class SimpleObjServer
  {
    static void Main(string[] args)
    {
      Console.WriteLine("***** SimpleRemoteObjectServer started! *****");
      Console.WriteLine("Hit enter to end.");

      // Зарегистрировать новый HttpChannel
      HttpChannel c = new HttpChannel(32469);
      ChannelServices.RegisterChannel(c);

      // Зарегистрировать тип WKO type, используя активизацию одиночки.
      RemotingConfiguration.RegisterWellKnownServiceType(
          typeof(SimpleRemotingAsm.RemoteMessageObject),
          "RemoteMsgObj.soap",
          WellKnownObjectMode.Singleton);
          Console.ReadLine();
    }
  }
}
```

`Main()` начинается с создание нового экземпляра типа `HttpChannel`, использующего произвольный идентификатор порта. Этот порт открывается на зарегистрированном канале через статический метод `ChannelServices.RegisterChannel()`. Как только канал зарегистрирован, удаленный сервер автоматически готов обрабатывать входящие сообщения через порт номер 32469.

На заметку! Номер, который вы назначаете порту, обычно выбирается произвольно вами (или системным администратором). Помните, однако, что порты с номерами ниже 1024 зарезервированы для системного использования.

Затем, чтобы зарегистрировать тип `SimpleRemotingAsm.RemoteMessageObject` как WKO, необходимо использовать метод `RemotingConfiguration.RegisterWellKnownServiceType()`. Первый аргумент этого метода несет информацию о типе, который подлежит регистрации. Второй параметр `RegisterWellKnownServiceType()` — просто строка (по вашему усмотрению), которая будет использована для идентификации объекта при коммуникациях через границы доменов приложений. Здесь вы информируете CLR, что этот объект должен быть реализован клиентом с использованием имени `RemoteMsgObj.soap`.

Последний параметр — член перечисления `WellKnownObjectMode`, который специфицируется как `WellKnownObjectMode.Singleton`. Напомним, что тип-одиночка WKO гарантирует, что единственный экземпляр `RemoteMessageObject` будет обслуживать все входящие запросы. После построения серверной сборки давайте перейдем к коду клиентской стороны.

Построение сборки SimpleRemoteObjectClient.exe

Теперь, когда есть слушатель, который размещает в себе удаленный объект, остается последний шаг — построение сборки, которая будет запрашивать доступ к его службам. Установите ссылку на `System.Runtime.Remoting.dll` и `SimpleRemotingAsm.dll`. Реализуйте `Main()` следующим образом:

```
using System;
using System.Runtime.Remoting;
using System.Runtime.Remoting.Channels;
using System.Runtime.Remoting.Channels.Http;
using SimpleRemotingAsm;
namespace SimpleRemoteObjectClient
{
  class SimpleObjClient
  {
    static void Main(string[] args)
    {
      Console.WriteLine("***** SimpleRemoteObjectClient started! *****");
      Console.WriteLine("Hit enter to end.");

      // Создать новый HttpChannel.
      HttpChannel c = new HttpChannel();
      ChannelServices.RegisterChannel(c);

      // Получить прокси на удаленный тип WKO.
      object remoteObj = Activator.GetObject(
         typeof(SimpleRemotingAsm.RemoteMessageObject),
         "http://localhost:32469/RemoteMsgObj.soap");

      // Использовать удаленный объект.
      RemoteMessageObject simple = (RemoteMessageObject)remoteObj;
      simple.DisplayMessage("Hello from the client!");
      Console.WriteLine("Server says: {0}", simple.ReturnMessage());
      Console.ReadLine();
    }
  }
}
```

Несколько замечаний относительного этого клиентского приложения. Во-первых, обратите внимание, что клиент также должен зарегистрировать канал HTTP, но не специфицирует идентификатор порта, поскольку конечная точка указана в URL активизации, посылаемом клиентом. Поскольку клиент взаимодействует с запрашиваемым типом WKO, вы ограничены вызовом конструктора по умолчанию. Для этого используйте метод `Activator.GetObject()`, передав ему два параметра. Первый — информация о типе, описывающая удаленный объект, с которым вы собираетесь взаимодействовать. Учитывая, что метод `Activator.GetObject()` требует метаданных описания объекта, не совсем ясно, какой смысл клиенту *вдобавок* иметь ссылку на общую сборку! В конце этой главы вы узнаете о различных способах прояснения этого аспекта клиентской сборки.

Второй параметр `Activator.GetObject()` называется *URL активизации*. URL активизации, которые описывают тип WKO, могут быть обобщены в следующем формате:

СхемаПротокола`://`*ИмяКомпьютера*`:`*Порт*`/`*UriОбъекта*

И, наконец, обратите внимание, что метод `Activator.GetObject()` возвращает общий тип `System.Object` и потому вынуждает применить явное приведение, чтобы получить доступ к членам `RemoteMessageObject`.

Тестирование приложения .NET Remoting

Чтобы протестировать приложение, начните с запуска серверного приложения, которое откроет канал HTTP и зарегистрирует `RemoteMessageObject` для удаленного доступа. Затем запустите экземпляр клиентского приложения. Если все пройдет нормально, серверное окно должно выглядеть, как показано на рис. 1.2, а клиентское приложение отобразит то, что представлено на рис. 1.3.

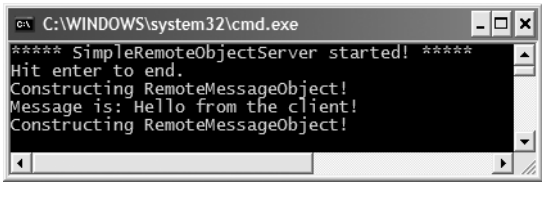

Рис. 1.2. Вывод сервера

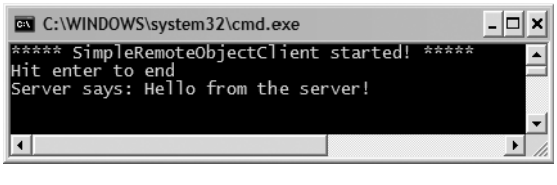

Рис. 1.3. Вывод клиента

Тип `ChannelServices`

Как вы видели, когда серверное приложение желает объявить об удаленном типе, оно использует тип `System.Runtime.Remoting.Channels.ChannelServices`. Этот тип предоставляет небольшой набор статических методов, которые участвуют в процессе регистрации канала удаленного взаимодействия, преобразовании и обнаружении URL. В табл. 1.4 описаны некоторые основные члены.

Таблица 1.4. Избранные члены типа `ChannelServices`

Член	Назначение
`RegisteredChannels`	Это свойство получает или устанавливает список зарегистрированных каналов, каждый из которых представлен интерфейсом `IChannel`.
`DispatchMessage()`	Этот метод выполняет диспетчеризацию входящие удаленные вызовы.
`GetChannel()`	Этот метод возвращает зарегистрированный канал с указанным именем.
`GetUrlsForObject()`	Этот метод возвращает массив всех URL, которые могут использоваться для обращения к специфицированному объекту.
`RegisterChannel()`	Этот метод регистрирует канал вместе с его службами.
`UnregisterChannel()`	Этот метод отменяет регистрацию конкретного канала, удаляя его из списка зарегистрированных.

В дополнение к методам с незатейливыми именами `RegisterChannel()` и `UnregisterChannel()`, тип `ChannelServices` определяет свойство `RegisteredChannels`. Этот член возвращает массив интерфейсов `IChannel`, каждый из которых представляет ссылку на зарегистрированный канал в данном домене приложений. Определение интерфейса `IChannel` достаточно прямолинейно:

```
public interface IChannel
{
  string ChannelName { get; }
  int ChannelPriority { get; }
  string Parse(string url, ref String objectURI);
}
```

Как видите, каждый канал получает дружественное строковое имя вместе с уровнем приоритета. Для иллюстрации, если добавить в метод `Main()` приложения `SimpleRemoteObjectClient` следующую логику:

```
// Список всех зарегистрированных каналов channels.
IChannel[] channelObjs = ChannelServices.RegisteredChannels;
foreach(IChannel i in channelObjs)
{
  Console.WriteLine("Channel name: {0}", i.ChannelName);
  Console.WriteLine("Channel Priority: {0}", i.ChannelPriority);
}
```

на консоли клиентской стороны можно будет увидеть то, что показано на рис. 1.4.

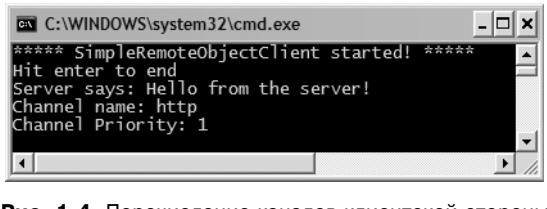

Рис. 1.4. Перечисление каналов клиентской стороны

Тип `RemotingConfiguration`

Другой ключевой тип в технологии .NET Remoting — это `RemotingConfiguration`, который, как указывает его имя, используется для конфигурирования различных аспектов распределенного приложения. К настоящему моменту вы уже видели этот тип в дей-

ствии на серверной стороне (через вызов метода `RegisterWellKnownServiceType()`). В табл. 1.5 перечислены дополнительные статические методы, представляющие интерес, и некоторые из них вы увидите в действии на протяжении оставшейся части главы.

Таблица 1.5. Члены типа `RemotingConfiguration`

Член	Назначение
`ApplicationId`	Получает идентификатор текущего исполняемого приложения.
`ApplicationName`	Получает или устанавливает имя распределенного приложения.
`ProcessId`	Получает идентификатор текущего выполняемого процесса.
`Configure()`	Читает конфигурационный файл и настраивает инфраструктуру .NET Remoting.
`GetRegisteredActivatedClientTypes()`	Извлекает массив типов объектов, зарегистрированных на клиенте, как типы, активизируемые удаленно.
`GetRegisteredActivatedServiceTypes()`	Извлекает массив типов объектов, зарегистрированных на конце службы и активизируемых по запросу клиента.
`GetRegisteredWellKnownClientTypes()`	Извлекает массив типов объектов, зарегистрированных на клиенте, как хорошо известные типы (well-known types).
`GetRegisteredWellKnownServiceTypes()`	Извлекает массив типов объектов, зарегистрированных на конце службы, как хорошо известные типы (well-known types).
`IsWellKnownClientType()`	Проверяет, зарегистрирован ли указанный тип объекта как хорошо известный клиентский тип.
`RegisterActivatedClientType()`	Регистрирует объект на стороне клиента как тип, который может быть активизирован на сервере.
`RegisterWellKnownClientType()`	Регистрирует объект на стороне клиента как хорошо известный тип (одиночный экземпляр или тип с одиночным вызовом).
`RegisterWellKnownServiceType()`	Регистрирует объект на стороне службы как хорошо известный тип (одиночный экземпляр или тип с одиночным вызовом).

Напомним, что слой .NET Remoting различает два типа объектов MBR: WKO (активизируемые сервером) и CAO (активизируемые клиентом). Более того, типы WKO могут конфигурироваться для использования активизации одиночного экземпляра или активизации одиночного вызова. Применяя функциональность типа `RemotingConfiguration`, можно динамически получать эту информацию во время выполнения. Например, если модифицировать метод `Main()` приложения `SimpleRemoteObjectServer` следующим образом:

```
static void Main(string[] args)
{
...
  // Установить дружественное имя для серверного приложения.
  RemotingConfiguration.ApplicationName = "First server app!";
  Console.WriteLine("App Name: {0}",
      RemotingConfiguration.ApplicationName);
  // Получить массив типов WellKnownServiceTypeEntry,
  // представляющих все зарегистрированные WKO.
  WellKnownServiceTypeEntry[] WKOs =
      RemotingConfiguration.GetRegisteredWellKnownServiceTypes();
  // Вывести их статистику.
  foreach(WellKnownServiceTypeEntry wko in WKOs)
  {
    Console.WriteLine("Asm name containing WKO: {0}", wko.AssemblyName);
                      // имя сборки, содержащей WKO
    Console.WriteLine("URL to WKO: {0}", wko.ObjectUri);      // URL-адрес WKO
    Console.WriteLine("Type of WKO: {0}", wko.ObjectType);    // Тип WKO
    Console.WriteLine("Mode of WKO: {0}", wko.Mode);          // Режим WKO
  }
}
```

то увидите список всех типов WKO, зарегистрированных данным доменом серверного приложения. По мере итерации по массиву типов `WellKnownServiceTypeEntry` можно выводить различную интересующую информацию относительно WKO. Учитывая, что серверное приложение зарегистрировало единственный тип (`SimpleRemotingAsm.RemoteMessageObject`), вы получите вывод, показанный на рис. 1.5.

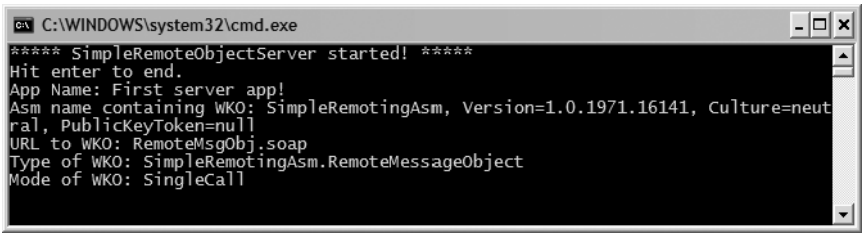

Рис. 1.5. Статистика серверной стороны

Другим важным методом типа `RemotingConfiguration` является `Configure()`. Как вы увидите чуть позже, этот статический член позволяет доменам серверного и клиентского приложений использовать удаленные конфигурационные файлы.

Еще раз о режиме активизации типов WKO

Напомним, что типы WKO могут конфигурироваться для работы в режиме активизации одиночки или с одиночным вызовом. В настоящее время серверное приложение зарегистрировало WKO для использования семантики активизации одиночки:

```
// Одиночки могут обслуживать множество клиентов.
RemotingConfiguration.RegisterWellKnownServiceType(
    typeof(SimpleRemotingAsm.RemoteMessageObject),
    "RemoteMsgObj.soap",
    WellKnownObjectMode.Singleton);
```

Одиночки WKO могут принимать запросы от множества клиентов. Поэтому объект-одиночка поддерживает отношение "один ко многим" между собой и удаленными клиентами. Чтобы протестировать это поведение, запустите серверное приложение (если оно еще не запущено) и запустите три отдельных клиентских приложения. Если вы посмотрите на вывод сервера, то обнаружите единственный вызов конструктора по умолчанию `RemoteMessageObject`.

Теперь, чтобы протестировать поведение объектов с одиночным вызовом, модифицируем сервер для регистрации WKO на поддержку активизации одиночного вызова:

```
// Типы с одиночным вызовом поддерживают отношение
// "один к одному" между клиентом и WKO.
RemotingConfiguration.RegisterWellKnownServiceType(
    typeof(SimpleRemotingAsm.RemoteMessageObject),
    "RemoteMsgObj.soap",
    WellKnownObjectMode.SingleCall);
```

Перекомпилируйте и запустите серверное приложение, после чего запустите три клиента. На этот раз можно увидеть, что новый объект `RemoteMessageObject` создается для каждого клиентского запроса. Как вы поняли, если нужно разделить данные состояния между множеством удаленных клиентов, единственный вариант состоит в использовании активизации одиночки; тогда все удаленные клиенты взаимодействуют с единственным экземпляром удаленного объекта.

Исходный код. Проекты `SimpleRemotingAsm`, `SimpleRemoteObjectServer` и `SimpleRemoteObjectClient` расположены в подкаталоге `Bonus Chapter 1`.

Развертывание сервера на удаленной машине

Пока вы преодолевали границы приложений и процессов на одной машине. Если вы подключены к другому компьютеру, давайте расширим этот пример, позволив клиенту взаимодействовать с типом `RemoteMessageObject` между разными машинами. Для этого понадобится выполнить следующие шаги.

1. На машине-сервере создайте и откройте для сетевого доступа папку со сборками серверной стороны.
2. Скопируйте сборки `SimpleRemoteObjectServer.exe` и `SimpleRemotingAsm.dll` в эту папку сервера.
3. Откройте рабочее пространство проекта `SimpleRemoteObjectClient` и измените URL активизации, указав имя удаленной машины. Например:

```
// Получить прокси удаленного объекта.
Object remoteObj = Activator.GetObject(
    typeof(SimpleRemotingAsm.RemoteMessageObject),
    "http://YourRemoteBoxName:32469/RemoteMsgObj.soap");
```

4. Запустите приложение `SimpleRemoteObjectServer.exe` на серверной машине.
5. Запустите приложение `SimpleRemoteObjectClient.exe` на клиентской машине.
6. Откиньтесь на спинку стула и улыбайтесь.

На заметку! URL-адреса активизации могут специфицировать IP-адрес вместо имени машины

Использование канала TCP

Сейчас ваш удаленный объект доступен по сетевому протоколу HTTP. Как уже упоминалось, этот протокол довольно дружественен по отношению к брандмауэрам, но результирующие пакеты SOAP несколько "раздуты" (ввиду природы представления данных XML). Чтобы сократить нагрузку, можете обновить клиентскую и серверную сборки для использования канала TCP, с применением типа `BinaryFormater` "за кулисами". Необходимые изменения серверной сборки показаны ниже.

На заметку! Когда вы определяете объект, доступный по URI через конечную точку TCP, принято (но не обязательно) использовать расширение `*.rem` (от "remote").

```
// Уточнение сервера!
using System.Runtime.Remoting.Channels.Tcp;
...
static void Main(string[] args)
{
...
  // Создать новый TcpChannel.
  TcpChannel c = new TcpChannel(32469);
  ChannelServices.RegisterChannel(c);

  // Зарегистрировать 'хорошо известный' объект в режиме одиночного вызова.
  RemotingConfiguration.RegisterWellKnownServiceType(
      typeof(SimpleRemotingAsm.RemoteMessageObject),
      "RemoteMsgObj.rem",
      WellKnownObjectMode.SingleCall);
  Console.ReadLine();
}
```

Обратите внимание, что теперь вы регистрируете для слоя .NET Remoting тип `System.Runtime.Remoting.Channels.Tcp.TcpChannel`. Также отметьте, что URI объекта изменен для поддержки более общего типа (`RemoteMsgObj.rem`) вместо SOAP-ориентированного расширения `*.soap`. Изменения в клиенте также просты:

```
// Уточнения клиента!
using System.Runtime.Remoting.Channels.Tcp;
...
static void Main(string[] args)
{
...
  // Создать новый TcpChannel.
  TcpChannel c = new TcpChannel();
  ChannelServices.RegisterChannel(c);

  // Получить прокси на удаленный объект.
  object remoteObj = Activator.GetObject(
      typeof(SimpleRemotingAsm.RemoteMessageObject),
      "tcp://localhost:32469/RemoteMsgObj.rem");

  // Использовать объект.
  RemoteMessageObject simple = (RemoteMessageObject)remoteObj;
  simple.DisplayMessage("Hello from the client!");
  Console.WriteLine("Server says: {0}", simple.ReturnMessage());
  Console.ReadLine();
}
```

Единственное, на что здесь следует обратить внимание: URL активизации клиента теперь должен специфицировать квалификатор канала `tcp://` вместо `http://`. Помимо этого большая часть кода остается идентичной прежней версии с логикой `HttpChannel`.

Исходный код. Проекты `TCPSimpleRemoteObjectServer` и `TCPSimpleRemoteObjectClient` находятся подкаталоге `Bonus Chapter 1` (оба проекта используют созданную ранее сборку `SimpleRemotingAsm.dll`).

Несколько слов о IpcChannel

Прежде чем перейти к изучению конфигурационных файлов .NET Remoting, напомним, что .NET 2.0 также предоставляет тип `IpcChannel`, который обеспечивает самый быстрый способ обмена информацией двух приложений на *одной и той же машине*. Учитывая, что эта глава посвящена теме распределенных программ, охватывающих два и более компьютеров, заинтересованного читателя мы отсылаем к теме `IpcChannel` в документации по .NET Framework 2.0 SDK (как вы можете догадаться, код останется почти таким же, как и в случае применения `HttpChannel` и `TcpChannel`).

Конфигурационные файлы .NET Remoting

К данному моменту вы успешно построили распределенное приложение, используя слой .NET Remoting. Одна проблема, которую вы, возможно, заметили в этих первых примерах, состоит в том, что клиентское и серверное приложения включают значительную часть жестко закодированной логики. Например, сервер специфицирует фиксированный идентификатор порта, фиксированный режим активизации и фиксированный тип канала. С другой стороны клиент жестко кодирует имя удаленного объекта, с которым он пытается взаимодействовать.

Вы наверняка согласитесь, что наивно было бы думать, что начальные положения дизайна останутся неизменными до момента развертывания готового приложения. В идеале такие детали, как идентификатор порта и режим активизации объекта должны меняться на лету, без необходимости перекомпилировать и заново развертывать клиентскую или серверную кодовую базу. В схеме .NET Remoting все эти проблемы могут быть преодолены за счет применения конфигурационного файла.

Файл `*.config` может быть использован для представления подсказок CLR, касающихся загрузки сборок, на которые установлены внешние ссылки. Одни и те же файлы `*.config` могут прмиеняться для информирования CLR о множестве деталей удаленного взаимодействия, как на стороне клиента, так и на стороне сервера.

Когда вы строите файл `*.config`, элемент `<system.runtime.remoting>` служит для хранения различных деталей, касающихся удаленного взаимодействия. Имейте в виду, что если строится приложение, которое уже имеет файл `*.config`, специфицирующий детали разрешения сборки, можно добавлять элементы `remoting` прямо в этот файл. Поэтому единственный файл `*.config`, содержащий информацию об удаленном взаимодействии и привязке должен выглядеть примерно так:

```
<configuration>
  <system.runtime.remoting>
    <!-- здесь конфигурируются настройки удаленного взаимодействия клиент/сервер -->
  </system.runtime.remoting>
  <runtime>
    <!-- здесь находятся настройки привязки сборки -->
  </runtime>
</configuration>
```

Если конфигурационный файл не нуждается в указании логики привязки сборки, можно опустить элемент <runtime> и применять такой скелет файла *.config:

```
<configuration>
  <system.runtime.remoting>
    <!-- здесь конфигурируются настройки удаленного взаимодействия клиент/сервер -->
  </system.runtime.remoting>
</configuration>
```

Построение файлов *.config серверной стороны

Конфигурационные файлы серверной стороны позволяют объявлять объекты, которые доступны удаленным вызовам, а также информацию о канале и порте. Применяя элементы <service>, <wellknown> и <channels>, можно заменить следующую логику серверной стороны:

```
// Жестко закодированная логика HTTP сервера.
HttpChannel c = new HttpChannel(32469);
ChannelServices.RegisterChannel(c);
RemotingConfiguration.RegisterWellKnownServiceType(
    typeof(SimpleRemotingAsm.RemoteMessageObject),
    "RemoteMsgObj.soap",
    WellKnownObjectMode.Singleton);
```

следующим файлом *.config:

```
<configuration>
  <system.runtime.remoting>
    <application>
      <service>
        <wellknown
          mode="Singleton"
          type="SimpleRemotingAsm.RemoteMessageObject, SimpleRemotingAsm"
          objectUri="RemoteMsgObj.soap"/>
      </service>
      <channels>
          <channel ref="http" port="32469"/>
      </channels>
    </application>
  </system.runtime.remoting>
</configuration>
```

Обратите внимание, что большая часть информации об удаленном взаимодействии находится в контексте элемента <service> (а не server). Дочерний элемент <wellknown> использует три атрибута (mode, type и objectUri) для спецификации хорошо известного объекта регистрации слоя .NET Remoting. Дочерний элемент <channels> содержит любое количество элементов <channel>, позволяющих определить тип канала (в данном случае — HTTP) для открытия на сервере. Каналы TCP просто используют строковую лексему tcp вместо http.

Поскольку файл SimpleRemoteObjectServer.exe.config содержит всю необходимую информацию, метод Main() серверной стороны существенно разгружается. Все, что понадобится сделать здесь — это добавить единственный вызов RemotingConfiguration.Configure() и специфицировать имя конфигурационного файла:

```
static void Main(string[] args)
{
  // Зарегистрировать 'хорошо известный' объект, используя файл *.config.
  RemotingConfiguration.Configure("SimpleRemoteObjectServer.exe.config");
  Console.WriteLine("Server started! Hit enter to end");
  Console.ReadLine();
}
```

Построение файлов *.config клиентской стороны

Клиенты также могут использовать файлы *.config с настройками удаленного взаимодействия. В отличие от конфигурационного файла серверной стороны, в клиентских конфигурационных файлах применяется элемент <client> для идентификации имени хорошо известного объекта, с которым желает взаимодействовать вызывающий код. В дополнение к обеспечению способности динамически изменять информацию об удаленном взаимодействии без необходимости в перекомпиляции кода, файлы *.config клиентской стороны позволяют создавать тип прокси непосредственно с использованием ключевого слова C# new вместо вызова метода Activator.GetObject(). Таким образом, если имеется следующий файл *.config клиентской стороны:

```
<configuration>
  <system.runtime.remoting>
    <application>
      <client displayName = "SimpleRemoteObjectClient">
        <wellknown
           type="SimpleRemotingAsm.RemoteMessageObject, SimpleRemotingAsm"
           url="http://localhost:32469/RemoteMsgObj.soap"/>
      </client>
      <channels>
        <channel ref="http"/>
      </channels>
    </application>
  </system.runtime.remoting>
</configuration>
```

можно изменить клиентский метод Main() так, как показано ниже:

```
static void Main(string[] args)
{
  RemotingConfiguration.Configure("SimpleRemoteObjectClient.exe.config");
  // Используя файл *.config, клиент может создавать
  // экземпляр типа непосредственным вызовом new.
  RemoteMessageObject simple = new RemoteMessageObject();
  simple.DisplayMessage("Hello from the client!");
  Console.WriteLine("Server says: {0}", simple.ReturnMessage());
  Console.WriteLine("Client started! Hit enter to end");
  Console.ReadLine();
}
```

Конечно, при запуске приложения вывод будет идентичен прежнему. Если клиент пожелает использовать канал TCP, свойство url элемента <wellknown> и свойство ref элемента <channel> должны использовать лексему tcp вместо http.

Исходный код. Проекты SimpleRemoteObjectServerWithConfig и SimpleRemoteObject ClientWithConfig находятся в подкаталоге Bonus Chapter 1 (оба используют созданную ранее сборку SimpleRemotingAsm.dll).

Работа с объектами MBV

Наше первое приложение с удаленным взаимодействием позволило клиенту обращаться к одиночному экземпляру типа WKO. Напомним, что типы WKO (по определению) являются типами MBR, и потому клиентский доступ к ним выполняется через промежуточный прокси. В отличие от этого, типы MBV — это локальные копии объектов серверной стороны, обычно возвращаемые методами типа MBR. Хотя вы уже знаете,

как конфигурировать тип MBV (пометить класс атрибутом [Serializable]), вы еще не видели примеров типа MBV в действии (помимо передачи строковых данных между участниками). Для иллюстрации совместной работы типов MBR и MBV давайте рассмотрим другой пример, включающий три сборки:

- общая сборка по имени CarGeneralAsm.dll
- клиентская сборка по имени CarProviderClient.exe
- серверная сборка по имени CarProviderServer.exe

Как и можно было предположить, код, лежащий в основе клиентского и серверного приложений, более или менее идентичен предыдущему примеру, особенно применением файлов *.config. Тем не менее, давайте пройдемся по процессу построения каждой сборки — шаг за шагом.

Построение общей сборки

Во время рассмотрения сериализации объектов был создали тип по имени JamesBondCar (вдобавок к связанным с ним классам Radio и Car). Библиотека кода CarGeneralAsm.dll использует повторно эти типы, поэтому начните с выбора пункта меню Project⇨Add Existing Item (Проект⇨Добавить существующий элемент) и включите эти файлы *.cs в новый проект библиотеки классов (автоматически сгенерированный файл Class1.cs может быть удален). Учитывая, что каждый из этих типов уже помечен атрибутом [Serializable], они готовы к маршализации по значению на удаленный клиент.

Все, что нужно теперь — это тип MBV, который обеспечит доступ в типу JamesBondCar. Однако чтобы сделать задачу чуть более интересной, объект MBR (CarProvider) будет поддерживать обобщенную коллекцию List<> элементов типа JamesBondCar. Также в CarProvider будут определены два члена, которые позволят вызывающему коду получить специфический экземпляр JamesBondCar, а также получить весь список List<> элементов. Вот полный код нового класса:

```
namespace CarGeneralAsm
{
  // Это объект MBR, обеспечивающий доступ
  // к связанным с ним объектам MBV.
  public class CarProvider : MarshalByRefObject
  {
    private List<JamesBondCar> theJBCars =
      new List<JamesBondCar>();
    // Добавить несколько автомобилей в список.
    public CarProvider()
    {
      Console.WriteLine("Car provider created");
      theJBCars.Add(new JamesBondCar("QMobile", 140, true, true));
      theJBCars.Add(new JamesBondCar("Flyer", 140, true, false));
      theJBCars.Add(new JamesBondCar("Swimmer", 140, false, true));
      theJBCars.Add(new JamesBondCar("BasicJBC", 140, false, false));
    }
    // Получить все JamesBondCar.
    public List<JamesBondCar> GetAllAutos()
    { return theJBCars; }
    // Получить один JamesBondCar.
    public JamesBondCar GetJBCByIndex(int i)
    { return (JamesBondCar)theJBCars[i]; }
  }
}
```

Обратите внимание, что метод GetAllAutos() возвращает внутренний экземпляр типа List<>. Очевидный вопрос, возникающий здесь: каким образом член пространства имен System.Collections.Generic маршализуется обратно вызывающему коду? Если взглянуть на описание этого типа в документации по .NET Framework 2.0 SDK, обнаружится, что List<> декорирован атрибутом [Serializable]:

```
[SerializableAttribute()]
public class List<T> : IList, ICollection, IEnumerable
```

Таким образом, все содержимое List<> может быть передано по значению вызывающему коду (предполагая, что хранимые в нем типы также сериализуемы)! Это позволяет сделать очень хороший вывод относительно .NET Remoting и членов библиотек базовых классов. В дополнение к специальным типам MBV и MBR, которые вы создаете самостоятельно, имейте в виду, что любой тип из библиотек базовых классов, декорированный атрибутом [Serializable], способен функционировать как тип MBV в архитектуре удаленного взаимодействия .NET. Аналогично любой тип, унаследованный (прямо или опосредованно) от MarshalByRefObject, будет работать как тип MBR.

На заметку! Имейте в виду, что SoapFormatter не поддерживает сериализацию обобщенных типов. Если вы построите методы, принимающие обобщенные типы (такие как List<>), то должны применить BinaryFormatter и объект TcpChannel.

Построение серверной сборки

Сборка серверного хоста (CarProviderServer.exe) имеет следующую логику внутри Main():

```
using System;
using System.Runtime.Remoting;
using System.Runtime.Remoting.Channels;
using System.Runtime.Remoting.Channels.Http;
using CarGeneralAsm;
namespace CarProviderServer
{
  class CarServer
  {
    static void Main(string[] args)
    {
      RemotingConfiguration.Configure("CarProviderServer.exe.config");
      Console.WriteLine("Car server started! Hit enter to end");
      Console.ReadLine();
    }
  }
}
```

Связанный файл *.config почти идентичен файлу *.config серверной стороны, который был создан в предыдущем примере:

```
<configuration>
  <system.runtime.remoting>
    <application>
      <service>
        <wellknown mode="Singleton"
                   type="CarGeneralAsm.CarProvider, CarGeneralAsm"
                   objectUri="carprovider.rem" />
      </service>
      <channels>
        <channel ref="tcp" port="32469" />
```

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

Построение клиентской сборки

И последнее: нужно построить клиентское приложение, которое использует тип MBR `CarProvider` для получения дискретных типов `JamesBondCar`, а также типа `List<>`. Получив тип от `CarProvider`, вы отправите его на обработку вспомогательной функции `UseCar()`:

```
using System;
using System.Runtime.Remoting;
using System.Runtime.Remoting.Channels;
using System.Runtime.Remoting.Channels.Http;
using CarGeneralAsm;
using System.Collections.Generic;
namespace CarProviderClient
{
  class CarClient
  {
    private static void UseCar(JamesBondCar c)
    {
      Console.WriteLine("-> Name: {0}", c.PetName);
      Console.WriteLine("-> Max speed: {0}", c.MaxSpeed);
      Console.WriteLine("-> Seaworthy? : {0}", c.isSeaWorthy);
      Console.WriteLine("-> Flight worthy? : {0}", c.isFlightWorthy);
      Console.WriteLine();
    }
    static void Main(string[] args)
    {
      RemotingConfiguration.Configure("CarProviderClient.exe.config");
      // Создать поставщика автомобилей.
      CarProvider cp = new CarProvider();
      // Получить первый JBC.
      JamesBondCar qCar = cp.GetJBCByIndex(0);
      // Получить все JBC.
      List<JamesBondCar> allJBCs = cp.GetAllAutos();
      // Использовать первый автомобиль.
      UseCar(qCar);
      // Использовать все автомобили из List<>.
      foreach(JamesBondCar j in allJBCs)
        UseCar(j);
      Console.WriteLine("Client started! Hit enter to end");
      Console.ReadLine();
    }
  }
}
```

Файл `*.config` клиентской стороны также выглядит ожидаемым образом. Просто измените URL активизации:

```
<configuration>
  <system.runtime.remoting>
    <application>
      <client displayName = "CarClient">
      <wellknown
         type="CarGeneralAsm.CarProvider, CarGeneralAsm"
         url="tcp://localhost:32469/carprovider.rem"/>
      </client>
```

```
      <channels>
        <channel ref="tcp"/>
      </channels>
    </application>
  </system.runtime.remoting>
</configuration>
```

Теперь запустите серверное и клиентское приложения (именно в таком порядке) и взгляните на их вывод. Консольное окно клиентской стороны пройдется по всем JamesBondCar и выведет статистику каждого экземпляра. Напомним, что взаимодействуя с типами List<> и JamesBondCar, вы оперируете с их членами внутри домена клиентского приложения, поскольку они оба помечены атрибутом [Serializable].

Чтобы убедиться в этом, модифицируйте вспомогательную функцию UseCar(), добавив вызов метода TurnOnRadio() на входящем JamesBondCar. Теперь снова запустите серверное и клиентское приложения. Обратите внимание, что окно сообщения появилось на клиентской машине! Если бы типы Car, Radio и JamesBondCar были сконфигурированы как типы MBR, то окно сообщения появилось бы на серверной машине. Если хотите убедиться в этом, унаследуйте каждый тип от MarshalByRefObject и перекомпилируйте все три сборки (чтобы гарантировать, что Visual Studio скопирует последнюю версию CarGeneralAsm.dll в каталог клиентского и серверного приложений). Когда после этого снова запустить приложение, окно сообщения появится на удаленной машине (сервере).

Исходный код. Проекты CarGeneralAsm, CarProviderServer и CarProviderClient находятся в подкаталоге Bonus Chapter 1.

Активизируемые клиентом объекты

Во всех текущих примерах удаленного взаимодействия использовались объекты WKO. Напомним, что WKO обладают следующими характеристиками:

- WKO могут быть сконфигурированы как одиночки или объекты с одиночным вызовом;
- WKO могут активизироваться только с использованием конструктора типа по умолчанию;
- экземпляры WKO создаются на сервере при первом вызове члена клиентом.

С другой стороны, экземпляры типов CAO могут создаваться с применением любого конструктора типа и создаются в момент использования клиентом ключевого слова C# new или вызова метода типа Activator. Более того, время жизни типов CAO отслеживается механизмом аренды .NET. Имейте в виду, что когда вы конфигурируете тип CAO, слой .NET Remoting генерирует специфический удаленный объект CAO для обслуживания каждого клиента. Большое отличие состоит в том, что CAO всегда существуют, и потому сохраняют состояние между отдельными вызовами методов.

Чтобы проиллюстрировать конструирование, размещение и использование типов CAO, давайте пересмотрим предыдущую общую сборку для работы с автомобилями. Предположим, что в MBR-классе CarProvider определен дополнительный конструктор, позволяющий клиенту передавать массив объектов JamesBondCar, которые будут применяться для наполнения обобщенной коллекции List<>:

```
public class CarProvider : MarshalByRefObject
{
  private List<JamesBondCar> theJBCars = new List<JamesBondCar>();
```

```
    public CarProvider(JamesBondCar[] theCars)
    {
      Console.WriteLine("Car provider created with custom ctor");
      theJBCars.AddRange(theCars);
    }
  ...
  }
```

Чтобы позволить вызывающему коду активизировать CarProvider, используя новый синтаксис конструктора, необходимо построить серверное приложение, которое зарегистрирует CarProvider как тип САО, а не тип WKO. Это можно сделать программно (методом RemotingConfiguration.RegisterActivatedServiceType()) либо через файл *.config серверной стороны. Если хотите жестко закодировать имя объекта САО в коде сервера, то все, что потребуется сделать — это передать информацию о типе (типах) после создания и регистрации канала, как показано ниже:

```
// Жестко закодировать тот факт, что CarProvider — тип САО.
RemotingConfiguration.RegisterActivatedServiceType(
    typeof(CAOCarGeneralAsm.CarProvider));
```

Если решено вместо этого использовать файл *.config, замените элемент <wellknown> на элемент <activated>, как показано ниже:

```
<configuration>
  <system.runtime.remoting>
    <application>
      <service>
        <activated type = "CAOCarGeneralAsm.CarProvider, CAOCarGeneralAsm"/>
      </service>
      <channels>
        <channel ref="tcp" port="32469" />
      </channels>
    </application>
  </system.runtime.remoting>
</configuration>
```

И, наконец, нужно модифицировать клиентское приложение, причем не только файл *.config (или код программы), чтобы запросить доступ к удаленному САО, но также инициировать вызов специального конструктора типа CarProvider. Вот необходимые обновления метода Main() клиентской стороны:

```
static void Main(string[] args)
{
  // Прочитать обновленный файл *.config.
  RemotingConfiguration.Configure("CAOCarProviderClient.exe.config");
  // Создать массив объектов для передачи поставщику.
  JamesBondCar[] cars =
  {
    new JamesBondCar("Viper", 100, true, false),
    new JamesBondCar("Shaken", 100, false, true),
    new JamesBondCar("Stirred", 100, true, true)
  };
  // Теперь вызвать специальный конструктор.
  CarProvider cp = new CarProvider(cars);
...
}
```

Обновленный файл *.config клиентской стороны также использует элемент <activated> вместо <wellknown>. Вдобавок элемент <client> теперь требует, чтобы свойство url определяло местоположение зарегистрированного САО. Напомним, что

когда сервер регистрировал CarProvider как WKO, клиент специфицировал эту информацию в элементе <wellknown>.

```
<configuration>
  <system.runtime.remoting>
    <application>
      <client displayName = "CarClient"
              url = "tcp://localhost:32469">
        <activated type = "CAOCarGeneralAsm.CarProvider, CAOCarGeneralAsm" />
      </client>
      <channels>
        <channel ref="tcp"/>
      </channels>
    </application>
  </system.runtime.remoting>
</configuration>
```

Если взамен жестко закодировать клиентский запрос типа CAO, то для этого можно использовать метод RegistrationServices.RegisterActivatedClientType():

```
static void Main(string[] args)
{
  // Использовать жестко закодированные значения.
  RemotingConfiguration.RegisterActivatedClientType(
      typeof(CAOCarGeneralAsm.CarProvider),
      "tcp://localhost:32469");
...
}
```

Если теперь запустить обновленные сборки сервера и клиента, обнаружится, что можно отправлять специальный массив экземпляров JamesBondCar удаленному CarProvider через перегруженный конструктор.

Исходный код. Проекты CAOCarGeneralAsm, CAOCarProviderServer и CAOCarProviderClient находятся в подкаталоге Bonus Chapter 1.

Жизненный цикл одиночных объектов CAO/WKO на основе аренды

Как вы уже видели, типы WKO, сконфигурированные на активизацию одиночным вызовом, доступны только на протяжении текущего вызова метода. С учетом этого факта, типы WKO одиночного вызова являются сущностями без состояния. Как только текущий вызов завершен, экземпляр одиночного вызова WKO становится кандидатом на сборку мусора.

С другой стороны, типы CAO и WKO, сконфигурированные на активизацию одиночки, по природе своей являются сущностями с состоянием. Исходя из этих двух конфигурационных настроек объектов, можно задать вопрос: каким образом серверный процесс узнает, когда следует разрушать эти объекты MBR? Ясно, что это может стать огромной проблемой, если сервер уничтожит с помощью сборщика мусора объекты MBR, которые в данный момент используются удаленным клиентом. Если серверная машина станет ждать слишком долго перед тем, как освободить ее набор экземпляров MBR, это может слишком загрузить систему, особенно если объекты MBR удерживают ценные системные ресурсы (подключения к базе данных, неуправляемые типы и т.п.).

Время жизни типа CAO- или WKO-одиночки MBR управляется "временем аренды", которое тесно интегрировано со сборщиком мусора .NET. Если время аренды типа CAO- или WKO-одиночки MBR истекает, объект готов к уничтожению при следующем цикле сборки мусора. Подобно любому типу .NET, если удаленный объект переопределяет `System.Object.Finalize()` (посредством синтаксиса деструктора C#), то исполняющая система .NET инициирует при этом логику финализации.

Поведение аренды по умолчанию

Типы CAO- или WKO-одиночки MBR обладают тем, что называется периодом *аренды по умолчанию* (default lease), составляющим пять минут. Если исполняющая система обнаруживает пять минут отсутствия активности типа CAO- или WKO-одиночки MBR, предполагается, что клиент уже не использует объект, и потому удаленный объект может быть подвергнут сборке мусора. Однако когда время аренды по умолчанию истекает, это не значит, что объект *немедленно* помечается готовым к сборке мусора. В действительности есть много способов повлиять на поведение аренды по умолчанию.

Самое главное, в любой момент, когда удаленный клиент вызывает член удаленного типа CAO- или WKO-одиночки MBR, лимит аренды возобновляется к его начальному значению в пять минут. Вдобавок к политике автоматического возобновления аренды после вызова клиента, исполняющая система .NET предлагает три дополнительных альтернативы:

- с помощью файлов `*.config` можно переопределять установки аренды по умолчанию для удаленных объектов;
- для работы с удаленным объектом, время аренды которого истекло, могут использоваться спонсоры аренды серверной стороны;
- для работы с удаленным объектом, время аренды которого истекло, могут использоваться спонсоры аренды клиентской стороны.

Мы рассмотрим эти опции в последующих разделах главы, а пока давайте изучим установки аренды по умолчанию для удаленного типа. Напомним, что базовый класс `MarshalByRefObject` определяет член по имени `GetLifetimeService()`. Этот метод возвращает ссылку на внутренне реализованный объект, поддерживающий интерфейс `System.Runtime.Remoting.Lifetime.ILease`. Несложно догадаться, что интерфейс `ILease` применяется для взаимодействия с поведением аренды определенного типа CAO- или WKO-одиночки. Вот его формальное определение:

```
public interface ILease
{
  TimeSpan CurrentLeaseTime { get; }
  LeaseState CurrentState { get; }
  TimeSpan InitialLeaseTime { get; set; }
  TimeSpan RenewOnCallTime { get; set; }
  TimeSpan SponsorshipTimeout { get; set; }
  void Register(System.Runtime.Remoting.Lifetime.ISponsor obj);
  void Register(System.Runtime.Remoting.Lifetime.ISponsor obj,
     TimeSpan renewalTime);
  TimeSpan Renew(TimeSpan renewalTime);
  void Unregister(System.Runtime.Remoting.Lifetime.ISponsor obj);
}
```

Интерфейс `ILease` не только позволяет получать информацию о текущей аренде (через `CurrentLeaseTime`, `CurrentState` и `InitialLeaseTime`), но также предоставляет возможность создавать "спонсоров" аренды (подробнее о них будет сказано ниже). В табл. 1.6 описаны роли членов `ILease`.

Таблица 1.6. Члены интерфейса `ILease`

Член	Назначение
`CurrentLeaseTime`	Получает период времени, ставшийся до деактивизации объекта, если он не получит дальнейших вызовов методов.
`CurrentState`	Получает текущее состояние аренды, представленное перечислением `LeaseState`.
`InitialLeaseTime`	Получает или устанавливает начальный период времени для данной аренды. Начальное время аренды объекта определяется периодом между начальной активизацией и истечением аренды, если не произошло никаких других вызовов методов.
`RenewOnCallTime`	Получает или устанавливает период времени, на который вызов удаленного объекта увеличивает `CurrentLeaseTime`.
`SponsorshipTimeout`	Получает или устанавливает период времени ожидания, пока спонсор вернет обновленное время аренды.
`Register()`	Перегружен. Регистрирует спонсора аренды.
`Renew()`	Обновляет аренду на заданное время.
`Unregister()`	Удаляет спонсора из списка спонсоров

Чтобы проиллюстрировать характеристики аренды по умолчанию удаленного объекта CAO- иди WKO-одиночки, предположим, что в текущем проекте `CAOCarGeneralAsm` определен новый внутренний класс по имени `LeaseInfo`. Этот `LeaseInfo` поддерживает статический член по имени `LeaseStats()`, который выводит избранную статистику о текущей аренде для типа `CarProvider` в консольное окно серверной стороны (не забудьте специфицировать директиву `using` для пространства имен `System.Runtime.Remoting.Lifetime`, чтобы проинформировать компилятор, где определен тип `ILease`:

```
internal class LeaseInfo
{
  public static void LeaseStats(ILease itfLease)
  {
    Console.WriteLine("***** Lease Stats *****");
    Console.WriteLine("Lease state: {0}", itfLease.CurrentState);
    Console.WriteLine("Initial lease time: {0}:{1}",
        itfLease.InitialLeaseTime.Minutes,
        itfLease.InitialLeaseTime.Seconds);
    Console.WriteLine("Current lease time: {0}:{1}",
        itfLease.CurrentLeaseTime.Minutes,
        itfLease.CurrentLeaseTime.Seconds);
    Console.WriteLine("Renew on call time: {0}:{1}",
        itfLease.RenewOnCallTime.Minutes,
        itfLease.RenewOnCallTime.Seconds);
    Console.WriteLine();
  }
}
```

Теперь, имея этот вспомогательный тип, представим, что `LeaseInfo.LeaseStats()` вызван изнутри методов `GetJBCByIndex()` и `GetAllAutos()` типа `CarProvider`. Как только вы перекомпилируете сборки сервера и клиента (просто чтобы гарантировать, что Visual Studio скопирует последнюю версию `CarGeneralAsm.dll` в каталоге приложений клиента и сервера), запустите приложение снова. Консольное окно сервера должно выглядеть примерно как показано на рис. 1.6.

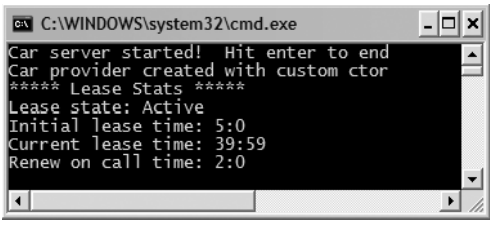

Рис. 1.6. Информация об аренде по умолчанию для `CarProvider`

Изменение характеристик аренды по умолчанию

Очевидно, что характеристики аренды по умолчанию типа CAO/WKO-одиночки могут не подойти каждому удаленному объекту соответствующего типа. Для изменения установок по умолчанию есть два выбора:

- можно поправить установки аренды по умолчанию в файле `*.config` серверной стороны;
- можно программно изменить установки аренды по умолчанию, переопределив члены базового класса `MarshalByRefObject`.

Хотя в обоих случаях действительно изменяются установки аренды по умолчанию, между ними есть ключевое различие. Когда вы используете файл `*.config` серверной стороны, установки аренды касаются *всех* объектов, размещенных в серверном процессе. В отличие от этого, когда вы изменяете установки избранных членов типа `MarshalByRefObject`, вы можете изменять эти установки для каждого объекта по отдельности.

Чтобы проиллюстрировать изменения установок аренды по умолчанию через файл `*.config`, предположим, что вы обновили данные XML серверной стороны в следующем элементе `<lifetime>`:

```
<configuration>
  <system.runtime.remoting>
    <application>
      <lifetime leaseTime = "15M" renewOnCallTime = "5M"/>
      <service>
          <activated type = "CarGeneralAsm.CarProvider, CarGeneralAsm"/>
      </service>
      <channels>
           <channel ref="tcp" port="32469" />
      </channels>
    </application>
  </system.runtime.remoting>
</configuration>
```

Обратите внимание, что свойства `leaseTime` и `renewOnCallTime` помечены суффиксом `M`, который, как несложно догадаться, указывает длительность в минутах каждой единицы времени, связанной с арендой. При желании элемент `<lifetime>` может также дополнять числовое значение суффиксом `MS` (миллисекунды), `S` (секунды), `H` (часы) или даже `D` (дни).

Теперь вспомните, что когда вы обновляете серверный файл `*.config`, то эффективно изменяете настройки аренды для каждого объекта CAO/WKO-одиночки, создаваемого на сервере. В качестве альтернативы можно программно переопределить метод `InitializeLifetime()` для конкретного удаленного типа:

```
public class CarProvider : MarshalByRefObject
{
  public override object InitializeLifetimeService()
  {
    // Получить текущую информацию об аренде.
    ILease itfLeaseInfo =
        (ILease) base.InitializeLifetimeService();
    // Скорректировать настройки.
    itfLeaseInfo.InitialLeaseTime = TimeSpan.FromMinutes(50);
    itfLeaseInfo.RenewOnCallTime = TimeSpan.FromMinutes(10);
    return itfLeaseInfo;
  }
...
}
```

Здесь `CarProvider` изменил свое значение `InitialLeaseTime` на 50 минут и значение `RenewOnCallTime` — на 10. Опять-таки, преимущество переопределения `InitializeLifetimeService()` состоит в возможности конфигурировать каждый удаленный тип индивидуально.

И, наконец, если отключить жизненный цикл на основе аренды для определенного типа САО/WKO-одиночки, можно переопределить `InitializeLifetimeServices()` и просто вернуть `null`. Если поступить подобным образом, будет сконфигурирован тип MBR, который *никогда* не уничтожится на протяжении жизни серверного приложения, в котором он создан.

Настройка аренды на стороне сервера

Как было только что показано, когда тип MBR переопределяет `InitializeLifetimeServices()`, он может изменить свое поведение аренды по умолчанию в момент активизации. Однако что если удаленный тип желает изменить эти параметры *после* активизации цикла? Например, предположим, что `CarProvider` имеет новый метод, реализация которого требует выполнения длительной операции (вроде подключения к удаленной базе данных и чтения большого количества записей). Прежде чем начать выполнять такую задачу, можно программно скорректировать настройки аренды, например, если осталось менее одной минуты времени аренды, оно увеличивается до, скажем, десяти минут. Для этого применяются унаследованные методы `MarshalByRefObject.GetLifetimeService()` и `ILease.Renew()`:

```
// Изменение установок аренды на стороне сервера.
// Предположим, это новый метод типа CarProvider.
public void DoLengthyOperation()
{
  ILease itfLeaseInfo = (ILease)this.GetLifetimeService();
  if(itfLeaseInfo.CurrentLeaseTime.TotalMinutes < 1.0)
      itfLeaseInfo.Renew(TimeSpan.FromMinutes(10));

  // Выполнить длительную задачу...
}
```

Настройка аренды на стороне клиента

В порядке дополнительного примечания к `ILease`: домен клиентского приложения может регулировать текущие свойства аренды для типов САО/WKO-одиночек, с которыми он взаимодействует по сети. Для этого клиент использует статический метод `RemotingServices.GetLifetimeService()`. В качестве параметра этого метода клиент передает ссылку на удаленный тип:

```
// Настройка аренды со стороны клиента.
CarProvider cp = new CarProvider(cars);
ILease itfLeaseInfo = (ILease)RemotingServices.GetLifetimeService(cp);
if(itfLeaseInfo.CurrentLeaseTime.TotalMinutes < 10.0)
    itfLeaseInfo.Renew(TimeSpan.FromMinutes(1000));
```

Этот подход может быть полезен, если домен клиентского приложения собирается запустить длительный процесс в том же потоке выполнения, который используется удаленным типом. Например, если однопоточное приложение собирается печатать 100-страничный документ, то велика вероятность, что удаленный тип САО/WKO-одиночка может выйти по таймауту во время этого процесса. Другое (более элегантное) решение, конечно, состоит в порождении нового потока выполнения, но основная идея должны быть ясна.

Спонсорство аренды на стороне сервера и на стороне клиента

Последняя тема, касающаяся времени жизни аренды объекта САО/WKO-одиночки — *спонсорство аренды* (lease sponsorship). Как было только что показано, каждая сущность САО/WKO-одиночки имеет установки аренды по умолчанию, которые могут быть изменены различными способами — как на стороне сервера, так и на стороне клиента. Теперь, независимо от конфигурации аренды типа, в конечном итоге время пользования объектом MBR истекает. В этот момент исполняющая система выполнит сборку мусора сущности... ну, почти.

На самом деле истина состоит в том, что прежде чем объект с истекшим временем аренды помечается для сборки мусора, исполняющая система проверит, нет ли у данного объекта MBR зарегистрированных спонсоров аренды. Просто говоря, спонсор — это тип, реализующий интерфейс `ISponsor`, определенный следующим образом:

```
public interface System.Runtime.Remoting.Lifetime.ISponsor
{
   TimeSpan Renewal(ILease lease);
}
```

Если исполняющая система обнаруживает, что объект MBR имеет спонсора, она не станет убирать его в мусор, а вместо этого вызовет метод `Renewal()` объекта-спонсора, чтобы добавить время к текущей аренде. С другой стороны, если MBR не имеет спонсора, время аренды объекта действительно истекло.

Предположим, что вы создали специальный класс, реализующий `ISponsor`, и он реализует `Renewal()` для возврата определенного периода времени (типа `TimeSpan`). Тогда возникает следующий вопрос: как именно ассоциировать тип с определенным удаленным объектом? Опять-таки, эта операция может быть выполнена либо доменом серверного приложения, либо доменом клиентского приложения.

Для этого заинтересованная часть получает ссылку `ILease` (через унаследованный метод `GetLifetimeService()` на сервере или статический метод `RemotingServices.GetLifetimeService()` на клиенте) и вызывает `Register()`:

```
// Регистрация спонсора на стороне сервера.
CarSponsor mySponsor = new CarSponsor();
ILease itfLeaseInfo = (ILease)this.GetLifetimeService();
itfLeaseInfo.Register(mySponsor);
// Регистрация спонсора на стороне клиента.
CarSponsor mySponsor = new CarSponsor();
CarProvider cp = new CarProvider(cars);
ILease itfLeaseInfo = (ILease)RemotingServices.GetLifetimeService(cp);
itfLeaseInfo.Register(mySponsor);
```

В любом случае, если клиент или сервер желает отменить спонсорство, он может сделать это с помощью метода ILease.Unregister(), например:

```
// Удалить спонсора данного объекта.
itfLeaseInfo.Unregister(mySponsor);
```

На заметку! Объекты, спонсированные клиентской стороной, в дополнение к реализации ISponsor должны наследоваться от MarshalByRefObject, поскольку клиент должен передать спонсора в удаленный домен приложения.

Как видите, управление временем жизни типов MBR с состоянием несколько сложнее, чем стандартная сборка мусора. Положительной стороной всего этого является масса возможностей контролировать, когда удаленный тип предстанет перед своим создателем. Однако, как вы наверно догадались, есть шанс, что удаленный тип может быть удален из памяти без ведома клиента. И тогда, если клиент попытается вызвать член объекта, уже удаленного из памяти, исполняющая система сгенерирует исключение System.Runtime.Remoting.RemotingException, после чего клиент сможет создать новый экземпляр удаленного типа или просто предпринять какие-то другие действия.

Исходный код. Проекты CAOCarGeneralAsmLease, CAOCarProviderServerLease и CAOCarProviderClientLease находятся в подкаталоге Bonus Chapter 1.

Альтернативные хосты для удаленных объектов

На протяжении этой главы было сконструировано множество консольных серверных хостов, которые обеспечивали доступ клиентов к определенному набору удаленных объектов. Если есть опыт работы с классической объектной моделью распределенных компонентов (Distributed Component Object Model — DCOM), этот шаг может показаться несколько странным. В мире DCOM нередко строился единственный COM-сервер серверной стороны, который размещал в себе удаленные объекты и также отвечал за прием входящих запросов от удаленного клиента. Это DCOM-приложение *.exe молча загружалось в фоновом режиме, не показывая никакого консольного окна.

Когда вы строите серверную сборку .NET, достаточно высоки шансы, что удаленной машине не понадобится отображать никакого пользовательского интерфейса. Взамен все, что требуется — это построить сущность серверной стороны, которая откроет нужные каналы и зарегистрирует необходимые удаленные объекты для клиентского доступа. Более того, при построении простого консольного хоста вы (или кто-то другой) должен будет вручную запустить эту сборку *.exe серверной стороны, потому что .NET Remoting не запустит автоматически *.exe на стороне сервера, когда поступит вызов от удаленного клиента.

С учетом этих двух трудностей возникает вопрос: как построить невидимый слушатель, который бы загружался автоматически? Программистам .NET доступны два основных варианта выбора, когда им нужно построить прозрачный хост для различных удаленных объектов:

- построить приложение .NET службы Windows для размещения удаленных объектов;
- использовать IIS в качестве хоста удаленных объектов.

Хостинг удаленных объектов с использованием службы Windows

Возможно, идеальным хостом для удаленных объектов является служба Windows, поскольку:

- она может быть настроена для автоматической загрузки при запуске системы;
- выполняется как невидимый фоновый процесс;
- может быть запущена от имени определенного пользователя.

К счастью, построить специальную службу Windows с использованием платформы .NET исключительно просто в отличие от того, как это делается в чистом Win32 API. Для иллюстрации давайте создадим проект службы Windows по имени `CarWinService` (рис. 1.7), который будет выполнять хостинг удаленных типов, содержащихся в сборке `CarGeneralAsm.dll`.

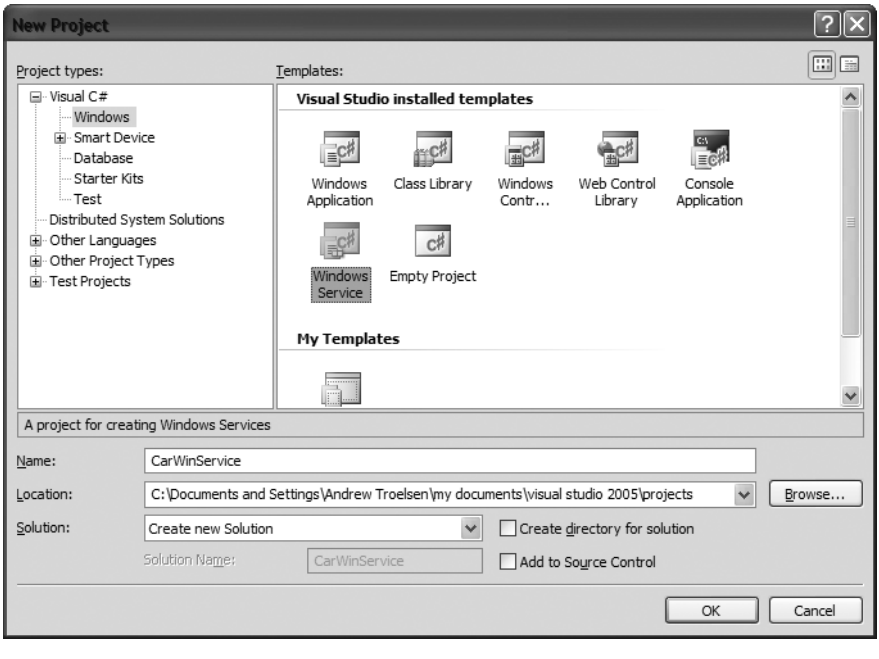

Рис. 1.7. Создание нового рабочего пространства проекта службы Windows

Среда Visual Studio отвечает генерацией частичного класса (по имени `Service1` по умолчанию), унаследованного от `System.ServiceProcess.ServiceBase`, и другого класса (`Program`), который реализует метод `Main()` службы. Учитывая, что `Service1` довольно невнятное имя для специальной службы, первое, что нужно сделать — изменить значения свойств `(Name)` и `ServiceName` на `CarService`, используя для этого окно Properties (Свойства). Отличие между этими двумя установками заключается в том, что `(Name)` используется для определения имени, по которому тип будет доступен в коде, а `ServiceName` помечает имя, отображаемое в инструментах конфигурирования служб Windows.

Прежде чем двигаться дальше, удостоверьтесь, что установлены ссылки на сборки CarGeneralAsm.dll и System.Remoting.dll и специфицированы следующие дополнительные директивы using в файле, содержащем определение класса CarService:

```
using System.Runtime.Remoting;
using System.Runtime.Remoting.Channels.Http;
using System.Runtime.Remoting.Channels;
using System.Diagnostics;
```

Реализация метода Main()

Метод Main() класса Program отвечает за запуск каждой службы, определенной в проекте, посредством передачи массива объектов типа ServiceBase в статический метод Service.Run(). Учитывая переименование специальной службы с Service1 на CarService, вы должны получить следующее определение класса (комментарии удалены для ясности):

```
static class Program
{
  static void Main()
  {
    ServiceBase[] ServicesToRun;
    ServicesToRun = new ServiceBase[] { new CarService() };
    ServiceBase.Run(ServicesToRun);
  }
}
```

Реализация CarService.OnStart()

Вероятно, вы уже представили, какого рода логика должна выполняться при запуске специальной службы на заданной машине. Напомним, что роль CarService состоит в выполнении тех же задач, что и разработанная ранее служба в виде консольного приложения. Поэтому, если вы хотите зарегистрировать CarService как тип WKO-одиночка, доступный через HTTP, можете добавить следующий код в метод OnStart() (как и следовало надеяться, тип RemotingConfiguration можно применять для загрузки конфигурационного файла *.config серверной стороны вместо того, чтобы жестко кодировать реализацию при развертывании удаленных объектов с использованием службы Windows):

```
protected override void OnStart(string[] args)
{
  // Создать новый HttpChannel.
  HttpChannel c = new HttpChannel(32469);
  ChannelServices.RegisterChannel(c);

  // Зарегистрировать WKO одиночного вызова.
  RemotingConfiguration.RegisterWellKnownServiceType(
      typeof(CarGeneralAsm.CarProvider),
      "CarProvider.soap",
      WellKnownObjectMode.SingleCall);

  // Протоколировать успешный запуск.
  EventLog.WriteEntry("CarWinService",
      "CarWinService started successfully!",
      EventLogEntryType.Information);
}
```

Обратите внимание, что как только тип зарегистрирован, в журнал событий Windows записывается специальное сообщение (через тип System.Diagnostics.EventLog) для документирования успешного запуска службы на машине-хосте.

Реализация OnStop()

С технической точки зрения CarService не требует никакой логики останова службы. В целях иллюстрации давайте отправим еще одно сообщение в EventLog, чтобы запротоколировать останов специальной службы Windows:

```
protected override void OnStop()
{
   EventLog.WriteEntry("CarWinService",
       "CarWinService stopped",
       EventLogEntryType.Information);
}
```

После того, как служба готова, остается инсталлировать ее на удаленной машине.

Добавление инсталлятора службы

Прежде чем можно будет инсталлировать службу на определенной машине, следует добавить дополнительный тип к текущему проекту CarWinService. В частности, любая служба Windows (написанная на .NET или Win32 API) требует нескольких записей реестра, чтобы позволить ОС взаимодействовать с собой. Вместо того чтобы создавать эти записи вручную, можно просто добавить тип Installer к проекту службы Windows, который корректно сконфигурирует тип-наследник ServiceBase после инсталляции на целевой машине.

Чтобы добавить инсталлятор к CarService, откройте редактор служб (двойным щелчком на файле CarService.cs в Solution Explorer), выполните щелчок правой кнопкой мыши в любом месте дизайнера и выберите в контекстном меню пункт Add Installer (Добавить инсталлятор), как показано на рис. 1.8.

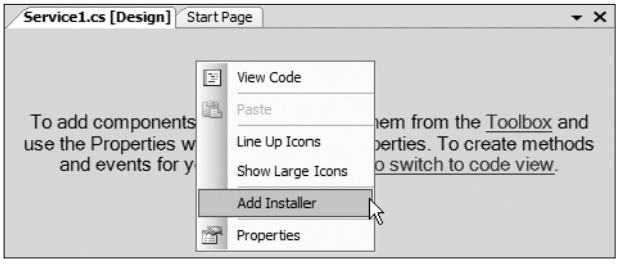

Рис. 1.8. Включение инсталлятора в службу Windows

Этот выбор добавит новый компонент, унаследованный от базового класса System.Configuration.Install.Installer. В дизайнере будет два компонента. Тип serviceInstaller1 представляет инсталлятор конкретной службы в проекте. Если выбрать эту пиктограмму и заглянуть в окно Properties, обнаружится, что свойство ServiceName было установлено в тип класса CarService.

Второй компонент (serviceProcessInstaller1) позволяет установить идентичность, под которой будет выполняться инсталлированная служба. По умолчанию свойство Account установлено в User. Используя окно Properties среды Visual Studio, измените это значение на LocalService (рис. 1.9).

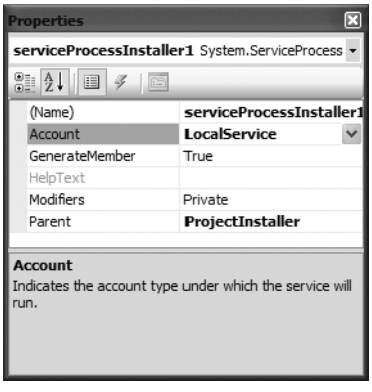

Рис. 1.9. Установка идентичности для `CarService`

Инсталляция `CarService`

Инсталляция `CarService.exe` на определенной машине (локальной или удаленной) требует выполнения двух шагов.

1. Перенос готовой сборки службы (и всех необходимых ей внешних сборок; в данном примере — `CarGeneralAsm.dll`) на удаленную машину.
2. Запуск инструмента командной строки `installutil.exe` с указанием службы в качестве аргумента.

Если первый шаг выполнен, откройте командное окно Visual Studio, перейдите к месту нахождения сборки `CarWinService.exe` и выполните следующую команду (обратите внимание, что тот же инструмент может применяться и для деинсталляции службы):

```
installutil carwinservice.exe
```

Как только служба Windows правильно установлена, можно запускать и конфигурировать ее в аплете Services (Службы) панели управления. Найдя `CarWinService` (рис. 1.10), щелкните на ссылке Start (Запуск) для загрузки и запуска программы службы.

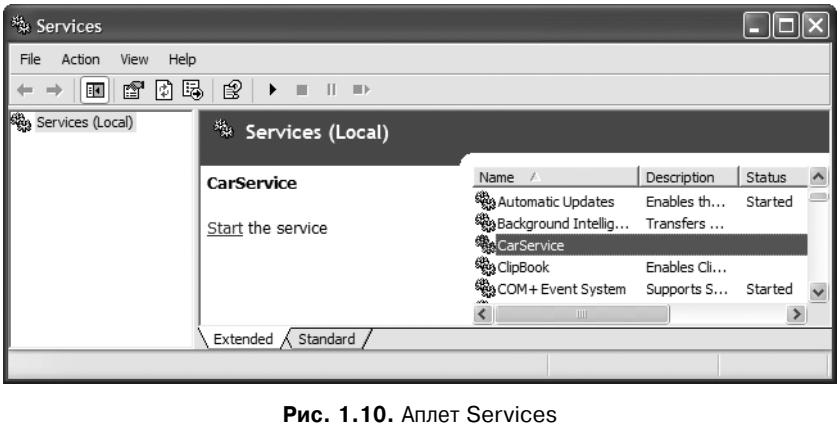

Рис. 1.10. Аплет Services

Исходный код. Проект `CarWinService` расположен в подкаталоге `Bonus Chapter 1`.

Хостинг удаленных объектов с использованием IIS

Хостинг удаленный службы под IIS еще проще, чем построение службы Windows, поскольку IIS предварительно запрограммирован на прием входящих запросов HTTP через порт 80. Теперь, учитывая, что IIS — это веб-сервер, должно быть ясно, что IIS может размещать в себе только удаленные объекты, использующий тип канала `HttpChannel` (в отличие от служб Windows, которые могут также применять тип `TcpChannel`). Если это не кажется ограничением, выполните следующие шаги, чтобы использовать поддержку удаленного взаимодействия IIS.

1. На жестком диске создайте новую папку для хранения `CarGeneralAsm.dll`. Внутри этой папки создайте подкаталог по имени `Bin`. Теперь скопируйте `CarGeneralAsm.dll` в этот подкаталог (например, `C:\IISCarService\Bin`).
2. Откройте аплет Internet Information Services панели управления на машине-хосте.
3. Выполните щелчок правой кнопкой мыши на узле Default Web Site (Веб-узел по умолчанию) и выберите в контекстном меню пункт New⇨Virtual Sirectory (Создать⇨Виртуальный каталог).
4. Создайте виртуальный каталог, отображаемый на созданную ранее папку (`C:\IISCarService`). Остальные установки по умолчанию, представленные мастером создания виртуальных каталогов (New Virtual Directory Wizard) можно оставить без изменений.
5. И, наконец, создайте новый конфигурационный файл по имени `web.config` для управления тем, как этот виртуальный каталог должен регистрировать удаленный тип (см. код ниже). Сохраните этот файл в корневой папке (в данном примере — в `C:\IISCarService`).

```
<configuration>
   <system.runtime.remoting>
      <application>
         <service>
            <wellknown mode="Singleton"
                type="CarGeneralAsm.CarProvider, CarGeneralAsm"
                objectUri="carprovider.soap" />
         </service>
         <channels>
            <channel ref="http"/>
         </channels>
      </application>
   </system.runtime.remoting>
</configuration>
```

Теперь сборка `CarGeneralAsm.dll` сконфигурирована для принятия запросов HTTP под IIS, и можно обновить файл `*.config` клиентской стороны, как показано ниже (используя имя IIS-хоста):

```
<configuration>
   <system.runtime.remoting>
      <application>
         <client displayName = "CarClient">
            <wellknown
              type="CarGeneralAsm.CarProvider, CarGeneralAsm"
              url="http://NameTheRemoteIISHost/IISCarHost/carprovider.soap"/>
         </client>
         <channels>
            <channel ref="http"/>
```

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

После этого можно запускать клиентское приложение, как и раньше.

Асинхронное удаленное взаимодействие

Для завершения картины давайте рассмотрим, как вызывать члены удаленного типа в асинхронном режиме. Ранее была представлена тема асинхронного вызова методов с использованием типов делегатов. Как и можно было ожидать, если клиентская сборка желает вызывать удаленный объект асинхронно, то первый шаг состоит в определении специального делегата для представления интересующего удаленного метода. В этот момент вызывающий код может использовать любой из рассмотренных ранее приемов, чтобы вызывать и принимать возвращаемое значение метода.

В качестве простой иллюстрации создайте новое консольное приложение (`AsyncWKOCarProviderClient`) и установите ссылку на первую итерацию сборки `CarGeneralAsm.dll`. Затем обновите класс `Program` следующим образом:

```
class Program
{
  // Делегат для метода GetAllAutos().
  internal delegate List<JamesBondCar> GetAllAutosDelegate();
  static void Main(string[] args)
  {
    Console.WriteLine("Client started! Hit enter to end");
    RemotingConfiguration.Configure
        ("AsyncWKOCarProviderClient.exe.config");

    // Создать поставщика машин.
    CarProvider cp = new CarProvider();

    // Создать делегат.
    GetAllAutosDelegate getCarsDel =
        new GetAllAutosDelegate(cp.GetAllAutos);

    // Вызвать GetAllAutos() асинхронно.
    IAsyncResult ar = getCarsDel.BeginInvoke(null, null);

    // Симулировать активность клиента.
    while(!ar.IsCompleted)
       { Console.WriteLine("Client working..."); }

    // Все готово! Получить возвращаемое значение от делегата.
    List<JamesBondCar> allJBCs = getCarsDel.EndInvoke(ar);

    // Использовать автомобили из List.
    foreach(JamesBondCar j in allJBCs)
       UseCar(j);
    Console.ReadLine();
  }
}
```

Обратите внимание, что клиентское приложение сначала объявляет делегат, соответствующий сигнатуре метода `GetAllAutos()` удаленного типа `CarProvider`. Когда делегат создан, вы передаете имя метода для вызова (`GetAllAutos`), как обычно. Затем вы инициируете метод `BeginInvoke()`, кэшируете результирующий интерфейс `IAsyncResult` и эмулируете некоторую деятельность клиентской стороны (напомним, что свойство `IAsyncResult.IsCompleted` позволяет отслеживать завершение обработки ассоциированным методом). И, наконец, как только работа клиента завершена, вы

получаете List<>, возвращенный методом CarProvider.GetAllAutos(), посредством вызова члена EndInvoke(), после чего передаете каждый экземпляр JamesBondCar статической вспомогательной функции по имени UseCar():

```
public static void UseCar(JamesBondCar j)
{
  Console.WriteLine("Can car fly? {0}", j.isFlightWorthy);
  Console.WriteLine("Can car swim? {0}", j.isSeaWorthy);
}
```

Красота типов делегатов .NET состоит в том, что здесь используется логика асинхронного вызова удаленных методов, идентичная процессу вызова локальных методов.

Исходный код. Проект AsyncWKOCarProviderClient находится в подкаталоге Bonus Chapter 1.

Роль атрибута [OneWay]

Предположим, что CarProvider имеет метод по имени AddCar(), принимающий входной параметр типа JamesBondCar и не возвращающий ничего. Последнее обстоятельство (не возвращает *ничего*) является ключевым. Как вы можете предположить исходя из имени класса System.Runtime.Remoting.Messaging.OneWayAttribute, слой удаленного взаимодействия .NET Remoting передает вызов удаленному однонаправленному методу, но не заботится об установке инфраструктуры, используемой для возврата значения (отсюда — *однонаправленный*). Ниже показаны необходимые изменения.

```
// Место расположения объявления атрибута [OneWay].
using System.Runtime.Remoting.Messaging;
...
namespace CarGeneralAsm
{
  public class CarProvider : MarshalByRefObject
  {
  ...
    // Клиент может вызвать этот метод и забыть.
    [OneWay]
    public void AddCar(JamesBondCar newJBC)
    { theJBCars.Add(newJBC);}
  }
}
```

Вызывающий код имеет дело с этим методом, как обычно:

```
// Создать поставщика машин.
CarProvider cp = new CarProvider();

// Добавить новую машину.
cp.AddCar(new JamesBondCar("Zippy", 200, false, false));
```

С точки зрения клиента вызов AddCar() полностью асинхронный, поскольку CLR обеспечит использование фонового потока для удаленного запуска метода. Учитывая, что AddCar() декорирован атрибутом [OneWay], клиент не может получить никакого возвращаемого значения от этого вызова. Поскольку AddCar() возвращает void, это не является проблемой.

В дополнение к этому ограничению также имейте в виду, что если есть метод [OneWay], который определяет выходные параметры или параметры, передаваемые по ссылке (через ключевое слово out или ref), то вызывающий код не сможет получить модификации, выполненные этим методом. Более того, если метод [OneWay] сгенерирует

исключение (любого типа), вызывающий код останется в полном неведении об этом. По сути, удаленные объекты могут помечать избранные методы атрибутом `[OneWay]`, чтобы позволить вызывающему коду ни о чем не заботиться после их вызова.

Резюме

В этой главе вы узнали, как конфигурировать различные сборки .NET для использования объектов через границы приложений. Как было показано, удаленный объект может быть сконфигурирован как тип MBV или MBR. Этот выбор в конечном итоге управляет реализацией удаленного типа в домене клиентского приложения (в виде копии или прозрачного прокси).

После конфигурирования типа для работы в виде сущности MBR появляется дальнейший выбор — WKO или CAO, одиночный вызов или объект-одиночка и т.д. В главе рассматривались все возможные варианты. К тому же вы ознакомились с процессом отслеживания времени жизни удаленного объекта через использование “аренды” и ее спонсорства. И, наконец, мы еще раз коснулись роли типов делегатов .NET для понимания асинхронных вызовов удаленных методов (процесс, к счастью, идентичный асинхронным вызовам методов локального объекта).