

# 54

## Технология .NET Remoting

### В ЭТОЙ ГЛАВЕ...

- Обзор .NET Remoting
- Контексты, используемые для группировки объектов с похожими требованиями к выполнению
- Реализация простого удаленного объекта, клиента и сервера
- Архитектура .NET Remoting
- Конфигурационные файлы .NET Remoting
- Хостинг объектов .NET Remoting в ASP.NET
- Использование утилиты Soapsuds для доступа к метаданным удаленных объектов
- Асинхронный вызов методов .NET Remoting
- Вызов методов на клиенте с помощью событий
- Использование `CallContext` для автоматической передачи данных на сервер

Эта глава посвящена технологии .NET Remoting, которая может использоваться для доступа к объектам в другом домене приложений (например, на другом сервере). Она предлагает более скоростной формат коммуникаций между приложениями .NET как на стороне клиента, так и на стороне сервера.

В главе вы разработаете объекты, клиенты и серверы .NET Remoting, используя каналы HTTP, TCP и IPC. Вы программно сконфигурируете клиент и сервер перед изменением приложения, чтобы применять конфигурационные файлы, для чего требуются только несколько методов .NET Remoting. Вы также напишете небольшие программы для использования .NET Remoting асинхронно и для вызова обработчиков в клиентских приложениях.

Классы .NET Remoting находятся в пространстве имен `System.Runtime.Remoting` и во вложенных подпространствах. Многие из этих классов заключены в сборку ядра `mscorlib`, а те, которые нужны только для межсетевых коммуникаций, доступны в сборке `System.Runtime.Remoting`.

## **Для чего используется .NET Remoting**

.NET Remoting — это технология коммуникаций между разными доменами приложений. Коммуникации между доменами приложений с помощью .NET Remoting могут происходить внутри одного процесса, между процессами в одной системе или между процессами в разных системах.

Для коммуникаций между клиентским и серверным приложением могут применяться несколько различных технологий. Для программирования приложений можно использовать сокеты или вспомогательные классы из пространства имен `System.Net`, которые облегчают работу с протоколами, IP-адресами и номерами портов (см. главу 24). В рамках этой технологии данные всегда должны передаваться по сети. Передаваемые данные могут быть оформлены в собственный специальный протокол, когда пакет интерпретируется сервером, так что сервер знает, какие методы должны быть вызваны. Придется не только иметь дело с передаваемыми данными, но также самостоятельно создавать потоки выполнения.

Используя веб-службы ASP.NET, можно отправлять сообщения по сети. Веб-службы обеспечивают независимость от платформы. Кроме того, достигается слабая связь между клиентом и сервером, а это означает облегчение работы с версиями. Веб-службы ASP.NET рассматриваются в главе 55.

Благодаря .NET Remoting, всегда имеется слабая связь между клиентом и сервером, потому что ими разделяются объекты одних и тех же типов. В рамках .NET Remoting функциональность объектов CLR перемещается в методы, которые вызываются между разными доменами приложений.

Функциональность .NET Remoting может быть описана типами приложений и поддерживаемых протоколов, а также *CLR Object Remoting*.

CLR Object Remoting (удаленное взаимодействие объектов CLR) — важный аспект технологии .NET Remoting. С удаленными объектами могут использоваться все языковые структуры (такие как конструкторы, делегаты, интерфейсы, методы, свойства и поля). Технология .NET Remoting расширяет функциональность объектов CLR на всю сеть. Данная технология имеет дело с активизацией, распределенной идентичностью, временем жизни и контекстами вызова. В этом состоит ее главное отличие от веб-служб XML. В веб-службах XML объекты абстрагируются, и клиенту не нужно знать реальные типы объектов на сервере.

Наилучший выбор для сетевых коммуникаций на сегодняшний день описан в главе 43, посвященной WCF. Технология WCF предлагает такие средства веб-служб ASP.NET, как независимость от платформы, обеспечивая высокую производительность и гибкость .NET Remoting для сетевых коммуникаций .NET. Область, в которой .NET Remoting все еще

имеет преимущество — это коммуникации между доменами приложений внутри процесса. Технология MAF (System.AddIn), которая обсуждается в главе 50, использует внутри себя .NET Remoting. И, конечно же, существует множество готовых решений .NET, основанных на .NET Remoting, которые ничего не выигрывают от переписывания в новой технологии.



*Хотя протокол SOAP предлагается .NET Remoting, не следует полагать, что этот протокол может применяться для обеспечения взаимодействия между разными платформами. Стил SOAP Document в .NET Remoting недоступен. Технология .NET Remoting предназначена для приложений .NET на обеих сторонах — на клиенте и сервере. Если нужна возможность взаимодействия, используйте веб-службы.*

.NET Remoting обладает исключительно гибкой архитектурой, которая может применяться с любыми типами приложений, работающими с любым транспортом, использующим любую кодировку содержимого.

Применение SOAP и HTTP — лишь один из способов вызова удаленных объектов. Транспортные каналы являются подключаемыми и могут легко заменяться. Вместе с .NET 4 вы получаете каналы HTTP, TCP и IPC, представленные классами `HttpChannel`, `TcpChannel` и `IpcChannel`, соответственно. Можно строить транспортные каналы на основе UDP, IPX, SMTP, механизма разделенной памяти и очередей сообщений — выбор полностью за вами.



*Термин “подключаемый” (pluggable) часто применяется в .NET Remoting. Подключаемость означает, что определенная часть спроектирована так, что может быть заменена специальной реализацией.*

Для транспортировки параметров вызовов метода используется полезная нагрузка (payload). Кодировка этой полезной нагрузки также может быть заменена. Microsoft предоставляет механизмы кодирования SOAP и двоичных данных. Можно использовать либо форматировщик SOAP с каналом HTTP, либо HTTP с двоичным форматировщиком. Разумеется, оба эти форматировщика могут использоваться с каналом TCP.



*Хотя SOAP используется в .NET Remoting, следует помнить, что поддерживается только стил SOAP RPC, в то время как веб-службы ASP.NET поддерживают как стил DOC (по умолчанию), так и стил RPC. В новых версиях SOAP стил RPC считается устаревшим.*

Технология .NET Remoting не только позволяет применять серверную функциональность в каждом приложении .NET. Эту технологию можно также использовать повсеместно (независимо от того, строится консольное приложение, Windows-приложение, служба Windows или компонент COM+). Вдобавок .NET Remoting также является хорошей технологией для одноранговых (peer-to-peer) коммуникаций.

## Объяснение терминов .NET

Технология .NET Remoting может применяться для обращения к объектам в другом домене. Эта технология всегда может использоваться там, где два объекта существуют внутри одного процесса, в разных процессах или на разных системах.

Удаленные сборки могут быть сконфигурированы для локальной работы в домене приложения или как часть удаленного приложения. Если сборка является частью удаленного приложения, вместо реального объекта клиент получает прокси для взаимодействия. Прокси является представителем удаленного объекта в клиентском процессе, использует

мым клиентским приложением для вызова его методов. Когда клиент вызывает метод на прокси, тот пересылает сообщение в канал, связывающий его с удаленным объектом.

Приложения .NET обычно функционируют внутри домена приложений. Домен приложения может восприниматься как подпроцесс внутри процесса. Традиционно процессы использовали в изолированных границах. Межпроцессные коммуникации необходимы для обеспечения взаимодействия приложений друг с другом. В .NET домен приложения — это новая безопасная область внутри процесса, потому что код MSIL безопасен в отношении типов и верифицируем. Как описано в главе 18, различные приложения могут выполняться внутри одного процесса, но в разных доменах приложений. Объекты, находящиеся в пределах одного домена приложений, могут взаимодействовать непосредственно. Прокси необходим для доступа к объектам в разных доменах приложений.

В следующем списке предлагается обзор ключевых элементов архитектуры .NET Remoting.

- *Удаленный объект* — это объект, выполняющийся на сервере. Клиент не вызывает методы этого объекта напрямую, а вместо этого использует прокси. В .NET легко отличить удаленные объекты от локальных: любой класс, который унаследован от `MarshalByRefObject`, никогда не покидает своего домена приложений. Клиент может вызывать методы удаленного объекта через прокси.
- *Канал* предназначен для коммуникаций между клиентом и сервером. У канала есть серверная и клиентская части. В .NET Framework 4 предлагаются типы каналов, которые взаимодействуют через TCP, HTTP или IPC. Можно создать собственный специальный канал, который взаимодействует посредством другого протокола.
- *Сообщения* посылаются в канал; они создаются для коммуникаций между клиентом и сервером. Эти сообщения содержат информацию об удаленном объекте, именах вызываемых методов и всех их аргументах.
- *Форматировщик* определяет то, как сообщения передаются в канал. В .NET 4 доступен форматировщик SOAP и двоичный форматировщик. Форматировщик SOAP применяется для коммуникаций с веб-службами, которые не основаны на .NET Framework. Двоичные форматировщики намного быстрее и могут эффективно использоваться в среде корпоративной сети. Разумеется, можно создавать и собственные специальные форматировщики.
- *Поставщик форматировщиков* используется для ассоциации форматировщика с каналом. При создании канала можно указать используемого поставщика форматировщиков, а это, в свою очередь, определяет форматировщик, применяемый для передачи данных в канал.
- Клиент вызывает методы на прокси, а не на удаленном объекте. Существуют два типа прокси: *прозрачный прокси* (transparent proxy) и *реальный прокси* (real proxy). Для клиента прозрачный прокси выглядит как удаленный объект. На прозрачном прокси клиент может вызывать методы, реализованные удаленными объектами. В свою очередь, прозрачный прокси вызывает метод `Invoke()` на реальном прокси. Метод `Invoke()` использует приемник сообщений (message sink) для передачи сообщения в канал.
- *Приемник сообщений*, или просто *приемник* — это объект-перехватчик. Перехватчики используются как клиентом, так и сервером. Приемник ассоциирован с каналом. Реальный прокси использует приемник сообщений для передачи сообщения в канал, так что приемник может осуществить какое-то вмешательство перед тем, как сообщение поступит в канал. В зависимости от места применения, его называют приемником агентов (envoy sink), приемником серверного контекста, приемником объектного контекста и так далее.

- Клиент может использовать *активатор* для создания удаленного объекта на сервере или получения прокси активизированного на сервере объекта.
- `RemotingConfiguration` — это служебный класс для конфигурирования удаленных серверов и клиентов. Он применяется либо для чтения конфигурационных файлов, либо для динамического конфигурирования удаленных объектов.
- `ChannelServices` — это служебный класс, используемый для регистрации каналов с последующим направлением сообщений в них.

На рис. 54.1 показана концептуальная картина взаимодействия всего перечисленного.

#### Домен приложений

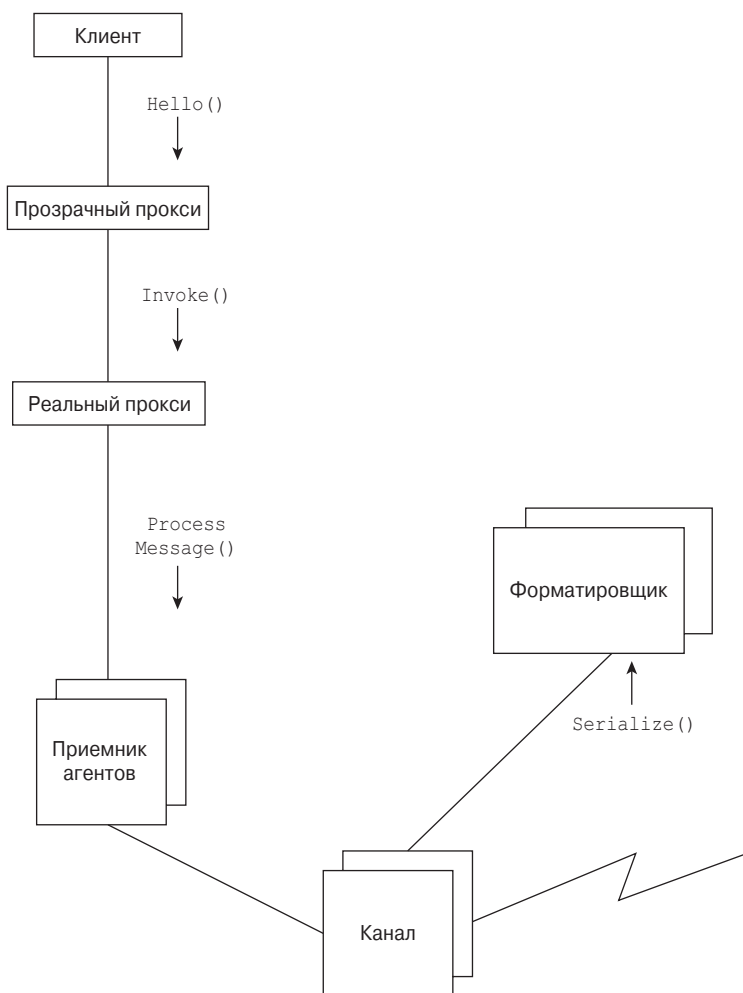


Рис. 54.1. Взаимодействие ключевых элементов архитектуры .NET Remoting

## Коммуникации клиентской стороны

Когда клиент вызывает методы на удаленном объекте, на самом деле он вызывает их на прозрачном прокси вместо реального объекта. Прозрачный прокси выглядит, как реальный объект — он реализует общедоступные методы реального объекта. Прозрачный прокси получает сведения об общедоступных методах реального объекта через механизм рефлексии, посредством которого читает метаданные из сборки.

В свою очередь, прозрачный прокси вызывает реальный прокси. Реальный прокси отвечает за отправку сообщения в канал. Этот прокси является подключаемым: он может заменяться специальной реализацией. Такая специальная реализация может использоваться для записи журнала, реализации другого способа поиска канала и т.д.

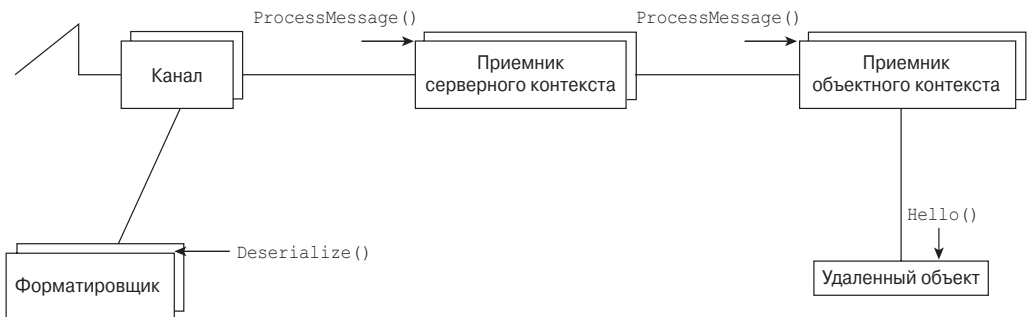
Реализация по умолчанию реального прокси находит коллекцию (или цепочку) приемников агентов и передает сообщение первому из них. Приемник агентов может вмешаться и изменить сообщение. Примерами таких приемников могут быть отладочные приемники, приемники безопасности и приемники синхронизации.

Последний из приемников агентов посылает сообщение в канал. Форматировщик определяет, как сообщения передаются по сети. Как упоминалось ранее, в .NET Framework доступны форматировщики SOAP и двоичные форматировщики. Форматировщик также является подключаемым. Канал отвечает за подключение к прослушивающему сокету на сервере и передачу форматированных данных. С помощью специального канала можно делать что-нибудь другое; для этого понадобится лишь реализовать код и предпринять необходимые действия для передачи данных противоположной стороне.

## Коммуникации серверной стороны

Мы продолжим рассмотрением серверной стороны, которая показана на рис. 54.2.

1. Канал принимает форматированные сообщения с клиента и использует форматировщик для демаршализации данных SOAP или двоичных данных в сообщения. Затем канал вызывает приемники серверного контекста.
2. Приемники серверного контекста образуют цепочку, в которой последний приемник передает вызов приемникам объектного контекста.
3. Последний приемник объектного контекста вызывает метод в удаленном объекте.



**Рис. 54.2.** Взаимодействие элементов серверной стороны

Обратите внимание, что приемники объектного контекста привязаны к объектному контексту, а приемники серверного контекста — к контексту сервера. Один приемник серверного контекста может применяться для доступа к множеству объектных приемников.

Протоколирующий приемник — лишь один пример, где этот механизм может быть полезен. С помощью протоколирующего приемника можно фиксировать отправляемые и принимаемые сообщения. Проверка доступности сервера и динамическое переключение на другой сервер может быть выполнено с помощью приемника на клиентской системе.



*.NET Remoting — исключительно настраиваемая технология: можно заменять реальный прокси, добавлять объекты приемников или заменять форматировщик и канал. Разумеется, можно также использовать то, что уже готово.*

Учитывая все эти уровни, вы можете обеспокоиться по поводу накладных расходов, но на самом деле их немного, если здесь ничего не происходит. Если вы добавите собственную функциональность, то накладные расходы будут зависеть от нее.

## Контексты

Прежде чем мы рассмотрим применение .NET Remoting для построения серверов и клиентов, взаимодействующих через сеть, в этом разделе мы обратимся к случаям, когда канал необходим внутри домена приложения: вызов объектов между контекстами.

Если вам приходилось ранее писать компоненты COM+, то уже должно быть известно о контекстах COM+. Контексты в .NET очень похожи. Контекст (context) — это ограниченная область, содержащая коллекцию объектов. Подобно контексту COM+, объекты в такой коллекции требуют выполнения некоторых правил пользования, которые определены в атрибутах контекста.

Как известно, один процесс может иметь несколько доменов приложений. Домен приложения — это нечто подобное подпроцессам с границами безопасности. Домены приложений обсуждаются в главе 18.

Домен приложения может иметь разные контексты. Контекст используется для группирования объектов со сходными требованиями к выполнению. Контексты составляются из набора свойств и используются для вмешательства (или перехвата (interception)): когда к *привязанному к контексту объекту* производится обращение из другого контекста, то *перехватчик* (interceptor) может выполнить некоторую работу перед тем, как вызов достигнет объекта. Это может применяться, например, для синхронизации потоков, транзакций и управления безопасностью.

Класс, унаследованный от MarshalByRefObject, привязан к домену приложений. За пределами домена приложений для доступа к объекту необходим прокси. Класс, унаследованный от ContextBoundObject, который сам наследуется от MarshalByRefObject, привязан к контексту. Вне контекста для доступа к объекту нужен прокси.

Привязанные к контексту объекты могут иметь *контекстные атрибуты*. Привязанный к контексту объект без контекстных атрибутов создается в контексте его создателя. Привязанный к контексту объект с контекстными атрибутами создается в новом контексте или в контексте создателя, если атрибуты совместимы с ним.

Для дальнейшего понимания контекстов вы должны быть знакомы со следующими понятиями.

- Создание домена приложения создает *контекст по умолчанию* для этого домена приложения. Если создается новый объект, которому нужны свойства другого контекста, создается новый контекст.
- *Атрибуты контекста* могут назначаться классу, унаследованному от ContextBoundObject. Можно создать специальный класс атрибута, реализовав интерфейс IContextAttribute. В .NET Framework имеется один класс атрибута контекста в пространстве имен System.Runtime.Remoting.Contexts по имени SynchronizationAttribute.

- Атрибуты контекста определяют *свойства контекста*, которые необходимы объекту. Класс свойства контекста реализует интерфейс `IContextProperty`. Активные свойства добавляют приемники сообщений в цепочку вызовов. Класс `ContextAttribute` реализует оба интерфейса — `IContextProperty` и `IContextAttribute` — и может служить базовым классом для специальных атрибутов.
- *Приемник сообщений* — это перехватчик вызова метода. С его помощью вызовы методов могут быть перехвачены. К приемникам сообщений могут добавляться свойства.

## Активизация

Новый контекст создается в том случае, если экземпляр создаваемого класса нуждается в контексте, отличном от вызывающего контекста. Классы атрибутов, ассоциированные с целевым классом, запрашиваются, если все свойства текущего контекста приемлемы. Если какие-то из них неприемлемы, исполняющая среда опрашивает все классы свойств, ассоциированные с классом атрибута, и создает новый контекст. Исполняющая среда затем опрашивает классы свойств о приемниках, которые они хотят установить. Класс свойства может реализовать один из интерфейсов `IContributeXXXSink` для построения объекта приемника. Несколько из этих интерфейсов доступны для прохода с разнообразными приемниками.

## Атрибуты и свойства

Свойства контекста определяются в атрибутах контекста. Класс контекста атрибута — это, прежде всего, атрибут. Дополнительные сведения об атрибутах можно найти в главе 14, посвященной рефлексии. Классы атрибутов контекста должны реализовывать интерфейс `IContextAttribute`. Специальный класс контекста атрибута может наследоваться от `ContextAttribute`, потому что этот класс уже имеет реализацию этого интерфейса по умолчанию.

В .NET Framework определены два класса атрибутов контекстов: `System.Runtime.Remoting.Contexts.SynchronizationAttribute` и `System.Runtime.Remoting.Activation.UrlAttribute`. Атрибут `Synchronization` определяет требования синхронизации; он задает свойство синхронизации, которое необходимо объекту. С помощью этого атрибута можно указать, что множество потоков не могут обращаться к объекту одновременно, но поток, обращающийся к объекту, может заменяться.

В конструкторе этого атрибута можно установить одно из следующих значений:

- `NOT_SUPPORTED` определяет, что экземпляры этого класса не могут создаваться в контексте, где установлена синхронизация;
- `REQUIRED` указывает на обязательность контекста синхронизации;
- `REQUIRES_NEW` всегда создает новый контекст;
- `SUPPORTED` означает, что независимо от того, какой имеется контекст, объект может находиться в нем.

## Коммуникации между контекстами

Каким образом происходят коммуникации между контекстами? Клиент использует прокси вместо реального объекта. Прокси создает сообщение, передаваемое в канал, а приемники могут его перехватывать. Звучит знакомо? Так и должно быть. Тот же механизм применяется для коммуникаций между разными доменами приложений на разных системах. Канал TCP или HTTP не обязателен для коммуникаций между контекстами, но канал здесь также используется. Класс `CrossContextChannel` может использовать ту же виртуальную память как на серверной, так и на клиентской стороне канала, и для пересечения контекстов форматировщики не требуются.



## Удаленные объекты, клиенты и серверы

Прежде чем углубляться в детали архитектуры .NET Remoting, в этом разделе мы кратко рассмотрим удаленные объекты и построим очень простое клиент-серверное приложение, использующее удаленный объект. После этого мы подробно обсудим все необходимые шаги и варианты.

На рис. 54.3 показаны основные классы .NET Remoting на стороне клиента и сервера. Удаленный объект, который будет реализован, называется `Hello`. Кроме того, `HelloServer` — главный класс приложения на сервере, а `HelloClient` используется для клиента.

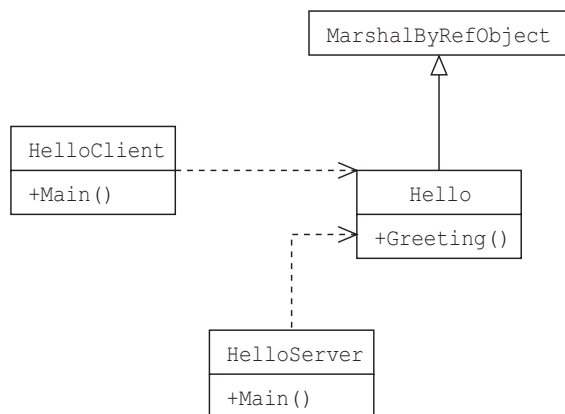


Рис. 54.3. Диаграмма классов простого клиент-серверного приложения

## Удаленные объекты

Удаленные объекты требуются для реализации распределенных вычислений. Объект, который планируется вызывать удаленно из другой системы, должен наследоваться от `System.MarshalByRefObject`. Объекты `MarshalByRefObject` *ограничены доменом приложений*, в котором они созданы. Это значит, что они никогда не выходят за пределы домена приложений; для доступа к удаленному объекту из другого домена приложений используется прокси-объект. При этом другой домен приложений может существовать внутри того же процесса, в другом процессе или на другой системе.

Удаленный объект имеет *распределенную идентичность*. По этой причине ссылка на него может передаваться другим клиентам, и они будут обращаться к тому же объекту. Прокси знает об идентичности удаленного объекта.

Вдобавок к методам, унаследованным от класса `Object`, класс `MarshalByRefObject` имеет методы для инициализации и получения служб времени жизни. Службы времени жизни (lifetime services) определяют, насколько долго существует удаленный объект. Службы времени жизни и средства аренды рассматриваются далее в главе.

Чтобы увидеть .NET Remoting в действии, создайте библиотеку классов для удаленного объекта. Класс `Hello` унаследован от `System.MarshalByRefObject`. В конструкторе на консоль выводится сообщение, в котором указывается информация о времени жизни объекта. Добавьте также метод `Greeting()`, который будет вызываться со стороны клиента.

Чтобы не путать сборку с классом в последующих разделах, назначьте им разные имена в аргументах вызовов метода. Именем сборки будет `RemoteHello`, а именем класса — `Hello`.

```

using System;
namespace Wrox.ProCSharp.Remoting
{
    public class Hello : System.MarshalByRefObject
    {
        public Hello()
        {
            Console.WriteLine("Вызван конструктор");
        }
        public string Greeting(string name)
        {
            Console.WriteLine("Вызван метод Greeting");
            return "Hello, " + name;
        }
    }
}

```

---

Фрагмент кода RemoteHello\RemoteHello\Hello.cs

---

## Простое серверное приложение

Для сервера создайте новое консольное приложение C# по имени HelloServer. Чтобы пользоваться классом TcpServerChannel, необходимо сослаться на сборку System.Runtime.Remoting. Также потребуется ссылка на сборку RemoteHello, которая была создана ранее в разделе “Удаленные объекты”.

В методе Main() создается объект типа System.Runtime.Remoting.Channels.Tcp.TcpServerChannel с портом номер 8086. Этот канал зарегистрирован с классом System.Runtime.Remoting.Channels.ChannelServices для обеспечения его доступности удаленным объектам. Тип удаленного объекта зарегистрирован вызовом метода RemotingConfiguration.RegisterWellKnownServiceType().

В примере клиент использует тип URI класса удаленного объекта с указанием режима. Режим WellKnownObjectMode.SingleCall означает, что новый экземпляр создается для каждого вызова метода; в примере приложения для удаленного объекта никакого состояния не удерживается.



*.NET Remoting позволяет создавать удаленные объекты без состояния и с состоянием. В первом примере применяется хорошо известный, однократно вызываемый объект, который не сохраняет состояние. Другой тип объекта называется активизируемым клиентом. Активизируемые клиентом объекты хранят состояние. Далее в этой главе, при рассмотрении последовательности активизации объектов, будут даны более подробные сведения об этих отличиях, а также о том, как используются эти типы объектов.*

После регистрации удаленного объекта необходимо сохранять сервер работающим до тех пор, пока не будет нажата любая клавиша.

```

using System;
using System.Collections.Generic;
using System.Runtime.Remoting;
using System.Runtime.Remoting.Channels;
using System.Runtime.Remoting.Channels.Tcp;
namespace Wrox.ProCSharp.Remoting
{
    class Program
    {
        static void Main()
        {

```

```

        var channel = new TcpServerChannel(8086);
        ChannelServices.RegisterChannel(channel, true);
        RemotingConfiguration.RegisterWellKnownServiceType(
            typeof(Hello), "Hi", WellKnownObjectMode.SingleCall);
        Console.WriteLine("Для завершения нажмите Enter");
        Console.ReadLine();
    }
}
}

```

---

Фрагмент кода *RemoteHello\HelloServer\Program.cs*

---

## Простое клиентское приложение

Клиентом будет также консольное приложение под названием `HelloClient`. В этом проекте понадобится сослаться на сборку `System.Runtime.Remoting`, чтобы использовать класс `TcpClientChannel`. Вдобавок нужно сослаться на сборку `RemoteHello`. Хотя объект будет создан на удаленном сервере, эта сборка необходима клиенту, чтобы прокси мог прочитать информацию о типе во время выполнения.

В клиентской программе создайте объект `TcpClientChannel`, зарегистрированный в `ChannelServices`. Для `TcpChannel` можно использовать конструктор по умолчанию, выбирающий свободный порт. Затем с помощью класса `Activator` возвращается прокси удаленного объекта.

Прокси имеет тип `System.Runtime.Remoting.Proxies.__TransparentProxy`. Этот объект выглядит как реальный объект, потому что предоставляет те же методы. Прозрачный прокси использует реальный прокси для отправки сообщений в канал.

```

➡ using System;
   using System.Runtime.Remoting.Channels;
   using System.Runtime.Remoting.Channels.Tcp;

   namespace Wrox.ProCSharp.Remoting
   {
       class Program
       {
           static void Main()
           {
               Console.WriteLine("Нажмите Enter после запуска сервера");
               Console.ReadLine();
               ChannelServices.RegisterChannel(new TcpClientChannel(), true);
               Hello obj = (Hello)Activator.GetObject(
                   typeof(Hello), "tcp://localhost:8086/Hi");

               if (obj == null)
               {
                   Console.WriteLine("не удастся найти сервер");
                   return;
               }

               for (int i=0; i< 5; i++)
               {
                   Console.WriteLine(obj.Greeting("Stephanie"));
               }
           }
       }
   }

```

---

Фрагмент кода *RemoteHello\HelloClient\Program.cs*

---



*Прокси — это объект, используемый клиентским приложением вместо удаленного объекта. Прокси, которые применяются в главе 55, имеют функциональность, похожую на прокси из настоящей главы. Однако реализации прокси для веб-служб и прокси для .NET Remoting существенно отличаются.*

Теперь можно запустить сервер, а затем клиент. Внутри консоли клиента текст Hello, Stephanie появляется пять раз. В окне консоли сервера можно видеть показанный ниже вывод. С каждым новым вызовом метода создается новый экземпляр, так как был выбран режим активизации WellKnownObjectMode.SingleCall.

```
Для завершения нажмите Enter
Вызван конструктор
Вызван метод Greeting
Вызван конструктор
Вызван метод Greeting
Вызван конструктор
Вызван метод Greeting
Вызван конструктор
Вызван метод Greeting
Вызван конструктор
Вызван метод Greeting
```

## Архитектура .NET Remoting

Теперь, увидев простое приложение клиента и сервера в действии, давайте взглянем на архитектуру .NET Remoting перед тем, как углубиться в детали. Отталкиваясь от ранее созданной программы, рассмотрим архитектуру и механизмы расширяемости.

В этом разделе будут раскрыты следующие темы.

- Функциональность и конфигурирование канала.
- Форматировщики и их применение.
- Служебные классы ChannelServices и RemotingConfiguration.
- Различные способы активизации удаленных объектов и использование в .NET Remoting объектов с состоянием и без.
- Функциональность приемника сообщений.
- Передача объектов по значению и по ссылке.
- Управление жизненным циклом объектов с состоянием через механизм аренды .NET Remoting.

## Каналы

Канал используется для коммуникаций между клиентом .NET и сервером. В .NET Framework 4 определены классы каналов, которые взаимодействуют с использованием TCP, HTTP или IPC. Разрешено создавать специальные каналы для других протоколов.

Канал HTTP использует для коммуникаций протокол HTTP. Поскольку обычно в брандмауэрах порт 80 открыт, чтобы клиенты могли иметь доступ к веб-серверам, веб-службы .NET Remoting могут прослушивать порт 80 и потому легко использоваться клиентами.

Также можно применять канал TCP в Интернете, но в этом случае брандмауэры должны быть сконфигурированы так, чтобы клиенты могли обращаться к указанному порту, который используется каналом TCP. В среде корпоративной сети канал TCP обеспечивает более эффективных коммуникаций, чем канал HTTP.

Канал IPC лучше всего подходит для коммуникаций в пределах одной системы, но между разными процессами. Он использует механизм межпроцессного взаимодействия Windows и потому работает быстрее других каналов.

Выполнение вызова метода на удаленном объекте заставляет объект клиентского канала отправлять сообщение удаленному объекту канала.

Как серверное, так и клиентское приложение должно создать канал. В следующем коде показано создание `TcpServerChannel` на стороне сервера:

```
using System.Runtime.Remoting.Channels.Tcp;
...
TcpServerChannel channel = new TcpServerChannel(8086);
```

Порт, на котором выполняет прослушивание сокет TCP, указан в аргументе конструктора. Серверный канал должен специфицировать хорошо известный порт, и клиент должен использовать этот порт для обращения к серверу. Однако чтобы создать `TcpClientChannel` на клиенте, не обязательно указывать хорошо известный порт. Конструктор `TcpClientChannel` по умолчанию выбирает доступный порт, который передается серверу во время соединения, так что сервер может затем вернуть данные клиенту.

Создание нового экземпляра канала немедленно переключает сокет в состояние ожидания, что можно проверить с помощью команды `netstat -a`.



*Очень полезный инструмент для проверки того, какие данные переданы по сети, является `tcpTrace`. Утилита `tcpTrace` доступна для загрузки по адресу [www.pocketsoap.com/tcptrace](http://www.pocketsoap.com/tcptrace).*

Каналы HTTP используются аналогично каналам TCP. Можно указывать порт, на котором сервер будет создавать сокет для прослушивания.

Сервер может прослушивать несколько каналов. Этот код создает и регистрирует каналы HTTP, TCP и IPC:

```
var tcpChannel = new TcpServerChannel(8086);
var httpChannel = new HttpServerChannel(8085);
var ipcChannel = new IpcServerChannel("myIPCPort");
// Регистрация каналов
ChannelServices.RegisterChannel(tcpChannel, true);
ChannelServices.RegisterChannel(httpChannel, false);
ChannelServices.RegisterChannel(ipcChannel, true);
```

Класс канала должен реализовывать интерфейс `IChannel`. Интерфейс `IChannel` имеет два свойства, которые описаны ниже.

- `ChannelName` — свойство только для чтения, возвращающее имя канала. Имя канала зависит от типа; например, канал HTTP называется HTTP.
- `ChannelPriority` — свойство только для чтения. Для коммуникаций между клиентом и сервером может использоваться более одного канала. Приоритет определяет порядок канала. На клиенте канал с более высоким приоритетом выбирается первым для подключения к серверу. Чем выше значение приоритета, тем выше приоритет. По умолчанию принято значение 1, но можно указывать отрицательные значения для создания более низких приоритетов.

Дополнительные интерфейсы реализуются в зависимости от того, является канал клиентским или серверным. Серверные версии каналов реализуют интерфейс `IChannelReceiver`, а клиентские — интерфейс `IChannelSender`.

Классы `HttpChannel`, `TcpChannel` и `IPCChannel` могут применяться как для клиента, так и для сервера. Они реализуют интерфейсы `IChannelSender` и `IChannelReceiver`. Эти интерфейсы унаследованы от `IChannel`.

`ISender` клиентской стороны имеет в дополнение к `ISender` единственный метод по имени `CreateMessageSink()`, который возвращает объект, реализующий `IMessageSink`. Интерфейс `IMessageSink` может быть использован для помещения синхронных и асинхронных сообщений в канал. С помощью интерфейса серверной стороны `IChannelReceiver` канал может быть переведен в режим прослушивания вызовом метода `StartListening()` и вновь остановлен вызовом метода `StopListening()`. Для доступа к полученным данным служит свойство `ChannelData`.

Используя свойства классов канала, можно получить информацию о конфигурации каналов. Для обоих каналов предоставляются свойства `ChannelName`, `ChannelProperty` и `ChannelData`. Свойство `ChannelData` применяется для получения информации об URI, которые хранятся в классе `ChannelDataStore`. В `HttpServerChannel` есть также свойство `Scheme`. Ниже приведен код вспомогательного метода `ShowChannelProperties()`, отображающего такую информацию.

```
static void ShowChannelProperties(IChannelReceiver channel)
{
    Console.WriteLine("Имя: {0}", channel.ChannelName);
    Console.WriteLine("Приоритет: {0}", channel.ChannelPriority);
    if (channel is HttpServerChannel)
    {
        HttpServerChannel httpChannel = channel as HttpServerChannel;
        Console.WriteLine("Схема: {0}", httpChannel.ChannelScheme);
    }
    if (channel is TcpServerChannel)
    {
        TcpServerChannel tcpChannel = channel as TcpServerChannel;
        Console.WriteLine("Защищенный: {0}", tcpChannel.IsSecured);
    }
    ChannelDataStore data = (ChannelDataStore) channel.ChannelData;
    if (data != null)
    {
        foreach (string uri in data.ChannelUris)
        {
            Console.WriteLine("URI: {0}", uri);
        }
    }
    Console.WriteLine();
}
```

### **Установка свойств канала**

Конструктор `HttpServerChannel(IDictionary, IServerChannelSinkProvider)` позволяет установить все свойства канала в списке. Обобщенный класс `Dictionary` реализует интерфейс `IDictionary`, поэтому с его помощью можно устанавливать свойства `Name`, `Priority` и `Port`. Чтобы использовать класс `Dictionary`, потребуется импортировать пространство имен `System.Collections.Generic`.

В дополнение к параметру `IDictionary` конструктору класса `HttpServerChannel` можно передать объект, реализующий интерфейс `IServerChannelSinkProvider`.

В приведенном ниже примере вместо поставщика `SoapServerFormatterSinkProvider`, который является выбором по умолчанию для `HttpServerChannel`, устанавливается `BinaryServerFormatterSinkProvider`.

Реализация класса `BinaryServerFormatterSinkProvider` по умолчанию ассоциирует класс `BinaryServerFormatterSink` с каналом, который использует объект `BinaryFormatter` для преобразования данных при передаче:

```
var properties = new Dictionary<string, string>();
properties["name"] = "HTTP Channel with a Binary Formatter";
properties["priority"] = "15";
```

```
properties["port"] = "8085";
var sinkProvider = new BinaryServerFormatterSinkProvider();
var httpChannel = new HttpServerChannel(properties, sinkProvider);
```

В зависимости от типов каналов указываются разные свойства. Каналы TCP и HTTP поддерживают свойства `name` и `property`, применявшиеся в показанном выше примере. Эти каналы также поддерживают другие свойства, такие как `bindTo`, указывающее IP-адрес, который может использоваться, если компьютер имеет несколько сконфигурированных IP-адресов. Свойство `rejectRemoteRequests` поддерживается серверным каналом TCP, разрешающим клиентские соединения только с локального компьютера.

### Подключаемость канала

Для отправки сообщений посредством транспортного протокола, отличного от HTTP, TCP или IPC, может быть создан специальный канал. Кроме того, существующие каналы могут быть расширены для предоставления дополнительной функциональности.

- Отправляющая часть должна реализовывать интерфейс `IChannelSender`. Наиболее важным элементом является метод `CreateMessageSink()`, посредством которого клиент отправляет URL, и здесь может быть установлено соединение с сервером. Должен быть создан приемник сообщений, который затем используется прокси для отправки сообщения в канал.
- Принимающая часть должна реализовывать интерфейс `IChannelReceiver`. Прослушивание должно начинаться в свойстве `ChannelData` `get`. Затем в отдельном потоке можно ожидать получения данных от клиента. После демаршализации сообщения можно воспользоваться методом `ChannelServices.SyncDispatchMessage()` для перенаправления сообщения объекту.

### Форматировщики

В .NET Framework предлагаются два класса форматировщиков:

- `System.Runtime.Serialization.Formatters.Binary.BinaryFormatter`
- `System.Runtime.Serialization.Formatters.Soap.SoapFormatter`

Форматировщики ассоциированы с каналами через объекты приемников форматировщиков и поставщики приемников форматировщиков.

Оба класса форматировщиков реализуют интерфейс `System.Runtime.Remoting.Messaging.IRemotingFormatter`, который определяет методы `Serialize()` и `Deserialize()` для передачи данных в канал и из канала.

Форматировщик также является подключаемым. При написании специального класса форматировщика экземпляр должен быть ассоциирован с каналом, который необходимо использовать. Это делается с помощью приемника форматировщика и поставщика приемника форматировщика.

Поставщик, например, `SoapServerFormatterSinkProvider`, может быть передан в качестве аргумента при создании канала, как было показано ранее. Поставщик приемника форматировщика реализует интерфейс `IServerChannelSinkProvider` для сервера и `IClientChannelSinkProvider` для клиента.

Оба эти интерфейса определяют метод `CreateSink()`, который может вернуть приемник форматировщика. `SoapServerFormatterSinkProvider` возвращает экземпляр класса `SoapServerFormatterSink`. На стороне клиента класс `SoapClientFormatterSink` использует методы `SyncProcessMessage()` и `AsyncProcessMessage()` класса `SoapFormatter` для сериализации сообщения. Класс `SoapServerFormatterSink` десериализует сообщение тоже с применением `SoapFormatter`.

Все эти классы приемников и поставщиков могут быть расширены и заменены специальными реализациями.

## ChannelServices и RemotingConfiguration

Служебный класс `ChannelServices` используется для регистрации каналов в исполняющей среде .NET Remoting. Через этот класс также можно получать доступ ко всем зарегистрированным каналам. Это исключительно полезно, если для конфигурирования канала применяются конфигурационные файлы, потому что, как будет показано позже, здесь канал создается неявно.

Канал регистрируется с помощью статического метода `ChannelServices.RegisterChannel()`. Первый параметр этого метода требует канала. Установка второго параметра в `true` обеспечивает проверку безопасности канала. Класс `HttpChannel` не имеет поддержки безопасности, потому для этого канала данный параметр устанавливается в `false`. Ниже приведен код сервера для регистрации каналов HTTP, TCP и IPC:

```
var tcpChannel = new TcpChannel(8086);
var httpChannel = new HttpChannel(8085);
var ipcChannel = new IpcChannel("myIPCPort");
ChannelServices.RegisterChannel(tcpChannel, true);
ChannelServices.RegisterChannel(httpChannel, false);
ChannelServices.RegisterChannel(ipcChannel, true);
```

Теперь служебный класс `ChannelServices` может использоваться для диспетчеризации синхронных и асинхронных сообщений, а также для закрытия регистрации определенных каналов. Свойство `RegisteredChannels` возвращает массив `IChannel` всех зарегистрированных каналов. Для получения специфического канала по его имени служит метод `GetChannel()`.

С помощью `ChannelServices` можно написать специальную административную утилиту, которая будет управлять каналами. Ниже приведен небольшой пример, демонстрирующий, как остановить прослушивание серверными каналами входящих запросов:

```
HttpServerChannel channel =
    (HttpServerChannel) ChannelServices.GetChannel("http");
channel.StopListening(null);
```

`RemotingConfiguration` — еще один служебный класс .NET Remoting. На стороне сервера он используется для регистрации удаленных типов для объектов, активизируемых сервером, и для маршализации удаленных объектов на ссылку маршализованного объекта типа `ObjRef`. Объект `ObjRef` — это сериализуемое представление объекта, который передается по сети. На стороне клиента применяется `RemotingServices` для демаршализации удаленного объекта и создания прокси из объектной ссылки.

### Сервер для хорошо известных объектов

Рассмотрим код серверной стороны для регистрации хорошо известного типа удаленного объекта в `RemotingServices`:

```
RemotingConfiguration.RegisterWellKnownServiceType(
    typeof(Hello), // тип
    "Hi", // URI
    WellKnownObjectMode.SingleCall); // режим
```

Первый аргумент `RegisterWellKnownServiceType()` — `typeof(Hello)` — указывает тип удаленного объекта. Второй аргумент — `"Hi"` — представляет собой URI удаленного объекта, используемый клиентом для доступа к этому объекту. Последний аргумент — `WellKnownObjectMode.SingleCall` — задает режим удаленного объекта. Режим может быть значением перечисления `WellKnownObjectMode: SingleCall` или `Singleton`.



- `SingleCall` означает, что объект не хранит состояния. При каждом вызове удаленного объекта создается новый экземпляр. Объект одиночного вызова создается на сервере методом `RemotingConfiguration.RegisterWellKnownServiceType()` с аргументом `WellKnownObjectMode.SingleCall`. Это очень эффективно для сервера, так как означает, что не нужно удерживать никаких ресурсов для, возможно, тысяч клиентов.
- При значении `Singleton` объект совместно используется всеми клиентами сервера; обычно объекты такого типа применяются, когда необходимо разделить некоторые общие данные между клиентами. Если речь идет о данных только для чтения, никаких проблем не возникает. Если же данные должны и читаться, и записываться, то следует помнить о блокировках и масштабируемости. Объект-одиночка (`Singleton`) создается сервером с помощью метода `RemotingConfiguration.RegisterWellKnownServiceType()` с аргументом `WellKnownObjectMode.Singleton`. Вы должны уделить внимание блокировке ресурсов, удерживаемых объектом-одиночкой, гарантировать, что данные не будут повреждены при параллельном доступе клиентов к объекту, и также стараться обеспечить достаточно эффективную блокировку, чтобы достичь требуемой масштабируемости.

### Сервер для активизируемых клиентом объектов

Если удаленный объект должен сохранять состояние определенного клиента, можно использовать объекты, активизируемые клиентом. В следующем разделе “Активизация объекта” описано, как вызываются на стороне клиента объекты, активизируемые сервером и клиентом. На стороне сервера активизируемые клиентом объекты должны регистрироваться иначе, чем объекты, активизированные сервером.

Вместо вызова `RemotingConfiguration.RegisterWellKnownServiceType()` требуется вызвать `RemotingConfiguration.RegisterActivatedServiceType()`. Для этого метода указывается только тип, а не URI. Причина в том, что для активизируемых клиентом объектов клиенты могут создавать экземпляры разных типов при одном и том же URI. Идентификатор URI для всех активизируемых клиентом объектов должен быть определен с использованием `RemotingConfiguration.ApplicationName`:

```
RemotingConfiguration.ApplicationName = "HelloServer";
RemotingConfiguration.RegisterActivatedServiceType(typeof(Hello));
```

### Активизация объекта

Клиенты могут использовать и создавать удаленный класс `Activator`. С помощью вызова метода `GetObject()` можно получить прокси на активизируемый сервером или хорошо известный удаленный объект. Метод `CreateInstance()` возвращает прокси на активизируемый клиентом удаленный объект.

Вместо класса `Activator` для активизации удаленных объектов также может применяться операция `new`. Чтобы такое стало возможным, удаленный объект также должен быть сконфигурирован внутри клиента с использованием класса `RemotingConfiguration`.

### URL приложения

Во всех сценариях активизации должен быть указан URL удаленного объекта. Это такой же URL, который используется в веб-браузере. Первая часть специфицирует протокол, за которым следует имя сервера или IP-адрес, номер порта и универсальный идентификатор ресурса (URI), который был задан при регистрации удаленного объекта на сервере:

```
протокол://сервер:порт/URI
```

В примерах кода постоянно используются три URL, в которых указаны протоколы `http`, `tcp` или `ipc`. В случае протоколов `HTTP` и `TCP` именем сервера является `localhost`, а номерами портов — 8085 и 8086. Для канала `IPC` имя хоста определять не нужно, потому что `IPC` работает только в пределах одной системы. Для всех протоколов значение URI выглядит как `Hi`:

```
http://localhost:8085/Hi
tcp://localhost:8086/Hi
ipc://myIPCPort/Hi
```

### **Активизация хорошо известных объектов**

В представленном ранее в разделе “Простое клиентское приложение” простом примере клиента активизируются хорошо известные объекты. Рассмотрим последовательность активизации более подробно:

```
TcpClientChannel channel = new TcpClientChannel();
ChannelServices.RegisterChannel(channel);
Hello obj = (Hello)Activator.GetObject(typeof(Hello), "tcp://localhost:8086/Hi");
```

`GetObject()` — статический метод класса `System.Activator`, который вызывает `RemotingServices.Connect()` для возврата прокси-объекта для удаленного объекта. Первый аргумент этого метода задает тип удаленного объекта. Прокси реализует все общедоступные и защищенные методы и свойства, так что клиент может вызывать эти методы, как делал бы это на реальном объекте. Во втором аргументе передается URL удаленного объекта. Здесь используется строка `tcp://localhost:8086/Hi`. Префикс `tcp` определяет используемый протокол, затем идет `localhost:8086` — имя хоста и номер порта и, наконец, URI объекта, который указан на сервере с помощью метода `RemotingConfiguration.RegisterWellKnownServiceType()`.

Вместо `Activator.GetObject()` можно было бы также непосредственно вызвать `RemotingServices.Connect()`:

```
Hello obj = (Hello)RemotingServices.Connect(typeof(Hello),
                                             "tcp://localhost:8086/Hi");
```

Если вы предпочитаете использовать операцию `new` для активизации хорошо известных удаленных объектов, то удаленный объект может быть зарегистрирован на клиенте с помощью `RemotingConfiguration.RegisterWellKnownClientType()`. Аргументы похожи: тип удаленного объекта и URI. Операция `new` на самом деле не создает новый удаленный объект, а возвращает прокси, подобно `Activator.GetObject()`. Если удаленный объект зарегистрирован с флагом `WellKnownObjectMode.SingleCall`, то правило остается прежним — удаленный объект создается при каждом вызове метода:

```
RemotingConfiguration.RegisterWellKnownClientType(typeof(Hello),
                                                    "tcp://localhost:8086/Hi");
Hello obj = new Hello();
```

### **Активизация объектов, активизируемых клиентом**

Удаленные объекты могут хранить состояние для клиента. `Activator.CreateInstance()` создает активизируемый клиентом удаленный объект. Используя метод `Activator.GetObject()`, вы создаете удаленный объект при вызове метода и разрушаете его по завершении метода. Объект не хранит состояние на сервере. В случае применения `Activator.CreateInstance()` ситуация отличается. При статическом методе `CreateInstance()` запускается последовательность активизации для создания удаленного объекта. Этот объект существует до тех пор, пока не истечет его время аренды (`lease time`) и не произойдет сборка мусора. Механизм аренды рассматривается в разделе “Управление временем жизни” далее в главе.

Некоторые из перегруженных методов `Activator.CreateInstance()` могут применяться только для создания локальных объектов. Чтобы создавать удаленные объекты, необходим метод, которому можно передать атрибуты активизации. Один из таких перегруженных методов используется ниже в примере. Этот метод принимает два строковых параметра и массив объектов. Первый параметр указывает имя сборки, а второй — тип класса удаленного объекта. В третьем параметре можно передавать аргументы конструктору класса удаленного объекта. Канал и имя объекта задаются в массиве объектов с помощью `UriAttribute`. Чтобы можно было применять класс `UriAttribute`, должно быть импортировано пространство имен `System.Runtime.Remoting.Activation`.

```
object[] attrs = {new UriAttribute("tcp://localhost:8086/HelloServer") };
ObjectHandle handle = Activator.CreateInstance(
    "RemoteHello", "Wrox.ProCSharp.Remoting.Hello", attrs);
if (handle == null)
{
    Console.WriteLine("не удается обнаружить сервер");
    return;
}
Hello obj = (Hello)handle.Unwrap();
Console.WriteLine(obj.Greeting("Christian"));
```

Разумеется, для активизируемых клиентом объектов вместо класса `Activator` можно использовать операцию `new`, но тогда этот активизируемый клиентом объект должен быть зарегистрирован с помощью `RemotingConfiguration.RegisterActivatedClientType()`. В рамках архитектуры активизируемых клиентом объектов операция `new` не только возвращает прокси, но также создает удаленный объект:

```
RemotingConfiguration.RegisterActivatedClientType(typeof(Hello),
    "tcp://localhost:8086/HelloServer");
Hello obj = new Hello();
```

## Прокси-объекты

Методы `Activator.GetObject()` и `Activator.CreateInstance()` возвращают клиенту прокси. На самом деле используется два прокси: прозрачный прокси и реальный прокси. Прозрачный прокси выглядит как удаленный объект — он реализует общедоступные методы удаленного объекта. Эти методы просто вызывают метод `Invoke()` объекта `RealProxy`, куда передается сообщение о вызове метода реального объекта. Реальный прокси отправляет это сообщение в канал с помощью приемников сообщений.

Метод `RemotingServices.IsTransparentProxy()` позволяет проверить, является ли объект прозрачным прокси. Для получения реального прокси служит метод `RemotingServices.GetRealProxy()`. Используя отладчик `Visual Studio`, теперь легко просмотреть все свойства реального прокси.

```
ChannelServices.RegisterChannel(new TCPChannel());
Hello obj = (Hello)Activator.GetObject(typeof(Hello),
    "tcp://localhost:8086/Hi");
if (obj == null)
{
    Console.WriteLine("не удается обнаружить сервер");
    return;
}
if (RemotingServices.IsTransparentProxy(obj))
{
    Console.WriteLine("Использование прозрачного прокси");
    RealProxy proxy = RemotingServices.GetRealProxy(obj);
    // proxy.Invoke(message);
}
```

## Подключаемость прокси

Реальный прокси может быть заменен специальным прокси. Специальный прокси может расширить базовый класс `System.Runtime.Remoting.Proxies.RealProxy`. Тип удаленного объекта принимается в конструкторе специального прокси. Вызов конструктора `RealProxy` создает прозрачный прокси в дополнение к реальному прокси. В конструкторе с помощью класса `ChannelServices` можно получить доступ к зарегистрированным каналам для создания приемника сообщений `IChannelSender.CreateMessageSink()`. Помимо реализации конструктора, специальный канал должен переопределять метод `Invoke()`. В `Invoke()` принимается сообщение, которое может быть проанализировано и отправлено в приемник сообщений.

## Сообщения

Прокси отправляет сообщение в канал. На стороне сервера вызов метода может быть выполнен после анализа сообщения. В этом разделе более подробно рассматриваются сообщения.

В .NET Framework определены классы сообщений для вызовов методов, ответов, возврата сообщений и т.п. Общим у классов сообщений является то, что они реализуют интерфейс `IMessage`. Этот интерфейс имеет единственное свойство — `Properties`. Данное свойство представляет словарь с интерфейсом `IDictionary`, который упаковывает URI в объект, `MethodName`, `MethodSignature`, `TypeName`, `Args` и `CallContext`.

На рис. 54.4 показана иерархия классов и интерфейсов сообщений. Сообщение, которое отправляется в реальный прокси — это объект типа `MethodCall`. С помощью интерфейсов `IMethodCallMessage` и `IMethodMessage` можно иметь более легкий доступ к свойствам сообщения, чем через интерфейс `IMessage`. Вместо использования интерфейса `IDictionary`, есть прямой доступ к имени метода, URI, аргументам и т.д. Реальный прокси возвращает прозрачному прокси объект `ReturnMessage`.

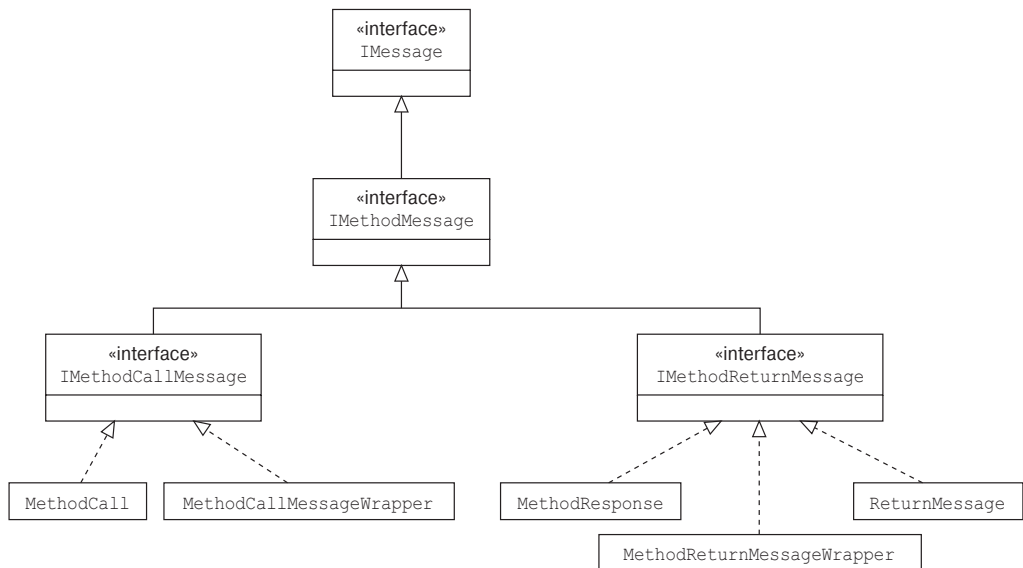


Рис. 54.4. Иерархия классов и интерфейсов сообщений

## Приемники сообщений

Метод `Activator.GetObject()` вызывает `RemotingServices.Connect()` для подключения к хорошо известному объекту. В методе `Connect()` происходит вызов `Unmarshal()`, где создается не только прокси, но и приемники агентов (*envoy sinks*). Прокси использует цепочку приемников агентов для передачи сообщения в канал. Все приемники представляют собой перехватчики, которые могут изменять сообщения и выполнять некоторые дополнительные действия, подобные установке блока, записи события, проверке безопасности и т.п.

- Свойство `NextSink` используется приемником для получения следующего приемника и передачи ему сообщения.
- Для синхронизации сообщений предыдущим приемником или инфраструктурой .NET Remoting вызывается `SyncProcessMessage()`. В нем есть параметр `IMessage` для отправки и возврата сообщения.
- Для асинхронных сообщений предыдущим приемником в цепочке или инфраструктурой вызывается `AsynkProcessMessage()`. Этот метод ожидает два параметра: сообщение и приемник сообщений, который принимает ответ.

В следующем разделе рассматриваются три разных приемника сообщений, которые доступны для использования.

### Приемник агентов

Получить цепочку приемников агентов можно с использованием интерфейса `IEnvoyInfo`. Маршилизованная объектная ссылка `ObRef` имеет свойство `EnvoyInfo`, которое возвращает интерфейс `IEnvoyInfo`. Список агентов создается из контекста сервера, поэтому сервер может внедрить функциональность в клиент. Агенты могут собирать информацию идентичности о клиенте и передавать ее серверу.

### Приемник серверного контекста

Когда сообщение получается на серверной стороне канала, оно передается приемникам серверного контекста. Последний из них маршрутизирует сообщения в цепочку приемников объектов.

### Приемник объекта

Приемник объекта ассоциируется с определенным объектом. Если класс объекта определяет некоторые атрибуты контекста, то для этого объекта создаются контекстные приемники.

## Передача объектов в удаленных методах

Типы параметров удаленных вызовов методов не ограничены лишь базовыми типами данных, а могут быть также определенными вами классами. Для удаленной работы следует различать три типа классов.

- **Классы, маршализуемые по значению.** Эти классы сериализуются через канал. Классы, которые должны быть маршализованы, нужно помечать атрибутом `Serializable`. Объекты этих классов не имеют удаленной идентичности, потому что полный объект маршализуется по каналу, а объект, сериализованный клиенту, не зависит от серверного объекта (и наоборот). Классы, маршализуемые по значению, также называют *несвязанными (unbound) классами*, потому что они не имеют данных, которые зависят от домена приложения. Сериализация обсуждается в главе 29.

- **Классы, маршализуемые по ссылке.** Эти классы имеют удаленную идентичность. Объекты не передаются по сети, а вместо этого возвращается прокси. Класс, маршализуемый по ссылке, должен наследоваться от `MarshalByRefObject`. Экземпляры `MarshalByRefObject` известны как *объекты, привязанные к домену приложения*. Специализированной версией `MarshalByRefObject` является `ContextBoundObject`: абстрактный класс `ContextBoundObject` наследуется от `MarshalByRefObject`. Если класс наследуется от `ContextBoundObject`, необходим прокси, даже если пересекаются границы контекста в пределах одного и того же домена приложений. Такие объекты называются *привязанными к контексту объектами*, и они действительны только в контексте создания.
- **Классы, не поддающиеся удаленному вызову (Nonremotable).** Эти классы не сериализуемы и не наследуются от `MarshalByRefObject`. Классы этих типов не могут использоваться в качестве параметров в общедоступных методах удаленных объектов. Эти классы привязаны к домену приложений, в котором они созданы. Не поддающиеся удаленному вызову классы должны применяться в том случае, если в классе есть член данных, который действителен только в домене приложений, такой как файловый дескриптор `Win32`.

Чтобы увидеть маршализацию в действии, измените удаленный объект таким образом, чтобы он отправлял объект на клиент: для примера будет отправляться класс `MySerialized` с маршализацией по значению. В методе значение выводится на консоль, так что можно увидеть, где произошел вызов — на клиенте или на сервере. Вдобавок класс `Hello` расширяется для возврата экземпляра `MySerialized`.

```

using System;
namespace Wrox.ProCSharp.Remoting
{
    [Serializable]
    public class MySerialized
    {
        public MySerialized(int val)
        {
            a = val;
        }
        public void Foo()
        {
            Console.WriteLine("Вызван метод MySerialized.Foo");
        }
        public int A
        {
            get
            {
                Console.WriteLine("Вызван метод MySerialized.A");
                return a;
            }
            set
            {
                a = value;
            }
        }
        protected int a;
    }
    public class Hello : MarshalByRefObject
    {
        public Hello()
        {

```

```

        Console.WriteLine("Вызван конструктор");
    }
    public string Greeting(string name)
    {
        Console.WriteLine("Вызван метод Greeting");
        return "Hello, " + name;
    }
    public MySerialized GetMySerialized()
    {
        return new MySerialized(4711);
    }
}

```

Фрагмент кода *PassingObjects\RemoteHello\Hello.cs*

Клиентское приложение также должно быть изменено, чтобы увидеть эффект от маршализуемых по значению и по ссылке объектов. Для извлечения новых объектов вызываются методы `GetMySerialized()` и `GetMyRemote()`. Также с помощью метода `RemotingServices.IsTransparentProxy()` осуществляется проверка того, является ли возвращенный объект прокси.

```

❶ ChannelServices.RegisterChannel(new TcpClientChannel(), false);
Hello obj = (Hello)Activator.GetObject(typeof(Hello),
    "tcp://localhost:8086/Hi");
if (obj == null)
{
    Console.WriteLine("не удастся обнаружить сервер");
    return;
}
MySerialized ser = obj.GetMySerialized();
if (!RemotingServices.IsTransparentProxy(ser))
{
    Console.WriteLine("ser не является прозрачным прокси");
}
ser.Foo();

```

Фрагмент кода *PassingObjects\HelloClient\Program.cs*

В окне клиентской консоли видно, что объект `ser` вызывается на клиенте. Этот объект не является прозрачным прокси, потому что сериализуется на стороне клиента:

```

Нажмите Enter после запуска сервера
ser не является прозрачным прокси
Вызван метод MySerialized.Foo

```

## Безопасность и сериализованные объекты

Одно важное отличие между .NET Remoting и веб-службами ASP.NET заключается в способе маршализации объектов. В веб-службах ASP.NET по сети передаются только общедоступные поля и свойства. Технология .NET Remoting применяет различные механизмы для сериализации всех данных, включая приватные. Злоумышленники могут использовать фазы сериализации и десериализации для взлома приложения.

Чтобы привлечь внимание эту проблему, при передаче объектов через границы .NET Remoting определены два уровня десериализации: низкий и полный.

По умолчанию используется низкоуровневая десериализация. При ней невозможно передавать объекты `ObjRef` и объекты, реализующие интерфейс `ISponsor`. Чтобы это стало возможным, необходимо изменить уровень десериализации на полный. Это можно сделать программно, создав поставщика форматирующего приемника и установив свойство `TypeFilterLevel`.

Для двоичного форматировщика классом поставщика будет `BinaryServerFormatterSinkProvider`, а для форматировщика SOAP — `SoapServerFormatterSinkProvider`.

В следующем коде показано, как создать канал TCP с поддержкой полной сериализации:

```

var serverProvider = new BinaryServerFormatterSinkProvider();
serverProvider.TypeFilterLevel = TypeFilterLevel.Full;
var clientProvider = new BinaryClientFormatterSinkProvider();
var properties = new Dictionary<string, string>();
properties["port"] = 6789;
var channel = new TcpChannel(properties, clientProvider, serverProvider);

```

Фрагмент кода *PassingObjects\HelloServer\Program.cs*

Сначала создается `BinaryServerFormatterSinkProvider`, в котором свойство `TypeFilterLevel` устанавливается в `TypeFilterLevel.Full`. Перечисление `TypeFilterLevel` определено в пространстве имен `System.Runtime.Serialization`. `Formatters`, поэтому понадобится добавить на него ссылку. Для клиентской стороны канала создается поставщик `BinaryClientFormatterSinkProvider`. Экземпляры поставщиков приемника форматировщика как клиентской, так и серверной стороны передаются конструктору `TcpChannel`, наряду со свойствами `IDictionary`, которые определяют атрибуты канала.

## Атрибуты направления

Удаленные объекты никогда не передаются по сети, в то время как типы значений и сериализуемые классы напротив — передаются. Иногда данные должны передаваться только в одном направлении. Это может быть особенно важно, когда данные передаются по сети. Например, если необходимо передать данные в коллекции на сервер, чтобы он произвел какие-то вычисления над этими данными и вернул простое значение клиенту, было бы не очень эффективно пересылать коллекцию обратно клиенту. В СОМ было возможно объявлять атрибуты направления `[in]`, `[out]` и `[in, out]` для аргументов, если данные должны были передаваться на сервер, на клиент либо в обоих направлениях.

В С# подобные атрибуты представляют собой часть языка: это параметры методов `ref` и `out`. Эти параметры могут использоваться для типов значений и для ссылочных типов, которые являются сериализуемыми. Благодаря параметру `ref`, аргумент маршализируется в обоих направлениях, а с помощью параметра `out` данные передаются с сервера на клиент. Если параметры не указаны, данные отправляются только на сервер.



Ключевые слова `out` и `ref` описаны в главе 3.

## Управление временем жизни

Каким образом клиент и сервер определяют, что другая сторона более недоступна, и с какими проблемами здесь можно столкнуться?

Для клиента ответ прост. Как только клиент производит вызов метода на удаленном объекте, генерируется исключение типа `System.Runtime.Remoting.RemotingException`. Вам нужно лишь обработать это исключение и сделать то, что необходимо; например, предпринять повторную попытку, внести запись в журнал, информировать пользователя и т.п.

А как насчет сервера? Когда сервер обнаруживает, что клиент отключился, значит ли это, что он может очищать все ресурсы, удерживаемые клиентом? Можно подождать, пока



не поступит следующий вызов метода от клиента, но, возможно, это вообще не произойдет. В мире COM протокол DCOM использует механизм опроса доступности. Клиент посылает пакет опроса на сервер с информацией об объектах, на которые он ссылается. Клиент может ссылаться на сотни объектов на сервере, так что информация в этом пакете опроса может быть очень объемной. Чтобы сделать этот механизм более эффективным, DCOM не отправляет всю информацию обо всех объектах, а только ту, что отличается от отправленной в предыдущем пакете опроса.

Это управление временем жизни работает только для активизируемых клиентом объектов и хорошо известных объектов-одиночек. Объекты одиночного вызова могут уничтожаться после каждого вызова метода, потому что они не хранят состояния. Активизируемые клиентом объекты имеют состояние и вы должны помнить об используемых ими ресурсах. Для активизируемых клиентом объектов, на которые есть ссылки извне домена приложения, создается механизм аренды. Он предполагает ограниченное время аренды, и когда оно истекает, удаленный объект отключается и, в конце концов, обрабатывается сборщиком мусора.

### **Продление аренды**

Если клиент вызывает метод на объекте, когда время аренды истекает, генерируется исключение. При наличии клиента, которому удаленный объект может понадобиться на более чем 300 секунд (время аренды по умолчанию), доступны три способа ее продления.

- **Неявное продление.** Автоматически выполняется, когда клиент вызывает метод на удаленном объекте. Если текущее время аренды меньше значения `RenewOnCallTime`, оно устанавливается в `RenewOnCallTime`.
- **Явное продление.** Клиент может указать новое время аренды. Это делается методом `Renew()` интерфейса `ILease`. Для получения экземпляра интерфейса `ILease` необходимо вызвать метод `GetLifetimeService()` прозрачного прокси.
- **Спонсированное продление.** В случае такого продления аренды клиент создает спонсора, который реализует интерфейс `ISponsor`, и регистрирует спонсора в службах аренды с помощью метода `Register()` интерфейса `ILease`. Спонсор определяет время продления аренды. Когда оно истекает, у спонсора запрашивается новое продление. Механизм спонсорства может применяться, если нужны долгоживущие объекты на сервере.

### **Конфигурационные значения аренды**

Для конфигурации аренды доступны перечисленные ниже значения.

- `LeaseTime` определяет время истечения срока аренды.
- `RenewOnCallTime` — время аренды устанавливается при вызове метода, если текущее время аренды оказывается меньше.
- Если спонсор недоступен в пределах периода времени `SponsorshipTimeout`, инфраструктура .NET Remoting ищет следующего спонсора. Если спонсор не найден, время аренды истекает.
- `LeaseManagerPollTime` определяет временной интервал, в течение которого диспетчер аренды проверяет устаревшие объекты.

Значения по умолчанию перечисленных параметров приведены в табл. 54.1.

**Таблица 54.1. Значения по умолчанию для конфигурации аренды**

Конфигурация аренды	Значение по умолчанию (секунды)
LeaseTime	300
RenewOnCallTime	120
SponsorshipTimeout	120
LeaseManagerPollTime	10

### **Классы, используемые для управления временем жизни**

Класс `ClientSponsor` реализует интерфейс `ISponsor`. Он может применяться на стороне клиента для продления аренды. Через интерфейс `ILease` можно получить всю информацию об аренде, все ее свойства, текущее время аренды и ее состояние. Состояние определено перечислением `LeaseState`. С помощью служебного класса `LifetimeServices` можно получать и устанавливать свойства аренды всех удаленных объектов в домене приложения.

### **Пример получения информации об аренде**

В следующем небольшом примере коде информация об аренде получается через метод `GetLifetimeService()` прозрачного прокси. Для интерфейса `ILease` потребуется добавить ссылку на пространство имен `System.Runtime.Remoting.Lifetime`. Для класса `UrlAttribute` необходимо импортировать пространство имен `System.Runtime.Remoting.Activation`.

Механизм аренды может использоваться только для объектов с состоянием (активируемых клиентом и объектов-одиночек). Объекты одиночного вызова все равно создаются при каждом вызове метода, так что к ним механизм аренды не применим. Чтобы предоставить активируемые клиентом объекты на сервере, можно изменить конфигурацию удаленного взаимодействия для вызова `RegisterActivatedServiceType()`:

```
RemotingConfiguration.ApplicationName = "Hello";
RemotingConfiguration.RegisterActivatedServiceType(typeof(Hello));
```

В клиентском приложении создание экземпляра удаленного объекта должно быть также изменено. Вместо метода `Activator.GetObject()` применяется метод `Activator.CreateInstance()` для вызова активируемых клиентом объектов:

```
ChannelServices.RegisterChannel(new TcpClientChannel(), true);
object[] attrs = {new UrlAttribute("tcp://localhost:8086/Hi") };
Hello obj = (Hello)Activator.CreateInstance(typeof(Hello), null, attrs);
```

Для получения времени аренды используется интерфейс `ILease`, возвращенный вызовом `GetLifetimeService()` от прокси объекта:

```
ILease lease = (ILease)obj.GetLifetimeService();
if (lease != null)
{
    Console.WriteLine("Конфигурация аренды:");
    Console.WriteLine("InitialLeaseTime: {0}", lease.InitialLeaseTime);
    Console.WriteLine("RenewOnCallTime: {0}", lease.RenewOnCallTime);
    Console.WriteLine("SponsorshipTimeout: {0}", lease.SponsorshipTimeout);
    Console.WriteLine(lease.CurrentLeaseTime);
}
```

### Изменение конфигурации аренды по умолчанию

Сам сервер может изменять конфигурацию аренды по умолчанию для всех удаленных объектов сервера, используя служебный класс `System.Runtime.Remoting.Lifetime.LifetimeServices`:

```
LifetimeServices.LeaseTime = TimeSpan.FromMinutes(10);
LifetimeServices.RenewOnCallTime = TimeSpan.FromMinutes(2);
```

Если нужны другие значения времени жизни, зависящие от типа удаленного объекта, можно изменить конфигурацию аренды удаленного объекта, переопределив метод `InitializeLifetimeService()` базового класса `MarshalByRefObject`:

```
public class Hello : System.MarshalByRefObject
{
    public override Object InitializeLifetimeService()
    {
        ILease lease = (ILease)base.InitializeLifetimeService();
        lease.InitialLeaseTime = TimeSpan.FromMinutes(10);
        lease.RenewOnCallTime = TimeSpan.FromSeconds(40);
        return lease;
    }
}
```


Настройка служб времени жизни также может выполняться с помощью конфигурационных файлов, о чем речь пойдет ниже.

## Конфигурационные файлы

Вместо описания конфигурации канала и объекта в исходном коде для этого можно использовать конфигурационные файлы. Таким способом может быть переконфигурирован канал, добавлены дополнительные каналы и т.д. — причем все это без изменения исходного кода. Подобно всем прочим конфигурационным файлам на платформе .NET, в них применяется формат XML. В качестве примера рассматривается то же приложение и конфигурационные файлы, которые были описаны в главах 18 и 21. Для .NET Remoting предусмотрено несколько дополнительных элементов и атрибутов XML, служащих для настройки канала и удаленных объектов. Отличие конфигурационного файла .NET Remoting в том, что эта конфигурация не должна содержаться в самом конфигурационном файле приложения; файл может иметь любое имя. Чтобы облегчить процесс сборки, в этой главе конфигурация удаленного взаимодействия сохраняется внутри конфигурационных файлов приложения, имеющих расширение `.config` после имени исполняемого файла программы.

В подкаталогах для клиента и сервера внутри каталога `ConfigurationFiles` среди примеров кода для настоящей главы содержатся следующие конфигурационные файлы: `clientactivated.config` и `wellknown.config`. В примере клиента также имеется файл `wellknownhttp.config`, который указывает канал HTTP для хорошо известного удаленного объекта. Чтобы использовать эти конфигурации, файлы должны быть переименованы в соответствии с именем, передаваемым в параметре метода `RemotingConfiguration.Configure()`, и помещены в каталог, содержащий исполняемый файл.

Ниже показан лишь один пример того, как может выглядеть конфигурационный файл:

```
 <configuration>
  <system.runtime.remoting>
    <application name="Hello">
      <service>
        <wellknown mode="SingleCall"
          type="Wrox.ProCSharp.Remoting.Hello, RemoteHello"
          objectUri="Hi" />
      </service>
```

```

    <channels>
      <channel ref="tcp" port="6791" />
      <channel ref="http" port="6792" />
      <channel ref="ipc" portName="myIPCPort" />
    </channels>
  </application>
</system.runtime.remoting>
</configuration>

```

Фрагмент кода *ConfigurationFiles\HelloServer\app.config*

`<configuration>` — это корневой элемент XML для всех конфигурационных файлов .NET. Все конфигурации удаленного взаимодействия расположены в подэлементе `<system.runtime.remoting>`. При этом `<application>` является подэлементом `<system.runtime.remoting>`.

В следующем списке перечислены основные элементы и атрибуты частей внутри `<system.runtime.remoting>`.

- В элементе `<application>` можно указывать имя приложения, используя для этого атрибут `name`. На стороне сервера это имя сервера, а на стороне клиента — имя клиентского приложения. В качестве примера серверной конфигурации `<application name="Hello">` определяет имя удаленного приложения `Hello`, которое используется клиентом как часть URL для доступа к удаленному объекту.
- На сервере применяется элемент `<service>`, в котором задается коллекция удаленных объектов. Он может иметь подэлементы `<wellknown>` и `<activated>`, которые специфицируют тип удаленного объекта как хорошо известного или активизируемого клиентом.
- Клиентской частью элемента `<service>` является `<client>`. Подобно элементу `<service>`, он может иметь подэлементы `<wellknown>` и `<activated>` для указания типа удаленного объекта. В отличие от `<service>`, элемент `<client>` имеет атрибут `url` для указания URL удаленного объекта.
- `<wellknown>` — это элемент, используемый на сервере и клиенте для указания хорошо известных удаленных объектов. Серверная часть должна выглядеть следующим образом:

```

<wellknown mode="SingleCall"
  type="Wrox.ProCSharp.Remoting.Hello, RemoteHello"
  objectURI="Hi" />

```

Хотя можно задать атрибут `mode` со значением `SingleCall` или `Singleton`, типом является тип удаленного класса, включая пространство имен `Wrox.ProCSharp.Remoting.Hello`. За ним следует имя сборки `RemoteHello`. Кроме того, `objectURI` — это имя удаленного объекта, зарегистрированного в канале.

- На клиенте атрибут `type` является тем же, что и в серверной версии. `mode` и `objectURI` не требуются, но атрибут `url` используется для определения пути к удаленному объекту: протокола, имени хоста, номера порта, имени приложения и URI объекта:

```

<wellknown type="Wrox.ProCSharp.Remoting.Hello, RemoteHello"
  url="tcp://localhost:6791/Hello/Hi" />

```

- Элемент `<activated>` применяется для активизируемых клиентом объектов. В атрибуте `type` должен быть указан тип, а сборка должна быть определена как в клиентском, так и серверном приложении:

```

<activated type="Wrox.ProCSharp.Remoting.Hello, RemoteHello" />

```

- Для указания канала используется элемент `<channel>`. Это подэлемент `<channels>`, так что для одного приложения может быть сконфигурирована коллекция каналов. Для клиентов и серверов его применение похоже. С помощью XML-атрибута `ref` производится ссылка на предварительно сконфигурированное имя канала. Для серверного канала с помощью XML-атрибута `port` потребуется установить номер порта. XML-атрибут `displayName` служит для указания имени канала, которое используется в инструменте .NET Framework Configuration, рассматриваемом далее в главе.

```
<channels>
  <channel ref="tcp" port="6791" displayName="TCP Channel" />
  <channel ref="http" port="6792" displayName="HTTP Channel" />
  <channel ref="ipc" portName="myIPCPort" displayName="IPC Channel" />
</channels>
```



*Предопределенные каналы имеют имена `tcp`, `http` и `ipc`, что определено классами `TcpChannel`, `HttpChannel` и `IpcChannel`.*

## Серверная конфигурация для хорошо известных объектов

В файле этого примера — `Wellknown_Server.config` — для свойства `name` выбрано значение `Hello`. В следующем конфигурационном файле канал TCP устанавливается на прослушивание порта 6791, а канал HTTP — на прослушивание порта 6792.

Канал IPC сконфигурирован с именем порта `myIPCPort`. Классом удаленного объекта является `Wrox.ProCSharp.Remoting.Hello` в сборке `RemoteHello`, объект в канале называется `Hi`, а в качестве режима объекта используется `SingleCall`.

```
<configuration>
  <system.runtime.remoting>
    <application name="Hello">
      <service>
        <wellknown mode="SingleCall"
          type="Wrox.ProCSharp.Remoting.Hello, RemoteHello"
          objectUri="Hi" />
      </service>
    </application>
  </system.runtime.remoting>
</configuration>
```

Фрагмент кода `ConfigurationFiles\HelloServer\Wellknown.config`

## Клиентская конфигурация для хорошо известных объектов

Для хорошо известных объектов необходимо специфицировать сборку и канал в клиентском конфигурационном файле `Wellknown_Client.config`. Типы для удаленного объекта находятся в сборке `RemoteHello`, `Hi` является именем объекта в канале, а URI удаленного типа `Wrox.ProCSharp.Remoting.Hello` выглядит как `ipc://myIPCPort/Hello/Hi`. На клиенте канал IPC также используется, но никакого порта не указывается, поэтому выбирается свободный. Канал, выбранный клиентом, должен соответствовать URL.

```

<configuration>
  <system.runtime.remoting>
    <application name="Client">
      <client displayName="Hello client for well-known objects">
        <wellknown type = "Wrox.ProCSharp.Remoting.Hello, RemoteHello"
          url="ipc://myIPCPort/Hello/Hi" />
      </client>
    <channels>
      <channel ref="ipc" displayName="IPC Channel (HelloClient)" />
    </channels>
    </application>
  </system.runtime.remoting>
</configuration>

```

*Фрагмент кода ConfigurationFiles\HelloClient\Wellknown\_Client.config*

После небольшого изменения в конфигурационном файле используется канал HTTP (как видно в WellknownHttp\_Client.config):

```

<client>
  <wellknown type="Wrox.ProCSharp.Remoting.Hello, RemoteHello"
    url="http://localhost:6792/Hello/Hi" />
</client>
<channels>
  <channel ref="http" displayName="HTTP Channel (HelloClient)" />
</channels>

```

*Фрагмент кода ConfigurationFiles\HelloClient\WellknownHttp\_Client.config*

## Серверная конфигурация для активизируемых клиентом объектов

Изменив только конфигурационный файл (ClientActivated\_Server.config), можно изменить серверную конфигурацию с активизируемых сервером объектов на объекты, активизируемые клиентом. Здесь указан подэлемент <activated> элемента <service>. В элементе <activated> серверной конфигурации должен присутствовать только атрибут type. Атрибут name элемента application определяет URI.

 <configuration>

```

  <system.runtime.remoting>
    <application name="HelloServer">
      <service>
        <activated type="Wrox.ProCSharp.Remoting.Hello, RemoteHello" />
      </service>
    <channels>
      <channel ref="http" port="6788"
        displayName="HTTP Channel (HelloServer)" />
      <channel ref="tcp" port="6789"
        displayName="TCP Channel (HelloServer)" />
      <channel ref="ipc" portName="myIPCPort"
        displayName="IPC Channel (HelloServer)" />
    </channels>
  </application>
</system.runtime.remoting>
</configuration>

```

*Фрагмент кода ConfigurationFiles\HelloServer\ClientActivated.config*

## Клиентская конфигурация для активизируемых клиентом объектов

В файле `ClientActivated_Client.config` определен активизируемый клиентом удаленный объект с использованием атрибута `url` элемента `<client>` и атрибута `type` элемента `<activated>`:

```
<configuration>
  <system.runtime.remoting>
    <application>
      <client url="http://localhost:6788/HelloServer"
        displayName="Hello client for client-activated objects">
        <activated type="Wrox.ProCSharp.Remoting.Hello, RemoteHello" />
      </client>
    <channels>
      <channel ref="http" displayName="HTTP Channel (HelloClient)" />
      <channel ref="tcp" displayName="TCP Channel (HelloClient)" />
      <channel ref="ipc" displayName="IPC Channel (HelloClient)" />
    </channels>
  </application>
</system.runtime.remoting>
</configuration>
```

Фрагмент кода `ConfigurationFiles\HelloClient\ClientActivated_Client.config`

## Серверный код, использующий конфигурационные файлы

В серверном коде удаленное взаимодействие должно конфигурироваться с использованием статического метода `Configure()` класса `RemotingConfiguration`. Здесь создаются все каналы, которые определены в конфигурационном файле и настроены исполняющей средой .NET Remoting. Возможно, также понадобится получить сведения о конфигурации каналов из серверного приложения — именно для этого предусмотрены статические методы `ShowActivatedServiceTypes()` и `ShowWellKnownServiceTypes()`; они вызываются после загрузки и запуска удаленной конфигурации.

```
public static void Main()
{
    RemotingConfiguration.Configure("HelloServer.exe.config", false);
    Console.WriteLine("Приложение: {0}", RemotingConfiguration.ApplicationName);
    ShowActivatedServiceTypes();
    ShowWellKnownServiceTypes();
    System.Console.WriteLine("Для завершения нажмите Enter");
    System.Console.ReadLine();
}
```

Фрагмент кода `ConfigurationFiles\HelloServer\Program.cs`

Эти две функции показывают конфигурационную информацию о хорошо известных и активизированных клиентом типах:

```
public static void ShowWellKnownServiceTypes()
{
    WellKnownServiceTypeEntry[] entries =
        RemotingConfiguration.GetRegisteredWellKnownServiceTypes();
    foreach (var entry in entries)
    {
        Console.WriteLine("Сборка: {0}", entry.AssemblyName);
        Console.WriteLine("Режим: {0}", entry.Mode);
        Console.WriteLine("URI: {0}", entry.ObjectUri);
        Console.WriteLine("Тип: {0}", entry.TypeName);
    }
}
```

```

public static void ShowActivatedServiceTypes()
{
    ActivatedServiceTypeEntry[] entries =
        RemotingConfiguration.GetRegisteredActivatedServiceTypes();
    foreach (var entry in entries)
    {
        Console.WriteLine("Сборка: {0}", entry.AssemblyName);
        Console.WriteLine("Тип: {0}", entry.TypeName);
    }
}

```

## Клиентский код, использующий конфигурационные файлы

В клиентском коде необходимо лишь сконфигурировать службы удаленного взаимодействия, используя конфигурационный файл `client.exe.config`. После этого можно применять операцию `new` для создания экземпляров удаленного класса `Hello`, независимо от того, работаете вы с удаленными объектами, активизируемыми сервером или клиентом. Однако есть небольшая разница: для активизируемых клиентом объектов теперь с операцией `new` можно использовать *конструкторы не по умолчанию*. Это невозможно для объектов, активизируемых сервером. Объекты одиночного вызова не имеют состояния, потому что они уничтожаются при каждом вызове; объекты-одиночки создаются лишь однажды. Вызов конструкторов не по умолчанию возможен только с активизируемыми клиентом объектами, поскольку это единственный тип, для которого создается экземпляр удаленного объекта посредством вызова операции `new`.

В методе `Main()` файла `HelloClient.cs` теперь можно изменить код удаленного взаимодействия для использования конфигурационного файла с `RemotingConfiguration.Configure()` и создать удаленный объект операцией `new`:

```

❏ RemotingConfiguration.Configure("HelloClient.exe.config", false);
Hello obj = new Hello();
if (obj == null)
{
    Console.WriteLine("не удается обнаружить сервер");
    return;
}
for (int i=0; i < 5; i++)
{
    Console.WriteLine(obj.Greeting("Stephanie"));
}

```

Фрагмент кода `ConfigurationFiles\HelloClient\Program.cs`

## Отложенная загрузка клиентских каналов

В .NET Remoting по умолчанию сконфигурированы три канала, которые применяются автоматически, если клиент явно не указывает канал:

```

<system.runtime.remoting>
  <application>
    <channels>
      <channel ref="http client" displayName="http client (delay loaded)"
        delayLoadAsClientChannel="true" />
      <channel ref="tcp client" displayName="tcp client (delay loaded)"
        delayLoadAsClientChannel="true" />
      <channel ref="ipc client" displayName="ipc client (delay loaded)"
        delayLoadAsClientChannel="true" />
    </channels>
  </application>
</system.runtime.remoting>

```



XML-атрибут `delayLoadAsClientChannel` со значением `true` указывает, что канал должен быть использован из клиента, который не конфигурирует канал. Исполняющая среда пытается подключиться к серверу, используя каналы с отложенной загрузкой. Поэтому необходимо настроить канал в клиентском конфигурационном файле. Такой клиентский конфигурационный файл для хорошо известного объекта, использованного ранее, может выглядеть следующим образом:

```
<configuration>
  <system.runtime.remoting>
    <application name="Client">
      <client url="tcp://localhost:6791/Hello">
        <wellknown type = "Wrox.ProCSharp.Remoting.Hello, RemoteHello"
          url="tcp://localhost:6791/Hello/Hi" />
      </client>
    </application>
  </system.runtime.remoting>
</configuration>
```

## Отладка конфигурации

Если конфигурационный файл содержит недопустимые настройки (например, указано неверное имя удаленной сборки), то ошибка не обнаруживается сразу после запуска сервера. Вместо этого она обнаружится, когда клиент создаст экземпляр удаленного объекта и вызовет его метод. Клиент получит исключение, говорящее о том, что сборка удаленного объекта не может быть найдена. Указывая конфигурацию `<debug loadTypes="true" />`, можно заставить загрузиться удаленный объект и создать его экземпляр на сервере при его запуске. Тогда в случае ошибки в конфигурационном файле ошибка на сервере возникает немедленно.

```
<configuration>
  <system.runtime.remoting>
    <application name="Hello">
      <service>
        <wellknown mode="SingleCall"
          type="Wrox.ProCSharp.Remoting.Hello, RemoteHello"
          objectUri="Hi" />
      </service>
      <channels>
        <channel ref="tcp" port="6791"
          displayName="TCP Channel (HelloServer)" />
      </channels>
    </application>
    <debug loadTypes="true" />
  </system.runtime.remoting>
</configuration>
```

## Службы времени жизни в конфигурационных файлах

Конфигурация аренды для удаленных серверов также описывается в конфигурационных файлах. Элемент `<lifetime>` имеет атрибуты `leaseTime`, `sponsorshipTimeOut`, `renewOnCallTime` и `pollTime`, как показано в примере:

```
<configuration>
  <system.runtime.remoting>
    <application>
      <lifetime leaseTime = "15M" sponsorshipTimeOut = "4M"
        renewOnCallTime = "3M" pollTime = "30s"/>
    </application>
  </system.runtime.remoting>
</configuration>
```

Благодаря конфигурационным файлам, можно изменить конфигурацию удаленного взаимодействия, редактируя их вместо работы с исходным кодом. Довольно легко изменить используемый канал с HTTP на TCP, поменять порт, имя канала и т.д. Добавив всего одну строку, можно заставить сервер прослушивать два канала вместо одного.

## Поставщики форматировщиков

Ранее в этой главе было показано, где должны быть изменены свойства поставщика форматировщика, чтобы поддержать маршализацию всех объектов по сети. Вместо того чтобы делать это программно, как мы поступали ранее, свойства поставщика форматировщика можно настроить в конфигурационном файле.

Следующий конфигурационный файл сервера содержит изменения в элементе `<channel>`, внутри которого определены дочерние элементы `<serverProviders>` и `<clientProviders>`. В элементе `<serverProviders>` содержатся ссылки на встроенные поставщики `wsdl`, `soap` и `binary`, а для поставщиков `soap` и `binary` свойство `typeFilterLevel` установлено в `Full`.

```
<configuration>
  <system.runtime.remoting>
    <application name="HelloServer">
      <service>
        <activated type="Wrox.ProCSharp.Remoting.Hello, RemoteHello" />
      </service>
      <channels>
        <channel ref="tcp" port="6789" displayName="TCP Channel (HelloServer)">
          <serverProviders>
            <provider ref="wsdl" />
            <provider ref="soap" typeFilterLevel="Full" />
            <provider ref="binary" typeFilterLevel="Full" />
          </serverProviders>
          <clientProviders>
            <provider ref="binary" />
          </clientProviders>
        </channel>
      </channels>
    </application>
  </system.runtime.remoting>
</configuration>
```

## Хостинг серверов в ASP.NET

До этого момента все примеры серверов запускались как саморазмещаемые серверы .NET. Саморазмещаемый сервер должен запускаться вручную. Сервер .NET Remoting также может запускаться во множестве других типов приложений. В Windows-службе сервер может запускаться автоматически при загрузке системы, и к тому же процесс может выполняться с регистрационными данными системной учетной записи. За дополнительными подробностями о Windows-службах обращайтесь в главу 25.

Предусмотрена специальная поддержка серверов .NET Remoting для ASP.NET. Платформа ASP.NET может применяться для автоматического запуска удаленных серверов. Размещенная в ASP.NET система удаленного взаимодействия использует другой конфигурационный файл, чем приложения в исполняемых файлах EXE, но поддерживает тот же самый синтаксис.

Чтобы использовать инфраструктуру Internet Information Services (IIS) и ASP.NET, требуется создать класс, унаследованный от `System.MarshalByRefObject` и имеющий конструктор по умолчанию. Код, использованный ранее для сервера, который создавал и регистрировал канал, более не нужен; это делается исполняющей средой ASP.NET. Вам

нужно лишь создать виртуальный каталог на веб-сервере, который отображается на каталог, куда будет помещен конфигурационный файл `Web.config`. Сборка удаленного класса должна располагаться в подкаталоге `bin`.

Для конфигурирования виртуального каталога на веб-сервере может использоваться оснастка IIS консоли управления MMC. Выберите **Default Web Site** (Веб-сайт по умолчанию) с помощью меню **Action** (Действие) создайте новый виртуальный каталог.

Конфигурационный файл `Web.config` на веб-сервере должен быть помещен в домашний каталог виртуального веб-сайта. При конфигурации IIS по умолчанию канал, который будет использоваться, прослушивает порт 80.

```
<configuration>
  <system.runtime.remoting>
    <application>
      <service>
        <wellknown mode="SingleCall"
          type="Wrox.ProCSharp.Remoting.Hello, RemoteHello"
          objectUri="HelloService.soap" />
      </service>
    </application>
  </system.runtime.remoting>
</configuration>
```



*Если удаленный объект размещен в IIS, его имя должно оканчиваться либо на `.soap`, либо на `.rem` — в зависимости от типа используемого форматировщика (SOAP или двоичный).*

Клиент может теперь подключаться к удаленному объекту с использованием приведенного ниже конфигурационного файла. URL, который должен быть указан для удаленного объекта — это веб-сервер `localhost`, за которым следует имя приложения `RemoteHello` (выбранное при создании виртуального веб-сайта) и URI удаленного объекта `HelloService.soap`, который определен в файле `Web.config`. Указывать номер порта 80 необязательно, так как это порт по умолчанию для протокола HTTP. Отсутствие раздела `<channel>` означает, что используется загружаемый с задержкой канал HTTP из конфигурационного файла `machine.config`.

```
<configuration>
  <system.runtime.remoting>
    <application>
      <client url="http://localhost/RemoteHello">
        <wellknown type="Wrox.ProCSharp.Remoting.Hello, RemoteHello"
          url="http://localhost/RemoteHello/HelloService.soap" />
      </client>
    </application>
  </system.runtime.remoting>
</configuration>
```



*Хостинг удаленных объектов в ASP.NET поддерживает только хорошо известные объекты; активизируемые клиентом объекты не допускаются.*

## Классы, интерфейсы и утилита Soapsuds

В примерах .NET Remoting, которые рассматривались до сих пор, сборка удаленного объекта всегда копировалась не только на сервер, но также и в клиентское приложение. Таким образом, код MSIL удаленного объекта оказывался как на клиентской, так и на серверной системе, хотя клиентскому приложению нужны лишь метаданные. Однако копиро-

вание сборки удаленного объекта означает невозможность независимого программирования клиента и сервера. Намного лучше использования метаданных является применение интерфейсов или утилиты `Soapsuds.exe`.

## Интерфейсы

Используя интерфейсы, можно добиться более четкого разделения клиентского и серверного кода. Интерфейс просто определяет методы без реализации. Подобным образом контракт (интерфейс) отделяется от реализации, и клиентской системе нужен только контракт. Ниже перечислены шаги, необходимые для использования интерфейса.

1. Определите интерфейс, который будет помещен в отдельную сборку.
2. Реализуйте интерфейс в классе удаленного объекта. Для этого необходимо сослаться на сборку интерфейса.
3. На стороне сервера никаких изменений не требуется. Сервер может программироваться и конфигурироваться обычным образом.
4. На стороне клиента сошлитесь на сборку интерфейса вместо сборки удаленного класса.
5. Клиент может теперь использовать интерфейс удаленного объекта вместо класса удаленного объекта. Объект может быть создан с применением класса `Activator`, как это делалось и ранее. В этом случае применять операцию `new` нельзя, поскольку создать экземпляр интерфейса невозможно.

Интерфейс определяет контракт между клиентом и сервером. Два приложения могут теперь разрабатываться независимо друг от друга. Если вы также привыкли к старым правилам COM относительно интерфейсов (что интерфейс никогда не может изменяться), то никакие проблемы с версиями возникать не будут.

## Утилита `Soapsuds`

Для получения метаданных от сборки, если используется канал HTTP и форматировщик SOAP, можете также использоваться утилита `Soapsuds`. Она позволяет преобразовывать сборки в схемы XML и схемы XML в классы-оболочки; эта утилита работает и в обратном направлении.

Следующая команда преобразует тип `Hello` из сборки `RemoteHello` в сборку `HelloWrapper`, где генерируется прозрачный прокси, который вызывает удаленный объект:

```
soapsuds -types:Wrox.ProCSharp.Remoting>Hello,RemoteHello -oa:HelloWrapper.dll
```

С помощью `Soapsuds` можно также получить информацию о типе непосредственно от работающего сервера, если используется канал HTTP и форматировщик SOAP:

```
soapsuds -url:http://localhost:6792/hello/hi?wsdl -oa:HelloWrapper.dll
```

На клиенте теперь можно сослаться на сгенерированную сборку вместо первоначальной. Некоторые опции `Soapsuds` перечислены в таблице 54.2.

## Асинхронное удаленное взаимодействие

Если серверным методам нужно некоторое время на завершение, а клиент в это же время должен делать какую-то другую работу, не обязательно запускать отдельный поток, чтобы выполнить удаленный вызов. Благодаря асинхронному вызову, можно заставить метод стартовать и немедленно вернуть управление клиенту. Асинхронные вызовы могут выполняться на удаленном объекте, как они выполняются на локальном объекте с помощью делегата. Для методов, которые не возвращают значения, можно также использовать атрибут `OneWay`.

Таблица 54.2. Опции утилиты Soapsuds

Опция	Описание
-url	Извлекает схему из указанного URL
-proxyurl	Если для доступа к серверу необходим прокси-сервер, он указывается с помощью этой опции
-types	Специфицирует тип и сборку для чтения информации схемы
-is	Входной файл схемы
-ia	Входной файл сборки
-os	Выходной файл схемы
-oa	Выходной файл сборки

## Использование делегатов с .NET Remoting

Чтобы создать асинхронный метод, необходимо использовать делегат с тем же аргументом и значением возврата, как в методе `Greeting()` удаленного объекта. Очень полезны для этого обобщенные делегаты `Func<T>` и `Action<T>`. Метод `Greeting()` принимает параметр `string` и возвращает тип `string`, `Func<string, string>`. Это определение делегата для вызова данного метода. Вызов `Greeting()` инициируется с помощью метода `BeginInvoke()` делегата. Второй аргумент `Greeting()` — это экземпляр `AsyncCallback`, определяющий метод `HelloClient.Callback()`. Он вызывается, когда удаленный метод завершает работу. В методе `Callback()` удаленный вызов завершается использованием `EndInvoke()`.

```
using System;
using System.Runtime.Remoting;
namespace Wrox.ProCSharp.Remoting
{
    public class HelloClient
    {
        private static string greeting;
        public static void Main()
        {
            RemotingConfiguration.Configure("HelloClient.exe.config");
            Hello obj = new Hello();
            if (obj == null)
            {
                Console.WriteLine("не удастся обнаружить сервер");
                return;
            }
            Func<string, string> d = new Func<string, string>(obj.Greeting);
            IAsyncResult ar = d.BeginInvoke("Stephanie", null, null);
            // Выполнить некоторую работу и затем ожидать
            ar.AsyncWaitHandle.WaitOne();
            string greeting = null;
            if (ar.IsCompleted)
            {
                greeting = d.EndInvoke(ar);
            }
            Console.WriteLine(greeting);
        }
    }
}
```

Дополнительные сведения о делегатах можно найти в главе 8, а об асинхронном их использовании — в главе 20.

## Атрибут `OneWay`

Метод, возвращающий `void` и имеющий только входные параметры, может быть помечен атрибутом `OneWay`. Атрибут `OneWay` (определенный в пространстве имен `System.Runtime.Remoting.Messaging`) автоматически делает метод асинхронным, независимо от того, как его вызывает клиент. Добавление метода `TakeAWhile()` к классу удаленного объекта `RemoteHello` создает метод, работающий в стиле “запустил и забыл” (`fire-and-forget`). Если клиент вызывает его через прокси, прокси немедленно возвращает управление клиенту. На сервере метод завершается несколько позже.

```
[OneWay]
public void TakeAWhile(int ms)
{
    Console.WriteLine("Метод TakeAWhile запущен");
    System.Threading.Thread.Sleep(ms);
    Console.WriteLine("Метод TakeAWhile завершен");
}
```

## Безопасность в .NET Remoting

Технология .NET Remoting поддерживает безопасность, начиная с версии .NET 2.0. В .NET Remoting обеспечивается конфиденциальность данных, передаваемых по сети, а также аутентификация пользователя.

Безопасность может быть сконфигурирована в каждом канале. TCP, HTTP и IPC поддерживают конфигурацию безопасности. В конфигурации сервера должны быть определены минимальные требования безопасности коммуникаций, в то время как конфигурация клиента определяет средства безопасности. Если клиент определяет конфигурацию безопасности, которая ниже минимальных требований, установленных на сервере, коммуникация не работает.

Начнем с конфигурации сервера. Ниже приведен фрагмент конфигурационного файла с настройкой безопасности на сервере.

С помощью XML-атрибута `protectionLevel` можно указать, требует ли сервер передачи по сети зашифрованных данных. Возможные значения, которые могут быть установлены атрибутом `protectionLevel` — это `None`, `Sign` и `EncryptAndSign`. При значении `Sign` создается подпись, используемая для верификации неизменности данных при их передаче по сети. Однако остается возможность перехвата данных сниффером, который читает сетевой трафик. При значении `EncryptAndSign` данные, которые пересылаются, также шифруются, что исключает возможность их чтения системами, которые официально не участвуют в передаче.

Атрибут `impersonate` может быть установлен в `true` или `false`. Если `impersonate` установлен в `true`, сервер может выступать от имени пользователем клиента для доступа к ресурсам:

```
<channels>
  <channel ref="tcp" port="9001"
    secure="true"
    protectionLevel="EncryptAndSign"
    impersonate="false" />
</channels>
```

В конфигурации клиента определяются возможности сетевой коммуникации. Если они не отвечают требованиям сервера, коммуникация не работает. В приведенном ниже примере конфигурации присутствуют настройки `protectionLevel` и `TokenImpersonationLevel`.

protectionLevel может устанавливаться в те же опции, как было показано в конфигурационном файле сервера.

TokenImpersonationLevel может устанавливаться в Anonymous, Identification, Impersonation и Delegation. При значении Anonymous сервер не может идентифицировать пользователя клиента. Если установлено значение Identification, сервер может определить идентичность пользователя сервера. Если сервер сконфигурирован с impersonate, установленным в true, коммуникация не работает, если клиент сконфигурирован с Anonymous или Identification. Установка TokenImpersonationLevel в Impersonation позволяет серверу выступать от имени клиента не только для доступа к локальным ресурсам сервера, но также использовать идентификацию пользователя для доступа к ресурсам сервера. Значение Delegation возможно, только если для регистрации пользователя используется Kerberos, как это имеет место в случае Active Directory.

```
<channels>
  <channel ref="tcp"
    secure="true"
    protectionLevel="EncryptAndSign"
    TokenImpersonationLevel="Impersonate" />
</channels>
```

В случае, когда канал создается программно, а не через конфигурационный файл, можно также программно определять настройки безопасности. Коллекция с настройками безопасности передается конструктору канала, как показано ниже. Класс коллекции должен реализовывать интерфейс IDictionary, например, обобщенный класс Dictionary, или класс Hashtable:

```
var dict = new Dictionary<string, string>();
dict.Add("secure", "true");
var clientChannel = new TcpClientChannel(dict, null);
```

Дополнительные сведения о безопасности даны в главе 21.

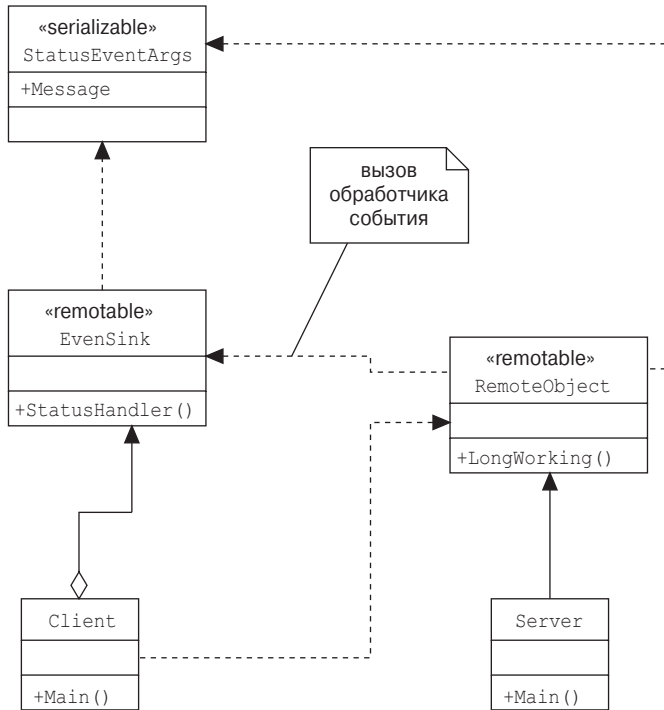
## Удаленное взаимодействие и события

Не только клиент может вызывать методы на удаленном объекте по сети, но и сервер может делать то же самое — вызывать методы на клиенте. Для этого используется известный механизм из базового набора средств языка, а именно — *делегаты и события*.

В принципе, архитектура проста. Сервер имеет удаленный объект, который может быть вызван клиентом, а клиент имеет удаленный объект, который может быть вызван сервером.

- Удаленный объект на сервере должен объявить внешнюю функцию (делегат) с сигнатурой метода, который клиент реализует в обработчике.
- Аргументы, передаваемые функции-обработчику на клиент, должны быть маршализуемыми, т.е. все данные, отправляемые клиенту, должны быть сериализуемыми.
- Удаленный объект должен также объявить экземпляр функции делегата, модифицированной ключевым словом event; клиент использует это для регистрации обработчика.
- Клиент должен создать объект приемника с методом-обработчиком, имеющим ту же сигнатуру, что и определенный делегат, и должен зарегистрировать объект приемника с событием в удаленном объекте.

Рассмотрим пример. Чтобы увидеть все части обработки события в .NET Remoting, создайте пять классов: Server, Client, RemoteObject, EventSink и StatusEventArgs. Зависимости между этими классами показаны на рис. 54.5.



**Рис. 54.5.** Зависимости между классами *Server*, *Client*, *RemoteObject*, *EventSink* и *StatusEventArgs*

Класс *Server* — это удаленный сервер, вроде тех, с которыми вы уже знакомы. Класс *Server* создает канал на основе информации из конфигурационного файла и регистрирует удаленный объект, реализованный в классе *RemoteObject*, в исполняющей среде .NET Remoting. Удаленный объект объявляет аргументы делегата и инициирует события в зарегистрированных функциях-обработчиках. Аргументы, которые переданы функции-обработчику, имеют тип *StatusEventArgs*. Класс *StatusEventArgs* должен быть сериализуемым, чтобы его можно было маршализовать на клиент.

Класс *Client* представляет клиентское приложение. Этот класс создает экземпляр класса *EventSink* и регистрирует метод *StatusHandler()* этого класса в качестве обработчика для делегата удаленного объекта. *EventSink* должен допускать удаленную работу, подобно классу *RemoteObject*, потому что этот класс будет также вызываться через сеть.

## Удаленный объект

Класс удаленного объекта реализован в файле *RemoteObject.cs*. Как уже известно из предыдущих примеров, класс удаленного объекта должен наследоваться от *MarshalByRefObject*. Чтобы позволить клиенту зарегистрировать обработчик событий, который может быть вызван из удаленного объекта, необходимо объявить внешнюю функцию с ключевым словом *delegate*. Объявите делегат *StatusEvent()* с двумя аргументами: *sender* (чтобы клиент знал об объекте, который инициировал событие) и переменной типа *StatusEventArgs*. Во второй аргумент можно помещать дополнительную информацию, которая должна быть передана клиенту.

Метод, который будет реализован клиентом, должен отвечать ряду строгих требований. Он должен иметь только входные параметры — тип возврата, параметры *ref* и *out* не разрешены — и типы аргументов должны быть либо *[Serializable]*, либо допускать уда-



ленное взаимодействие (т.е. наследоваться от `MarshalByRefObject`). Этим требованиям отвечают параметры, определенные делегатом `EventHandler<StatusEventArgs>`.

Внутри класса `RemoteObject` объявите имя события `Status` типа `EventHandler<StatusEventArgs>`. Клиент должен добавить обработчик события `Status` для получения информации о состоянии от удаленного объекта:

```

❏ public class RemoteObject : MarshalByRefObject
{
    public RemoteObject()
    {
        Console.WriteLine("Вызван конструктор RemoteObject");
    }
    public event EventHandler<StatusEventArgs> Status;

```

Фрагмент кода *RemotingAndEvents\RemoteObject\RemoteObject.cs*

Метод `LongWorking()` проверяет, зарегистрирован ли обработчик события, прежде чем событие будет инициировано вызовом `Status(this, e)`.

Чтобы проверить, вызывается ли событие асинхронно, запустите его в начале метода, перед вызовом `Thread.Sleep()`, и после него.

```

public void LongWorking(int ms)
{
    Console.WriteLine("RemoteObject: метод LongWorking() запущен");
    StatusEventArgs e = new StatusEventArgs(
        "Сообщение для клиента: метод LongWorking() запущен");
    // Инициировать событие
    if (Status != null)
    {
        Console.WriteLine("RemoteObject: сгенерировано событие старта");
        Status(this, e);
    }
    System.Threading.Thread.Sleep(ms);
    e.Message = "Сообщение для клиента: метод LongWorking() завершен";
    // Инициировать завершающее событие
    if (Status != null)
    {
        Console.WriteLine("RemoteObject: сгенерировано событие завершения");
        Status(this, e);
    }
    Console.WriteLine("RemoteObject: метод LongWorking() завершен");
}

```

## Аргументы события

Как было показано в классе `RemoteObject`, класс `StatusEventArgs` используется в качестве аргумента для делегата. С помощью атрибута `Serializable` экземпляр этого класса может быть передан из сервера на клиент. Ниже приведен код простого свойства типа `string` для отправки сообщения клиенту:

```

❏ [Serializable]
public class StatusEventArgs : EventArgs
{
    public StatusEventArgs(string message)
    {
        this.Message = message;
    }
    public string Message { get; set; }
}

```


Фрагмент кода *RemotingAndEvents\RemoteObject\RemoteObject.cs*

## Сервер

Сервер реализован в консольном приложении.

С помощью метода `RemotingConfiguration.Configure()` читается конфигурационный файл и тем самым устанавливается канал и удаленные объекты. За счет вызова `Console.ReadLine()` сервер ожидает, пока пользователь не остановит приложение.

```

 using System;
using System.Runtime.Remoting;
namespace Wrox.ProCSharp.Remoting
{
    class Server
    {
        static void Main()
        {
            RemotingConfiguration.Configure("Server.exe.config", true);
            Console.WriteLine("Для завершения нажмите Enter");
            Console.ReadLine();
        }
    }
}


```

Фрагмент кода *RemotingAndEvents\RemoteServer\Program.cs*

## Конфигурационный файл сервера

Как уже обсуждалось, создается также и конфигурационный файл сервера `Server.exe.config`. Следует отметить только один важный момент: удаленный объект должен сохранять состояние клиента, поскольку клиент сначала регистрирует обработчик события и после этого вызывает удаленный метод. Использовать с событиями объекты одиночного вызова невозможно, поэтому класс `RemoteObject` сконфигурирован как активизируемый клиентом тип. Также для поддержки делегатов понадобится включить полную сериализацию, указав атрибут `typeFilterLevel` с элементом `<provider>`.

```

 <configuration>
  <system.runtime.remoting>
    <application name="CallbackSample">
      <service>
        <activated type="Wrox.ProCSharp.Remoting.RemoteObject,
                      RemoteObject" />
      </service>
      <channels>
        <channel ref="tcp" port="6791">
          <serverProviders>
            <provider ref="binary" typeFilterLevel="Full" />
          </serverProviders>
        </channel>
      </channels>
    </application>
  </system.runtime.remoting>
</configuration>

```

Фрагмент кода *RemotingAndEvents\RemoteServer\app.config*

## Приемник событий

Библиотека приемника событий должна использоваться клиентом и вызываться сервером. Приемник событий реализует обработчик `StatusHandler()`, который определен с

делегатом. Как упоминалось ранее, метод может иметь только входные параметры и возвращать `void`. Класс `EventSink` должен также вызываться удаленно с сервера.

```

using System;
using System.Runtime.Remoting.Messaging;
namespace Wrox.ProCSharp.Remoting
{
    public class EventSink : MarshalByRefObject
    {
        public EventSink()
        {
        }
        public void StatusHandler(object sender, StatusEventArgs e)
        {
            Console.WriteLine("EventSink: произошло событие: " + e.Message);
        }
    }
}

```

Фрагмент кода *RemotingAndEvents\EventSink\EventSink.cs*

## Клиент

Клиент читает клиентский конфигурационный файл с помощью класса `RemotingConfiguration`, что не отличается от клиентов, которые обсуждались до сих пор. Клиент создает экземпляр класса удаленного приемника `EventSink` локально. Метод, который должен быть вызван из удаленного объекта на сервере, передается удаленному объекту.

```

using System;
using System.Runtime.Remoting;
namespace Wrox.ProCSharp.Remoting
{
    class Client
    {
        static void Main()
        {
            RemotingConfiguration.Configure("Client.exe.config", true);
            Console.WriteLine("ожидание сервера...");
            Console.ReadLine();
        }
    }
}

```

Фрагмент кода *RemotingAndEvents\RemoteClient\Program.cs*

А вот здесь начинаются отличия. Требуется создать экземпляр класса приемника `EventSink` локально. Поскольку этот класс не будет сконфигурирован элементом `<client>`, его экземпляр создается локально. Затем создается экземпляр объекта удаленного класса `RemoteObject`. Этот класс сконфигурирован в элементе `<client>`, поэтому его экземпляр создается на удаленном сервере.

```

var sink = new EventSink();
var obj = new RemoteObject();
if (!RemotingServices.IsTransparentProxy(obj))
{
    Console.WriteLine("проверка удаленной конфигурации");
    return;
}

```

Теперь можно зарегистрировать метод-обработчик объекта `EventSink` в удаленном объекте. `StatusEvent` — это имя делегата, который был определен в сервере. Метод `StatusHandler()` имеет те же аргументы, что и определены в `StatusEvent`.

Вызывая метод `LongWorking()`, сервер вызовет `StatusHandler()` в начале и конце метода:


```
// Зарегистрировать клиентский приемник на сервере, т.е. подписаться на событие
obj.Status += sink.StatusHandle;
obj.LongWorking(5000);
```

Теперь вы не заинтересованы в получении событий с сервера, поэтому отменяете подписку на событие. Когда в следующий раз будет вызван метод `LongWorking()`, никакого события не будет получено:

```
// Отписаться от события
obj.Status -= sink.StatusHandler;
obj.LongWorking(5000);
Console.WriteLine("Для завершения нажмите Enter");
Console.ReadLine();
}
}
}
```

## Клиентский конфигурационный файл

Конфигурационный файл клиента `client.exe.config` — почти такой же, как конфигурационный файл для активизируемых клиентом объектов. Отличие связано с определением номера порта для канала. Поскольку сервер должен добираться до клиента через известный порт, номер порта для канала должен быть определен как атрибут элемента `<channel>`. Определение раздела `<service>` для класса `EventSink` необходимо, потому что экземпляр класса будет создан из клиента за счет локального применения операции `new`. Сервер не имеет доступа к этому объекту по его имени; вместо этого он получит маршализованную ссылку на экземпляр.



```
<configuration>
  <system.runtime.remoting>
    <application name="Client">
      <client url="tcp://localhost:6791/CallbackSample">
        <activated type="Wrox.ProCSharp.Remoting.RemoteObject,
          RemoteObject" />
      </client>
      <channels>
        <channel ref="tcp" port="0">
          <serverProviders>
            <provider ref="binary" typeFilterLevel="Full" />
          </serverProviders>
        </channel>
      </channels>
    </application>
  </system.runtime.remoting>
</configuration>
```

Фрагмент кода `RemotingAndEvents\RemoteClient\app.config`

## Запуск программ

Ниже приведен результирующий вывод сервера. Конструктор удаленного объекта вызывается однажды, потому что в наличии активизируемый клиентом объект. Затем можно видеть начало вызова `LongWorking()` и инициацию события для клиента. Следующий запуск метода `LongWorking()` не иницирует события, так как клиент уже закрыл регистрацию на него.

```
Для завершения нажмите Enter
Вызван конструктор RemoteObject
RemoteObject: метод LongWorking() запущен
RemoteObject: сгенерировано событие старта
RemoteObject: сгенерировано событие завершения
```

```
RemoteObject: метод LongWorking() завершен  
RemoteObject: метод LongWorking() запущен  
RemoteObject: метод LongWorking() завершен
```

В выводе клиента можно видеть, как события передаются по сети:

```
ожидание сервера...  
EventSink: произошло событие: Сообщение для клиента: метод LongWorking() запущен  
EventSink: произошло событие: Сообщение для клиента: метод LongWorking() завершен
```

## Контекст вызова

Активируемые клиентом объекты могут хранить состояние для определенного клиента. С активируемыми клиентом объектами сервер выделяет ресурсы для каждого клиента. С активируемыми сервером объектами `SingleCall` новый экземпляр создается для каждого вызова, и никакие ресурсы на сервере не удерживаются; эти объекты не могут хранить состояние для клиента. Для управления состоянием можно сохранять состояние на стороне клиента; при каждом вызове метода информация о состоянии пересылается на сервер. Чтобы реализовать такое управление состоянием, не нужно изменять все сигнатуры методов для добавления дополнительного параметра, который будет передавать состояние на сервер; это делается автоматически с помощью *контекста вызова*.

Контекст вызова сопровождает логический поток и передается с каждым вызовом метода. *Логический поток* начинается с вызова потока и продолжается через все вызовы методов, которые начинаются с вызывающего потока, проходя через разные контексты, различные домены приложений и различные процессы.

Для присваивания данных контексту вызова применяется `CallContext.SetData()`. Класс объекта, используемого в качестве данных для метода `SetData()`, должен реализовывать интерфейс `ILogicalThreadAffinative`. Эти данные можно получать в том же логическом потоке (но, возможно, в другом физическом потоке) с помощью метода `CallContext.GetData()`.

Для данных контекста вызова создайте новую библиотеку классов C# с классом `CallContextData`. Этот класс будет применяться для передачи некоторых данных от клиента серверу при каждом вызове метода. Такой класс, передаваемый с контекстом вызова, должен реализовывать интерфейс `System.Runtime.Remoting.Messaging.ILogicalThreadAffinative`. Этот интерфейс не имеет метода; он просто служит меткой для исполняющей среды для определения, что экземпляры класса должны сопровождать логический поток. Класс `CallContextData` также должен быть помечен атрибутом `Serializable`, чтобы его можно было передавать по каналу.

```
using System;  
using System.Runtime.Remoting.Messaging;  
namespace Wrox.ProCSharp.Remoting  
{  
    [Serializable]  
    public class CallContextData : ILogicalThreadAffinative  
    {  
        public CallContextData()  
        {  
        }  
        public string Data { get; set; }  
    }  
}
```

В классе удаленного объекта `Hello` измените метод `Greeting()` для обращения к контексту вызова. Чтобы использовать класс `CallContextData`, необходимо сослаться на ранее созданную сборку `CallContextData` из файла `CallContextData.dll`. Для работы с классом `CallContextData` должно быть открыто пространство имен `System.Runtime`.

Remoting.Messaging. Переменная `cookie` хранит данные, которые передаются от клиента серверу. Имя `cookie` выбрано потому, что контекст работает подобно cookie-наборам браузера, в которых клиент автоматически передает данные веб-серверу.

```
public string Greeting(string name)
{
    Console.WriteLine("Метод Greeting запущен");
    CallContextData cookie = (CallContextData)CallContext.GetData("mycookie");
    if (cookie != null)
    {
        Console.WriteLine("Значение cookie: " + cookie.Data);
    }
    Console.WriteLine("Метод Greeting завершен");
    return "Hello, " + name;
}
```

В коде клиента информация контекста вызова устанавливается с помощью `CallContext.SetData()`. Посредством этого метода экземпляр класса `CallContextData` присваивается для передачи на сервер. Теперь при каждом вызове метода `Greeting()` в цикле `for` данные контекста автоматически передаются на сервер.

```
CallContextData cookie = new CallContextData();
cookie.Data = "information for the server";
CallContext.SetData("mycookie", cookie);
for (int i=0; i < 5; i++)
{
    Console.WriteLine(obj.Greeting("Christian"));
}
```

Контекст вызова может использоваться для отправки сведений о пользователе, имени клиентской системы или просто для передачи уникального идентификатора на сторону сервера для получения некоторой информации о состоянии из базы данных.

## Резюме

В этой главе было показано, каким образом .NET Remoting упрощает задачу вызова методов через сеть. Удаленный объект должен быть унаследован от `MarshalByRefObject`. В серверном приложении нужен только один метод для загрузки конфигурационного файла, что позволит настроить и запустить каналы и удаленные объекты. Внутри клиента вы загружаете конфигурационный файл и можете использовать операцию `new` для создания экземпляров удаленного объекта.

Также была продемонстрирована работа .NET Remoting без помощи конфигурационных файлов. На сервере был просто создан канал и зарегистрирован удаленный объект. На клиенте создавался канал и этот удаленный объект использовался.

Вдобавок было показано, что архитектура .NET Remoting является гибкой и расширяемой. Все части этой технологии, такие как каналы, прокси, форматировщики, приемники сообщений и т.п., являются подключаемыми и могут быть заменены специальными реализациями.

Вы научились применять каналы HTTP, TCP и IPC для коммуникаций по сети, а форматировщики SOAP и двоичные форматировщики — для форматирования параметров перед их отправкой.

Вы узнали о типах объектов с состоянием и без состояния, которые используются хорошо известными и активизируемыми клиентом объектами. Для активизируемых клиентом объектов было продемонстрировано применение механизма аренды для спецификации времени жизни удаленных объектов.

Также было показано, что .NET Remoting очень хорошо интегрируется с другими частями .NET Framework, в том числе вызовом асинхронных методов, выполнением обратных вызовов с помощью ключевых слов `delegate` и `event` и т.п.