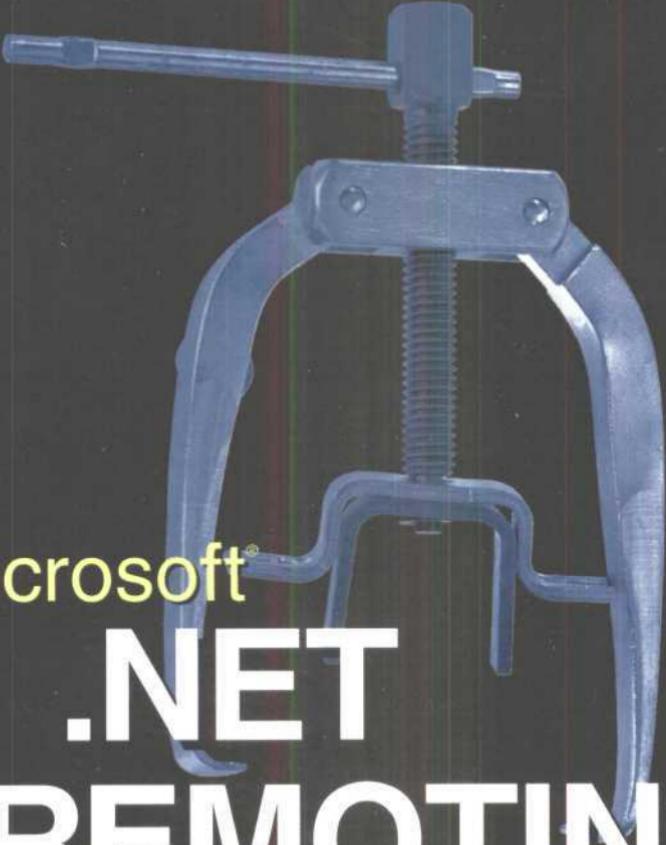


Скотт Маклин
Джеймс Нафтел
Ким Уильямс



Microsoft[®]
.NET
REMOTING

Microsoft
.net

 РУССКАЯ РЕДАКЦИЯ

Microsoft[®]

Моим родителям, никогда не сомневавшимся во мне, и моей жене Нэнси — любви всей моей жизни.

Скотт

Я посвящаю эту книгу моим дочерям, Миган и Эмме, моей жене Эприл и моей семье. Все вы ~ главное, что у меня есть в жизни.

Джеймс

Моей жене Пэтти и сыну Шону. Я люблю вас обоих и очень ценю вашу поддержку.

Ким

Scott McLean
James Naftel
Kim Williams

MICROSOFT®
.NET
REMOTING

Microsoft Press

Скотт Маклин
Джеймс Нафтел
Ким Уильяме

MICROSOFT®
**.NET
REMOTING**

Москва 2003

 РУССКАЯ РЕДАКЦИЯ

УДК 004.45
ББК 32.973.26-018.2
M15

Маклин С., Нафтел Дж., Уильямс К.

M15 Microsoft .NET Remoting/Пер. с англ. — М.: Издательско-торговый дом «Русская Редакция», 2003. — 384 с.: ил.

ISBN 5-7502-0229-1

В этой книге обсуждаются особенности функционирования и архитектуры .NET Remoting — новейшей технологии для построения распределенных приложений. Вы узнаете, как использовать беспрецедентные возможности этой технологии для построения отказоустойчивых, масштабируемых, защищенных, быстрых и простых в сопровождении и администрировании распределенных Интернет-приложений.

Книга состоит из 8 глав и предметного указателя.

Предназначена разработчикам, имеющим опыт работы с Microsoft .NET Framework и C#.

УДК 004.45
ББК 32.973.26-018.2

Подготовлено к изданию по лицензионному договору с Microsoft Corporation, Редмонд, Вашингтон, США.

ActiveX, BackOffice, JScript, Microsoft, Microsoft Press, MSDN, NetShow, Outlook, PowerPoint, Visual Basic, Visual C++, Visual InterDev, Visual J++ , Visual SourceSafe, Visual Studio, Win32, Windows и Windows NT являются товарными знаками или охраняемыми товарными знаками корпорации Microsoft в США и/или других странах. Все другие товарные знаки являются собственностью соответствующих фирм.

Все названия компаний, организаций и продуктов, а также имена лиц, используемые в примерах, вымышлены и не имеют никакого отношения к реальным компаниям, организациям, продуктам и лицам,

- © Оригинальное издание на английском языке, Scott McLean, James Naftel, Kim Williams, 2003
- © Перевод на русский язык, Microsoft Corporation, 2003
- © Оформление и подготовка к изданию, издательско-торговый дом «Русская Редакция», 2003

ISBN 0-7356-1778-3 (англ.)
ISBN 5-7502-0229-1

Оглавление

Благодарности	XII
Введение	XIII
Кому предназначена эта книга.....	XIV
Структура книги.....	XIV
Системные требования.....	XVI
Файлы примеров.....	XVII
 ГЛАВА 1 ОСНОВЫ РАСПРЕДЕЛЕННЫХ	
ПРИЛОЖЕНИЙ	1
Исторический экскурс.....	2
Распределенные архитектуры.....	2
Модульное программирование.....	3
Клиент-серверные архитектуры.....	3
Многоуровневые архитектуры.....	4
Одноранговые архитектуры.....	5
Распределенные технологии.....	7
Сокеты.....	7
Удаленные вызовы <i>процедур</i>	8
Распределенные объекты — удобная абстракция.....	8
Преимущества распределенных приложений.....	10
Отказоустойчивость.....	10
Масштабируемость.....	11
Администрирование.....	11
Проблемы распределенных приложений.....	12
Производительность.....	12
Защита.....	15
Открытость и сетевые форматы данных.....	15
Интернет и брандмауэры.....	16
Конфигурирование.....	17
Независимость от расположения.....	17
Управление временем жизни объекта.....	17
Решение проблем распределенных приложений	
в .NET Remoting.....	18

Производительность.....	19
Расширение и настройка удаленного взаимодействия.....	19
Конфигурирование.....	20
Преимущества CLR и CTS.....	21
Открытость.....	22
Защита.....	24
Аутентификация.....	24
Защита данных.....	24
Защита без IIS.....	25
Управление временем жизни.....	25
Сервисы масштаба предприятия.....	25
Заключение.....	26
ГЛАВА 2 АРХИТЕКТУРА .NET REMOTING.....	27
Границы .NET Remoting.....	27
Домены приложений.....	28
Контексты.....	28
Пересечение границ.....	28
Недистанцируемые типы.....	29
Дистанцируемые типы.....	29
Активизация объекта.....	33
Серверная активизация.....	34
Режим Singleton.....	34
Режим SingleCall.....	36
Клиентская активизация.....	37
Лицензия объекта.....	39
Лицензия.....	40
Диспетчер лицензий.....	41
Спонсоры.....	42
Пересечение границ приложений.....	43
Маршалинг ссылок на удаленные объекты посредством ObjRef.....	43
URI.....	45
Метаданные.....	45
Информация о канале.....	45
Использование прокси для взаимодействия с удаленными объектами.....	46
Прозрачный прокси.....	46
Реальный прокси.....	47
Сообщения — основа удаленного взаимодействия.....	48
Транспортировка сообщений каналами.....	49
TCP... ..	49

HTTP.....	50
Обработка сообщений цепочками канальных приемников ...	50
Сериализация сообщений форматирующими приемниками.....	52
Реализация сетевого интерфейса с помощью транспортных приемников.....	53
Заключение.....	55

ГЛАВА 3 РАСПРЕДЕЛЕННЫЕ ПРИЛОЖЕНИЯ

НА ОСНОВЕ .NET REMOTING.....	56
Проектирование приложения.....	57
Реализация приложения <i>JobServer</i>	57
Логика приложения.....	58
Структура <i>JobInfo</i>	58
Интерфейс <i>IJobServer</i>	58
Класс <i>JobEventArgs</i>	59
Класс <i>JobServerImpl</i>	60
Добавление средств .NET Remoting.....	63
Реализация дистанцируемого типа.....	64
Выбор серверного домена приложения.....	65
Выбор модели активизации.....	68
Выбор канала и порта.....	71
Выбор способа получения клиентами метаданных сервера.....	72
Настройка параметров .NET Remoting для сервера.....	72
Реализация приложения <i>JobClient</i>	74
Выбор клиентского домена приложения.....	74
Получение метаданных сервера.....	83
Настройка .NET Remoting для приложения <i>JobClient</i>	85
Программная настройка.....	86
Конфигурационный файл.....	88
Реализация класса <i>JobServerImpl</i> в виде Web-сервиса.....	90
Необходимые изменения.....	93
Удаление обратных вызовов клиентов.....	93
Выбор метода активизации.....	94
Настройка виртуального каталога.....	94
Настройка файла <i>Web.config</i>	95
Развертывание.....	95
Использование <i>SOAPSuds</i>	96
Сборка с реализацией.....	96
Сборка с метаданными.....	96
XML-схема (<i>WSDL</i>).....	97

Компилируемый класс.....	97
Защита Web-сервиса.....	98
Изменения настроек виртуального каталога.....	99
Изменения в файле <code>Web.config</code>	99
Изменения в приложении <code>JobClient</code>	99
Использование защиты на основе ролей с <code>.NET Remoting</code>	101
Использование объектов с клиентской активизацией.....	104
Класс <code>JobNotes</code>	104
Изменения в приложении <code>JobClient</code>	106
Настройка клиента для работы с объектами с клиентской активизацией.....	109
Программная настройка.....	110
Конфигурационный файл.....	110
Настройка сервера при использовании объектов с клиентской активизацией.....	111
Программная настройка.....	111
Конфигурационный файл.....	112
Спонсор лицензии.....	112
Инициализация лицензии.....	114
Реализация интерфейса <code>ISponsor</code>	114
Регистрация спонсора.....	115
Проблемы зависимости от метаданных.....	116
Устранение зависимости <code>JobServer</code> от метаданных <code>JobClient</code>	116
Разработка класса-дублера, публикуемого вместо метаданных <code>JobServerImpl</code>	119
Удаленный доступ к интерфейсу <code>IJobServer</code>	120
Заключение.....	122
 ГЛАВА 4 SOAP И ОБМЕН СООБЩЕНИЯМИ	123
Протокол SOAP.....	123
Нужно ли знать SOAP?.....	125
RPC на основе HTTP.....	125
Элементы сообщения SOAP.....	126
Конверт SOAP.....	126
Заголовок SOAP	127
Тело SOAP.....	128
Ошибки в SOAP.....	129
Документ-литеральный SOAP.....	129
Обмен сообщениями.....	ПО
Сообщение-запрос <code>add JobEvent</code>	130
Сообщение-ответ <code>add JobEvent</code>	136

Сообщение-запрос <code>GetJobs</code>	137
Сообщение-ответ <code>GetJobs</code>	137
Сообщение-запрос <code>CreateJob</code>	139
Сообщение-ответ <code>CreateJob</code>	139
Сообщение-запрос <code>UpdateJobState</code>	140
Сообщение-ответ <code>UpdateJobState</code>	141
Сообщение-запрос на активизацию <code>JobNotes</code>	141
Сообщение-ответ на активизацию <code>JobNotes</code>	143
Сообщение-запрос <code>removeJobEvent</code>	145
Сообщение-ответ <code>removeJobEvent</code>	150
Заключение.....	150

ГЛАВА 5 СООБЩЕНИЯ И ПРОКСИ.....151

Сообщения.....	151
Сообщения вызовов конструкторов.....	152
Сообщения вызовов методов.....	153
Типы сообщений.....	153
Прокси.....	156
<code>TransparentProxy</code>	156
<code>RealProxy</code>	158
Расширение <code>RealProxy</code>	158
Специализированные прокси на практике.....	159
Пример с активизацией.....	160
Пример со сменой канала.....	166
Пример с распределением нагрузки.....	171
Использование контекста вызова.....	174
Контекст вызова в потоке сообщений.....	179
Заключение.....	180

ГЛАВА 6 ПРИЕМНИКИ СООБЩЕНИЙ

И КОНТЕКСТЫ.....181

Приемники сообщений.....	181
<code>IMessageSink</code>	182
Синхронная обработка сообщения.....	184
Асинхронная обработка сообщения.....	184
Контексты.....	186
Установление контекста.....	186
Атрибуты и свойства контекста.....	187
Контексты и удаленное взаимодействие.....	191
Динамические контекстные приемники.....	193
Создание динамического приемника.....	194

Клиентская контекстная цепочка	195
Создание приемника для клиентской контекстной цепочки.....	196
Серверная контекстная цепочка приемников	196
Создание приемника для серверной контекстной цепочки.....	197
Пример; контекст с регистрацией исключений.....	198
Серверная объектная цепочка приемников.....	205
Создание серверного объектного приемника.....	207
Пример: трассировка всех вызовов методов объекта.....	207
Цепочка агентских приемников.....	212
Создание агентского приемника.....	214
Пример: проверка параметров метода.....	215
Заключение.....	224

ГЛАВА 7 КАНАЛЫ И КАНАЛЬНЫЕ

ПРИЕМНИКИ.....225

Построение каналов.....	225
Терминология каналов.....	226
HttpChannel.....	227
HttpServerChannel.....	230
HttpServerTransportSink.....	232
HttpClientChannel.....	233
HttpClientTransportSinkProvider.....	234
HttpClientTransportSink.....	234
Создание нестандартных каналов.....	235
Этапы создания нестандартного канала .NET Remoting.....	236
Создание нестандартного канала FileChannel.....	237
Реализация класса FileClientChannel.....	238
Реализация класса FileClientChannelSinkProvider.....	243
Реализация класса FileClientChannelSink.....	244
Реализация класса FileServerChannel.....	249
Реализация класса FileServerChannelSink.....	254
Реализация класса FileChannel.....	259
Реализация класса FileChannelHelper.....	261
Создание класса транспорта FileChannel.....	263
Нестандартные каналные приемники.....	267
Создание приемника контроля времени доступа.....	269
Реализация класса AccessTimeServerChannelSink.....	270
Реализация класса AccessTimeServerChannelSinkProvider	274

Добавление <code>AccessTimeServerChannelSink</code> в конфигурационный файл.....	275
Заключение.....	276

ГЛАВА 8 ФОРМАТИРОВЩИКИ

СЕРИАЛИЗАЦИИ.....	277
Сериализация объектов.....	277
Атрибут <code>Serializable</code>	278
Расширение механизма сериализации объекта.....	279
Сериализация графов объектов.....	283
Десериализация графа объектов.....	285
Уведомление о завершении десериализации.....	285
Суррогаты сериализации и селекторы суррогатов.....	286
Суррогаты.....	286
Селекторы суррогатов.....	287
Суррогат <code>TimeStamper</code>	287
<code>RemotingSurrogateSelector</code>	290
Форматировщики сериализации.....	290
Получение сериализуемых членов типа.....	291
Обход графа объектов.....	293
Идентификация объектов с использованием класса	
<code>ObjectIDGenerator</code>	294
Каталогизация объектов для сериализации.....	295
Использование класса <code>ObjectManager</code>	296
Использование класса <code>Formatter</code>	299
Реализация собственного форматировщика.....	301
Определение формата сериализации.....	301
Реализация интерфейса <code>IFormatter</code>	305
Реализация метода <code>IFormatter.Serialize</code>	307
Реализация метода <code>IFormatter.Deserialize</code>	320
Создание приемника форматировщика.....	329
Клиентский приемник форматировщика.....	330
<code>ClientFormatterSinkProvider</code>	335
Серверный приемник форматировщика.....	336
<code>ServerFormatterSinkProvider</code>	341
Заключение.....	342
Предметный указатель.....	343
Об авторах.....	357

Благодарности

Мы не написали бы эту книгу без помощи многих людей.

Прежде всего мы говорим огромное спасибо сотрудникам Microsoft Press, которые сделали возможным издание этой книги: выпускающему редактору Даниэль Берд, редактору проекта Кэтлин Аткинс, редактору рукописи Мишель Гудман, техническому редактору Дейлу Мэджу-младшему, художнику Робу Нанси, составителю Кэрри Деваль и Марку Янгу, техническому редактору, прочитавшему наш текст на начальной стадии проекта. Мы благодарим наших коллег, предоставивших свои отзывы, Аллена Джонса и Адама Фримена. Мы также признательны ребятам из Moore Literary Agency: Майку Михэну, Клодетте Мур и Дэби Маккенне за их помощь в издании этой книги.

И еще: работа над книгой отнимает массу времени — в том числе и то, которое мы обычно посвящаем общению со своими близкими. Мы отлично понимаем, что без их всесторонней поддержки мы бы не справились с нашей фундаментальной задачей. И выражаем им нашу искреннюю признательность.

Введение

Распределенные вычисления в наши дни стали неотъемлемой частью любого программного продукта. До появления .NET Remoting, рекомендуемым методом разработки распределенных приложений для платформ Microsoft была DCOM. К сожалению, среднему разработчику непросто изучить DCOM и работать с ней. .NET Remoting — это объектно-ориентированная архитектура для поддержки распределенных приложений в Microsoft .NET. Подобно тому как .NET Framework заменяет COM в качестве средства разработки компонентов, .NET Remoting заменяет DCOM в качестве средства создания распределенных приложений на основе .NET Framework. Более того .NET Remoting является основой для .NET Web-сервисов. Таким образом, понимание основ .NET Remoting совершенно необходимо для разработки на основе .NET Framework распределенных приложений, в том числе для Интернета.

В этой книге подробно рассказано об архитектуре .NET Remoting и представлены конкретные примеры на языке C#, демонстрирующие способы ее расширения и настройки. Мы рассмотрим возможности, предоставляемые .NET Remoting, и разработаем программы, демонстрирующие настройку ключевых средств .NET Remoting, то есть познакомим вас с тем, что составляет истинное преимущество .NET Remoting. Кроме того, стоит учесть, что архитектура .NET Remoting имеет множество точек расширения (extensibility hooks), позволяющих использовать различные протоколы и конфигурации.

Начав работать с .NET Framework, мы были приятно удивлены тому, как легко создавать распределенные приложения с помощью .NET Remoting. Гораздо проще и удобней, чем в DCOM! Более того, начав расширять инфраструктуру .NET Remoting, мы быстро поняли, в чем ее истинная мощь. В общем, мы обнаружили, что логичная и согласованная объектная модель .NET Remoting поддерживает как простые изменения конфигурации,

так и замысловатые расширения инфраструктуры .NET Remoting. Кроме того, .NET Remoting поддерживает открытые Интернет-стандарты, такие, как Web-сервисы и протокол SOAP (Simple Object Access Protocol). К сожалению, мир несовершенен — каждая новая технология грешит своими проблемами. Тем не менее практически для всех тех, что возникли в ходе нашей работы, удавалось найти разумные обходные решения. (В книге такие обходные варианты отмечены *особо*.) Нам удалось на практике воспользоваться новыми технологиями, и мы считаем .NET Remoting достойной заменой своей предшественнице (DCOM), а также мощным инструментом для создания распределенных приложений в современном мире открытых Интернет-решений,

Кому предназначена эта книга

Книгу освоит любой, кто имеет некоторый опыт написания программ для .NET Framework и желает научиться создавать распределенные приложения с помощью .NET Remoting. .NET Remoting рассматривается подробно — от читателя не требуется никаких предварительных знаний по этому предмету. Все примеры написаны на C#, поэтому базовые сведения об этом языке необходимы; однако мы не будем интенсивно использовать его сложные возможности. Хотя определенные познания в .NET Framework и C# все-таки требуются, эту книгу легко поймет любой, имеющий опыт работы на C++ , Microsoft Visual Basic .NET или Java. Если вам приходилось писать распределенные приложения с использованием одного из этих языков, то ваших познаний должно хватить для понимания большинства материалов книги.

Структура книги

Книга состоит из восьми глав. Первые две главы носят концептуальный характер. В остальных описаны более сложные возможности и демонстрируется, как использовать фантастические возможности расширения, предоставляемые .NET Remoting.

- **Глава 1: Основы распределенных приложений** Здесь закладываются основы и обсуждается история архитектуры и технологии распределенных приложений. В этой главе описаны технологии RPC, DCOM, RMI (Remote Method Invocation) и SOAP/XML. Цель главы — рассказать о преимуществах и не-

достатках этих технологий. Затем подробно рассматривается, каким образом .NET Remoting решает как традиционные, так и новые задачи, возникающие при разработке распределенных приложений.

- **Глава 2: Архитектура .NET Remoting** Посвящена основным архитектурным компонентам инфраструктуры .NET Remoting, которые подробно разбираются в последующих главах. Эта глава является одновременно введением и справочником по базовым концепциям .NET Remoting. Здесь представлены вводные сведения по основным компонентам архитектуры .NET Remoting: типы активизации (серверная и клиентская); *маршalling по ссылке*, *маршalling по значению*, *лицензии*, каналы, сообщения и форматировщики.
- **Глава 3: Распределенные приложения на основе .NET Remoting** Содержит подробный обзор методов построения распределенных приложений на основе различных стандартных средств .NET Remoting. Мы разработаем гипотетическое приложение *распределения заданий*, на примере которого продемонстрируем фундаментальные концепции .NET Remoting, такие, как клиентская и серверная активизация объектов. Кроме того, здесь рассказано о том, как использовать .NET Remoting для *создания Web-сервисов*, а также о том, как обеспечить защиту приложений .NET Remoting на основе мощных средств защиты Microsoft Internet Information Services (IIS), и демонстрируется, как сделать из удаленного объекта Web-сервис.
- **Глава 4: SOAP и обмен сообщениями** Эта глава познакомит вас с основами SOAP, здесь рассматриваются *сообщения*, которыми обмениваются клиентские и серверные приложения, разработанные в главе 3. Дополнительно мы продемонстрируем внешние данные, генерируемые и потребляемые .NET Remoting.
- **Глава 5: Сообщения и прокси** Вначале рассматриваются сообщения, которые являются *основой* для расширения и настройки инфраструктуры .NET Remoting. Далее речь пойдет о прокси, выполняющих роль мостов между локальными и удаленными объектами. Код клиента вызывает *объект-прокси*, который, в свою очередь, вызывает методы удаленного объек-

та. Мы продемонстрируем три метода создания специализированных прокси и способы их подключения к инфраструктуре .NET Remoting. С помощью специализированных прокси разработаем два приложения, одно из которых динамически переключается с TCP на HTTP, если работа по TCP невозможна (например, из-за брандмауэра), а другое обеспечивает равномерное распределение нагрузки.

- **Глава 6: Приемники сообщений и контексты** Здесь рассказывается о том, как использовать контекст .NET Remoting для управления правилами и поведением объектов, исполняющихся в этом контексте. Мы объясним, что такое цепочки приемников сообщений и почему они являются основной точкой расширения архитектуры .NET Remoting, предоставляя фундамент, на котором построены мощные возможности контекстов по перехвату событий. Кроме того, мы расскажем о каждом из связанных с контекстом сообщений и покажем, как их использовать.
- **Глава 7: Каналы и каналные приемники** Каналы являются фундаментальными компонентами .NET Remoting. Чтобы вы лучше поняли, как создается специализированный канал, эту главу мы начнем с описания архитектуры канала *HttpChannel* и поддерживающих его классов. Затем мы рассмотрим пример расширения .NET Remoting с помощью специализированного канала, использующего в качестве механизма транспорта сообщений .NET Remoting файловую систему. В заключение главы мы создадим специализированный приемник, блокирующий вызовы методов в течение заданного пользователем промежутка времени.
- **Глава 8: Форматировщики сериализации** В заключительной главе на основе материала, представленного в предыдущих главах, подробно рассказывается о форматировщиках сериализации. После общих понятий сериализации мы покажем, как расширить .NET Remoting, создав специализированный форматировщик сериализации и приемник форматирования.

Системные требования

Для компоновки и исполнения примеров программ вам понадобится Microsoft Visual Studio .NET. Для запуска Web-сервисов и

демонстрации приемов защиты, обсуждающихся в главе 3, вам также придется установить IIS. Хотя многие возможности .NET Remoting лучше всего демонстрируются в сети из двух или более компьютеров, для запуска примеров из этой книги достаточно единственного компьютера.

Файлы примеров

Файлы примеров для этой книги вы найдете в Интернете по адресу <http://www.microsoft.com/mspress/books/6172.asp>. Войдя на этот сайт, для загрузки примеров к книге щелкните ссылку Companion Content в меню More Information, расположенном в правой части Web-страницы. По этой команде будет загружена страница Companion Content, содержащая ссылки для загрузки файлов примеров.

Основы распределенных приложений

Разработчикам пришлось потрудиться над технологиями построения распределенных приложений, такими, как DCOM, Java RMI и CORBA, пока те не стали более-менее полно удовлетворять требования пользователей. Современная технология распределенных приложений должна быть эффективной, расширяемой, поддерживающей транзакции, позволяющей взаимодействовать с другими технологиями, обладающей большими возможностями настройки, поддерживать работу в Интернету и т. д. Однако такой широкий набор возможностей требуется далеко не ото всех приложений. Для поддержки простых систем достаточно, чтобы технология распределенных приложений предоставляла стандартные решения и обеспечивала максимально легкую настройку.

Кажется, что одно отдельно взятое решение не в состоянии обеспечить весь этот список требований. Фактически большинство современных технологий распределенных приложений вначале отвечали более скромным требованиям, и лишь со временем в них появлялась поддержка и дополнительных возможностей.

Иногда лучше все стереть и написать заново. Так и сделали разработчики .NET Remoting. Эта технология предоставляет согласованную модель объектов, имеющую точки расширения для поддержки систем, которые до настоящего времени создавались на основе DCOM. Проектировщики .NET Remoting имели возможность учесть последние требования технологии, которые не были известны создателям DCOM.

Хотя в этой главе мы не ставили перед собой задачу помочь вам в выборе технологии распределенных приложений, нужно отметить, что .NET Remoting имеет явные преимущества. Если все новые разработки вы ведете на основе .NET и для реализации как клиента, так и сервера используете .NET Framework, то .NET Remoting, вероятно, будет наилучшим выбором. С другой стороны, .NET Remoting предоставляет беспрецедентный уровень совместимости со старыми технологиями на тот случай, если ваше распределенное приложение уже реализовано с использованием иной технологии удаленного доступа.

- Если приложение применяет COM/DCOM, то слой взаимодействия .NET-COM с .NET Framework предоставляет полный набор средств и прост в работе. Данный слой позволяет переходить к использованию .NET Remoting постепенно.
- Если ваша система построена не на основе распределенной технологии Microsoft, а на базе, например, Java Remote Method Invocation (RMI) или Common Object Request Broker Architecture (CORBA), то и здесь все не так плохо. Поддержка в .NET Remoting открытых стандартов, таких, как XML и Simple Object Access Protocol (SOAP), обеспечивает взаимодействие между приложениями на разных платформах и от разных поставщиков по мере перехода последних к использованию открытых стандартов. На данный момент доступно множество средств разработки (toolkit) для SOAP на Java. Хотя поддержка SOAP в CORBA запаздывает, в настоящий момент Object Management Group (OMG) работает над официальным стандартом CORBA/SOAP.

Исторический экскурс

В самом широком смысле *распределенное приложение* — это то, в котором обработка происходит на двух или нескольких компьютерах. А значит, и *обрабатываемые* данные также являются распределенными.

Распределенные архитектуры

.NET Remoting предшествовало множество решений для создания распределенных приложений. Эти ранние технологии, самой

последней инкарнацией которых стал .NET Remoting, стали фундаментом, позволившим извлечь множество уроков о природе распределенных вычислений.

Модульное программирование

Управление сложностью является неотъемлемой частью разработки любых программ, за исключением самых тривиальных. Один из фундаментальных приемов ограничения сложности — разделение кода на отдельные части по функциональному назначению. Такой прием можно применять на разных уровнях путем объединения кода в процедуры, процедур в классы, классов в компоненты, а компонентов в более крупные подсистемы. Распределенные приложения не только получили большие преимущества от использования данной концепции, но во многих случаях и помогают обеспечить ее применение, так как модульность необходима для распределения кода по разным компьютерам. Фактически основные категории распределенных архитектур различаются степенью ответственности, возлагаемой на отдельные модули, и способами взаимодействия последних.

Клиент-серверные архитектуры

«Клиент — сервер» — это наиболее ранняя и наиболее фундаментальная из распределенных архитектур. В широком смысле это просто клиентский процесс, запрашивающий обслуживание у серверного процесса. Обычно клиентский процесс отвечает за уровень представления (или пользовательский интерфейс). На этом уровне выполняется проверка введенных пользователем данных, выполнение вызова сервера и, возможно, применение некоторых бизнес-правил. Сервер выступает в качестве механизма исполнения клиентских запросов путем применения прикладных алгоритмов и взаимодействия с такими ресурсами, как базы данных и файловые системы. Очень часто у одного сервера бывает много клиентов. Хотя эта книга посвящена разработке распределенных приложений, следует отметить, что роли клиента и сервера в общем случае не обязательно исполняются разными компьютерами. Разделение функциональности приложения зарекомендовало себя как хороший способ и для процессов, исполняющихся на одном компьютере.

Многоуровневые архитектуры

Клиент-серверные приложения еще *называют двухуровневыми*, так как клиент *взаимодействует* с сервером непосредственно. Обычно двухуровневые архитектуры легко реализуемы, но имеют проблемы масштабируемости. В прошлом разработчикам часто приходилось использовать многоуровневые архитектуры следующим образом. Приложение исполнялось на одном компьютере. Кто-то *решал*, что оно должно стать распределенным. Например, из-за *намерения* поддерживать более одного клиента, ограничить доступ к ресурсу или *использовать* большие вычислительные возможности одного мощного компьютера. Как правило, первый прототип — *двухуровневый*; он работал отлично и двухуровневое решение принималось. Но по мере *увеличения* числа клиентов скорость работы немного падала. Если *клиентов* становилось еще больше, система полностью останавливалась. Проблему пытались решить путем модернизации аппаратных средств сервера, но этот вариант был *дорогостоящим* и *лишь* отсрочивал решение главной проблемы.

Тогда пришли к выводу, что стоит изменить архитектуру на трех- или многоуровневую. На рис. 1-1 показано, каким образом промежуточный уровень в трехуровневых архитектурах используется для решения различных задач. Одним из вариантов является помещение на промежуточный уровень бизнес-логики. В этом случае промежуточный слой проверяет правильность данных, переданных клиентом, и *обрабатывает* их в соответствии с *бизнес-правилами*. Эта обработка может требовать взаимодействия с уровнем данных или выполнения локальных вычислений. Если все идет нормально, то *промежуточный* уровень обычно передает результаты на уровень данных для хранения или возвращает клиенту результаты. Главное преимущество *такой* архитектуры — более четком распределении ответственности при обработке.

Логическое разделение функций системы дает преимущества даже тогда, когда более одного уровня многоуровневой системы находится на одном и том же компьютере. Разработчики или администраторы получают возможность работать с разными уровнями по отдельности, удалять их совсем или переносить на другие машины в соответствии с новыми требованиями масштабируемости. Вот почему *трехуровневые* (или по-настоящему

многоуровневые) архитектуры оптимальны с точки зрения как масштабируемости, так и гибкости поддержки и развертывания программных систем.

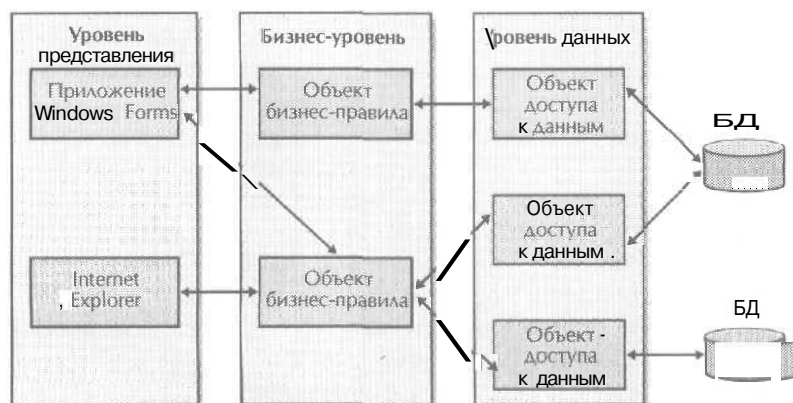


Рис. 1-1. Трехуровневая архитектура

Одноранговые архитектуры

В описанных ранее архитектурах каждый уровень имеет свою явно выраженную роль. Уровни клиента и сервера можно с тем же успехом называть уровнями ведущего/ведомого или поставщика/потребителя. В многоуровневой модели ролями уровней чаще всего считаются представление, бизнес-логика и хранение данных. Однако в некоторых случаях целесообразно использовать модель более тесного взаимодействия, в которой границы между клиентом и сервером размыты. Подобная организация применяется в сценариях рабочих групп, так как основная задача подобных распределенных приложений — совместное использование информации и средств обработки.

Исключительно одноранговая (peer-to-peer) среда состоит из множества отдельных узлов без центрального сервера, как показано на рис. 1-2. В отсутствие общеизвестного (well-known) главного сервера необходим механизм, позволяющий узлам отыскивать друг друга. Обычно для этого используются приемы на основе широковещательных сообщений или некоторые предопределенные параметры конфигурации.

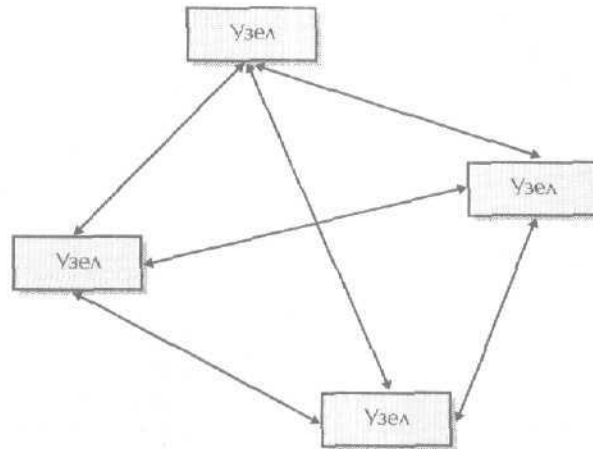


Рис. 1-2. Одноранговая архитектура

Интернет можно считать классической клиент-серверной средой, в которой монолитный Web-сервер обслуживает множество тонких клиентов. Однако Интернет дал жизнь и некоторым квази-одноранговым приложениям, таким, как Napster и Gnutella. Подобные системы обеспечивают непосредственный обмен данными между равноправными компьютерами. Для поиска других узлов используется централизованный сервер, как показано на рис. 1-3. Не будучи одноранговыми архитектурами в чистом виде, эти гибридные модели обычно масштабируются гораздо лучше, чем полностью децентрализованная модель, обеспечивая аналогичные возможности непосредственного взаимодействия.



Рис. 1-3. Одноранговая архитектура с центральным сервером поиска

Преимущества от менее строгого разделения на клиенты и серверы очевидны даже в многоуровневых архитектурах. Очень часто клиентский модуль также является и сервером, а сервер — клиентом. Мы подобно поговорим о размывании понятий клиента и сервера, когда будем обсуждать клиентские обратные вызовы и события в главе 3.

Распределенные технологии

На протяжении многих лет для реализации обсуждавшихся выше распределенных архитектур применялись разнообразные технологии. Хотя набор базовых типов архитектур не изменялся, значительный прогресс наблюдается в технологии разработки распределенных приложений. Достаточно вспомнить инструменты и абстракции, использовавшиеся для разработки распределенных приложений 10 лет назад! Сегодня, мы тратим гораздо больше времени на решение прикладной проблемы, а не на конструирование инфраструктуры пересылки данных с одного компьютера на другой. Так как же далеко мы продвинулись? Об этом — в следующих разделах.

Сокеты

Сокеты — это одно из фундаментальных понятий современных сетевых приложений. Они изолируют программиста от низкоуровневых деталей, связанных с сетью, так как при их использовании сетевое взаимодействие выглядит, как потоковый ввод-вывод. Хотя сокеты предоставляют максимальные возможности управления процессом взаимодействия, для построения на их основе сложных, полномасштабных распределенных приложений требуется слишком много усилий. Применение потокового ввода-вывода для обмена данными означает, что разработчику необходимо создать систему сообщений, а также формировать и выполнять разбор потоков данных. Это слишком утомительно для большинства распределенных приложений общего назначения. Разработчикам требуется более высокий уровень абстракции, который создавал бы иллюзию вызова локальной функции или процедуры.

Удаленные вызовы процедур

В среде DCE (Distributed Computing Environment), предложенной Open Group (бывшая Open Software Foundation), в числе других технологий была определена спецификация выполнения удаленных вызовов процедур (remote procedure calls, RPC). С использованием RPC, а также с возможностью настройки надлежащих конфигурационных параметров и ограничений типов данных, разработчикам удается реализовать удаленное взаимодействие, во многом аналогичное вызовам локальных процедур. В RPC введен ряд фундаментальных понятий, являющихся основой всех современных распределенных технологий, включая DCOM, CORBA, Java RMI и наконец .NET Remoting.

- **Заглушки (stub)** Это фрагменты кода, исполняющиеся на клиенте и на сервере, благодаря которым удаленные вызовы процедур выглядят, как локальные. Например, клиентский код вызывает процедуры заглушки, которые выглядят в точности так же, как процедуры, реализованные на сервере. Заглушка затем отправляет вызов удаленной процедуре.
- **Маршалинг** Так называется процесс передачи параметров из одного контекста в другой. В RPC параметры функции для передачи по сети сериализуются в пакеты.
- **Interface Definition Language (IDL)** Этот язык предоставляет стандартные средства описания синтаксиса вызова и типов данных удаленно вызываемых процедур, не зависящие от языка программирования. Java RMI позволяет обойтись без IDL, так как данная технология распределенных приложений поддерживает только один язык: Java.

RPC — это огромный шаг вперед, существенно упростивший удаленное взаимодействие по сравнению с сокетами. Однако с течением времени объектно-ориентированная разработка стала превалировать над процедурным программированием. Появление распределенных объектных технологий стало неизбежным.

Распределенные объекты — удобная абстракция

Сегодня большинство разработчиков считают объектно-ориентированное проектирование и программирование основами совре-

менной разработки программ. Использование эффективных абстракций необходимо всегда, когда человеку приходится иметь дело с чем-нибудь столь же сложным, как создание больших программных систем. В наши дни объекты и есть та самая фундаментальная и повсеместно используемая абстракция. Но если преимущества использования объектов общепризнанны, то имеет смысл применить их и при распределенной обработке.

Распределенные объектные технологии позволяют обращаться к объектам, исполняющимся на некотором компьютере, из приложений или объектов, исполняющихся на других компьютерах. Так же как RPC делает удаленные вызовы процедур похожими на локальные, так и распределенные объектные технологии делают похожими на локальные удаленные объекты. Примерами распределенных объектных технологий являются DCOM, CORBA, Java RMI и .NET Remoting. Хотя все они реализованы совершенно по-разному и в их основе лежит разная философия, у них необычайно много общих черт.

- Они основаны на объектах, которые обладают идентифицируемостью (identity) и имеют либо могут иметь состояние. Разработчики оперируют удаленными объектами практически так же, как локальными. Это дает простую, унифицированную модель, что упрощает распределенное программирование. Там где это возможно, разработчики выделяют конструкции языка, специфичные для распределенного программирования, и помещают их на конфигурационный уровень.
- Они связаны с некоторой моделью компонентов. Термин компонент имеет множество определений, но здесь мы считаем компонентом отдельный функциональный элемент, который может распространяться как двоичный модуль. Компоненты представляют собой переход к повторному использованию кода в режиме «черного ящика». Благодаря наличию строго определенных открытых протоколов использования, компоненты обычно менее зависимы, кроме того, их можно создавать и перемещать как функциональные единицы. Применение компонентов повышает гибкость при разработке, а также позволяет выделять сервисы общего пользования.
- Они связаны с сервисами масштаба предприятия. Обычно сервисы этого уровня предоставляют такую поддержку, как

транзакции, пулы объектов, управление параллелизмом и поиск объектов. Эти сервисы решают общие проблемы систем высокой производительности и весьма сложны в реализации. По достижении определенной степени клиентской нагрузки на распределенную систему наличие таких сервисов становится критически важным для обеспечения масштабируемости и целостности данных. Так как эти сервисы сложно реализовать, но потребность в них очевидна, то обычно они выводятся из области ответственности прикладного программиста и предоставляются распределенной объектной технологией, сервером приложений или операционной системой.

Преимущества распределенных приложений

Вероятно, вы знаете, как используются распределенные приложения, раз уж решили прочитать книгу по .NET Remoting. Однако для полноты мы все-таки приведем несколько основных причин, по которым стоит предпочесть приложения этого типа.

Отказоустойчивость

Отказоустойчивость — это одно из преимуществ распределенных приложений, одновременно порождающее и массу проблем при их использовании. Несмотря на простоту этого понятия, отказоустойчивым алгоритмам и архитектурам посвящено целое направление исследований. Отказоустойчивость означает, что система должна сохранять работоспособность при сбоях в ней. Одним из краеугольных камней при построении отказоустойчивой системы является избыточность. Например, у автомобиля — две фары. Если одна из них перегорит, то вторая, вероятно, еще некоторое время будет светить, что позволит водителю доехать до места назначения. У нас есть основания надеяться, что водитель заменит перегоревшую фару до того, пока не откажет другая!

Сама природа распределенных приложений позволяет строить отказоустойчивые программные системы, применяя такой способ, как избыточность. Распределение копий функциональных модулей — или, в случае объектно-ориентированного приложения, копий объектов — по разным узлам повышает вероятность

того, что сбой одного узла не повлияет на избыточные объекты, исполняющиеся на других узлах. Функции отказавшего узла может взять на себя один из избыточных объектов, что позволит системе в целом продолжить работу.

Масштабируемость

Масштабируемость — это способность системы выдерживать увеличивающуюся нагрузку лишь с небольшим снижением производительности. Подобно тому, как распределенные приложения позволяют создавать отказоустойчивые системы, они обеспечивают масштабируемость путем распределения разных функциональных частей приложения по разным узлам. Это сокращает объем обработки, выполняемой одним узлом, и позволяет выполнять больший объем работ параллельно — конечно же, он зависит от природы приложения.

Самые современные и мощные модели оборудования всегда значительно дороже, чем модели предпоследней серии. Как уже говорилось при обсуждении трехуровневых архитектур, разделение монолитного приложения на отдельные модули, исполняющиеся на разных машинах, обеспечивает гораздо лучшее соотношение «производительность/стоимость». Таким образом, несколько дорогих и мощных серверов способны обслуживать множество более дешевых и менее мощных клиентских машин. Дорогие процессоры сервера заняты обработкой множества параллельных клиентских запросов, тогда как более дешевые клиентские процессы простаивают в ожидании пользовательского ввода.

Администрирование

Среди задач, связанных с информационными технологиями, найдется немного столь же сложных, как управление аппаратной и программной конфигурацией большой сети ПК. Задача поддержания одинаковых версий программ на множестве географически разнесенных компьютеров требует больших трудозатрат, и здесь легко допустить ошибку. Гораздо проще перенести наиболее часто изменяющиеся программы в централизованное хранилище и обеспечить к нему удаленный доступ.

В такой модели изменения в бизнес-правила можно вносить на сервере, при этом работа клиентов практически не прерывается.

ся. Наиболее очевидный пример *данной* модели — революция тонких клиентов. В архитектурах с тонким клиентом (*как* правило, на основе браузера) большинство, если не все, бизнес-правила *реализуются* централизованно на сервере. В системах на основе браузеров затраты на установку клиентов практически *отсутствуют*, так как даже код уровня представления *находится* на Web-серверах и загружается клиентами при каждом обращении.

Принцип сокращения затрат на администрирование при реализации бизнес-правил на сервере остается верным даже в традиционных архитектурах с толстым клиентом. Если *толстый* клиент отвечает в основном за отображение данных и проверку *вводимых* значений, то приложение можно разбить таким образом, чтобы реализовать на сервере те алгоритмы, изменения в которых наиболее вероятны.

Проблемы распределенных приложений

Разработка распределенных приложений без сомнения относится к сложным задачам. Хотя данный раздел посвящен проблемам, которые следует решать *при* помощи технологии *распределенных приложений*, мы также рассмотрим и некоторые вопросы, решения которых относятся к компетенции разработчика.

Производительность

На производительность распределенного приложения влияет множество факторов. Например, за пределами программной системы это скорость сети, ее загрузка и аппаратные проблемы, связанные с отдельными компьютерами (*проблемы процессоров*, подсистем ввода-вывода, размер и скорость работы оперативной памяти).

На текущем уровне развития технологий распределенных приложений производительность и открытость являются *взаимоисключающими* целями. Если ваше распределенное приложение должно работать очень быстро, то его следует установить за *брандмауэром*, а сервер и клиент должны использовать одну и ту же платформу. Это позволит распределенному приложению *задействовать* эффективный *сетевой* протокол, например TCP, и передавать данные в специализированном двоичном формате. По-

добные форматы значительно эффективнее текстовых, обычно применяемых открытыми стандартами.

В предположении, что конфигурация аппаратных средств распределенной системы (включая сеть) оптимальна, для создания масштабируемых высокопроизводительных приложений совершенно необходимо применять надлежащие приемы программирования. С точки зрения оптимизации следует избегать удаленных вызовов всегда, когда это возможно. Обычно о подобной оптимизации говорят, как о выборе между «болтливостью и тучностью» (*chatty vs. chunky*). Большинство традиционных приемов объектно-ориентированного программирования позволяют элегантно решать часто встречающиеся проблемы программирования. Их следует применять, когда взаимодействующие объекты расположены рядом (внутри одного процесса или, как в случае .NET, внутри одного домена приложения).

Например, если ваш коллега сидит рядом с вами, то для решения возникающих проблем вы можете общаться друг с другом так часто, как это необходимо, например обсудить возникшую идею, изменить ранее принятые решения и вообще свободно переговариваться на протяжении всего рабочего дня. С другой стороны, если ваш партнер живет на другом континенте, то стиль работы потребует радикальных изменений. В этом случае вам придется выполнить максимально возможный объем работ самостоятельно, тщательно проверить результаты и постараться получить максимальную отдачу от редких сеансов связи с партнером. Это не так просто, как общение с коллегой, сидящим за соседним столом, а значит, необходимо осваивать новые приемы работы. В тех случаях, когда расстояние является одним из факторов, для повышения эффективности работы вам придется отправлять и получать большие объемы информации менее часто.

Использование локальных объектов предоставляет вам несколько возможностей.

- **Неограниченное использование свойств** Чтобы задать состояние объекта, вы можете установить значения многих его свойств, используя по одному обращению к объекту для каждого из них. А значит, клиент получает возможность изменять столько свойств (много или мало), сколько требуется в конкретном случае.

- **Неограниченное использование обратных вызовов** Так как временем обмена данными с локальным объектом разрешается пренебречь, то объект может вызвать каждый клиентский объект даже для обновления тривиального фрагмента информации о состоянии. Подобные клиентские объекты вызывают друг друга для обновления или получения требуемой информации, особенно при этом не беспокоясь о потере производительности.

При использовании удаленных объектов вам придется придерживаться следующих правил.

- **Не использовать свойства очень часто** Вместо этого состояние удаленного объекта следует изменять путем передачи их значений в качестве параметров одного или нескольких методов. Фактически некоторые распределенные приложения, обладающие очень большой масштабируемостью, используют методы с длинными списками параметров, которые в случае локальных вызовов выглядят нелепо.
- **Тщательно продумывать использование обратных вызовов** Во избежание дорогостоящих удаленных вызовов большое число обратных вызовов можно объединить в один или несколько вызовов методов с большим числом параметров.

Вывод очевиден: хорошо спроектированное распределенное приложение часто выглядит, как плохо спроектированное объектно-ориентированное приложение. Просто не все приемы работы с локальными объектами применимы для удаленных объектов без ущерба для производительности.

ПРИМЕЧАНИЕ Конечно, никогда не стоит писать неряшливый, расточительный код. В объектно-ориентированном программировании следует разумно ограничивать объем взаимодействия между объектами. Из-за противоречия между «болтливостью и тучностью» для достижения масштабируемости при работе с удаленными объектами зачастую придется отказаться от многих шаблонов, к которым вы, возможно, привыкли, работая с локальными объектами.

Защита

В последнее время ни один из параметров распределенных систем не привлекал к себе большего внимания, чем защита. С увеличением масштаба выхода корпоративных сетей и данных в Интернет внимание к защите будет только расти. Защищенным считается распределенное приложение, в котором выполняются три основных требования, связанных с защитой.

- **Аутентификация** Серверам необходим способ убедиться в том, что клиент на самом деле является тем, кем он себя называет.
- **Криптография** После аутентификации клиента сервер должен быть способен защитить канал передачи данных.
- **Управление доступом** После аутентификации клиента сервер должен быть способен определить, что данный клиент может делать. Например, какие операции он способен выполнять и к каким файлам у него есть доступ для чтения или записи.

Благодаря тесной интеграции с системой защиты Microsoft Windows NT, DCOM обеспечивает превосходную поддержку аутентификации, криптографии и управления доступом. Хотя DCOM предоставляет надежную и всеобъемлющую модель защиты, ее эффективное практическое применение вовсе не так просто. Когда сложность и масштаб решений на основе DCOM выходят на уровень реальных задач, конфигурирование защиты может значительно усложниться. В связи с большой важностью защиты для распределенных приложений способы ее реализации должны быть максимально простыми и безопасными.

Открытость и сетевые форматы данных

Большинство технологий распределенных приложений, включая DCOM, CORBA и Java RMI, имеют собственные сетевые форматы, главной целью при разработке которых обычно считалась производительность. Несколько лет назад открытость казалась далеко не такой важной, как стремление застолбить территорию и по возможности привязать пользователей к одному поставщику или технологии. Предпринимались попытки создания «мостов», которые позволили бы организациям, «завязанным» на опреде-

ленную технологию, взаимодействовать с внешним миром. Однако ни одно из подобных решений не стало бы столь же прозрачным и простым в использовании, если бы поддержка открытого взаимодействия систем не была встроена в технологии распределенных приложений.

Интернет и брандмауэры

Большинство популярных технологий распределенных приложений первоначально разрабатывались для работы в закрытых сетях. Несмотря на то, что общедоступный Интернет существует уже многие годы, до последнего времени он применялся в основном для передачи файлов, электронной почты и просмотра HTML-страниц, предоставляемых Web-серверами. Большинство людей не использовало Интернет как сеть для работы распределенных приложений. С течением времени компании начали защищать свои внутренние сети от любого трафика кроме HTTP, обычно только через порт 80. Вероятно, на данный момент справедливо утверждать, что большинство клиентов работают по протоколу HTTP. Это не означает, что HTTP так уж эффективен. Это просто соглашение, которое возникло в связи с ростом популярности Интернета.

Старые сетевые форматы данных и протоколы обычно требуют открытия доступа через брандмауэры к «небезопасным» портам. Кроме того, подобные форматы и протоколы, как правило, используют не один порт, но несколько портов или диапазоны номеров портов. Среди специалистов по защите бытует мнение, что настройка брандмауэров на пропуск подобного трафика фактически противоречит самому их назначению.

Таким образом, следующим шагом стало связывание закрытых сетей посредством HTTP. Подобная задача решалась и решается до сих пор посредством туннелирования нестандартных сетевых форматов через HTTP, для чего и пишутся специализированные клиенты и серверы либо используются Web-серверы. Все эти варианты не слишком привлекательны. Они представляют собой трудоемкие, подверженные ошибкам «заплатки» для преодоления ограничений сетевых форматов данных.

Подобные обстоятельства сделали использование специализированных форматов в Интернете неудобным. Нравится это кому-

то или нет, общепринятым стандартом для проникновения через брандмауэр стало написание распределенных приложений, работающих по HTTP.

Конфигурирование

Обычно реальные распределенные приложения очень сложны. Вам потребуется управлять множеством факторов только для установления удаленного соединения и еще большим, чтобы приложение по-настоящему заработало. К таким факторам относятся настройка конечных точек (endpoint), правила активизации, параметры защиты и протоколы. Различные распределенные технологии выработали разные приемы конфигурирования, но на данный момент общепринятым считается требование к таким системам обеспечить конфигурирование как программно, так и административно.

Например, DCOM поддерживает программное конфигурирование посредством API COM. К сожалению, для хранения конфигурационной информации DCOM требуется реестр. Так как редактирование реестра связано с ошибками и небезопасно, Microsoft поставляет программу Dcomcnfg, упрощающую редактирование параметров конфигурации DCOM. Но даже при наличии этой программы использование реестра для хранения параметров распределенного приложения затрудняет развертывание, так как настройка требует вмешательства человека или использования сценариев установки.

Независимость от расположения

Все современные технологии распределенных объектов обладают средствами представления удаленных объектов как локальных. Это задача распределенных систем считается весьма важной, так как позволяет перемещать или реплицировать серверные объекты без дорогостоящих изменений в клиентских объектах.

Управление временем жизни объекта

Сети по своей природе ненадежны. Клиентские приложения «падают», пользователи покидают свои рабочие места, а сети иногда теряют работоспособность. Драгоценные ресурсы сервера нельзя удерживать дольше необходимого, иначе пострадает масштабируемость, а аппаратные требования к поддержанию задан-

ной нагрузки станут чрезмерными. Технологии распределенных приложений обязаны предоставить способы управления временем жизни объектов и обнаружения клиентских сбоев, чтобы серверные объекты удавалось удалять из памяти как можно быстрее.

Управление временем жизни объекта в DCOM основано на сочетании опроса (pinging) и подсчета клиентом ссылок. К сожалению, желал переложить ответственность за время жизни серверных объектов на клиента, программисты (по крайней мере, на C++) обычно вставляют в свой код вызовы *AddRef* и *Release*. Как и управление защитой, отслеживание баланса *AddRef* и *Release* становится крайне сложным по мере того, как интерфейсы передаются между объектами, а число объектов и интерфейсов возрастает.

Работа с несколькими интерфейсами разных объектов, передача ссылок на них другим объектам, локальным и удаленным, обязательный вызов *Release* в случае ошибок и предотвращение его вызовов в неподходящее время — распространенные проблемы в сложных COM-приложениях.

Хотя подсчет ссылок позаботится о том, чтобы серверный объект не «умер», необходим также способ выявления клиентов, потерпевших крах перед тем, как освободить удерживаемые ими ссылки на серверные объекты. Для этого DCOM использует опрос клиентов. Хотя он и приводит к периодическому увеличению трафика, механизм опроса в COM существенно оптимизирован таким образом, чтобы вставлять опрос в другие вызовы данного клиентского компьютера.

Решение проблем распределенных приложений в .NET Remoting

В конечном счете все решают деньги. Предприятие заработает больше, если оно способно создавать лучшие решения за меньшее время и если ему не приходится искать суперпрограммистов, которые в состоянии совладать со всеми разрозненными технологиями, необходимыми для построения больших, промышленных решений. Хотя в DCOM имеются все средства для решения сложных проблем распределенных приложений, работать с ним сможет только опытный программист. Не все эти проблемы удастся решить с помощью мастеров и простых приемов разра-

ботки, но многое удастся сделать для упрощения и улучшения средств конфигурации, модели программирования и расширяемости DCOM. В этом состоит самая сильная сторона .NET Remoting. Эта архитектура значительно упрощает, или, лучше того, организует, методы создания и расширения распределенных приложений. Этот уровень организации позволяет разработчикам стать более продуктивными, системам более управляемыми, а компаниям, возможно, чуть более богатыми.

Производительность

Если настроить приложение .NET Remoting на оптимальную производительность, то скорость работы сравняется со скоростью DCOM — она станет очень быстрой. Конечно, у сконфигурированных таким образом приложений .NET Remoting остаются те же ограничения по открытости, что и у сравнимых DCOM-приложений. По счастью, настройка баланса между производительностью и открытостью для приложений .NET Remoting осуществляется просто путем редактирования нескольких записей в конфигурационном файле.

ПРИМЕЧАНИЕ Производительность .NET Remoting отличается от производительности DCOM, когда клиент и сервер располагаются на одном и том же компьютере. Инфраструктура DCOM/COM определяет, что процессы расположены локально, и переключается на использование методов взаимодействия COM (более быстрых). .NET Remoting при взаимодействии между расположенными на одном компьютере доменами приложений по-прежнему продолжает использовать сетевой протокол (например, TCP), заданный конфигурацией.

Расширение и настройка удаленного взаимодействия

Обеспечение легкости использования системы в простых, распространенных сценариях при возможности логического расширения и настройки для более сложных случаев — одна из общих проблем проектирования программ. Решение данной проблемы в .NET Remoting — вероятно, огромный «плюс» этой архитектуры. Она поддерживает множество стандартных сценариев, тре-

буя лишь незначительных объемов работы или настройки. Это позволяет легко получить работоспособное распределенное приложение- Подключаемые модули можно считать дальнейшим развитием компонентного подхода. Компоненты обеспечивают инкапсуляцию объектов в строительные блоки, из которых собирают приложения. Аналогично архитектуры подключаемых компонентов обеспечивают взаимозаменяемость целых подсистем, поддерживающих определенный интерфейс подключения. Так, архитектура .NET Remoting позволят вам подключить нужный тип канала (например, HTTP или TCP) и тип форматировщика (двоичный или SOAP). Таким образом, исходя из критериев производительности и открытости, вы можете выбирать из стандартных, но мощных конфигураций, просто подключая другой модуль.

Конфигурирование

В настоящий момент .NET поддерживает три варианта настройки распределенных приложений.

- С помощью конфигурационных файлов Настройка удаленного взаимодействия легко выполняется посредством файлов в формате XML. Использовать для хранения конфигурационной информации открытого стандарта, такого, как XML, значительно лучше, чем реестра Windows. Например, различные экземпляры удаленных приложений, сосуществующие на одном компьютере, можно настраивать по отдельности средствами конфигурационных файлов, расположенных в их каталогах. Кроме того, конфигурационные файлы позволяют обеспечить так называемую *Хсору-установку* приложений .NET Remoting. *Хсору-установка* — это метод установки приложений путем простого копирования дерева каталогов на целевой компьютер без необходимости написания программ установки, которые должны запускаться пользователем для настройки приложений. После того как приложение установлено, сопровождать его гораздо проще, так как изменение параметров осуществляется простым копированием нового конфигурационного файла в тот каталог, где находится исполняемый файл приложения. Подобная простота конфигурирования недостижима для COM-объектов, использующих реестр.

ПРИМЕЧАНИЕ В зависимости от области действия, выделяется три разновидности конфигурационных файлов:

- **Machine.config** содержит параметры для всех приложений на компьютере;
 - **Web.config** используется для конфигурирования приложений ASP.NET и объектов .NET Remoting, доступ к которым осуществляется посредством Microsoft Internet Information Server (IIS);
 - **конфигурационный файл приложения** содержит параметры конфигурации конкретного дома на приложения.
-

- **Программно** Если изменение настройки распределенного приложения нежелательно, то разработчики могут полностью управлять ими непосредственно из кода программы.
- **С помощью оснастки .NET Framework консоли управления (ММС)** Программная настройка и настройка с помощью конфигурационных файлов считаются средствами для разработчика, тогда как администраторам для конфигурирования распределенных приложений предоставляется специальная оснастка (snap-in) MMC. Хотя этот инструмент обладает меньшими возможностями по сравнению с программной настройкой и файлами конфигурации, он позволяет выполнять все действия, необходимые системным администраторам.

Преимущества CLR и CTS

Значительная часть мощи и простоты использования объектной технологии обусловлена лежащими в ее основе системой типов и моделью объектов. В DCOM действуют те же ограничения, которые присущи системе типов и объектно-ориентированным возможностям COM. COM не поддерживает наследование реализации, за исключением конструкций, подобных агрегированию и включению. Средства обработки ошибок ограничены кодами возврата, так как COM не поддерживает исключения. Система типов COM также является непоследовательной и разрозненной. COM-системы на основе языка C++ используют описания типов

в исходных текстах (IDL), тогда как Visual Basic и языки сценариев — двоичные описания типов (библиотеки типов). Ни IDL, ни библиотеки типов не могут считаться безусловным стандартом, так как каждый из этих вариантов поддерживает конструкции, не поддерживаемые другим вариантом. Наконец, COM не поддерживает целый ряд объектно-ориентированных конструкций, таких, как модификаторы `static`, виртуальные функции и перегружаемые методы,

В противоположность DCOM .NET Remoting является мощной и простой в использовании технологией во многом благодаря общей системе типов (common type system, CTS) и общезыковой исполняющей среде (common language runtime, CLR). Информация о типах — метаданные — стандартизована и доступна. CTS определяет набор базовых типов, которые должны поддерживаться всеми .NET-совместимыми языками. Эти элементы так же надежны при удаленном взаимодействии, как и при взаимодействии между классами внутри одной программы. Более того, метаданные имеют унифицированный формат и хранятся вместе с определяющей их сборкой (assembly), поэтому удаленным объектам не требуются отдельные описания типов, как в DCOM и CORBA,

Благодаря возможности использовать всю мощь объектно-ориентированных средств .NET, .NET Remoting полностью поддерживает наследование реализации, свойства, а также статические, виртуальные и перегружаемые методы. CLR и CTS позволяют разработчикам использовать одну объектную систему как для локальных, так и для удаленных объектов и избегать проектных решений, в которых удаленное расположение объектов ограничивает свободу использования объектно-ориентированного подхода. Наконец, .NET полностью поддерживает распространение исключений между удаленными процессами, что значительно упрощает обработку ошибок для распределенных объектов по сравнению с кодами возврата DCOM.

Открытость

Требования пользователей к технологиям масштаба предприятия, таким, как распределенные объектные технологии, за последние несколько лет изменились. Теперь, чтобы добиться успеха, технология должна поддерживать баланс между производительно-

стью и простотой использования в закрытых системах и возможностью взаимодействия с другими, в некоторых случаях нестандартными и устаревшими системами. В этом заключается одна из основных целей разработки .NET Remoting. Открытость .NET Remoting достигается за счет поддержки открытых стандартов, таких, как HTTP, SOAP, Web Service Description Language (WSDL) и XML. Для преодоления брандмауэров вы можете подключить канал HTTP. (О каналах поговорим в главе 7.) Для взаимодействия с клиентами и серверами, разработанными не на основе .NET, допустимо подключить форматировщик SOAP. (Форматировщики рассматриваются в главе 8.) Удаленные объекты могут быть представлены как Web-сервисы — об этом речь пойдет в главе 3. Среда .NET Remoting и IIS позволяют генерировать описания Web-сервисов, используя которые разработчики клиентов, выполненных не на основе .NET, смогут взаимодействовать с объектами .NET Remoting по Интернету.

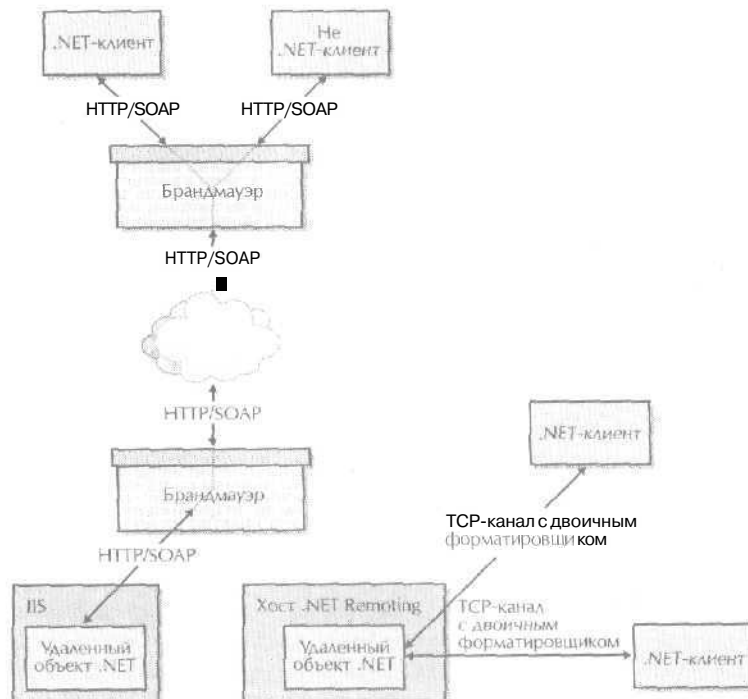


Рис. 1-4. Открытость и производительность

На рис. 1-4 демонстрируются два распространенных сценария использования .NET Remoting: канал HTTP/форматовщик SOAP обеспечивают максимальную открытость, а TCP-канал/двоичный форматовщик — максимальную производительность. Организация обеих конфигураций осуществляется путем простого подключения нужных модулей,

Защита

Модель защиты для систем .NET Remoting существенно изменилась по сравнению со сложной и требующей больших объемов настройки модели DCOM. В первой версии .NET Framework предлагается помещать удаленный сервер внутрь IIS. Основное преимущество IIS заключается в возможности использовать его надежные средства защиты, не внося изменения в коды клиента или сервера. То есть вам удастся защитить распределенную систему, лишь установив IIS в качестве базовой среды и передав параметры регистрации (имя пользователя, пароль и, возможно, домен) с помощью параметров конфигурации клиента. Пример использования IIS мы рассмотрим в главе 3, а пока поговорим о некоторых возможностях защиты, предоставляемых IIS.

Аутентификация

IIS предоставляет поддержку различных механизмов аутентификации, в том числе Windows-интегрированную (NTLM), базовую, Passport и на основе сертификатов. NTLM предоставляет ту же самую надежную систему защиты, которой обладают Windows NT и ее потомки. Это идеальный выбор для приложений в интрасетях, так как NTLM не поддерживает аутентификацию через брандмауэры и прокси-серверы. Однако при работе через Интернет и в других случаях, когда задействованы брандмауэры, вам придется использовать базовую или Passport-аутентификацию.

Защита данных

Следующая после аутентификации пользователя задача — защита секретных данных, передаваемых между удаленными компьютерами. IIS и в этой области предоставляет надежное решение — Secure Sockets Layer (SSL) — промышленный стандарт шифрования данных, который полностью поддерживается IIS с помощью серверных сертификатов. Клиентам .NET Remoting доста-

точно лишь указать в URL сервера в качестве протокола *https* вместо *http*.

Защита без IIS

В первой версии .NET Framework нет готового механизма защиты для распределенных приложений, не использующих IIS. Однако архитектура подключаемых модулей позволяет вам написать собственный модуль защиты. Порядок разработки такого модуля выходит за рамки данной книги. Но, несомненно, вскоре для .NET будет создан целый ряд подобных решений.

Управление временем жизни

Решение .NET Remoting G области управления временем жизни объекта представляет собой прекрасный пример философии «простота для распространенных случаев, логичные расширения — для сложных». Во-первых, подсчет ссылок и опрос клиентов теперь не используются. Их заменили лицензии и спонсоры. Хотя модель DCOM на основе подсчета ссылок проста концептуально, на практике она часто оказывается сложной, подверженной ошибкам и плохо настраиваемой. Лицензии и спонсоры .NET просты в использовании и позволяют при необходимости участвовать в управлении временем жизни объектов как серверам, так и клиентам. Как и другие параметры .NET Remoting, сроки жизни объектов можно настраивать программно или посредством конфигурационных файлов. (Подробно управление временем жизни рассматривается в главе 2.)

Сервисы масштаба предприятия

Сервисы масштаба предприятия подобны лекарствам: если в них нет нужды, нечего их и применять. Но если они нужны, то без них не обойтись. Если вам действительно необходимы сервисы масштаба предприятия, такие, как распределенные транзакции, пулы объектов и средства защиты, то первая версия .NET Framework позволяет использовать проверенные средства COM+.

Если и клиент, и сервер работают в среде CLR, то для доступа к средствам COM+ достаточно создать класс, производный от *System.EnterpriseServices.ServicedComponent*. Так как *ServicedComponent* в конечном счете является производным от *MarshalByRefObject* (базовый тип для всех удаленных объектов .NET), то все

объекты COM+ з .NET автоматически поддерживают удаленный доступ. Любой производный от *ServiceComponent* объект способен использовать сервисы COM+, добавляя атрибуты к своему классу или его методам. Программирование посредством атрибутов является мощной парадигмой, позволяющей разработчику помимо других вещей описывать свой код таким образом, чтобы инструментальные средства и другие программы могли получать информацию о его намерениях и требованиях. Например, для участия в транзакции удаленному объекту .NET достаточно просто установить атрибут удаленного объекта, работающего в среде COM+.

ПРИМЕЧАНИЕ Удаленные объекты .NET могут входить в состав сервисов COM+ и обслуживать как клиентов .NET Remoting, так и традиционных COM-клиентов. Однако, при работе с COM-клиентами объекты .NET в COM+ используют для передачи сообщений COM, а не .NET Remoting. Кроме того, такие .NET-объекты должны иметь строгие имена и регистрироваться как COM-объекты посредством программы Regasm.

Заключение

В этой главе мы рассмотрели задачи, которые должна решать любая технология удаленного доступа: как критически важные (производительность, открытость и защита), так и желательные (простота настройки). .NET Remoting предоставляет следующие средства для решения этих проблем:

- готовая поддержка наиболее популярных сценариев, таких, как «высокая производительность» или «максимальная открытость»;
- возможность использования надежных средств защиты IIS;
- архитектура подключаемых модулей, позволяющая применять специализированные подсистемы;
- логичная модель объектов, позволяющая расширять и настраивать поведение удаленных приложений.

Глава 2

Архитектура .NET Remoting

В первой главе был представлен обзор распределенных приложений, мы рассмотрели различные архитектуры, их преимущества и проблемы. Эта глава посвящена архитектуре .NET Remoting, здесь описаны различные сущности и понятия, используемые при разработке распределенных приложений на базе .NET Remoting. Их глубокое понимание совершенно необходимо для изучения материала остальных глав книги. В главе приводятся несколько кратких фрагментов кода, которые дадут вам представление о программных элементах, определяемых инфраструктурой .NET Remoting, однако обсуждение законченных реализаций отложено до главы 3. Если вы уже знакомы с архитектурой .NET Remoting, то просто просмотрите эту главу и переходите к главе 3.

Границы .NET Remoting

В неуправляемом мире операционная система Microsoft Windows изолирует приложения по отдельным процессам. По сути дела, процесс формирует границу вокруг кода и данных приложения. Все данные и адреса памяти определяются относительно процесса, и код, исполняющийся внутри одного процесса, не имеет доступа к памяти другого процесса, если не используется какой-либо механизм коммуникаций между процессами (interprocess communication, IPC). Значительное преимущество подобной изоляции адресных пространств — повышенная отказоустойчивость, так как сбой одного процесса не влияет на другие. Изоляция адресных пространств также не позволяет коду внутри одного процесса напрямую манипулировать данными другого процесса.

Так как CLR выполняет контроль типов з управляемом коде и проверяет, не обращается ли *управляемый* код к недопустимым адресам, то несколько управляемых приложений могут исполняться *внутри* одного процесса с сохранением тех же преимуществ изоляции приложений, что и в не управляемой модели «одно приложение — один процесс». CLR определяет для приложений .NET два логических подразделения: *домен приложения и контекст*.

Домены приложений

Домен приложения можно рассматривать как *логический* процесс. Мы говорим так потому, что один процесс Win32 способен содержать более одного домена приложения. Код и *объекты*, исполняющиеся в одном домене приложения, не имеют непосредственного доступа к коду и *объектам*, исполняющимся в другом домене приложения. Это обеспечивает защиту, так как *сбой* в одном домене приложения не влияет на другие домены приложения в том же процессе. Разделение между доменами приложения формирует границу .NET Remoting.

Контексты

CLR подразделяет домен приложения на контексты. Контекст гарантирует соблюдение общего набора ограничений и правил использования для всех *объектов* внутри него. Например, контекст синхронизации позволяет исполняться внутри себя в любой момент времени не более чем одному потоку. Это означает, что *объектам* внутри контекста синхронизации не нужен дополнительный код для *управления* многопоточным доступом. В любом домене приложения имеется как минимум один контекст, называемый *контекстом по умолчанию*. CLR создает объект в контексте по умолчанию, если только этот объект явно не требует иного контекста. Контексты подробно рассматриваются в главе 6. На данный момент вам достаточно лишь знать, что наряду с разделением между доменами приложений разделение между контекстами формирует *границу* .NET Remoting.

Пересечение границ

.NET Remoting позволяет объектам, *исполняющимся* внутри разных доменов приложений и *контекстов*, *взаимодействовать* друг с другом через границы .NET Remoting. Граница .NET Remoting

ведет себя, как полупроницаемая мембрана: в некоторых случаях она позволяет экземпляру пройти сквозь нее без изменений; в других — заставляет экземпляр объекта за пределами домена или контекста взаимодействовать с внутренними объектами по строго определенному протоколу.

С точки зрения инфраструктуры .NET Remoting объекты разделяются на две категории: *дистанцируемые* (*remotable*) и *недистанцируемые* (*nonremotable*). Тип является дистанцируемым тогда и только тогда, когда выполняется хотя бы одно из следующих условий:

- экземпляры типа способны пересекать границы .NET Remoting;
- другие объекты могут обращаться к экземплярам данного типа через границы .NET Remoting.

И наоборот, если тип не обладает ни одним из перечисленных выше качеств, то он недистанцируемый.

Недистанцируемые типы

Не всякий тип — дистанцируемый. Экземпляры недистанцируемых типов не могут пересекать границы .NET Remoting ни при каких условиях. При попытке передать экземпляр недистанцируемого типа другому домену приложения или контексту, инфраструктура .NET Remoting сгенерирует исключение. Более того, экземплярам объектов, расположенных вне домена приложения или контекста, содержащего экземпляр объекта недистанцируемого типа, не удастся получить непосредственный доступ к этому экземпляру.

Дистанцируемые типы

Разные категории дистанцируемых типов либо проникают сквозь границы .NET Remoting, либо доступны через эти границы. .NET Remoting определяет три категории дистанцируемых типов: *передаваемые по значению* (*marshal-by-value*), *передаваемые по ссылке* (*marshal-by-reference*) и *контекстно-связанные* (*context-bound*).

Передача по значению

Экземпляры типов, передаваемых по значению, пересекают границы .NET Remoting с помощью процесса, известного под назва-

нием *сериализация*. Сериализация — это процесс представления текущего состояния объекта в виде последовательности бит. После того как объект сериализован, инфраструктура .NET Remoting пересылает полученную последовательность бит через границы .NET Remoting в другой домен приложения или контекст, где инфраструктура выполняет их десериализацию в экземпляр типа, содержащий точную копию первоначального состояния. В .NET тип считается сериализуемым, если он объявлен с атрибутом *Serializable*. В следующем фрагменте кода данный атрибут используется для объявления класса сериализуемым:

```
[Serializable]
class SomeSerializableClass
{
    :
}
```

Кроме того, тип с атрибутом *Serializable* может реализовать интерфейс *ISerializable* для выполнения нестандартной сериализации. Подробно сериализация рассматривается в главе 8. На рис. 2-1 показаны сериализация и десериализация экземпляра объекта при передаче его из одного домена приложения в другой.

Передача по ссылке

В некоторых случаях передача по значению — это то, что надо. Но иногда необходимо, чтобы все вызовы объекта, созданного в некотором домене приложения, обращались именно к экземпляру в данном домене, а не к его копии в другом домене. Например, объекту могут требоваться ресурсы, доступные только объектам, исполняющимся на данном компьютере. В этих случаях используются типы, передаваемые по ссылке, для которых инфраструктура .NET Remoting передает ссылку на экземпляр объекта, а не его сериализованную копию. .NET Framework требует, чтобы типы, передаваемые по ссылке, наследовали от *System.MarshalByRefObject*. Простое наследование от этого класса обеспечивает возможность удаленного доступа к экземплярам производного типа. Ниже показан фрагмент объявления типа, передаваемого по ссылке:

```
class SomeMBRType : MarshalByRefObject
{
    :
}
```

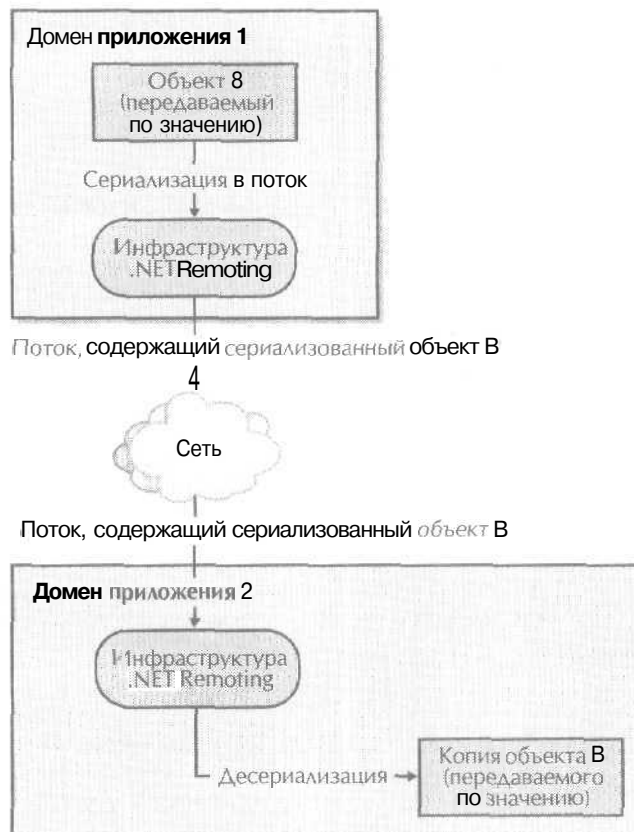


Рис. 2-1. Передача по значению: сериализация экземпляра объекта при передаче его между доменами приложений

На рис. 2-2 показано, что передаваемый по ссылке экземпляр удаленного объекта остается в своем «домашнем» домене приложения и взаимодействует с объектами вне этого домена средствами инфраструктуры .NET Remoting.

Контекстное связывание

Контекстно-связанный тип является частным случаем типа, передаваемого по ссылке. Если тип является производным от *System.ContextBoundObject*, то его экземпляры должны оставаться

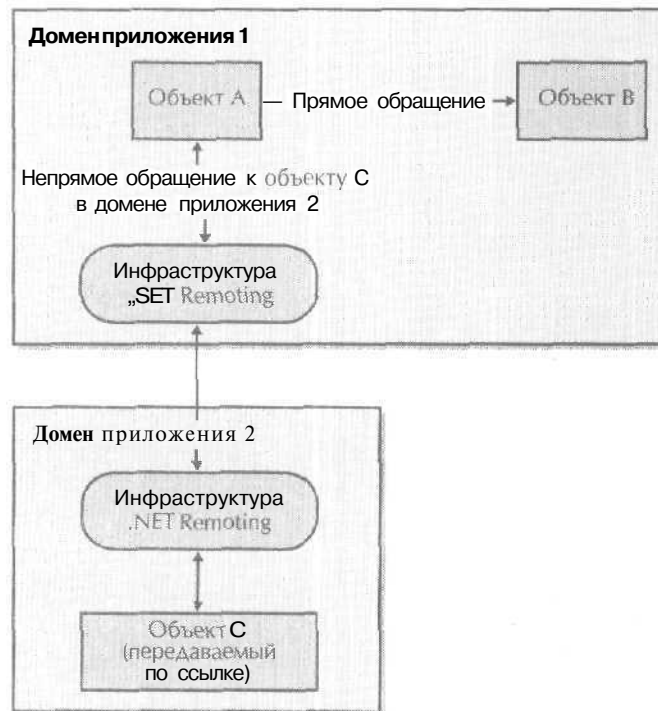


Рис. 2-2. Передача по ссылке: экземпляр объекта остается в домашнем домене приложения

в пределах определенного контекста. Внешние по отношению к данному контексту объекты не имеют непосредственного доступа к типам *ContextBoundObject*, даже если они находятся в одном и тот же домене приложения. Контекстно-связанные типы подробно обсуждаются в главе 6. Ниже приведен пример объявления контекстно-связанного типа:

```
class SomeContextBoundType : ContextBoundObject
{
    ...
}
```

На рис. 2-3 показано взаимодействие между контекстно-связанным объектом и объектами вне его контекста.

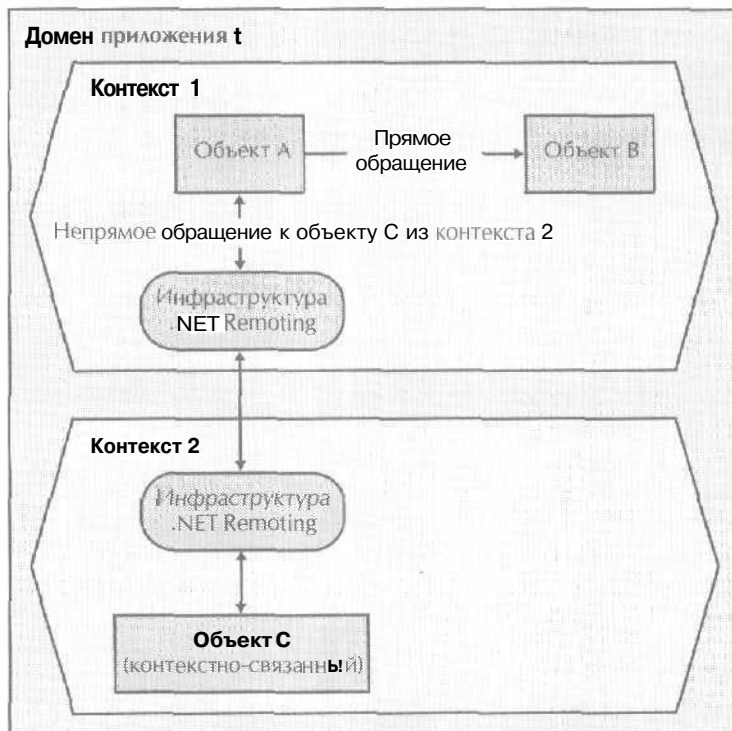


Рис. 2-3. Контекстное связывание: удаленные объекты, связанные с контекстом, взаимодействуют с объектами внеданного контекста средствами инфраструктуры .NET Remoting.

Активизация объекта

Прежде чем работать с экземпляром дистанцируемого типа, его следует создать и инициализировать путем так называемой *активизации* (activation). В .NET Remoting типы, передаваемые по ссылке, поддерживают два вида активизации: серверную и клиентскую. Типы, передаваемые по значению, не требуют особого механизма активизации, так как они копируются в процессе сериализации и фактически активизируются в результате десериализации.

ПРИМЕЧАНИЕ В .NET Remoting режим активизации типа определяется не самим типом, а настройкой инфраструктуры. Например, один и тот же тип может в одном приложении быть сконфигурирован для активизации сервером, а в другом — для активизации клиентом.

Серверная активизация

В терминах инфраструктуры .NET Remoting типы, активизируемые сервером, называются *общеизвестными* (well-known) типами, так как прежде чем активизировать экземпляры объекта серверное приложение публикует такой тип при помощи общеизвестного URI (Uniform Resource Identifier). Серверный процесс, в котором находится дистанцируемый тип, отвечает за конфигурирование данного типа как общеизвестного, публикуя его по определенным общеизвестным конечным точкам или адресам и активизируя экземпляры типа только при необходимости. .NET Remoting поддерживает два режима серверной активизации: *Singleton* и *SingleCall*.

Режим Singleton

В отдельный момент времени может быть активен не более чем один экземпляр типа, сконфигурированного в режиме Singleton. Единственный экземпляр активизируется при первом обращении к нему клиента в отсутствие другого экземпляра. Пока этот экземпляр активен, он обрабатывает все последующие запросы, как от того же клиента, так и от других. Экземпляр типа Singleton способен сохранять состояние в промежутке между вызовами его методов.

Ниже показан пример программного конфигурирования объектного типа в режиме Singleton в серверном приложении, которое реализует этот дистанцируемый объектный тип:

```
RemotingConfiguration.RegisterWellKnownServiceType(  
    typeof( SomeMBRType ),  
    "SomeURI",  
    WellKnownObjectMode.Singleton );
```

В данном примере класс *System.Runtime.Remoting.RemotingConfiguration* используется для регистрации типа *SomeMBRType* как общеизвестного объекта в режиме Singleton. Клиент также

должен сконфигурировать *SomeMBRType* как общеизвестный объект в режиме *Singleton*, это показано ниже:

```
RemotingConfiguration.RegisterWellKnownClientType(
    typeof( SomeMBRType ),
    "http://SomeWellKnownURL/SomeURI" );
```

ПРИМЕЧАНИЕ Существует два механизма конфигурирования инфраструктуры .NET Remoting: программный и с помощью файлов конфигурации. Мы подробно рассмотрим их оба в главе 3.

На рис. 2-4 показана обработка клиентских запросов дистанцируемым объектным типом, сконфигурированным в режиме *Singleton*.

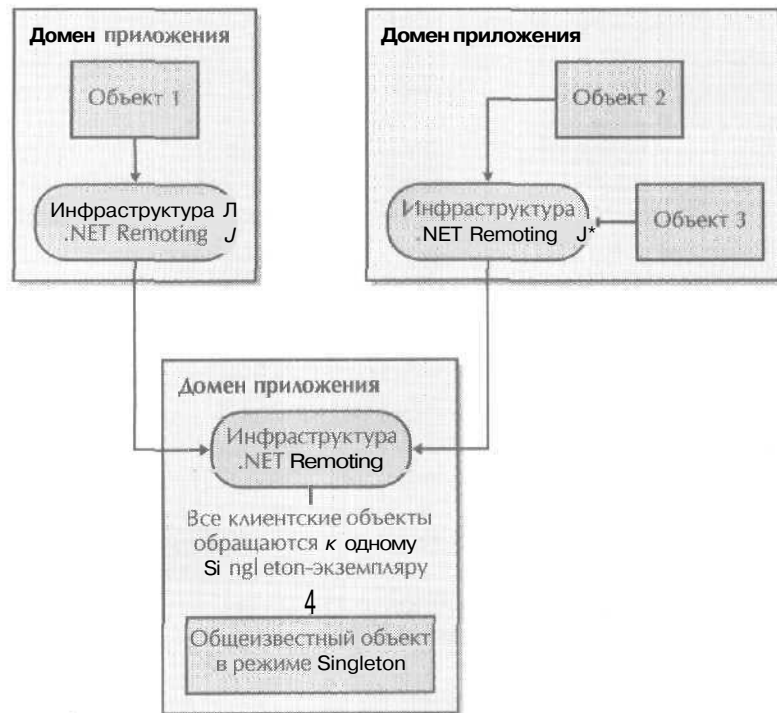


Рис. 2-4. Дистанцируемый объект с серверной активизацией в режиме *Singleton*

ВНИМАНИЕ! Система управления временем жизни в .NET Remoting назначает типам Singleton, активируемым сервером, стандартное время жизни. Из этого следует, что последующие клиентские вызовы могут быть обработаны разными экземплярами типа Singleton. Однако вы вправе изменить стандартное время жизни для своего типа, сконфигурированного в режиме Singleton. Подробнее об этом мы поговорим в главе 3.

Режим SingleCall

Второй режим серверной активизации — SingleCall — обеспечивает поддержку модели программирования без сохранения состояния. Для типа, сконфигурированного как SingleCall, инфраструктура .NET Remoting будет активизировать новый экземпляр при каждом вызове клиентом метода этого типа. После того как вызов метода обработан, инфраструктура .NET Remoting делает экземпляр типа доступным сборщику мусора. Следующий фрагмент кода демонстрирует программный способ конфигурирования дистанцируемого объектного типа в режиме SingleCall в приложении, которое реализует этот дистанцируемый объектный тип:

```
RemotingConfiguration.RegisterWellKnownServiceType(  
    typeof( SomeMBRType ),  
    "SomeURI",  
    WellKnownObjectMode.SingleCall );
```

За исключением последнего параметра этот вызов идентичен использовавшемуся ранее для регистрации *SomeMBRType* в режиме Singleton. Конфигурирование *SomeMBRType* в качестве общеизвестного объекта в режимах SingleCall и Singleton на стороне клиента осуществляется одинаково. Дистанцируемый объект с серверной активизацией в режиме SingleCall показан на рис. 2-5. Инфраструктура .NET Remoting гарантирует, что каждый запрос на вызов метода будет обрабатываться новым экземпляром удаленного объекта.

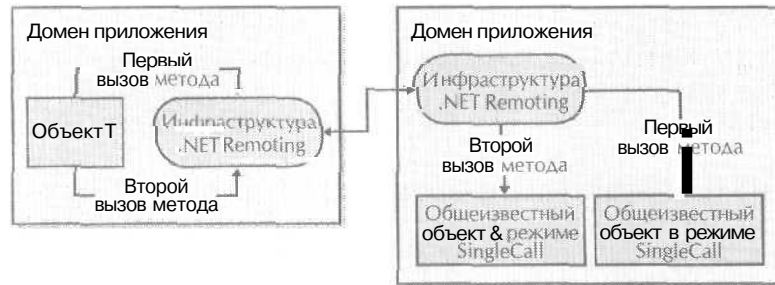


Рис. 2-5. Удаленный объект с серверной активизацией в режиме SingleCall

Клиентская активизация

В некоторых случаях необходимо, чтобы каждый клиент работал с отдельным экземпляром удаленного объекта. Для этой цели .NET Remoting предоставляет клиентскую активизацию. В отличие от общеизвестных типов с серверной активизацией, инфраструктура .NET Remoting назначает URI каждому экземпляру типа с клиентской активизацией.

Экземпляры типов с клиентской активизацией могут оставаться активными в промежутках между вызовами методов и использовать ту же схему управления временем жизни, что и типы Singleton. Однако, вместо того чтобы обрабатывать все клиентские запросы с помощью единственного экземпляра типа, здесь ссылка каждого клиента отображается на отдельный экземпляр дистанцируемого типа,

Ниже приведен пример программной конфигурации дистанцируемого объектного типа как активизируемого клиентом в приложении, которое реализует этот дистанцируемый объектный тип:

```
RemotingConfiguration.RegisterActivatedServiceType(
    typeof( SomeMBRType ));
```

Соответствующий конфигурационный код в клиентском приложении будет таким:

```
RemotingConfiguration.RegisterActivatedClientType(
    typeof( SomeMBRType ),
    "http://SomeURL");
```

Подробнее примеры настройки и использования объектов с клиентской активизацией рассматриваются в главе 3.

ПРИМЕЧАНИЕ Методы класса *RemotingConfiguration*, предназначенные для регистрации удаленных объектов, используют следующие схемы именования:

- методы *RegisterXXXXClientType* регистрируют дистанцируемые объектные типы, которые будут применяться клиентским приложением;
- методы *RegisterXXXXServiceType* регистрируют дистанцируемые объектные типы, публикуемые серверным приложением.
- XXXX может быть либо *WellKnown*, либо *Activated*. *WellKnown* означает, что метод регистрирует тип, активизируемый сервером; *Activated* — тип, активизируемый клиентом. Подробнее класс *RemotingConfiguration* рассматривается в главе 3.

На рис. 2-6 показано, что у каждого клиента имеется ссылка на отдельный экземпляр типа с клиентской активизацией.

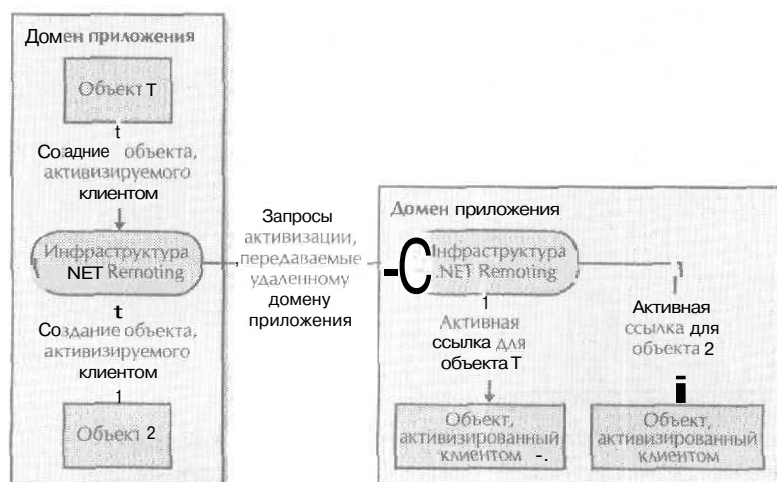


Рис. 2-6. Клиентская активизация

Лицензия объекта

Для управления временем жизни удаленных объектов .NET Remoting использует вариант распределенной сборки мусора на основе *лицензий* (lease). Чтобы понять причину выбора подобной схемы управления временем жизни, рассмотрим случай взаимодействия множества клиентов с удаленным объектом, активизируемым сервером в режиме Singleton. Схемы управления временем жизни, не использующие лицензии, могут применять для определения необходимости ликвидации такого объекта сочетание опроса клиентов и подсчета ссылок. Число ссылок соответствует числу подключенных *клиентов*, тогда как опрос гарантирует, что клиенты все еще функционируют. В подобной схеме *сетевой* трафик, связанный с опросом клиентов, иногда негативно сказывается на *общей* работе распределенного приложения. В противоположность этому система управления временем жизни на основе лицензий использует лицензии, спонсоров и диспетчер лицензий. Отсутствие в данной схеме опроса позволяет повысить общую производительность. Архитектура распределенно-



Рис. 2-7. NET Remoting для реализации распределенной сборки мусора использует систему управления временем жизни на основе лицензий

го управления временем жизни, применяемая .NET Remoting, показана на рис. 2-7.

В каждом домене приложения на рис. 2-7 имеется *диспетчер лицензий* (lease manager). Он содержит ссылки на *объект-лицензию* для каждого объекта Singleton с серверной активизацией и для каждого объекта с клиентской активизацией, работающего в домене приложения этого диспетчера. С каждой лицензией связан нуль или более спонсоров, которые способны продлить лицензию, когда диспетчер лицензий определяет, что ее срок истек.

Лицензия

Лицензия — это объект, инкапсулирующий значения *TimeSpan*, используемые инфраструктурой .NET Remoting для управления временем жизни удаленного объекта. Для реализации этой функциональности инфраструктура .NET Remoting предоставляет интерфейс *ILease*. Когда CLR активизирует экземпляр общеизвестного объекта Singleton или удаленного объекта с клиентской активизацией, она запрашивает у объекта срок лицензии, вызывая его метод *InitializeLifetimeServices*, наследуемый от *System.MarshalByRefObject*. Вы можете переопределить этот метод, чтобы вернуть сроки лицензии, отличные от стандартных. Пример такого переопределения в классе *SomeMBRType* показан ниже:

```
class SomeMBRType : MarshalByRefObject
{
    public override object InitializeLifetimeService()
    {
        // null означает, что срок лицензии не кончается никогда.
        return null;
    }
}
```

Другой пример переопределения метода *InitializeLifetimeService* приведен в главе 3.

Интерфейс *ILease* определяет следующие свойства, используемые инфраструктурой .NET Remoting для управления временем жизни объекта:

- *InitialLeaseTime*;
- *RenewOnCallTime*;
- *SponsorshipTimeout*;
- *CurrentLeaseTime*.

Пример работы со свойствами лицензий мы рассмотрим в главе 3. На данный момент важно понять назначение каждого из этих свойств. Свойство *InitialLeaseTime* — это значение типа *TimeSpan*, определяющее первоначальный срок лицензии. Когда инфраструктура .NET Remoting в первый раз выдает лицензию на объект, значение *CurrentLeaseTime* равно *InitialLeaseTime*. Если *InitialLeaseTime* равно 0, то срок лицензии никогда не заканчивается.

Свойство *RenewOnCallTime* используется инфраструктурой .NET Remoting для продления лицензии при каждом вызове соответствующего удаленного объекта клиентом. При вызове клиентом метода удаленного объекта инфраструктура .NET Remoting определяет, сколько времени осталось до конца лицензии. Если это время меньше, чем *RenewOnCallTime*, инфраструктура .NET Remoting продлевает лицензию на интервал времени, указанный *RenewOnCallTime*.

Свойство *SponsorshipTimeout* задает тайм-аут, определяющий, как долго инфраструктура .NET Remoting будет ждать после уведомления спонсора о том, что срок лицензии истек. Мы поговорим о спонсорах дальше.

Свойство *CurrentLeaseTime* указывает интервал времени до окончания лицензии. Оно доступно только для чтения.

Диспетчер лицензий

В каждом домене приложения имеется диспетчер лицензий, управляющий лицензиями экземпляров дистанцируемых объектов этого домена. При активизации удаленного объекта инфраструктура .NET Remoting передает его лицензию диспетчеру лицензий. Диспетчер содержит объект *System.Hashtable*, отображающий лицензии на экземпляры класса *System.DateTime*, которые соответствуют моментам окончания действия лицензий. Диспетчер периодически просматривает список лицензий, сравнивая время окончания действия лицензий с текущим временем. По умолча-

нию диспетчер лицензий выполняет проверку окончания лицензии каждые 10 секунд, но периодичность опроса можно изменить. В следующем фрагменте кода периодичность опроса, выполняемого диспетчером лицензий, устанавливается равной 5 минутам:

```
LifetimeServices.LeaseManagerPollTime =  
    System.TimeSpan.FromMinutes(5);
```

Каждая лицензия, срок действия которой истек, уведомляется об этом диспетчером лицензий, после чего лицензия пытается продлить себя, опрашивая своих спонсоров. Если у лицензии нет спонсоров или никто из них не продлил ее, то лицензия отменяет себя, выполняя следующие действия;

1. устанавливает свое состояние в *System.Runtime.Remoting.Lifetime.LeaseState.Expired*;
2. уведомляет диспетчер лицензий о том, что ее следует удалить из таблицы лицензий;
3. отключает удаленный объект от инфраструктуры .NET Remoting;
4. отключает объект-лицензию от инфраструктуры .NET Remoting.

После этого у инфраструктуры .NET Remoting больше нет ссылок ни на удаленный объект, ни на его лицензию, так что оба объекта становятся доступны сборщику мусора. Посмотрим, что произойдет, если клиент попытается вызвать метод объекта, лицензия которого закончилась. Результаты зависят от режима активизации удаленного объекта. Если это объект, активизируемый сервером в режиме Singleton, то выполняется активизация нового экземпляра удаленного объекта. Если это объект, активизируемый клиентом, то инфраструктура .NET Remoting сгенерирует исключение из-за попытки обращения к объекту, который ей больше неизвестен.

Спонсоры

Как уже упоминалось, спонсоры — это объекты, которые могут продлить лицензии удаленных объектов. Для создания типа, способного выступать в качестве спонсора, необходимо реализовать интерфейс *ISponsor*. Обратите внимание, что поскольку спонсор получает обратный вызов из домена приложения удаленного объекта, то и тип спонсора должен быть производным от Sys-

tem.MarshalByRefObject. Созданного спонсора разрешается связать с лицензией, вызвав метод */Lease.Register*. У лицензии может быть несколько спонсоров.

Для удобства разработчиков .NET Framework содержит класс *ClientSponsor* в пространстве имен *System.Runtime.Remoting.Lifetime*, который вы вправе использовать в своих программах. Этот класс является производным от *System.MarshalByRefObject* и реализует интерфейс *ISponsor*. Класс *ClientSponsor* позволяет регистрировать ссылки на удаленные объекты, которые следует спонсировать. Когда методу *ClientSponsor.Register* передается ссылка на удаленный объект, этот метод регистрирует экземпляр *ClientSponsor* в качестве спонсора лицензии удаленного объекта и запоминает ссылку на лицензию удаленного объекта во внутренней хэш-таблице. Интервал времени, на который спонсор продлит лицензию, задается свойством *ClientSponsor.RenewalTime*. Ниже показан пример использования класса *ClientSponsor*:

```
// Использование класса ClientSponsor : предполагается,  
// что someMBR ссылается на экземпляр типа,  
// производного от MarshalByRefObject.  
ClientSponsor cp = new ClientSponsor(TimeSpan.FromMinutes(5));  
cp.Register(someMBR);
```

Пересечение границ приложений

Ранее в этой главе говорилось, что границы .NET Remoting определяются доменами приложений и контекстами. Большую часть инфраструктуры .NET Remoting составляют средства, позволяющие объектам взаимодействовать через эти границы. Определив в предыдущих разделах ряд базовых понятий, мы можем теперь рассмотреть общую последовательность событий, происходящих, когда клиент активизирует удаленный объект и вызывает его метод.

Маршалинг ссылок на удаленные объекты посредством *ObjRef*

Ранее уже говорилось, что объекты в одном подразделении .NET Remoting не имеют непосредственного доступа к экземплярам типов, передаваемых по ссылке, в другом подразделении .NET Remoting. Каким же образом .NET Remoting позволяет объектам

взаимодействовать через границы .NET Remoting? В общих словах, клиент использует объект-прокси для взаимодействия с удаленным объектом посредством некоторого механизма коммуникаций между процессами. Мы скоро рассмотрим прокси подробнее, но прежде обсудим, каким образом инфраструктура .NET Remoting выполняет **маршалинг** ссылки на объект, передаваемый по ссылке из одного подразделения в другое.

Есть как минимум три случая, когда может потребоваться передать ссылку на объект через границу .NET Remoting:

- использование передаваемого по ссылке объекта в качестве аргумента функции;
- возврат из функции объекта, передаваемого по ссылке;
- создание объекта с клиентской активизацией, передаваемого по ссылке.

В этих случаях инфраструктура .NET Remoting задействует средства, предоставляемые типом *System.Runtime.Remoting.ObjRef*. Маршалинг — это процесс передачи ссылки на объект из одного подразделения .NET Remoting в другое. Для **маршалинга** ссылки на объект, передаваемый по ссылке из одного подразделения в другое, инфраструктура .NET Remoting выполняет следующие действия:

1. создает экземпляр *ObjRef*, который полностью описывает передаваемый по ссылке объект;
2. сериализует *ObjRef* в виде потока бит;
3. пересылает сериализованный *ObjRef* в целевое подразделение .NET Remoting.

При получении сериализованного *ObjRef* инфраструктура .NET Remoting на принимающей стороне выполняет следующие действия:

1. десериализует *ObjRef* и создает экземпляр класса *ObjRef*;
2. по информации из *ObjRef* создает объект-прокси, посредством которого клиент может обращаться к удаленному объекту.

Для поддержания описанной функциональности тип *ObjRef* является сериализуемым и инкапсулирует важную информацию,

необходимую инфраструктуру .NET Remoting для создания экземпляра прокси в домене приложения клиента.

URI

При активизации экземпляра объекта, передаваемого по ссылке, инфраструктура .NET Remoting назначает ему URI (Uniform Resource Identifier— унифицированный идентификатор ресурса), используемый клиентом при всех последующих обращениях по ссылке на данный объект. Для типов, активизируемых сервером, URI соответствует опубликованной серверным приложением общеизвестной конечной точке. Для типов, активизируемых клиентом, в качестве URI используется GUID, который генерируется инфраструктурой .NET Remoting и связывается с экземпляром удаленного объекта.

Метаданные

Метаданные — это ДНК .NET — базовые строительные блоки общезыковой исполняющей среды. *ObjRef* содержит информацию о типе или метаданные, описывающие передаваемый по ссылке тип. Информация о типе содержит полное квалифицированное имя типа, имя сборки, которая предоставляет его реализацию, а также информацию о версии, региональных стандартах и открытом ключе сборки. Информация о типе также содержит сведения обо всех типах в иерархии наследования и реализуемых интерфейсах, но не саму реализацию типа.

Из набора сведений, содержащихся в информации о типе, передаваемой внутри экземпляра *ObjRef*, можно сделать неочевидный, но важный вывод: так как *ObjRef* несет в себе информацию о сборке, содержащей тип, и об иерархии наследования, но не содержит реализацию типа, принимающий домен приложения должен иметь доступ к сборке, определяющей реализацию типа. Из этого требования вытекает ряд правил развертывания удаленного объекта, которые мы рассмотрим в главе 3.

Информация о канале

Наряду с URI и информацией о типе *ObjRef* содержит для принимающего подразделения .NET Remoting данные о том, каким образом получить доступ к удаленному объекту. Для пересылки через границы .NET Remoting сериализованных экземпляров

ObjRef, а также иной информации используются каналы. Мы скоро поговорим о каналах, пока же вам достаточно знать, что *ObjRef* содержит два набора информации о канале:

- * данные, идентифицирующие контекст, домен приложения и процесс, содержащий удаленный объект;
- сведения, задающие тип транспорта (например, HTTP), адрес и порт для запросов.

Использование прокси для взаимодействия с удаленными объектами

Как говорилось ранее, когда *ObjRef* поступает в клиентское подразделение .NET Remoting, инфраструктура десериализует экземпляр *ObjRef* и создает по нему экземпляр объекта-прокси. Клиент использует прокси для взаимодействия с удаленным объектом, представленным *ObjRef*. Подробно прокси рассматриваются в главе 5, пока же ограничимся концептуальными особенностями прокси, что позволит вам лучше понимать его роль в .NET Remoting.

На рис. 2-8 показан клиентский объект и прокси двух типов: *прозрачный* (transparent) и *реальный* (real). Эти два типа используются инфраструктурой .NET Remoting для обеспечения бесшовного взаимодействия между клиентом и удаленным объектом.

Прозрачный прокси

Прозрачный прокси — это тот, к которому клиент имеет непосредственный доступ. Во время создания прокси по информации из *ObjRef* «на лету» создается экземпляр *TransparentProxy* с интерфейсом, идентичным интерфейсу удаленного объекта. Клиент никоим образом не ощущает того, что взаимодействует с чем-то отличным от подлинного типа удаленного объекта. Внутренне *TransparentProxy* определяется и реализуется инфраструктурой .NET Remoting как тип *System.Runtime.Remoting.Proxies.__TransparentProxy*.

При вызове клиентом метода прозрачного прокси последний преобразует вызов в объект-сообщение, который мы рассмотрим далее. Затем прозрачный прокси передает сообщение прокси второго типа — реальному прокси.

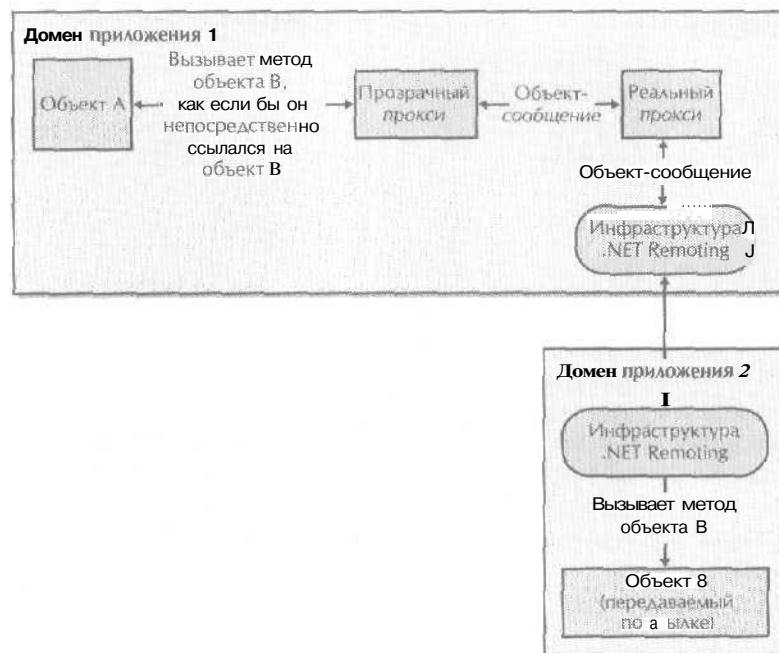


Рис. 2-8, Использование .NET Remoting двух видов прокси: прозрачных и реальных — для обеспечения взаимодействия клиентов с удаленными объектами

Реальный прокси

Реальный прокси — это «рабочая лошадка», принимающая сообщение, созданное прозрачным прокси, и отправляющее его инфраструктуре .NET Remoting для доставки удаленному объекту.

Тип `System.Runtime.Remoting.Proxies.RealProxy` является абстрактным классом, поэтому его экземпляры создавать нельзя. Этот класс — базовый для всех типов прокси, подключаемых к инфраструктуре .NET Remoting. Фактически инфраструктура .NET Remoting определяет класс `RemotingProxy`, представляющий собой расширение `RealProxy`. Класс `RemotingProxy` используется инфраструктурой для выполнения роли `RealProxy`, однако, вы можете создать собственный класс, производный от `RealProxy`, и применять его вместо стандартного прокси, предоставляемого инфраструктурой. О нестандартных прокси речь пойдет в главе 5.

Сообщения — основа удаленного взаимодействия

Давайте немного отвлечемся от .NET Remoting и посмотрим, что происходит при вызове метода локального объекта. С логической точки зрения, при вызове метода объекта вы даете объекту сигнал на выполнение некоторой функции. В некотором смысле вы посылаете объекту сообщение, содержащее значения аргументов метода. Адресом назначения этого сообщения является адрес точки входа метода. На самом низком уровне вызывающий помещает аргументы метода в стек вместе с адресом, по которому следует вернуть управление по окончании работы метода. Затем выполняется вызов метода путем установки указателя текущей команды приложения на точку входа метода. Так как вызывающая процедура и метод используют заранее определенное соглашение о вызове, то методу известно, как получить значения аргументов из стека в надлежащем порядке. Фактически при вызове метода стек играет роль транспортного уровня, передавая аргументы и результаты между вызывающим и вызываемым.

Инкапсуляция информации о вызове метода в объекте-сообщении позволяет абстрагировать и моделировать концепцию «вызов метода как сообщение» объектно-ориентированным способом. Объект-сообщение переносит имя метода, аргументы и другую информацию, связанную с вызовом метода, от вызывающего вызываемому. Подобная схема используется в .NET Remoting для обеспечения взаимодействия удаленных объектов друг с другом. Объекты-сообщения инкапсулируют вызовы всех методов, входные параметры, вызовы конструкторов, возвращаемые значения методов, выходные аргументы, исключения и т.д.

Типы объектов-сообщений .NET Remoting реализуют интерфейс *System.Runtime.Remoting.Messages.IMessage*, и их можно сериализовать. Интерфейс *IMessage* определяет единственное свойство типа *IDictionary* под названием *Properties*. Это словарь, который содержит поименованные свойства и значения, описывающие различные аспекты вызываемого метода. Обычно словарь содержит такую информацию, как URI удаленного объекта, имя вызываемого метода, а также параметры последнего. При пересылке сообщения через границы .NET Remoting инфраструктура сериализует значения, хранящиеся в словаре. Инфраструктурой .NET

Remoting определено несколько типов сообщений, производных от *IMessage*. Подробнее сообщения и их типы рассматриваются в главе 5.

ПРИМЕЧАНИЕ Вы помните, что границы .NET Remoting могут пересекать только экземпляры сериализуемых типов. Имейте в виду, что для пересылки объекта-сообщения через границы .NET Remoting инфраструктура его сериализует. Это означает, что все объекты, помещенные в словарь *Properties* объекта-сообщения, должны быть сериализуемыми, чтобы их удалось передать через границы .NET Remoting вместе с сообщением.

Транспортировка сообщений каналами

.NET Remoting пересылает сериализованные объекты-сообщения через границы .NET Remoting посредством каналов. Объекты-каналы по обе стороны границы предоставляют транспортный механизм, обладающий большими возможностями по расширению и потенциальной возможностью поддержки самых разнообразных протоколов и сетевых форматов данных. Инфраструктура .NET Remoting предоставляет два стандартных типа каналов, которые вы можете использовать в своих приложениях: TCP и HTTP. Если эти каналы не удовлетворяют вашим требованиям, вам предоставлено право создать собственный транспорт и подключить его к инфраструктуре .NET Remoting. Подробнее о расширяемой архитектуре каналов мы поговорим в главе 7.

TCP

Для тех случаев, когда требуется максимальная эффективность, инфраструктура .NET Remoting предоставляет транспорт на основе сокетов, использующий для передачи сериализованного потока сообщений через границы .NET Remoting протокол TCP. Тип канала *TcpChannel* определен в пространстве имен *System.Runtime.Remoting.Channels.Tcp* и реализует интерфейсы *IChannel*, *IChannelReceiver* и *IChannelSender*. Это означает, что тип *TcpChannel* поддерживает как отправку, так и прием данных через границы .NET Remoting. По умолчанию для сериализации объектов-сообщений тип *TcpChannel* использует двоичный фор-

мат сетевых данных. В следующем фрагменте кода домен приложения настраивается на использование канала *TcpChannel*, ожидающего поступления входящих вызовов на порту 2000:

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

```
TcpChannel c = new TcpChannel( 2000 );  
ChannelServices.Register(c);
```

HTTP

Для тех случаев, когда требуется максимальная открытость, инфраструктура .NET Remoting предоставляет транспорт, использующий протокол HTTP для пересылки сериализованного потока сообщений по Интернету и через брандмауэры. Функциональность транспорта HTTP реализована в типе *HttpChannel*, который определен в пространстве имен *System.Runtime.Remoting.Channels.Http*. Как и *TcpChannel*, канал *HttpChannel* способен и принимать, и передавать данные через границы .NET Remoting. По умолчанию для сериализации объектов-сообщений канал *HttpChannel* использует сетевой формат SOAP. В следующем фрагменте кода домен приложения настраивается на использование канала *HttpChannel*, ожидающего поступления входящих вызовов на порту 80:

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

```
HttpChannel c = new HttpChannel( 80 );  
ChannelServices.Register(c);
```

Обработка сообщений цепочками канальных приемников

Большая гибкость архитектуры .NET Remoting достигается за счет четкого разделения обязанностей между объектами. Гибкость архитектуры каналов обеспечивается использованием последовательности объектов — канальных приемников (channel sink), объединенных в цепочку приемников. Каждый приемник играет строго определенную роль в обработке сообщения. В общем он выполняет следующие операции:

1. принимает сообщение и поток от предыдущего приемника в цепочке;
2. в зависимости от содержимого сообщения или потока выполняет некоторые действия;
3. передает сообщение и поток следующему приемнику в цепочке,

Для транспортировки сериализованных сообщений через границы .NET Remoting каналы используют, как минимум, два объекта-приемника. На рис. 2-9 показана архитектура канала на стороне клиента.



Рис. 2-9. Архитектура канала на стороне клиента

На рис. 2-9 клиентский объект вызывает прозрачный прокси, который преобразует вызов метода в объект-сообщение и передает сообщение реальному прокси — фактически это *RemotingProxy*, производный от *RealProxy*. *RemotingProxy* передает объект-сообщение группе специализированных цепочек сообщения внутри контекста (не показано на рис. 2-9), которые мы подробно рассмотрим в главе 6. Объект-сообщение проходит по цепочкам контекстных приемников, пока не достигнет первого приемника из цепочки канальных приемников — это форматирующий приемник, отвечающий за сериализацию объекта-сообщения в поток байт определенного сетевого формата данных. Затем форматирующий приемник передает поток для дальнейшей обработки следующему приемнику в цепочке. Последний приемник в цепочке отвечает за пересылку потока байт по сети с использованием определенного транспортного протокола.

Сериализация сообщений форматирующими приемниками

.NET Remoting предоставляет два типа форматирующих приемников для сериализации сообщений: *BinaryFormatter* и *SoapFormatter*. Выбор одного из них во многом зависит от конфигурации сети, связывающей распределенные объекты. Благодаря наращиваемой архитектуре .NET Remoting вы можете создавать собственные форматирующие приемники. Подобная гибкость позволяет потенциально поддерживать самые разнообразные сетевые форматы данных. Создание специализированных форматов рассматривается в главе 8. Здесь же кратко опишем готовые стандартные средства .NET Remoting.

Для сетевых транспортов, которые позволяют отправлять и принимать двоичные данные (например, TCP/IP), годится тип *BinaryFormatter*, определенный в пространстве имен *System.Runtime.Serialization.Formatters.Binary*. Как следует из названия, данный форматировщик сериализует сообщения в двоичном формате. Это может быть наиболее эффективный и компактный способ представления объекта-сообщения для транспортировки по сети.

Некоторые сетевые транспорты не поддерживают прием и пересылку двоичных данных. Для использования таких транспортов приложение должно преобразовать двоичные данные перед отправкой по сети в текстовое ASCII-представление. В подобных

ситуациях или при необходимости обеспечить максимальную открытость .NET Remoting предоставляет тип *SoapFormatter*, определенный в пространстве имен *System.Runtime.Serialization.Formatters.Soap*. Данный форматировщик использует для сериализации сообщений представление в формате SOAP. Подробнее SOAP рассматривается в главе 4.

Реализация сетевого интерфейса с помощью транспортных приемников

Транспортный приемник обладает информацией о том, как передать данные своему двойнику на противоположной стороне границы .NET Remoting с помощью конкретного транспортного протокола. Например, *HttpChannel* использует транспортный приемник, умеющий отправлять и принимать HTTP-запросы.

Транспортный приемник завершает цепочку канальных приемников на клиентской стороне. Когда такой приемник получает сериализованное сообщение, он сначала передает по сети заголовочную информацию транспортного протокола, а затем и само сообщение, в результате чего сериализованное сообщение проходит границу .NET Remoting и попадает в подразделение .NET Remoting на серверной стороне.

Архитектура канала на серверной стороне показана на рис. 2-10. Как видите, она похожа на архитектуру канала клиента.

Как показано на рис. 2-10, первый приемник в серверной цепочке канальных приемников, к которому попадает сериализованное сообщение, — это транспортный приемник, считывающий из потока заголовки транспортного протокола и сериализованные данные сообщения. После приема данных из сети, транспортный приемник передает полученную информацию следующему приемнику в цепочке. Приемники в цепочке обрабатывают полученное сообщение и передают его дальше, пока оно не достигнет форматизирующего приемника. Тот в свою очередь десериализует сообщение в объект */Message*, передаваемый затем объекту *StackBuilderSink* инфраструктуры .NET Remoting, который и выполняет фактический вызов метода удаленного объекта. По окончании работы метода *StackBuilderSink* упаковывает возвращаемое значение и выходные параметры в объект-сообщение типа *System.Runtime.Remoting.Messaging.ReturnMessage*, который про-

делывает затем путь по цепочке в обратном направлении, пока не достигнет прокси в вызывающем подразделении .NET Remoting.

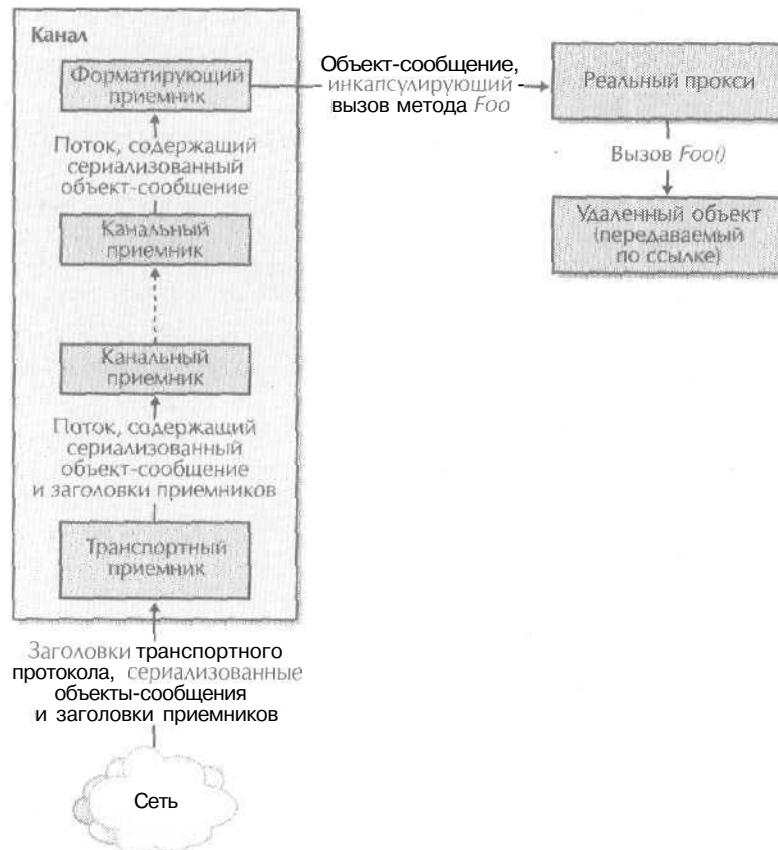


Рис. 2-10. Архитектура канала на стороне сервера

Заключение

В этой главе мы в общих чертах рассмотрели все основные архитектурные компоненты и концепции инфраструктуры .NET Remoting. .NET Remoting включает стандартную поддержку взаимодействия между распределенными объектами посредством транспортов TCP и HTTP с использованием представления потока данных в двоичном или SOAP-формате. Кроме того, .NET Remoting предоставляет большие возможности расширения. Архитектура позволяет подключать специализированные компоненты практически в любой точке обработки удаленного вызова метода. Способам использования возможностей расширения .NET Remoting в приложениях посвящены главы с 5 по 8.

Теперь, когда вы познакомились с архитектурой .NET Remoting, мы можем перейти к рассказу о применении .NET Remoting для создания распределенных приложений.

Глава 3

Распределенные приложения на **ОСНОВЕ** .NET Remoting

В предыдущей главе мы рассмотрели общую архитектуру .NET Remoting и ее основные компоненты. В этой главе речь пойдет о том, как с помощью .NET Remoting создать приложение для распределения заданий между сотрудниками.

На примерах этой главы мы покажем приемы применения различных возможностей технологии .NET Remoting, обсуждавшихся в главе 2, для реализации концепций построения распределенных приложений, которые излагались в главе 1. Во второй части книги на примере этого же приложения демонстрируются возможности расширения .NET Remoting путем создания специализированных прокси, канала и форматировщика.

В процессе разработки приложения мы поясним и продемонстрируем способы решения следующих задач:

- определения удаленных типов;
- создания сервера удаленных объектов;
- передачи событий через границы .NET Remoting;
- публикации и использования удаленных объектов;
- представления удаленного объекта как Web-сервиса;
- минимизации зависимостей от метаданных.

Проектирование приложения

Приложение для распределения задания состоит из двух частей: сервера и клиента. Клиент работает только с одним сервером, тогда как сервер обслуживает произвольное число клиентов. Серверное приложение должно поддерживать для каждого клиента состояние сессии, кроме того, необходимо, чтобы:

- клиенты имели возможность выбирать задания для исполнения;
- клиенты имели возможность сообщить о том, что выполнение задания завершено;
- сервер в реальном времени уведомлял клиентов о поступлении новых заданий;
- сервер отслеживал задания, взятые на обработку и выполненные каждым клиентом.

Основная задача клиента — ввод данных, следовательно, ему потребуется пользовательский интерфейс. Главное окно такого интерфейса должно отображать список всех заданий, зарегистрированных в данный момент на сервере, а также содержать элементы управления для создания, выбора и обновления состояния заданий. От клиента дополнительно требуется, чтобы:

- пользователь имел возможность выбирать задания из списка доступных заданий;
- пользователь имел возможность сообщить о завершении обработки задания;
- выполнялась обработка поступающих в реальном времени уведомлений о новых заданиях;
- обрабатывались поступающие в реальном времени уведомления о взятии заданий на исполнение.

Реализация приложения JobServer

Основная задача приложения JobServer — поддержка распределенного объекта *JobServerImpl*. Обратите внимание, что интерфейсы, структуры и классы в листингах кода данного раздела не

содержат никаких ссылок на .NET Remoting. Одним из важных достоинств .NET Remoting является ее «ненавязчивость».

При разработке листингов примеров этой главы мы сначала создали простое клиент-серверное приложение (з книге это не показано), в котором и клиент, и сервер располагались в одном домене приложения. Данный подход позволяет убедиться в правильной работе приложения до того, как в нем появятся дополнительные источники возможных сбоев. Кроме того, отладку гораздо легче вести в пределах одного домена приложения.

Логика приложения

Приложение JobServer содержит структуру *JobInfo*, интерфейс *IJobServer* и классы *JobEventArgs* и *JobServerImpl*. Серверное приложение, которое мы скоро рассмотрим, публикует экземпляр класса *JobServerImpl* как удаленный объект — остальные типы должны поддерживать этот класс.

Структура *JobInfo*

В этом разделе показана структура *JobInfo*, содержащая уникальный идентификатор задания, его описание, пользователя-исполнителя и статус:

```
public struct JobInfo
{
    public JobInfo(int nID, string sDescription,
                  string sAssignedUser, string sStatus)
    {
        m_nID = nID;
        m_sDescription = sDescription;
        m_sAssignedUser = sAssignedUser;
        m_sStatus = sStatus;
    }

    public int m_nID;
    public string m_sDescription;
    public string m_sAssignedUser;
    public string m_sStatus;
}
```

Интерфейс *IJobServer*

Здесь показан интерфейс *IJobServer*, определяющий взаимодействие клиентов с экземпляром *JobServerImpl*:

```
public interface IJobServer
{
    event JobEventHandler JobEvent;
    void CreateJob( string sDescription);
    void UpdateJobState(int nJobID,
                       string sUser,
                       string sStatus);
    ArrayList GetJobs ();
}
```

Как следует из названия, метод *IJobServer.CreateJob* позволяет клиентам создавать новые задания, указывая их описания, а метод *IJobServer.UpdateJobState* — установить клиенту текущий статус задания: «Assigned» (выполняется) либо «Completed» (выполнено). Метод *IJobServer.GetJobs* возвращает список всех текущих заданий, представленных экземплярами *JobInfo*.

Реализация *IJobServer* должна генерировать событие *JobEvent* всякий раз, когда клиент создает новое задание или обновляет статус существующего задания. Реализация интерфейса *IJobServer* обсуждается далее в разделе «Класс *JobServerImpl*».

Класс *JobEventArgs*

Класс *JobEventArgs* служит для передачи обработчикам события *JobEvent* информации о новых или изменивших свое состояние заданиях. Он определен следующим образом:

```
public class JobEventArgs : System.EventArgs
{
    public enum ReasonCode { NEW, CHANGE };
    private ReasonCode m_Reason;
    private JobInfo m_JobInfo;

    public JobEventArgs( JobInfo NewJob, ReasonCode Reason )
    {
        m_JobInfo = NewJob;
        m_Reason = Reason;
    }

    public JobInfo Job
    {
        get
        { return m_JobInfo; }

        set
```

```

        { m_JobInfo = value; }
    }

    public ReasonCode Reason
    {
        get
        { return m_Reason; }
    }
}
!

```

Так как в нашей реализации интерфейса *IJobServer* событие *JobEvent* генерируется как при создании нового, так и при обновлении состояния старого задания, то для индикации типа события, произошедшего с заданием, которое описано полем *m_JobInfo*, служит поле *m_Reason*.

Обратите внимание на то, что класс *JobEventArgs* является производным от *System.EventArgs*. Это не обязательно, но рекомендуется в тех случаях, когда отправителю сообщения необходимо передавать получателю информацию, специфичную для данного события. В последующих разделах данной главы класс *JobEventArgs* рассмотрен подробнее.

Класс *JobServerImpl*

Класс *JobServerImpl* — главный для приложения *JobServer*, которое служит сервером для удаленного объекта этого класса. Реализация класса *JobServerImpl* показана ниже:

```

public class JobServerImpl : IJobServer
{
    private int      m_nNextJobNumber;
    private ArrayList m_JobArray;

    public JobServerImpl()
    {
        m_nNextJobNumber = 0;
        m_JobArray        = new ArrayList();
    }

    // Вспомогательная функция генерации IJobServer.JobEvent.
    private void NotifyClients(JobEventArgs args)
    {
        // Определяется позже...
    }
}

```

```

// Реализация интерфейса IJobServer.
public event JobEventHandler JobEvent;

public ArrayList GetJobs()
{
    // Определяется позже...
}

public void CreateJob( string sDescription )
{
    // Определяется позже...
}

public void UpdateJobState( int    nJobID,
                           string sUser,
                           string sStatus )
{
    // Определяется позже...
}
}

```

Все экземпляры *JobInfo* хранятся в массиве *JobServerImpl.m_JobArray*. Член *JobServerImpl.m_nNextJobNumber* служит для генерации уникального идентификатора нового задания.

Вот как реализован метод *GetJobs*:

```

public ArrayList GetJobs()
{
    return m_JobArray;
}

```

Методы *CreateJob* и *UpdateJobState* для генерации события *JobEvent* при создании клиентом нового или обновлении им состояния старого задания используют вспомогательный метод *NotifyClients*. Далее представлена его реализация:

```

private void NotifyClients(JobEventArgs args)
{
    //
    // Все обработчики событий вызываются вручную для
    // идентификации отключившихся клиентов.
    System.Delegate[] invkList = JobEvent.GetInvocationList();

    IEnumerator ie = invkList.GetEnumerator();
    while (ie.MoveNext())
    {

```

```

        JobEventHandler handler = (JobEventHandler)ie.Current;
        try
        {
            IAsyncResult ar =
                handler.BeginInvoke( this, args, null, null);
        }
        catch(System.Exception e)
        {
            JobEvent -= handler;
        }
    }
}

```

Обратите внимание, что вместо использования простой формы генерации события метод *NotifyClients* просматривает список обработчиков и вызывает каждый из них вручную. Это необходимо для обработки ситуации, когда клиент стал недоступным после подписки на событие. Если клиент стал недоступен, то в список обработчиков *JobEvent* добавляется делегат, ссылающийся на удаленный объект, с которым нет связи. При вызове такого делегата инфраструктура генерирует исключение из-за невозможности обращения к удаленному объекту. В результате последующие делегаты в списке не вызываются и соответствующие клиенты не получают уведомлений. Во избежание такой проблемы следует вызывать каждый делегат вручную, удаляя при этом те, при вызове которых произошло исключение. В коде реального приложения необходимо отслеживать типы возникающих ошибок и обрабатывать их соответствующим образом.

Ниже показана реализация метода *CreateJob*, позволяющего создавать новое задание, в классе *JobServerImpl*:

```

public void CreateJob( string sDescription )
{
    // Создание нового экземпляра JobInfo.
    JobInfo oJobInfo = new JobInfo( m_nNextJobNumber,
                                    Description,
                                    "" );

    // Увеличить номер следующего задания.
    m_nNextJobNumber++;

    // Добавить экземпляр JobInfo в массив JobArray.
    m_JobArray.Add( oJobInfo );
}

```

```

        // Уведомить клиентов о новом задании,
        NotifyClients( new JobEventArgs( oJobInfo,
                                           JobEventArgs.ReasonCode.NEW ));
    }

```

Метод *UpdateJobState*, посредством которого клиенты устанавливают для задания имя пользователя и статус, реализован так:

```

public void UpdateJobState( int nJobID,
                           string sUser,
                           string sStatus )
{
    // Извлечь указанное задание из массива.
    JobInfo oJobInfo = C JobInfo ) m_JobArray[ nJobID ];

    // Обновить имя пользователя и статус.
    oJobInfo.m_sAssignedUser = sUser;
    oJobInfo.m_sStatus = sStatus;

    // Обновить элемента массива, так как JobInfo - размерный тип.
    m_JobArray[ nJobID ] = oJobInfo;

    // Уведомить клиентов об изменении статуса задания.
    NotifyClients( new JobEventArgs( oJobInfo,
                                       JobEventArgs.ReasonCode.CHANGE));
}

```

Добавление средств .NET Remoting

До настоящего момента мы реализовали ряд типов безотносительно .NET Remoting. Действия, необходимые для добавления в приложение jobServer поддержки .NET Remoting, перечислены ниже:

1. реализация дистанцируемого типа;
2. выбор серверного домена приложения;
3. выбор модели активизации;
4. выбор канала и порта;
5. выбор способа получения метаданных сервера клиентами;
6. настройка параметров .NET Remoting для сервера.

Рассмотрим их подробнее,

Реализация дистанцируемого типа

Для реализации поддержки .NET Remoting нам потребуется внести ряд расширений в класс *JobServerImpl* и связанные с ним типы. Начнем со структуры *JobInfo*. Класс *JobServerImpl* передает экземпляры структуры *JobInfo* клиенту, следовательно, структура должна быть сериализуемой. В .NET Framework для обеспечения сериализуемости объекта достаточно использовать атрибут *[serializable]*.

Кроме того, необходимо сделать *System.MarshalByRefObject* базовым классом для нашего удаленного объекта — *JobServerImpl*. Как вы, вероятно, помните из главы 2, экземпляр типа, производного от *System.MarshalByRefObject*, взаимодействует с объектами в удаленных доменах приложений посредством прокси. Открытые методы *System.MarshalByRefObject* перечислены в табл. 3-1.

Таблица 3-1. Открытые методы *System.MarshalByRefObject*

Открытый метод	Описание
<i>CreateObjRef</i>	Виртуальный метод, возвращающий экземпляр <i>System.Runtime.Remoting.ObjRef</i> , который используется для маришалинга ссылки на объект через границы .NET Remoting. В своих производных типах вы можете переопределить эту функцию, чтобы возвращать специализированную версию <i>ObjRef</i>
<i>GetLifetimeService</i>	Данный метод применяется для получения интерфейса <i>ILease</i> лицензии, связанной с данным экземпляром <i>MarshalByRefObject</i>
<i>InitializeLifetimeService</i>	Инфраструктура .NET Remoting вызывает этот виртуальный метод во время активизации для получения объекта типа <i>ILease</i> . Как уже говорилось в главе 2 и показано далее в этой главе, в разделе «Спонсор лицензии», в производных классах этот метод можно переопределить для управления начальными параметрами времени жизни объекта

Мы переопределили *InitializeLifetimeService* следующим образом:

```
public override object InitializeLifetimeService()
{
    return null;
}
```

Возвращаемое значение *null* сообщает .NET Remoting о том, что объект должен *существовать* неопределенно долго. Далее в *этой* главе, в разделе «Настройка клиента для работы с объектами с клиентской активизацией», мы покажем еще один вариант реализации данного метода.

Выбор серверного домена приложения

Теперь нужно принять решение, каким образом экземпляры *JobServerImpl* будут *предоставляться* клиентскому приложению. Выбор зависит от ответов на следующие вопросы:

- будет ли приложение работать постоянно;
- будут ли сервер и клиент размещены в интрасети;
- есть ли у серверного приложения *пользовательский* интерфейс;
- будет ли удаленный тип реализован как Web-сервис;
- как часто будет запускаться приложение;
- будет ли приложение доступно клиентам только через брандмауэр?

К счастью, нам доступно множество средств, включая следующие:

- консольное приложение;
- Windows Forms;
- служба Windows;
- Internet Information Services (IIS)/ASP.NET;
- COM+.

Вероятно, для проведения быстрых тестов и разработки прототипов предпочтительнее использовать консольное приложение из-за его простоты. Консольные приложения — это полноценные серверы .NET Remoting, но их применение в реальных задачах ограничено одним большим недостатком: их нужно запускать вручную. Однако при тестировании и отладке возможность запускать и останавливать сервер, а также выводить информации в консольном окне — это как раз то, что нужно.

В приложениях Windows Forms те же преимущества и ограничения, что и у консольных приложений, однако они имеют графиче-

ческий пользовательский интерфейс. Обычно приложения Windows Forms применяются не как серверные приложения, но как клиенты. Возможность работы приложения с графическим пользовательским интерфейсом в качестве сервера .NET Remoting — наглядный пример стирания границ между клиентом и сервером. Конечно, все серверы .NET Remoting способны функционировать либо как сервер и клиент одновременно, либо только как сервер,

В реальных системах сервер должен иметь возможность автоматически зарегистрировать свои каналы и ожидать вызовы от клиентов. Вероятно, вам знакома модель серверов DCOM, в которой Service Control Manager (SCM) автоматически запускает серверы при обращении к ним первого клиента. В отличие от этого, серверы .NET Remoting следует запускать до того, как первый клиент попытается соединиться с ними. Такую возможность предоставляют остальные обсуждающиеся здесь варианты серверных приложений.

Службы Windows — превосходный способ реализации постоянно исполняющегося сервера, так как они способны запускаться автоматически и не требуют входа пользователя в систему на компьютере-сервере. Недостатки служб Windows состоят в том, что их сложнее разрабатывать, а для их развертывания требуется утилита установки.

Самый простой в написании сервер .NET Remoting — это тот, который уже написан, — IIS. Так как IIS — это служба, он может служить постоянно исполняющимся сервером удаленных объектов. Кроме того, IIS предоставляет ряд уникальных средств, таких, как простая настройки защиты удаленных приложений и возможность изменения конфигурационного файла сервера без его перезапуска. Самый крупный недостаток применения MS в качестве сервера .NET Remoting — то, что IIS поддерживает лишь канал *HttpChannel* (см. глазу 2), хотя его производительность можно повысить, используя двоичный форматировщик.

Наконец, если вам требуется доступ к службам масштаба предприятия, то в качестве серверов удаленных объектов стоит использовать сервисы COM+. Фактически объекты .NET, применяющие сервисы COM+, автоматически являются дистанцируе-

мыми, так как обязаны наследовать от класса *System.EnterpriseServices.ServicedComponent*, который в конечном счете наследует от *System.MarshalByRefObject*. Лакмусовой бумажкой, позволяющей определить, когда следует в качестве серверной среды применять COM+, можно считать необходимость доступа к сервисам COM+, таким, как распределенные транзакции и пулы объектов. Если эти сервисы вам не нужны, то вероятно, потеря производительности, связанная с работой в среде COM+, необоснованна.

Для иллюстрации концепции .NET Remoting мы решили использовать самый простой вариант создания, исполнения и отладки сервера для JobServer: консольное приложение. Для создания приложения JobClient (об этом — в следующем разделе этой главы) мы применили Windows Forms. При обсуждении Web-сервисов далее в этой главе мы также рассмотрим использование IIS в качестве серверной среды.

Исходный текст точки входа приложения JobServer таков:

```
namespace JobServer
{
    class JobServer
    {
        /// <summary>
        /// Главная точка входа приложения.
        /// </summary>
        static void Main(string[] args)
        {
            // Вставить код .NET Remoting.

            // Работать до получения команды на завершение,
            System.Console.WriteLine( "Press Enter to exit" );

            // Ждать, пока пользователь не нажмет клавишу Enter.
            System.Console.ReadLine();
        }
    }
}
```

Комментарий «Вставить код .NET Remoting» мы далее заменим соответствующим кодом. В конце раздела приведен окончательный вариант метода *Main*. Вы удивитесь, насколько простым он окажется.

Выбор модели активизации

В главе 2 мы рассмотрели два типа активизации объектов, передаваемых по ссылке: серверную и клиентскую. Нам нужно, чтобы клиент создавал удаленный объект один раз и чтобы этот объект продолжал работать независимо от того, каким клиентом он создан. Как вы помните из главы 2, временем жизни объектов с клиентской активизацией управляет клиент. Очевидно, что клиентская активизация нам не подойдет. Следовательно, мы выбираем серверную активизацию. Необходимо принять еще одно решение: выбрать режим активизации — Singleton или SingleCall. Из главы 2 вы знаете, что в режиме SingleCall каждый запрос обрабатывается отдельным экземпляром объекта. Нам этот режим не подходит, так как здесь все экземпляры объектов используют собственные данные. С другой стороны, режим Singleton — это то, что нам нужно. В этом режиме приложение при первом обращении клиента создает единственный экземпляр *JobServerImpl*. Для настройки параметров типа, связанных с .NET Remoting, инфраструктура .NET Remoting предоставляет класс *RemotingConfiguration*. Открытые члены этого класса перечислены в табл. 3-2.

Таблица 3-2. Открытые члены класса *System.Runtime.RemotingConfiguration*

Член	Тип члена	Описание
<i>ApplicationId</i>	Свойство только для чтения	Строка, содержащая GUID приложения
<i>ApplicationName</i>	Свойство для чтения и записи	Строка, задающая название приложения. Это название является частью URI для удаленных объектов
<i>Configure</i>	Метод	Данный метод используется для настройки инфраструктуры .NET Remoting с помощью информации из конфигурационного файла

см. след. стр.

Таблица 3-2. (продолжение)

Член	Тип члена	Описание
<i>GetRegisteredActivatedClientTypes</i>	Метод	Возвращает массив зарегистрированных в настоящий момент типов с клиентской активизацией, используемых данным доменом приложения
<i>GetRegisteredActivatedServiceTypes</i>	Метод	Возвращает массив зарегистрированных в настоящий момент типов с серверной активизацией, опубликованных этим доменом приложения
<i>GetRegisteredWellKnownClientTypes</i>	Метод	Возвращает массив зарегистрированных в настоящий момент типов с серверной активизацией, используемых данным доменом приложения
<i>GetRegisteredWellKnownServiceTypes</i>	Метод	Возвращает массив зарегистрированных в настоящий момент общеизвестных типов, опубликованных этим доменом приложения
<i>IsActivationAllowed</i>	Метод	Позволяет определить, поддерживает ли текущий домен приложения клиентскую активизацию для заданного типа
<i>IsRemotelyActivatedClientType</i>	Метод	Возвращает экземпляр <i>ActivatedClientTypeEntry</i> , если текущий домен приложения зарегистрировал заданный тип для клиентской активизации

см. след. стр.

Таблица 3-2. (окончание)

Член	Тип члена	Описание
<i>IsWellKnownClientType</i>	Метод	Возвращает экземпляр <i>WellKnownClientTypeEntry</i> , если текущий домен приложения зарегистрировал заданный тип для серверной активизации
<i>ProcessId</i>	Свойство только для чтения	Строка в формате GUID, уникально идентифицирующая текущий процесс
<i>RegisterActivatedClientType</i>	Метод	Регистрирует тип с клиентской активизацией, клиентом которого является данный домен приложения
<i>RegisterActivatedService Type</i>	Метод	Регистрирует тип с клиентской активизацией, публикуемый данным доменом приложения
<i>RegisterWellKnown ClientType</i>	Метод	Регистрирует тип с серверной активизацией, клиентом которого является данный домен приложения
<i>RegisterWellKnownServiceType</i>	Метод	Регистрирует тип с серверной активизацией, публикуемый данным доменом приложения

Далее показана настройка *JobServerImpl* как типа с серверной активизацией, публикуемого с URI «*JobURL*» в режиме активизации Singleton:

```
RemotingConfiguration.RegisterWellKnownServiceType(
    typeof( JobServerImpl ),
```

```
"JobURI",
WellKnownObjectMode.Singleton );
```

Выбор канала и порта

Как говорилось в главе 2, .NET Remoting предоставляет два стандартных канала — *HttpChannel* и *TcpChannel*. Выбор канального транспорта обычно несложен, так как во многих случаях тип транспорта не имеет значения. При выборе типа канала необходимо учесть несколько факторов:

- необходимо ли пересылать данные через брандмауэр;
- вызывает ли пересылка данных открытым текстом проблемы защиты;
- нужны ли средства защиты, поддерживаемые IIS.

Поток сообщений между клиентом и сервером рассматривается в главе 4. С учетом этого мы выбираем *HttpChannel* (использующий по умолчанию *SOAPFormatter*), чтобы формат сообщений стал читабельным. Следующий фрагмент кода показывает, как легко настроить канал:

```
HttpChannel oJobChannel = new HttpChannel( 4000 );
ChannelServices.RegisterChannel( oJobChannel );
```

Сначала мы создаем экземпляр класса *HttpChannel*, передавая его конструктору значение 4000. Таким образом, 4000 — это номер порта, на котором сервер будет ожидать вызовы от клиентов. Создания объекта-канала недостаточно для того, чтобы канал начал принимать входящие сообщения. Для этого канал нужно зарегистрировать с использованием статического метода *ChannelServices.RegisterChannel*. Часть открытых членов класса *ChannelServices* показана в табл. 3-3; другие открытые члены этого класса, предоставляющие доступ к расширенным возможностям, рассматриваются в главе 7.

Таблица 3-3. Открытые члены *System.Runtime.Remoting.Channels.ChannelServices*

Член	Тип члена	Описание
<i>GetChannel</i>	Метод	Возвращает объект типа <i>IChannel</i> для зарегистрированного канала, задаваемого по имени

см. след. стр.

Таблица 3-3. (окончание)

Член	Тип члена	Описание
<i>GetUrlsForObject</i>	Метод	Возвращает массив всех URI, по которым доступен тип
<i>RegisterChannel</i>	Метод	Регистрирует канал для использования доменом приложения
<i>RegisteredChannels</i>	Свойство только для чтения	Возвращает массив интерфейсов <i>IChannel</i> для всех каналов, зарегистрированных доменом приложения
<i>UnregisterChannel</i>	Метод	Отменяет регистрацию канала в домене приложения

Выбор способа получения клиентами метаданных сервера

У клиента, использующего удаленный тип, должна быть возможность получения метаданных, описывающих этот тип. Метаданные требуются, в основном, для достижения двух целей:

- обеспечения компиляции клиентского кода, ссылающегося на тип удаленного объекта;
- обеспечения .NET Framework возможности сгенерировать класс прокси, используемый клиентом для взаимодействия с удаленным объектом.

Они достижимы несколькими способами, простейшим из которых считается использование сборки, содержащей реализацию удаленного объекта. Однако, с точки зрения разработчика удаленного объекта, предоставлять клиенту доступ к реализации не всегда желательно. В этом случае используют один из вариантов представления метаданных, которые мы рассмотрим далее в разделе «Проблемы зависимости от метаданных». Пока же клиент получает доступ к сборке *JobServerLib*, содержащей реализацию типа *JobServerImpl*.

Настройка параметров .NET Remoting для сервера

К настоящему моменту мы программно настроили приложение *JobServer* для удаленного взаимодействия. Далее показано тело функции *Main* приложения *JobServer*:

```
{
// Регистрация канала.
```

```
HttpChannel oJobChannel = new HttpChannel( 4000 );

ChannelServices.RegisterChannel( oJobChannel );

// Регистрация общеизвестного типа.
RemotingConfiguration.RegisterWellKnownServiceType(
    typeof( JobServerImpl ),
    "JobURI",
    WellKnownObjectMode.Singleton );
}
```

Казалось бы, неплохо, но что если мы захотим поменять номер порта? Тогда придется перекомпилировать сервер. Возможно, вы подумали: «Номер порта можно просто передать в командной строке». Этот прием работает, но не решает таких проблем, как добавление новых каналов. Необходим способ переноса этих деталей настройки из кода программы в конфигурационный файл. Использование конфигурационного файла позволяет администратору настраивать поведение приложения без необходимости перекомпилировать его. Но самое главное, что вы можете заменить весь предыдущий код одной строкой! Взгляните на новую функцию *Main*:

```
{
RemotingConfiguration.Configure( @"..\..\JobServer.exe.config" );
}
```

Новая версия *Main* состоит из одной строки. Вся конфигурационная информация .NET Remoting хранится теперь в файле *JobServer.exe.config*.

ПРИМЕЧАНИЕ По соглашению имя конфигурационного файла образуется путем добавления к имени исполняемого файла приложения строки «.config».

Ниже показан конфигурационный файл *JobServer.exe.config*:

```
<configuration>
  <system.runtime.remoting>
    <application name="JobServer">
      <service>
        <wellknown mode="Singleton">
```

```

        type="JobServerLib.JobServerImpl, JobServerLib"
        objectUri="JobURI" />
    </service>
    <channels>
        <channel ref="http"
            port="4000" />
    </channels>
</application>
</system.runtime.remoting>
</configuration>

```

Обратите внимание на то, как соотносится содержимое конфигурационного файла и код удаленного взаимодействия, добавленный ранее. Элемент `<channel>` содержит ту же самую информацию для настройки канала, которая была использована в первоначальном варианте программной конфигурации. Программная регистрация общеизвестного объекта заменена на элемент `<well-known>` конфигурационного файла. Информация для регистрации объектов с серверной активизацией заключена в элементе `<service>`. И элемент `<service>`, и элемент `<channels>` могут содержать несколько вложенных элементов. Как видно из данного примера, возможности конфигурационных файлов просто поразительны.

Реализация приложения JobClient

Теперь мы перейдем к реализации клиентского приложения, выполняющего вызовы методов удаленных объектов, которые предоставляет приложение `JobServer`. Вновь подчеркнем «ненавязчивость» .NET Remoting: средства удаленного взаимодействия в .NET как бы находятся в тени и для их применения не требуется писать много вспомогательного кода. Теперь рассмотрим основные этапы реализации поддержки удаленного взаимодействия в клиентском приложении.

Выбор клиентского домена приложения

Мы уже рассказывали о нескольких вариантах реализации сервера для экземпляра удаленного объекта `JobServerImpl`. Те же самые варианты доступны и при реализации клиентского приложения.

Мы решили реализовать приложение `JobClient` как приложение Windows Forms на языке C#. Это простое приложение, состоя-

щее из глазной формы, с элементом управления *ListView*, который содержит колонку для каждого члена структуры *JobInfo*: *JobID*, *Description*, *User* и *Status*.

На форме также есть три кнопки, позволяющие выполнять следующие действия:

- создавать новое задание;
- назначать задание для исполнения;
- завершать задание.

Далее подразумевается, что мы создали новый проект C# Windows Application в Microsoft Visual Studio .NET. После создания проекта добавьте к форме *Form1* элемент управления *System.Windows.Forms.ListView* и три элемента управления *System.Windows.Forms.Button*, чтобы внешний вид формы стал похож на показанный на рис. 3-1.

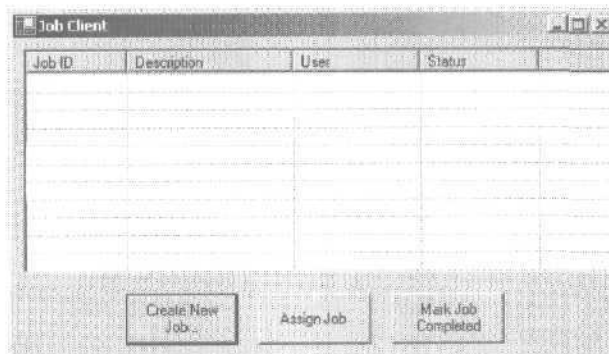


Рис. 3-1. Главная форма приложения **JobClient**

Приложение *JobClient* взаимодействует с экземпляром класса *JobServerImpl*, разработанного нами в предыдущем разделе. Таким образом, для того чтобы клиент мог использовать интерфейс *IJobServer*, класс *JobServerImpl*, структуру *JobInfo* и класс *JobEventArgs*, нам придется добавить ссылку на сборку *JobServerLib.dll*.

Так как класс *Form1* будет взаимодействовать с экземпляром класса *JobServerImpl*, добавим член типа *JobServerImpl* и метод *GetJobServer* в класс *Form1*, размещенный в файле *Form1.cs*:

```

using JobServerLib;

public class Form1 : System.Windows.Forms.Form
{
    // Это поле содержит ссылку на интерфейс IJobServer
    // удаленного объекта.
    private IJobServer m_IJobServer;

    private IJobServer GetIJobServer()
    {
        return (IJobServer)new JobServerImpl();
    }
}

```

Хотя *JobServerImpl* — густанцируемый, так как является производным от *MarshalByRefObject*, экземпляр *JobServerImpl*, создаваемый методом *GetIJobServer*, — локальный для домена приложения программы *JobClient*. Чтобы он стал удаленным, нам нужно настроить сервисы .NET Remoting, чем мы и займемся в конце раздела после реализации алгоритмов работы клиентского приложения. Пока же мы разработаем все клиентское приложение с использованием локального экземпляра класса *JobServerImpl*. Как говорилось в начале раздела, это дает ряд преимуществ, одно из которых — возможность быстро разработать прототип приложения, отложив решение проблем, связанных с .NET Remoting. Ниже показан конструктор класса *Form1*:

```

public Form1()
{
    //
    // Необходим для поддержки Windows Form Designer.
    //
    InitializeComponent();

    // Получить ссылку на удаленный объект.
    m_IJobServer = GetIJobServer();

    // Подписаться на событие JobEvent.
    m_IJobServer.JobEvent +=
        new JobEventHandler(this.MyJobEventHandler);
}

```

Последний оператор выполняет подписку на событие *JobEvent*. Подписка будет отменена, когда пользователь завершит приложение. Далее приведен обработчик сообщения события *Form.Close*:

```
private void OnClosed(object sender, System.EventArgs e)
{
    // Отмена подписки на JobEvent.
    m_IJobServer.JobEvent -= new
        JobEventHandler(this.MyJobEventHandler);
}
```

Как вы помните из предыдущего раздела, экземпляр *JobServerImpl* генерирует *IJobServer.JobEvent* всякий раз, когда клиент создает новое задание или изменяет статус задания на «Assigned» или «Complete». Вот реализация метода *MyJobEventHandler*:

```
public void MyJobEventHandler(object sender, JobEventArgs args)
{
    switch(args.Reason)
    {
        case JobEventArgs.ReasonCode.NEW:
            AddJobToListView(args.Job);
            break;
        case JobEventArgs.ReasonCode.CHANGE:
            UpdateJobInListView(args.Job);
            break;
    }
}
```

MyJobEventHandler использует два вспомогательных метода, о которых мы скоро поговорим. В зависимости от значения свойства *Reason* экземпляра *JobEventArgs* метод либо добавляет новое задание в окно списка, либо обновляет в списке строку существующего задания.

ВНИМАНИЕ! Объявление метода *MyJobEventHandler* с типом доступа, отличным от открытого, приведет к генерации исключения *System.Runtime.Serialization.SerializationException* при попытке клиентского приложения подписаться на *JobEvent*. Соответствующее сообщение об ошибке гласит «Serialization will not deserialize delegates to nonpublic methods.» Это важно с точки зрения защиты, так как

иначе появилась бы «лазейка», позволяющая незаконно вызывать неоткрытые методы.

Обратите внимание, что обратный вызов произойдет в потоке, отличном от того, который создал элемент управления *Form1*. Из-за ограниченной поддержки многопоточности в элементах управления большая часть методов последних не является потокобезопасными и вызов метода *Control.xxxx* из потока, отличного от потока-создателя, может иметь непредсказуемые результаты, в том числе взаимоблокировку. К счастью, тип *System.Windows.Forms.Control* предоставляет ряд методов (таких, как *Invoke*), посредством которых другие потоки могут заставить поток-создатель вызывать методы экземпляра элемента управления. Метод *Invoke* содержит два параметра: экземпляр вызываемого делегата и массив объектов, передаваемых как параметры целевому методу.

Метод *AddJobToListView* использует *ListView.Invoke* для вызова *ListView.Items.Add* из потока-создателя. Прежде чем использовать *ListView.Invoke* для вызова некоторого метода, необходимо определить делегат для этого метода. Далее показано определение делегата для метода *ListView.Items.Add*:

```
// Делегат для метода ListView.Items.Add .
delegate ListViewItem dlgListViewItemsAdd(ListViewItem lvItem);
```

Метод *AddJobToListView* использует *Invoke* для добавления к окну списка информации о новом задании, как показано ниже:

```
// Добавление задания в окно списка.
void AddJobToListView(JobInfo ji)
{
    // Создать делегат для метода listView1.Items.Add.
    dlgListViewItemsAdd lvadd =
        new dlgListViewItemsAdd( listView1.Items.Add );

    // Поместить данные JobInfo в экземпляр ListViewItem.
    ListViewItem lvItem =
        new ListViewItem(new string[] { ji.m_nID.ToString(),
                                         ji.m_sDescription,
                                         ji.m_sAssignedUser,
                                         ji.m_sStatus } );

    // Добавить ListViewItem к окну списка посредством Invoke.
    listView1.Invoke( lvadd, new object[] { lvItem } );
}
```

Реализация *UpdateJobInListView*, использующая ту же самую модель для вызова метода *GetEnumerator* класса набора *ListView.Items*, показана далее:

```
// Обновить информацию о задании в окне списка.
void UpdateJobInListView(JobInfo ji)
{
    IEnumerator ie = (IEnumerator)listView1.Invoke(new
        dlgItemsGetEnumerator(listView1.Items.
            GetEnumerator()));

    while( ie.MoveNext() )
    {
        // Найти в окне списка элемент, соответствующий
        // данному JobInfo.
        ListViewItem lvItem = (ListViewItem)ie.Current;
        if ( ! lvItem.Text.Equals(ji.m_nID.ToString()) )
        {
            continue;
        }

        // Нашли. Теперь обновим подэлементы ListViewItem.
        IEnumerator ieSub = lvItem.SubItems.GetEnumerator();
        ieSub.MoveNext(); // Пропустить JobID.

        // Обновить описание.
        ieSub.MoveNext();
        if ( ((ListViewItem.ListViewSubItem)ieSub.Current).Text !=
            ji.m_sDescription )
        {
            ((ListViewItem.ListViewSubItem)ieSub.Current).Text =
                Ji.m_sDescription;
        }

        // Обновить исполнителя.
        ieSub.MoveNext();
        if ( ((ListViewItem.ListViewSubItem)ieSub.Current).Text !=
            ji.m_sAssignedUser )
        {
            ((ListViewItem.ListViewSubItem)ieSub.Current).Text =
                ji.m_sAssignedUser;
        }

        // Обновить статус.
        ieSub.MoveNext();
        if ( ((ListViewItem.ListViewSubItem)ieSub.Current).Text !=
            ji.m_sStatus )
        {
            ((ListViewItem.ListViewSubItem)ieSub.Current).Text =
                ji.m_sStatus;
        }
    }
}
```

```

    {
        ((ListViewItem.ListViewSubItem)ieSub.Current).Text =
            ji.m_sStatus;
    }
} // End while
}

```

Метод *UpdateJobInListView* выполняет перебор элементов набора *ListView.Items*, отыскивая тот *ListViewItem*, который соответствует идентификатору задания из *JobInfo*. Когда такой элемент найден, выполняется обновление значений его подэлементов, каждый из которых соответствует колонке при отображении в режиме таблицы.

Итак, мы создали экземпляр класса *JobServerImpl* и сохранили ссылку на его интерфейс *IJobServer*. Мы написали код обработки сообщения *JobEvent*, а также рассмотрели использование метода *ListView.Invoke*, посредством которого удастся обновлять содержимое элемента управления *ListView* из потока, отличного от того, который создал экземпляр элемента управления. Теперь нам нужно:

- получить набор всех текущих задач для заполнения *ListView* при загрузке формы;
- реализовать обработчики событий *Button.Click*, чтобы пользователь мог создавать, назначать и завершать задания.

Прежде чем заняться реализацией обработчиков *Button.Click*, полезно рассмотреть ряд вспомогательных функций. Следующий код реализует метод *GetSelectedJob*, возвращающий экземпляр *JobInfo*, соответствующий выбранному в данный момент *ListViewItem*:

```

private JobInfo GetSelectedJobInfo()
{
    JobInfo ji = new JobInfo();

    // Определить выбранное задание.
    IEnumerator ie = listView1.SelectedItems.GetEnumerator();
    while( ie.MoveNext() )
    {
        // Наше окно списка не поддерживает множественное
        // выделение, поэтому можно выбрать
        // не более одного задания.
    }
}

```

```

        ji =
            ConvertListViewItemToJobInfo( (ListViewItem)ie.
                                          Current );
    }
    return ji;
}

```

Этот метод, в свою очередь, использует метод *ConvertListViewItemToJobInfo*, который принимает экземпляр *ListViewItem* и возвращает *JobInfo*, заполненный значениями подэлементов *ListViewItem*:

```

private JobInfo ConvertListViewItemToJobInfo(ListViewItem
                                              lvItem)
{
    JobInfo ji = new JobInfo();
    IEnumerator ieSub = lvItem.SubItems.GetEnumerator();
    ieSub.MoveNext();
    ji.m_nID =
        Convert.ToInt32(
            ((ListViewItem.ListViewSubItem)ieSub.Current).Text);
    ieSub.MoveNext();
    ji.m_sDescription =
        ((ListViewItem.ListViewSubItem)ieSub.Current).Text;
    ieSub.MoveNext();
    ji.m_sAssignedUser =
        ((ListViewItem.ListViewSubItem)ieSub.Current).Text;
    ieSub.MoveNext();
    ji.m_sStatus =
        ((ListViewItem.ListViewSubItem)ieSub.Current).Text;
    return ji;
}

```

Создав вспомогательные функции, мы можем реализовать обработчики событий *Button.Click* для кнопок Assign, Complete и Create:

```

private void buttonAssign_Click(object sender,
                                System.EventArgs e)
{
    // Определить выбранное задание.
    JobInfo ji = GetSelectedJobInfo();
    m_IJobServer.UpdateJobState(ji.m_nID,
                                System.Environment.MachineName,
                                "Assigned");
}

```

```
private void buttonComplete_Click(object sender,
                                System.EventArgs e)
{
    // Определить выбранное задание.
    JobInfo ji = GetSelectedJobInfo();
    m_IJobServer.UpdateJobState(ji.m_nID,
                                System.Environment.MachineName,
                                "Completed");
}
```

Обработчики для кнопок Assign и Complete просто получают текущее выбранное задание и вызывают *IJobServer.UpdateJobState*. Вызов последнего имеет двойной результат:

- экземпляр *JobServerImpl* устанавливает статус задания с указанным идентификатором з «Assigned» или «Completed»;
- экземпляр *JobServerImpl* генерирует событие *JobEvent*.

Наконец, ниже показана реализация для кнопки Create New Job;

```
private void buttonCreate_Click(object sender,
                                System.EventArgs e)
{
    // Отобразить форму Create New Job.
    FormCreateJob frm = new FormCreateJob();
    if ( frm.ShowDialog(this) == DialogResult.OK )
    {
        // Создать задание на сервере.
        string s = frm.JobDescription;
        if ( s.Length > 0 )
        {
            m_IJobServer.CreateJob(frm.JobDescription);
        }
    }
}
```

Метод *buttonCreate_Click* отображает на экране другую форму *FormCreateJob*, которая предлагает пользователю ввести описание нового задания. После того как пользователь ее закроет, метод *buttonCreate_Click* получает введенное пользователем описание. Если описание введено, то вызывается метод *IJobServer.Createjob* экземпляра *JobServerImpl*, который создает новое задание и генерирует событие *JobEvent*.

Для реализации *FormCreateJob* нужно добавить к проекту новую форму. Форма *FormCreateJob* показана на рис. 3-2.

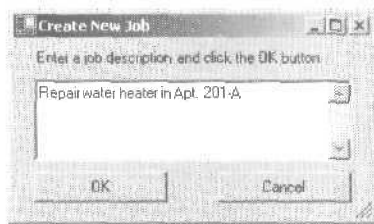


Рис. 3-2. Форма Create New Job приложения JobClient.

Далее показан дополнительный код, необходимый для реализации класса *FormCreateJob*:

```
public class FormCreateJob : System.Windows.Forms.Form
{
    private System.Windows.Forms.Button button1;
    private System.Windows.Forms.Button button2;
    private System.Windows.Forms.TextBox textBox1;
    private System.Windows.Forms.Label label1;

    private string m_sDescription;
    public string JobDescription
    {
        get{ return m_sDescription; }
    }

    private void button1_Click(object sender,
                               System.EventArgs e)
    {
        m_sDescription = textBox1.Text;
    }

    private void button2_Click(object sender,
                               System.EventArgs e)
    {
        this.Hide();
    }
}
```

Получение метаданных сервера

Теперь следует получить метаданные, описывающие удаленный тип. Как говорилось в разделе «Реализация приложения JobServer», метаданные необходимы для двух целей: для компиляции кода клиента, ссылающегося на тип удаленного объекта, а также

для генерации .NET Framework класса прокси, используемого клиентом для взаимодействия с удаленным объектом. В нашем примере включается ссылка на сборку *JobServerLib* и, таким образом, используется реализация типа *JobServerImpl*. Иные способы получения метаданных удаленного объекта мы рассмотрим далее в этой главе, в разделах «Реализация класса *JobServerImpl* в виде Web-сервиса» и «Проблемы зависимости от метаданных».

К этому моменту вы должны суметь скомпилировать приложение, запустить его и протестировать путем создания, назначения и завершения нескольких заданий. На рис. 3-3 показано, как выглядит приложение *JobClient* после того, как пользователь создал несколько заданий.

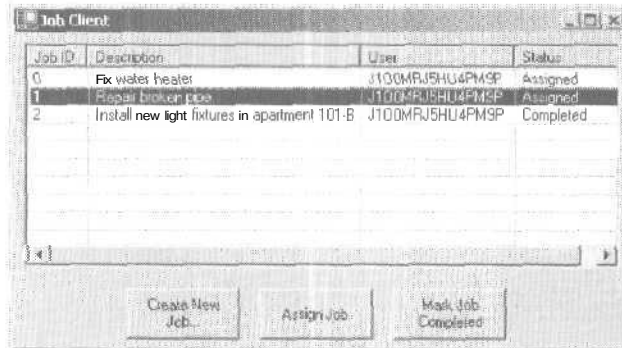


Рис. 3-3. Внешний вид приложения *JobClient* после создания нескольких заданий

Итак, чего же нам удалось добиться? Мы реализовали приложение *JobClient* как приложение Windows Forms на C#. При его запуске метод *GetJobServer* создает на самом деле экземпляр *JobServerImpl* в домене клиентского приложения; таким образом, этот экземпляр не является удаленным. В этом можно убедиться в отладчике, как показано на рис. 3-4.

Как вам уже известно из главы 2, клиент взаимодействует с экземплярами удаленных объектов посредством прокси. В настоящий момент член *m_JobServer* ссылается на экземпляр класса *JobServerLib.JobServerImpl*. Так как класс *JobServerImpl* — производный от *System.MarshalByRefObject*, его экземпляры — *дистанци-*

объекта. Настройку можно выполнять либо программно, либо с помощью конфигурационных файлов.

Программная настройка

Для настройки удаленного взаимодействия .NET Framework предоставляет класс *RemotingConfiguration* из пространства имен *System.Runtime.Remoting*. Как говорилось в разделе «Реализация приложения *JobServer*», этот класс содержит различные методы, обеспечивающие программную настройку инфраструктуры удаленного взаимодействия. Рассмотрим методы, предоставляемые этим классом для клиентов.

Для настройки удаленного взаимодействия в приложении *JobClient* можно модифицировать метод *GetJobServer* следующим образом:

```
private IJobServer GetJobServer()
{
    //
    // Регистрация канала.
    HttpChannel channel = new HttpChannel(0);
    ChannelServices.RegisterChannel(channel);

    //
    // Регистрация типа JobServerImpl как общеизвестного объекта.
    WellKnownClientTypeEntry remotetype =
        new WellKnownClientTypeEntry(typeof(JobServerImpl),
            "http://localhost:4000/JobURI");
    RemotingConfiguration.RegisterWellKnownClientType
        (remotetype);

    return (IJobServer)new JobServerImpl();
}
```

Здесь создается экземпляр класса *HttpChannel*, конструктору которого передается 0. Значение 0 заставляет канал выбрать любой доступный порт для приема входящих запросов на соединение. Если использовать конструктор по умолчанию (без параметров), то канал не будет ожидать входящих вызовов на каком-либо порту и годен только для исходящих вызовов удаленного объекта. Так как приложение *JobClient* подписывается на событие *JobEvent*, генерируемое экземпляром *JobServerImpl*, то необходимо задействовать канал, способный принять обратный вы-

зов при генерации события *JobEvent*. Если обратный вызов должен происходить на определенном порту, то стоит указать номер этого порта вместо 0. После возврата управления из конструктора приложение ожидает входящие вызовы либо на *заданном*, либо на любом свободном порту.

ПРИМЕЧАНИЕ Наиболее распространенные типы .NET Remoting *определены* в сборке *mscorlib.dll*. Однако некоторые дополнительные типы, такие, как *HttpChannel* в пространстве имен *System.Runtime.Remoting.Channels.Http*, определены в другой сборке — *System.Runtime.Remoting.dll*.

После создания экземпляра *HttpChannel* его необходимо зарегистрировать с помощью метода *RegisterChannel* класса *ChannelServices*. Этот метод добавляет интерфейс *ICChannel* экземпляра *HttpChannel* к внутренней структуре данных, представляющей каналы, зарегистрированные в домене клиентского приложения. После регистрации канал передает сообщения .NET Remoting между клиентом и сервером. Канал также принимает обратные вызовы от сервера на *порту ожидания* (listening port).

ПРИМЕЧАНИЕ Разрешается зарегистрировать более одного канала при условии уникальности их имен. Например, доступ к одному и тому же удаленному объекту может предоставляться одновременно как по *HttpChannel*, так и по *TcpChannel*. Таким образом, один сервер будет обслуживать как клиентов за брандмауэром (посредством *HttpChannel*), так и клиентов .NET в интрасети (посредством более скоростного *TcpChannel*).

Далее следует настроить инфраструктуру .NET Remoting таким образом, чтобы *run JobServerImpl* рассматривался как удаленный объект, расположенный за пределами домена приложения Job-Client. Для настройки общеизвестных объектов на клиентской стороне класс *RemotingConfiguration* предоставляет метод *RegisterWellKnownClientType*. Имеются две версии этого метода. Та, которая используется в нашем примере, принимает экземпляр

класса *System.Runtime.Remoting.WellKnownClientTypeEntry*. При создании экземпляра этого класса указывается тип удаленного объекта — в данном случае, *typeof(JobServerImpl)*— и общеизвестный URL объекта.

ПРИМЕЧАНИЕ Класс *RemotingConfiguration* предоставляет вариант метода *RegisterWellKnownClientType* с двумя аргументами: экземпляром *System.Type*, соответствующим типу удаленного объекта, и строкой URL:

```
RemotingConfiguration.RegisterWellKnownClientType(
    typeof(JobServerImpl),
    http://localhost:4000/JobURI );
```

Этот вариант метода *RegisterWellKnownClientType* применяет свои аргументы для создания экземпляра *WellKnownClientType*, который затем передается первому варианту *RegisterWellKnownClientType*.

К настоящему моменту мы настроили удаленное взаимодействие в приложении *JobClient*. Теперь приложение может подключиться к объекту *JobServerImpl*, расположенному в приложении *JobServer*, без необходимости каких-либо дальнейших изменений в коде приложения. Регистрация типа удаленного объекта требуется не всегда, а лишь если вы хотите для создания экземпляров удаленного типа использовать ключевое слово *new*. Другой вариант — применение метода *Activator.GetObject*, который мы рассмотрим далее в разделе «Удаленный доступ к интерфейсу *IJobServer*».

Конфигурационный файл

Второй метод, предоставляемый .NET Remoting для настройки параметров удаленного взаимодействия, использует конфигурационные файлы. Мы уже говорили об использовании конфигурационных файлов в разделе «Реализация приложения *JobServer*». При настройке клиентского приложения применяются другие теги. Вот так выглядит XML-код конфигурационного файла приложения *JobClient*:

```

<configuration>
  <system.runtime.remoting>
    <application name="JobClient">
      <client>
        <wellknown
          type="JobServerLib.JobServerImpl, JobServerLib"
          url="http://localhost:4000/JobURI" />
        </client>
      <channels>
        <channel ref="http" port="0" />
      </channels>
    </application>
  </system.runtime.remoting>
</configuration>

```

Клиентский конфигурационный файл содержит элемент *<client>*, необходимый для задания удаленных объектов, которые используются клиентским приложением. Показанный выше элемент *<client>* содержит дочерний элемент *<wellknown>*, отвечающий за описание общеизвестных серверных объектов, используемых данным клиентом. Обратите внимание на полное соответствие атрибутов элемента *<wellknown>* параметрам конструктора *WellKnownClientTypeEntry*, вызов которого мы рассматривали ранее, когда выполняли программную настройку. Элемент *<client>* также может содержать элемент *<activated>*, задающий объекты с клиентской активизацией — подробнее об этом далее в разделе «Использование объектов с клиентской активизацией».

Для настройки канала применяются те же самые элементы, что и в конфигурационном файле сервера. Вы можете задать канал HTTP, используя свойство *ref* и значение порта 0, которое позволит инфраструктуре .NET Remoting выбрать любой свободный порт. Далее показано использование файла *JobClient.exe.config* для настройки удаленного взаимодействия путем замещения предыдущей реализации метода *GetJobServer* вызовом *RemotingConfiguration.Configure*:

```

private IJobServer GetIJobServer()
{
    RemotingConfiguration.Configure(
        @"..\..\JobClient.exe.config" );
    return (IJobServer)new JobServerImpl();
}

```

Реализация класса *JobServerImpl* в виде Web-сервиса

Архитектура приложений *JobServer* и *JobClient* во многом похожа на архитектуру традиционных DCOM-приложений. Здесь так же высок уровень связанности, так как и клиент, и сервер должны работать под управлением общезыковой исполняющей среды. Применение CLR на всех уровнях распределенного приложения позволяет использовать все возможности системы типов .NET Framework, единую среду разработки программ, а также выбирать наиболее эффективные варианты коммуникации. Кроме того, мы подразумевали отсутствие между *JobServer* и *JobClient* брандмауэра (что требовалось в случае DCOM). Подобные допущения дали нам большую свободу выбора проектных решений, и подобная гибкость очень хороша для решений на базе интрасетей или специализированных сетей. Если при разработке приложений .NET Remoting подобные допущения справедливы, то возможности и производительность DCOM отлично реализуются при применении более простой и гораздо более расширяемой модели программирования.

Со временем, однако, требования к приложению для внутреннего использования могут измениться так, что вам придется предоставить доступ извне ко всей системе или ее части. Обычно при этом следует учитывать влияние таких факторов, как Интернет, брандмауэры и неизвестные, неконтролируемые клиенты, которые не обязательно работают под CLR. Для серверных приложений, реализованных с помощью DCOM, подобный переход совсем непрост. Обычное решение — код ASP (Active Server Pages) или теперь ASP.NET, который вызывает DCOM-объекты и реализует для внешнего мира интерфейс на базе HTML. Однако внешним пользователям могут потребоваться сервисы внутреннего приложения непосредственно для интеграции этих сервисов в свое приложение или для создания собственного пользовательского интерфейса. Если в качестве внутренних сервисов доступен только предназначенный для отображения HTML, то интеграция с другими приложениями трудна и плохо переносит какие-либо изменения. Обычным решением для внешних разработчиков будет разбор HTML, который предназначен только для визуального просмотра. Подобные решения очень ненадежны,

так как работе приложения может помешать любое изменение в пользовательском интерфейсе HTML. Кроме того, внешний вид Интернет-приложений корректируется очень часто. Для большинства сценариев взаимодействия между организациями или внутри одной организации прямой доступ к сервисам гораздо более мощное, гибкое и надежное решение по сравнению с фиксированным пользовательским интерфейсом HTML.

В этом состоит мощь Web-сервисов. Они обеспечивают доступ по Интернету к компонентным сервисам, а не просто передают информацию для отображения. На основе этих сервисов можно строить приложения большого масштаба, аналогичные традиционным программным компонентам. Web-сервисы являются абстракцией высокого уровня, для реализации которой необходим ряд технологий, таких, как SOAP; HTTP; XML; Web Service Description Language (WSDL) и Universal Description, Discovery, and Integration (UDDI). Использование Web-сервисами HTTP и SOAP позволяет преодолевать брандмауэры. WSDL требуется для описания открытых методов, параметров и расположения Web-сервисов, а UDDI предоставляет каталог для поиска Web-сервисов (WSDL-файлов) в Интернете.

ПРИМЕЧАНИЕ WSDL - это XML-грамматика, позволяющая создавать описания Web-сервисов, не зависящие от платформы и языка. Документ WSDL позволяет клиентам, использующим любые платформы, отыскивать Web-сервис, исполняющийся на любой платформе, и вызывать его открытые методы. Таким образом, WSDL схож с Interface Definition Language (IDL) за исключением того, что WSDL задает и расположение (URL) сервиса. Инструменты WSDL позволяют разработчикам создавать .wsdl-файл для указанного Web-сервиса и автоматически генерировать клиентский код, вызывающий этот сервис. В .NET Framework имеется два таких инструмента: WSDL.exe и SOAPSuds.exe. SOAPSuds.exe применяется для Web-сервисов на основе .NET Remoting, тогда как WSDL.exe поддерживает только Web-сервисы на основе ASP.NET.

Все эти технологии являются открытыми, и все отправлены в комитет W3C на рассмотрение (подробнее — на <http://www.w3c.org>). Разработчик, поддерживающий набор общих стандартов и облагающий инструментарием, совместимым с этими стандартами, должен иметь возможность на любом языке и на любой платформе создавать Web-сервис, доступный программам, написанным на любом языке и исполняющимся на любой платформе. Из-за изменений в некоторых, еще не окончательно установленных стандартах скорости обновления инструментальных средств их поставщиками, такая универсальная открытость на момент написания данной книги достигнута лишь частично. Подобнее о написании Web-сервисов с наибольшей степенью открытости — в статье MSDN «Designing Your Web Service for Maximum Interoperability».

Ограничения Web-сервисов

- Как мы уже отмечали, Web-сервисы могут взаимодействовать с клиентами, исполняющимися на любых платформах, которые в свою очередь написаны практически на любом языке, а также способны проникать через брандмауэры. Тогда почему же не использовать их всегда? Ответ прост; для достижения такой открытости приходится поступаться частью богатых функциональных возможностей, предоставляемых .NET Remoting. Подобно тому, как IDL или библиотеки типов предоставляют описания интерфейсов и типов данных COM-объектов, используя подход «наименьшего общего знаменателя», WSDL представляет собой компромиссное описание синтаксиса вызова и типов данных для дистанцируемых объектов. .NET Remoting позволяет работать со всеми типами CLR, тогда как Web-сервисы ограничены общими типами, которые легче отображаются на системы типов большинства языков.

Из-за ограничений WSDL описание удаленных объектов .NET с его помощью требует отказа от использования свойств. Хотя объект по-прежнему может поддерживать свойства для других .NET-клиентов, клиентам Web-сервиса следует предоставлять иной механизм для чтений и изменения этих данных.

Работа через брандмауэры требует не только использования HTTP. Обратные вызовы клиентов могут стать проблемой, так как требуют соединения от сервера к клиенту. При обратном вызове клиент становится сервером, но если брандмауэром закрыт порт, на котором клиент ожидает обратного вызова, то обратный вызов невозможен. Хотя для подобных случаев существуют обходные приемы, они не решают всех проблем, когда клиенты расположены за брандмауэрами NAT (Network Address Translation). Более того подобные обходные приемы не годятся для большинства клиентов Web-сервисов из-за того, что необходимый нестандартный клиентский код запатентован. Так как такие аппаратные брандмауэры способны транслировать IP-адреса, сервер не может выполнить обратный вызов клиента по переданному последним IP-адресу.

Необходимые изменения

Для тестирования Web-сервиса JobServer мы используем модифицированную версию приложения jobClient. Хотя наш тестовый клиент исполняется под управлением CLR, любой клиент не на основе .NET при наличии утилиты, поддерживающей создание кода, вызываемого клиентом из WSDL-файла, сможет работать с Web-сервисом JobServer.

Далее описаны изменения в нашем приложении, необходимые для реализации поддержки Web-сервисов.

Удаление обратных вызовов клиентов

Так как наш Web-сервис не поддерживает обратные вызовы клиентов, нам потребуется иной способ контроля за изменениями списка заданий. С этой целью мы проведем опрос, вызывая метод *GetJobs* каждые 5 секунд. Данный прием весьма распространен в клиентских Web-приложениях и легко реализуем с помощью элемента управления Timer, предоставляемого Windows Forms. Наличие опроса также несколько упрощает клиентское приложение. Вместо инкрементального обновления списка *JobInfo* новой информацией о заданиях, мы будем получать при каждом опросе весь список заново. Кроме того, так как клиент не обрабатывает удаленные события, то нет и второго потока, а следовательно

но, и необходимости задействовать для обновления пользовательского интерфейса метод *Invoke*.

Выбор метода активизации

Обычно мы реализуем Web-сервисы как серверы с режимом активизации *SingleCall*, использующие для сохранения состояния некоторое постоянное хранилище, например базу данных. Для простоты мы зададим режим активизации *Singleton* и будем хранить состояние в памяти. Кроме того, можно реализовать Web-сервис с поддержкой состояния сессии на основе объектов с клиентской активизацией, однако это усложнит или сделает невозможным использование такого Web-сервиса некоторыми клиентами не на основе .NET.

Теперь, когда приложение *JobServer* удовлетворяет требованиям к Web-сервисам, нам нужно настроить IIS в качестве сервера объектов *JobServerImpl*. Это не обязательно для Web-сервисов, но необходимо при использовании IIS в качестве сервера для любого удаленного объекта.

Настройка виртуального каталога

Для настройки виртуального каталога сначала создадим новое Web-приложение с помощью оснастки MMC (Microsoft Management Console) для IIS. Выделите узел дерева Default Web Site, выберите *Action/New/Virtual Directory* и укажите псевдоним (*alias*) виртуального каталога *JobWebService*. Затем следует настроить защиту Web-приложения. По умолчанию IIS применяет для новых Web-приложений Windows-интегрированную аутентификацию (NTLM). Это сделает наше приложение недоступным для не-Windows-клиентов и пользователей, не обладающих достаточными правами на физический NTFS-каталог, псевдонимом которого служит *JobWebService*. Чтобы обеспечить доступ к нашему Web-сервису для всех клиентов, мы «отключим» защиту, установив анонимный доступ. (Защита будет включена снова в разделе «Защита Web-сервиса».) Сначала выделите приложение *JobWebService* в узле дерева Default Web Site. Выберите *Action/Properties*. В диалоговом окне свойств укажите *Directory Security* и щелкните кнопку *Edit*. Уберите отметку из поля *Integrated Windows Authentication* и поставьте ее в поле *Anonymous Access*.

Настройка файла Web.config

Теперь нужно внести изменения в конфигурационный файл. Для конфигурирования удаленного приложения, доступ к которому осуществляется посредством MS, необходим файл *Web.config*, расположенный в корневом виртуальном каталоге приложения:

```
<configuration>
  <system.runtime.remoting>
    <application>
      <service>
        <wellknown mode="SingleCall"
          type="JobServerLib.JobServerImpl, JobServerLib"
          objectUri="JobServer.soap" />
      </service>
    </application>
  </system.runtime.remoting>
</configuration>
```

Элемент *<wellknown>* очень похож на одноименный элемент, использованный нами ранее. Важное отличие, что атрибут *objectUri* имеет расширение *.soap*. Общеизвестные объекты доступные посредством IIS должны иметь URI, заканчивающиеся либо на *.rem*, либо на *.soap*.

Обратите внимание на отсутствие элемента *<channel>* и вспомните, что в предыдущих примерах этот тег требовался для настройки типа канала и номера порта. По умолчанию для удаленных объектов, доступ к которым осуществляется посредством IIS, используется канал *HttpChannel*, что также обязательно требуется для Web-сервисов. Этот канал автоматически обращается к тому же порту, что и IIS (по умолчанию — порт 80). Для настройки IIS на использование другого порта, снова запустите оснастку MMC для IIS, выделите Default Web Site, выберите Action/Properties и задайте номер порта на вкладке Web Site.

Развертывание

Удаленные объекты, для доступа к которым требуется IIS, должны находиться либо в подкаталоге *\bin* виртуального каталога, либо в глобальном кэше сборок (global assembly cache, GAC). Для простоты мы поместим *JobServerImpl* объект в каталог *\bin*.

Использование SOAPSuds

SOAPSuds — утилита Microsoft для получения описания Web-сервисов на основе .NET Remoting в различных форматах. SOAPSuds годится как для локальной сборки, так и для конечной точки объекта .NET Remoting, доступ к которому осуществляется посредством IIS. Основных варианта формата результатов четыре:

- сборка с реализацией;
- сборка с метаданными;
- XML-схема (WSDL);
- компилируемый класс.

Рассмотрим каждый из этих форматов.

Сборка с реализацией

Сборка, содержащая реализацию класса `JobServerLib`, создается показанной ниже командой. Вы можете запустить SOAPSuds непосредственно для локальной сборки `JobServerLib`, используя опцию `-types`:

```
Soapsuds -types:JobServerLib.JobServerImpl,JobServerLib
        -oa:JobServerLib.dll
```

Синтаксис опции `-types` таков:

ПространствоИмен.ИмяКласса,ИмяСборки

Опция `-oa` (сокращение от *output assembly*) вызывает генерацию сборки, содержащую реализацию класса `JobServerImpl`.

Более интересен запуск SOAPSuds с указанием конечной точки удаленного объекта, доступ к которому осуществляется через IIS:

```
Soapsuds -url:http://localhost/JobWebService/JobServer.soap?wsdl
        -oa:JobServerLib.dll
```

Сборка с метаданными

SOAPSuds также предоставляет простой способ генерации сборки, содержащей только метаданные. Вот как это можно сделать для приложения `JobWebService`:

```
Soapsuds -url:http://localhost/JobWebService/JobServer.soap?wsdl
        -oa:JobServerLib.dll
```

Многие разработчики считают этот прием наиболее простым способом генерации сборки, содержащей только необходимый минимум информации для клиента .NET Remoting.

XML-схема (WSDL)

Конечно, сборку .NET имеет смысл генерировать только для клиентов .NET. Нам также необходим способ генерации WSDL-описания нашего Web-сервиса для клиентов других типов. SOAPSuds генерирует файл схемы с описанием Web-сервиса, если ему задан флаг `-os`:

```
Soapsuds -url:http://localhost/JobWebService/JobServer.soap?wsdl  
-os:JobServerLib.wsdl
```

Кстати, то же самое WSDL-описание вы получите, обратившись с помощью Microsoft Internet Explorer к конечной точке Web-сервиса следующим образом:

```
http://localhost/JobWebService/JobServer.soap?wsdl
```

Затем щелкните правой кнопкой клиентскую область браузера и выберите View Source из контекстного меню. Данный файл функционально идентичен WSDL-файлу, сгенерированному SOAP-Suds. Его можно передать соответствующему инструментальному средству для генерации прокси. Клиенты, разработанные не для .NET, применяют этот способ для создания кода для вызова Web-сервиса, созданного на основе .NET Remoting.

Компилируемый класс

SOAPSuds также может преобразовать WSDL-файл в компилируемый исходный текст для .NET, если ему заданы флаги `-is` и `-gc`:

```
Soapsuds -is:JobServer.wsdl -dc
```

SOAPSuds также способен преобразовать WSDL-файл в сборку непосредственно:

```
Soapsuds -is:JobServer.wsdl -oa:JobServerLib.dll
```

ПРИМЕЧАНИЕ Вы, вероятно, помните из главы 2, что канал *HttpChannel* по умолчанию использует сетевой формат SOAP для сериализации сообщения. Как следует из названия, основным назначением

SOAPSuds является генерация метаданных для объектов, поддерживаемых серверами .NET Remoting, которые используют *HttpChannel*. Именно по этой причине по умолчанию SOAPSuds генерирует так называемый *закрытый прокси* (wrapped proxy). Последний содержит жестко зашитый URL сервера и поддерживает только *HttpChannel*, что удобно для нашего клиента Web-сервиса. Однако SOAPSuds также может генерировать и *открытый* (nonwrapped) прокси, который поддерживает *TcpChannel* и допускает явное задание URL сервера из вызывающей программы. Для генерации открытой сборки с метаданными используйте опцию *-nowp*;

```
Soapsuds -ia:InputAssemblyName -oa:OutputAssemblyName.dll
        -nowp
```

При применении сгенерированной таким образом сборки необходимо указать URL сервера и нужный канал либо программно, либо с помощью конфигурационного файла.

Защита Web-сервиса

Как говорилось ранее, простота настройки защиты — одна из основных причин использования IIS в качестве среды для приложений .NET Remoting. Защита для Web-сервисов на основе .NET Remoting настраивается так же, как и для всех удаленных объектов, доступ к которым осуществляется посредством MS. Таким образом, вы можете применять один и тот же порядок действий и для настройки защиты, и для других случаев применения IIS в качестве серверной среды, например для объектов с клиентской активизацией или использующих несовместимые с Web-сервисами форматовщики, такие, как двоичный форматовщик.

Далее показан пример настройки Web-сервиса *JobServer* для использования NTLM-аутентификации в IIS. В результате IIS выполняет аутентификацию запросов со стороны *JobClient* на основании учетных данных пользователя Windows NT. Иными словами, объект *JobServerImpl* оказывается доступным только тем клиентам, которые могут предоставить учетные данные пользователя,

имеющего достаточные права доступа к виртуальному каталогу `JobWebService`. Имейте в виду, что NTLM подходит только для интрасети. Это связано с требованием наличия у клиентов учетных данных Windows, а также с тем, что NTLM не работает через брандмауэры или прокси-серверы.

Изменения настроек виртуального каталога

Выберите приложение `JobWebService` в узле дерева Default Web Site, затем — Action/Properties. В диалоговом окне свойств укажите Directory Security и щелкните Edit. На этот раз установите отметку в поле Windows Integrated Authentication и уберите ее из поля Anonymous Access.

Изменения в файле `Web.config`

Добавьте следующие строки в файл `Web.config` приложения `JobWebService`:

```
<system.web>
  <authentication mode="Windows"/>
  <identity impersonate="true"/>
</system.web>
```

Эти параметры настраивают ASP.NET на применение Windows-аутентификации и олицетворение пользователя браузера во время обработки на стороне сервера.

Изменения в приложении `JobClient`

Клиенты .NET Framework могут посылать серверу учетные данные (имя пользователя, пароль и имя домена, если используется) двух типов: заданные по умолчанию и явно заданные.

В случае учетных данных по умолчанию применяются идентификатор и пароль текущего пользователя, при этом не требуется, а возможно, и не допускается, явное их указание. Для настройки клиента с помощью конфигурационного файла на использование учетных данных по умолчанию добавьте атрибут `useDefaultCredentials` для канала HTTP:

```
<channels>
  <channel ref="http" useDefaultCredentials="true"/>
</channels>
```

Для программной установки режима учетных данных по умолчанию установите свойство *useDefaultCredentials* канала при его создании:

```
IDictionary props = new Hashtable();
props["useDefaultCredentials"] = true;

HttpChannel channel = new HttpChannel(
    props,
    null,
    new SoapServerFormatterSinkProvider()
);
```

Вместо автоматической передачи серверу учетных данных текущего пользователя может потребоваться явное указание имени пользователя, пароля и имени домена. Так как применение явно задаваемых учетных параметров связано с пересылкой пароля по сети открытым текстом, то данный вариант возможен только в сочетании с каким-либо методом шифрования, например SSL (Secure Sockets Layer).

Учетные параметры можно задать программно, установив свойства приемника канала, как показано ниже. (Подробно приемники канала обсуждаются в главе 7.)

```
IJobServer obj = (IJobServer)Activator.GetObject(
    typeof(IJobServer),
    "http://localhost/JobWebService/JobServer.soap" );

ChannelServices.GetChannelSinkProperties(obj)
    ["username"] = "Bob";
ChannelServices.GetChannelSinkProperties(obj)
    ["password"] = "test";
ChannelServices.GetChannelSinkProperties(obj)["domain"] = "";
```

Конечно, в реальном приложении учетные параметры не защищены жестко в код. Они, вероятно, запрашиваются у пользователя или считываются из защищенного источника.

На этом действия, необходимые для защиты Web-сервиса *JobServer* от неавторизованных пользователей, закончены. Несмотря на то, что Web-сервис теперь защищен, в нашу схему авторизации можно внести еще одно усовершенствование. При текущей конфигурации у пользователя есть либо полный доступ к нашему приложению, либо никакого. Однако может потребо-

ваться поддержка различных уровней доступа, например разрешение удалять незавершенные задания только администраторам. Для осуществления тонкого управления правами доступа пользователей к ресурсам сервера мы можем воспользоваться защитой .NET Framework на основе ролей.

Использование защиты на основе ролей с .NET Remoting

Польза групп известна каждому, кому приходилось управлять правами доступа сколько-нибудь значительного числа Windows-пользователей. Группы применяются потому, что доступ к ресурсам, таким, как файлы и базы данных, а также права на выполнение большинства операций практически никогда не требуется разграничивать на уровне отдельных пользователей. Вместо этого им зачастую выделяются общие уровни доступа, которые можно разбить на роли или группы.

Так как подход на основе ролей или групп часто применяется сетевыми администраторами, то его целесообразно использовать для управления доступом и при разработке приложений. Именно в этом и состоит защита на основе ролей, которая представляет собой особенно эффективный способ управления доступом в приложениях .NET Remoting. За исключением случаев, когда аутентифицированные клиенты имеют доступ ко всем ресурсам приложения, после аутентификации клиента необходимо реализовать управление доступом. Как и в случае других сценариев защиты, в .NET Remoting использование защиты на основе ролей требует применения в качестве серверной среды IIS.

При надлежащей настройке параметров защиты, удаленный объект может использовать объект *Principal*, связанный с текущим потоком, для идентификации текущего клиента и определения его принадлежности к ролям. Затем на основании информации о роли клиента можно запрещать или разрешать исполнение того или иного кода. Программных методов управления доступом три;

- декларативный;
- императивный;
- прямой опрос пользователя.

Декларативное программирование означает, что вы декларируете свои намерения в коде посредством атрибутов, которые при компиляции попадают в метаданные сборки. Используя класс *System.Security.Permissions.PrincipalPermissionAttribute* и перечисление *System.Security.Permissions.SecurityAction*, можно указать, что для исполнения метода вызывающий клиент должен выступать в определенной роли:

```
[PrincipalPermissionAttribute(SecurityAction.Demand,
    Role="BUILTIN\Administrators")]
public void MySecureMethod()
{
    // ...
}
```

Если клиент, вызывающий метод *MySecureMethod*, не относится к группе *BUILTIN\Administrators*, то система генерирует исключение *System.Security.SecurityException*.

Императивное программирование — это традиционное программирование с применением условных проверок ролей в теле метода, как показано ниже:

```
PrincipalPermission AdminPermission =
    new PrincipalPermission("Allen",
        "Administrator");

AdminPermission.Demand();
```

При императивной защите на основе ролей необходимо создать объект *System.Security.Permissions.PrincipalPermission*, которому передается имя пользователя и имя роли. Если клиент является членом заданной роли, то при вызове метода *Demand* объекта *PrincipalPermission* исполнение продолжится, в противном случае система сгенерирует исключение *System.Security.SecurityException*.

Возможны случаи, когда генерация исключительных ситуаций при проверке ролей нежелательна. Они снижают производительность удаленного приложения, поэтому к ним следует прибегать лишь в особых ситуациях. Например, вряд ли разумно проверять список ролей пользователя, создавая несколько объектов *PrincipalPermission* и перехватывая неизбежные исключения. Лучше всего декларативная и императивная защита работают в тех случа-

ях, когда предполагается наличие у пользователя некоторых прав и вы проверяете это предположение.

В тех случаях, когда при проверке ролей пользователя велика вероятность того, что проверка будет неудачной, следует использовать *прямой опрос пользователя* (direct principal access). Вот пример:

```
IPrincipal ClientPrincipal = Thread.CurrentThread.  
                                CurrentPrincipal;  
If ( ClientPrincipal.IsInRole( "Administrator" ) )  
{  
    RunMe();  
}
```

Thread.CurrentThread.CurrentPrincipal возвращает интерфейс *IPrincipal*, идентифицирующий клиента. Данный интерфейс содержит метод *IsInRole*, проверяющий наличие у пользователя заданной роли и возвращающий булево значение.

Защита доступа к коду при удаленном взаимодействии

Исполняя программу, установленную на вашем компьютере, вы должны быть абсолютно уверены, что она не принесет вреда. Сегодня исполняемый код можно получать из различных источников, многие из которых заслуживают мало доверия. Одним из таких источников является Интернет. Это означает, что коду должен быть присвоен уровень доверия, причем их должно быть гораздо больше, чем полное доверие и полное недоверие. Именно в этом состоит задача .NET Code Access Security. Подсистема защиты доступа к коду в .NET Framework предоставляет гибкие и мощные средства управления правами, необходимыми для исполнения кода, а также налагает ограничения на права кода, поступающего из различных зон.

Мы больше не будем обсуждать защиту доступа к коду в этой книге по той простой причине, что она не работает с .NET Remoting. Следовательно, между клиентом и сервером приложения .NET Remoting должен быть высокий уровень доверия.

Как отмечалось ранее, для применения защиты на основе ролей необходимо использовать в качестве серверной среды IIS. При настройке IIS на олицетворение клиента и применении некоторой схемы аутентификации, идентификационная информация клиента пересечет границу .NET Remoting. Так как для других серверных сред, например Windows-службы, клиента .NET Remoting аутентифицировать невозможно, то *Thread.CurrentThread.CurrentPrincipal* содержит пустой *GenericIdentity*. Так как нельзя идентифицировать клиента, то не удастся и управлять доступом. Следовательно, при проектировании удаленных объектов следует тщательно проанализировать, какой тип серверного приложения будет использоваться. Вряд ли вы захотите разрешить неаутентифицированным клиентам вызовы методов своего объекта с ограниченным доступом.

Использование объектов с клиентской активизацией

Ранее мы рассмотрели различные методы создания объектов с серверной активизацией и показали клиентский код, необходимый для взаимодействия с ними. Как говорилось в главе 2, .NET Framework предоставляет еще одну разновидность удаленных объектов: объекты с клиентской активизацией. Они «активизируются» по требованию клиента для каждого клиента, для каждой активизации создается отдельный экземпляр, и между вызовами эти объекты могут сохранять состояние.

Для демонстрации объектов с клиентской активизацией расширим приложение *JobClient*, введя возможность добавления примечаний к выбранному заданию. С этой целью добавим класс *JobNotes*, производный от *MarshalByRefObject*, который настроим как объект с клиентской активизацией. Способность сохранения состояния объектами с клиентской активизацией позволит тексту примечаний сохраняться в промежутках между вызовами методов, пока существует экземпляр объекта *JobNotes*.

Класс *JobNotes*

Реализация объекта с клиентской активизацией аналогична реализации объекта с серверной активизацией: достаточно сделать его класс производным от *System.MarshalByRefObject*. Будет ли

активизация объекта клиентской или серверной, зависит от настройки .NET Remoting в серверном приложении.

Далее показан новый класс *JobNotes*, производный от *System.MarshalByRefObject*:

```
using System.Collections;

public class JobNotes : MarshalByRefObject
{
    private Hashtable m_HashJobID2Notes;

    public JobNotes()
    {
        m_HashJobID2Notes = new System.Collections.Hashtable();
    }

    public void AddNote(int id, string s)
    {
        // Определяется позже...
    }

    public ArrayList GetNotesCint id)
    {
        // Определяется позже...
    }
}
```

Класс *JobNotes* позволяет клиентам связать с выбранным идентификатором задания текстовые примечания. Класс содержит член типа *System.Collections.Hashtable*, связывающий идентификатор задания с массивом строк *System.Collections.ArrayList*, хранящим примечания к этому заданию.

Метод *AddNote* добавляет примечание к заданию с определенным идентификатором:

```
public void AddNote(int id, string s)
{
    // Найти список примечаний.
    ArrayList al = (ArrayList)m_HashJobID2Notes[id];
    if (al == null)
    {
        al = new ArrayList();
        m_HashJobID2Notes[id] = al;
    }
}
```

```

    }

    // Вставить метку времени.
    s = s.Insert(0, Environment.NewLine);
    s = s.Insert(0, System.DateTime.Now.ToString());

    // Добавить s к списку примечаний.
    al.Add(s);
}

```

Реализация метода *GetNotes* такова:

```

public ArrayList GetNotes(int id)
{
    // Найти примечания по ID задания.
    ArrayList notes = (ArrayList)m_HashJobID2Notes[id];
    if (notes != null )
    {
        return notes;
    }
    return new ArrayList();
}

```

Изменения в приложении jobClient

Для того чтобы воспользоваться классом *JobNotes*, потребуются некоторые изменения в клиентском приложении. Необходимо предоставить пользовательский интерфейс для ввода нового примечания к выбранному заданию. Для этого понадобится еще одна кнопка, при нажатии которой будет отображаться форма ввода нового примечания. В этом приложении мы разрешим пользователю добавлять примечания только в том случае, если данное задание закреплено за ним.

Внесем в пользовательский интерфейс изменения, необходимые для поддержки ввода примечаний к выбранному в данный момент заданию. Для начала создадим новую форму *FormAddNote*, отображающую список текущих примечаний к данному заданию и дающую возможность ввода нового примечания. Создайте форму, похожую на ту, что изображена на рис. 3-6.

Добавьте элемент управления *TextBox* с именем *textBoxNotes*, который будет содержать текущий список примечаний. Данное поле должно допускать только чтение. Добавьте еще один элемент управления *TextBox* с именем *textBoxAddNote*, где пользо-

ватель будет вводить текст примечания. И, наконец, добавьте стандартные кнопки OK и Cancel.

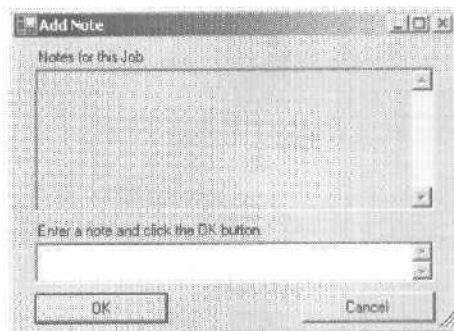


Рис. 3-6. Внешний вид формы *FormAddNote*

Для отображения текущих примечаний к заданию, а также для получения текста нового примечания клиентский код устанавливает свойство *Description* класса *FormAddNote* перед отображением формы и считывает значение этого свойства после закрытия формы. Определение свойства *Description* таково:

```
private string m_sDescription;

public string Description
{
    get
    { return m_sDescription; }

    set
    { m_sDescription = value; }
}
```

При загрузке формы значение свойства *Description* помещается в поле, на которое ссылается член *textBoxNotes*. Ниже показан соответствующий код обработчика события *Form.Load*:

```
private void FormAddNote_Load(object sender, System.EventArgs e)
{
    this.textBoxNotes.Text = m_sDescription;
}
```

Кроме того, для каждой из кнопок потребуются обработчики событий *Button.Click*. Далее показан такой обработчик для события *Click* кнопки Cancel:

```
private void button2_Click(object sender, System.EventArgs e)
{
    this.Hide();
}
```

Обработчик события *Click* для кнопки ОК сохраняет текст, введенный пользователем в *textBoxAddNote*, в поле *m_sDescription*, значение которого затем может быть получено посредством свойства *Description*:

```
private void button1_Click(object sender, System.EventArgs e)
{
    m_sDescription = this.textBoxAddNote.Text;
}
```

Так как каждый экземпляр приложения *JobClient* работает с собственным удаленным экземпляром объекта *JobNotes* с клиентской активизацией, то вы можете добавить переменную-член типа *JobNotes* в класс *Form1* и инициализировать ее в конструкторе этого класса.

Для добавления примечания к выбранному заданию добавим кнопку *buttonAddNote* к *Form1*. Реализация метода *buttonAddNote_Click* показана ниже.

```
private void buttonAddNote_Click(object sender,
                                System.EventArgs e)
{
    // Создать форму ввода примечаний.
    FormAddNote frm = new FormAddNote();

    // Проверить, что задание закреплено за текущим пользователем,
    JobInfo ji = GetSelectedJobInfo();
    if ( ji.m_sAssignedUser != System.Environment.MachineName )
    {
        MessageBox.Show("You are not assigned to that job");
        return;
    }

    // Получить примечания, связанные с текущим заданием
    // и отобразить их в диалоговом окне.
    ArrayList notes = m_JobNotes.GetNotes(ji.m_nID);
    IEnumerator ie = notes.GetEnumerator();
    while(ie.MoveNext())
    {
        frm.Description += (string)ie.Current;
    }
}
```

```
        frm.Description += Environment.NewLine;
        frm.Description += Environment.NewLine;
    }

    // Отобразить форму и получить новое примечание.
    if ( frm.ShowDialog(this) == DialogResult.OK )
    {
        string s = frm.Description;
        if ( s.Length > 0 )
        {
            m_JobNotes.AddNote(ji.m_nID, frm.Description);
        }
    }
}
```

Теперь следует запустить клиент и протестировать работу класса *JobNotes* посредством экземпляра *JobNotes*, создаваемого приложением *JobClient* локально. На рис. 3-7 показана новая форма после того, как пользователь добавил к заданию несколько примечаний.

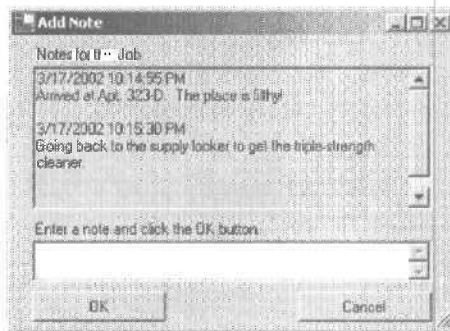


Рис. 3-7. Добавление примечаний к заданию

Настройка клиента для работы с объектами с клиентской активизацией

Чтобы экземпляры типа *JobNotes* стали удаленными, необходимо сообщить инфраструктуре о том, что тип *JobNotes* нужно обрабатывать как объект с клиентской активизацией. Займемся этим, настроив приложение *JobClient* на использование типа *JobNotes* как объекта с клиентской активизацией.

Программная настройка

Класс *RemotingConfiguration* содержит метод *RegisterActivatedClientType*, позволяющий клиентам зарегистрировать тип как активизируемый клиентом. Для этого добавим следующий код в конструктор *Form1* непосредственно перед созданием нового экземпляра *JobNotes*:

```
ActivatedClientTypeEntry acte =
    new ActivatedClientTypeEntry( typeof (JobNotes),
                                "http://localhost:4000" );
RemotingConfiguration.RegisterActivatedClientType( acte );
```

Сначала создается экземпляр типа *ActivatedClientTypeEntry*, конструктору которого передается два параметра:

- тип удаленного объекта;
- URL конечной точки, в которой будут активизироваться экземпляры удаленного типа.

Затем экземпляр *ActivatedClientTypeEntry* передается методу *RegisterActivatedClientType*, который регистрирует тип *JobNotes* как объект с клиентской активизацией. Совмещенный вариант метода *RegisterActivatedClientType* принимает те же самые два параметра, что и конструктор *ActivatedClientTypeEntry*.

Конфигурационный файл

Как уже говорилось, альтернатива программной настройке .NET Remoting — использование конфигурационного файла. Элемент *<client>* необходим для регистрации как общеизвестных объектов, так и объектов с клиентской активизацией. Внутри этого тега добавляется тег *<activated>*, содержащий ту же информацию, которая задается при программной настройке.

Изменим файл *JobClient.exe.config* таким образом, чтобы объявить *JobNotes* объектом с клиентской активизацией:

```
<configuration>
  <system.runtime.remoting>
    <application name="JobClient">
      <client>
        <wellknown
          type="JobServerLib.JobServerImpl, JobServerLib"
          url="http://localhost:4000/JobURI" />
        </client>
```

```

    <client url = "http://localhost:4000">
      <activated type="JobServerLib.JobNotes, JobServerLib"/>
    </client>
  </channels>
  <channel ref="http" port="0" />
</channels>
</application>
</system.runtime.remoting>
</configuration>

```

Необходимо добавить элемент `<client>`, задающий URL конечной точки активизации посредством атрибута `url`. Элемент `<client>` содержит дочерний элемент `<activated>`. Так как приложение `JobClient` использует единственный тип с клиентской активизацией, то конфигурационный файл содержит только один элемент `<activated>`. Если вашему приложению нужно активизировать несколько типов в одной и той же конечной точке, то внутри одного элемента `<client>` придется задать несколько элементов `<activated>` — по одному на тип. Аналогично, если вам нужно активизировать несколько типов в разных конечных точках, потребуется несколько элементов `<client>` — по одному на точку. Тип объекта с клиентской активизацией задается средствами атрибута `type` элемента `<activated>`. В предыдущем листинге можно добавить элемент `<activated>` к существующему элементу `<client>`, который содержит элемент `<wellknown>`, но в этом нет необходимости.

Настройка сервера при использовании объектов с клиентской активизацией

Теперь нам необходимо изменить приложение `JobServer`, чтобы настроить тип `JobNotes` как объект с клиентской активизацией. И снова мы можем выполнить это либо программно, либо посредством конфигурационного файла.

Программная настройка

Класс `RemotingConfiguration` содержит метод `RegisterActivatedServiceType`, позволяющий программно настраивать тип на клиентскую активизацию. Для настройки приложения `JobServer.exe` на клиентскую активизацию экземпляров класса `JobNotes` добавьте следующую строку кода в метод `Main` этого приложения:

```
RemotingConfiguration.RegisterActivatedServiceType(
    typeof(JobNotes));
```

Исполнение этой строки кода регистрирует тип *JobNotes* как объект с клиентской активизацией. Это означает, что сервер будет обрабатывать клиентские запросы на активизацию типа *JobNotes*. При получении запроса на активизацию инфраструктура .NET Remoting создаст экземпляр класса *JobNotes*.

Конфигурационный файл

Для настройки серверного приложения на клиентскую активизацию объектов можно также воспользоваться конфигурационным файлом. Чтобы зарегистрировать тип как объект с клиентской активизацией, следует использовать элемент `<activated>`. Для регистрации класса *JobNotes* как объекта с клиентской активизацией нужно добавить тег `<activated>` в файл *JobServer.exe.config*, как показано ниже:

```
<configuration>
  <system.runtime.remoting>
    <application name="JobServer">
      <service>
        <wellknown mode="Singleton"
          type="JobServerLib.JobServerImpl, JobServerLib"
          objectUri="JobURI" />
        <activated type="JobServerLib.JobNotes, JobServerLib" />
      </service>
    </application>
  </system.runtime.remoting>
</configuration>
```

Спонсор лицензии

Итак, мы реализовали класс *JobNotes* и настроили его на клиентскую активизацию. Теперь пришло время рассмотреть параметры времени жизни для этого класса.

В главе 2 описывалась применяемая инфраструктурой .NET Remoting система управления временем жизни удаленных объектов на основе лицензий. Класс *System.MarshalByRefObject* содержит виртуальный метод *InitializeLifetimeService*, который можно

переопределить в производных классах, чтобы изменить стандартные параметры лицензий и таким образом повлиять на продолжительность жизни объекта.

Стандартная реализация *InitializeLifetimeService* инициализирует объект, реализующий *ILease*, значениями по умолчанию. Свойство *InitialLeaseTime* по умолчанию равно 5 минутам, то есть лицензия объекта не истечет в течение 5 минут. Значение по умолчанию для свойства *RenewOnCallTime* равно 2 минутам. Если в течение времени *RenewOnCallTime* срок действия лицензии закончится, то вызов любого метода объекта продлит ее, устанавливая *CurrentLeaseTime* в *RenewOnCallTime*. Это означает, что при всяком вызове метода объекта его лицензия может быть продлена на время, задаваемое значением свойства *RenewOnCallTime*, если оставшийся срок лицензии меньше значения этого свойства. Свойство *SponsorshipTimeout* по умолчанию равно 2 минутам. То есть если в момент окончания действия лицензии у нее есть спонсоры, то вызов метода *ISponsor.Renewal* завершится по тайм-ауту через две минуты, если за это время не будет получен отклик от спонсора.

Вспомните для сравнения, что класс *JobServerImpl* переопределяет метод *InitializeLifetimeService* и возвращает *null*, указывая, что экземпляры этого класса должны существовать неопределенно долго, пока серверное приложение не завершится. Для общеизвестных объектов в режиме Singleton, таких, как *JobServerImpl*, подобное неограниченное время жизни имеет смысл.

Однако использовать неограниченное время жизни для объектов с клиентской активизацией, таких, как *JobNotes*, нецелесообразно, так как нет смысла сохранять экземпляры *JobNotes* после завершения клиентского приложения. Однако для сохранения примечаний к заданиям на постоянной основе потребуется механизм, с помощью которого отключенным экземплярам *JobNotes* удалось бы сохранить свои данные, прежде чем они будут уничтожены сборщиком мусора. Затем при активизации клиентским приложением нового экземпляра класса *JobNotes* конструктор должен восстановить ранее сохраненные данные. В такой ситуации, вероятно, имеет смысл изменить реализацию класса *JobNotes* и использовать активизацию общеизвестного объекта в режиме *SingleCall*. Но в нашем примере приложения это не требуется.

Тем не менее предположим, что объекты *JobNotes* должны существовать дольше, чем определено стандартной лицензией. Для этого потребуется переопределить метод *InitializeLifetimeService* и инициализировать лицензию объекта значениями, отличными от стандартных.

Инициализация лицензии

Класс *JobNotes* переопределяет метод *InitializeLifetimeService*, унаследованный от *System.MarshalByRefObject*. Как говорилось в разделе «Реализация приложения *JobServer*», инфраструктура .NET Remoting вызывает данный метод для получения информации о лицензии удаленного объекта. Реализация этого метода в классе *JobServerImpl* возвращает *null*, что соответствует неограниченному времени жизни. Вариант реализации *InitializeLifetimeService* в классе *JobNotes*, задающий нестандартные начальные параметры времени жизни объекта, показан ниже:

```
public override Object InitializeLifetimeService()
{
    ILease lease = (ILease)base.InitializeLifetimeService();
    if ( LeaseState.Initial == lease.CurrentState )
    {
        lease.InitialLeaseTime    = TimeSpan.FromMinutes(4);
        lease.SponsorshipTimeout = TimeSpan.FromMinutes(1);
        lease.RenewOnCallTime    = TimeSpan.FromMinutes(3);
    }

    return lease;
}
```

Сначала вызовом *InitializeLifetimeService* из базового класса мы получаем стандартную лицензию. Для изменения параметров лицензии она должна быть в состоянии *LeaseState.Initial*. Если лицензия находится в другом состоянии, то попытка изменения ее свойств вызовет исключение.

Реализация интерфейса *ISponsor*

Чтобы использовать все возможности, предоставляемые механизмом управления временем жизни .NET Remoting, можно создать спонсора, реализовав интерфейс *ISponsor* и зарегистрировав спонсора лицензии удаленного объекта. Когда действие лицензии истекает, инфраструктура вызывает методы *ISpon-*

sor.Renewal зарегистрированных спонсоров, давая каждому из них возможность продлить лицензию.

В приложении JobClient можно сделать спонсором класс *Form1*, для чего он должен быть производным от интерфейса *ISponsor* и предоставить реализацию последнего, как показано ниже:

```
public TimeSpan Renewal(ILease lease)
{
    !
    return TimeSpan.FromMinutes(5);
}
```

Метод *ISponsor.Renewal* возвращает экземпляр *System.TimeSpan*, представляющий 5-минутный интервал, в результате чего лицензия продлевается еще на 5 минут. В зависимости от потребностей вашего приложения, этот интервал может быть длиннее или короче. Использование более длинного интервала позволяет уменьшить частоту вызовов *ISponsor.Renewal* и, таким образом, сократить сетевой трафик, особенно в тех случаях, когда клиентское приложение является спонсором большого числа объектов с клиентской активизацией. Недостаток этого в том, что объекты с клиентской активизацией могут продолжать существовать на сервере дольше, чем это необходимо и, вероятно, занимать при этом дефицитные ресурсы. Более короткий интервал позволяет сократить время, в течение которого эти, так называемые, зомби не уничтожаются, но требует более частых вызовов метода *ISponsor.Renewal* и, таким образом, порождает более интенсивный трафик.

Регистрация спонсора

Завершив реализацию интерфейса *ISponsor*, мы можем зарегистрировать экземпляр *Form1* в качестве спонсора лицензии объекта *JobNotes*. Для этого сначала нужно получить интерфейс *ILease* для удаленного объекта, а затем вызвать метод *ILease.Register*, передав интерфейс *ISponsor* спонсора. Добавим следующий код в конструктор *Form1*:

```
ILease lease = (ILease) RemotingServices.GetLifetimeService(
    m_JobNotes);
lease.Register((ISponsor) this);
```

Теперь, когда приложение JobClient запускается и создает экземпляр *JobNotes*, лицензия последнего будет первоначально дей-

ствовать 4 минуты. Это связано с тем, что класс *JobNotes* переопределил метод *InitializeLifetimeService* и установил свойство *InitialLeaseTime* в 4 минуты. Если в течение первых 4 минут клиент не вызовет ни одного метода экземпляра *JobNotes*, то лицензия закончится и инфраструктура вызовет метод *ISponsor.Renewal* класса *Form1*, который обновит лицензию, продлив время жизни экземпляра *JobNotes* еще на 5 минут.

Проблемы зависимости от метаданных

Ранее в этой главе мы использовали простой подход к проблеме зависимости от метаданных с учетом того, что оба приложения — *JobClient* и *JobServer* — зависят от метаданных друг друга. А именно, *JobClient* нужны метаданные класса *JobServerImpl* и связанных с ним типов. Аналогично, так как клиент подписывается на событие *JobEvent*, программе *JobServer* требуются метаданные класса *Form7*, определенного в приложении *JobClient*. По сути дела, при получении обратного вызова в результате события *JobEvent* программа *JobClient* выступает в качестве сервера. Подобный тип архитектуры иногда называют «сервер — сервер».

Такой способ работы с метаданными вполне годится для приложения-примера, но в реальном мире передача метаданных клиента серверу или метаданных сервера клиенту может быть нежелательной. В этом разделе мы рассмотрим ряд способов решения данной проблемы.

Устранение зависимости *JobServer* от метаданных *JobClient*

Важно, чтобы вы понимали причину, по которой возникает зависимость приложения *JobServer* от метаданных *JobClient*. Ее вызывает следующая строка кода в конструкторе *Form1*:

```
m_IJobServer.JobEvent += new JobEventHandler(
    this.MyJobEventHandler);
```

Единственная причина зависимости *JobServer* от метаданных *JobClient* состоит в том, что класс *Form1*, определенный приложением *JobClient*, подписывается на сообщение *JobServerImpl.JobEvent*. Очевидно, что одним из методов устранения зависимости является замена подписки на событие *JobEvent* опросом. Мы по-

ступили так в разделе «Реализация класса *JobServerImpl* в вуге Web-сервиса», где *JobClient* был изменен для работы с Web-сервисом. Допустим, что данное решение неприемлема.

В этом случае нам потребуется тип, который служил бы связью между *Form1.MyEventHandler* и событием *JobServerImpl.JobEvent*. Нужный класс определяется таким образом:

```
public class JobEventRepeater : MarshalByRefObject
{
    //
    // Событие, на которое подписываются клиенты.
    public event JobEventHandler JobEvent;

    //
    // Обработчик для IJobServer.JobEvent.
    public void Handler(object sender, JobEventArgs args)
    {
        if (JobEvent != null)
        {
            JobEvent(sender, args);
        }
    }

    //
    // Запрет на уничтожения объекта службами
    // управления временем жизни.
    public override object InitializeLifetimeService()
    {
        return null;
    }
}
```

Класс *JobEventRepeater* выступает в качестве *повторителя* (repeater) события *JobEvent*. Класс содержит член *JobEvent* и метод *RepeatEventHandler*, который генерирует *JobEvent*. Чтобы воспользоваться этим классом, клиент создает его новый экземпляр и подписывается на его событие *JobEvent*, а не на *JobServerImpl.JobEvent*. Затем клиент подписывает данный экземпляр *JobEventRepeater* на событие *JobServerImpl.JobEvent*, так что при генерации сервером события *JobEvent* он вызывает метод *JobEventRepeater.RepeatEventHandler*.

Если мы добавим к классу *Form1* член типа *JobEventRepeater* с именем *m_JobEventRepeater*, то для использования класса *Job-*

EventRepeater можно изменить конструктор *Form1* следующим образом:

```
m_JobEventRepeater = new JobEventRepeater();
m_JobEventRepeater.JobEvent +=
    new JobEventHandler(this.MyJobEventHandler);
m_IJobServer.JobEvent +=
    new JobEventHandler(m_JobEventRepeater.Handler);
```

Взаимосвязь между экземпляром *Form1*, экземпляром *JobEventRepeater* и экземпляром *JobServerImpl* изображена на рис. 3-8.

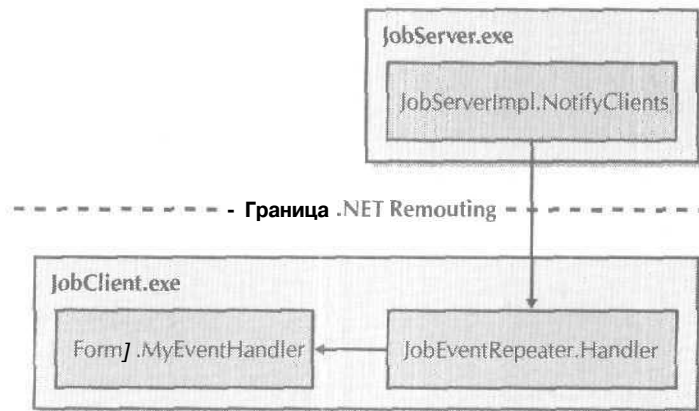


Рис. 3-8. Экземпляр класса *JobEventRepeater* служит связующим звеном между обработчиком события в классе *Form1* и *JobServerImpl.JobEvent*

Теперь при генерации сервером события *JobServerImpl.JobEvent* он вызывает обработчик в экземпляре класса *JobEventRepeater*. В свою очередь этот экземпляр *JobEventRepeater* генерирует свое сообщение *JobEvent*, которое повторяет обратный вызов для метода *Form1.MyJobEventHandler*.

Конечно, необходимо изменить и код, с помощью которого *Form1* прекращает подписку на событие *JobEvent*. Для этого следует заменить первоначальную строку кода в *Form1.OnClosed* на следующий код:

```
// Отмена подписки на JobEvent.
m_IJobServer.JobEvent -= new JobEventHandler(
```

```

        m_JobEventRepeater.Handler);
m_JobEventRepeater.JobEvent += new JobEventHandler(
    this.MyJobEventHandler);

```

Разработка класса-дублера, публикуемого вместо метаданных *JobServerImpl*

Иногда предоставление клиентскому приложению доступа к реализации удаленного объекта невозможно или нежелательно. В таких случаях можно создать так называемый *класс-дублер* (stand-in), который определяет тип удаленного объекта, но не содержит реализации.

В текущем варианте наше клиентское приложение зависит от метаданных типа *JobServerImpl*, так как оно создает новый экземпляр *JobServerImpl* в методе *GetJobServer* класса *Form1*. Клиентское приложение содержит ссылку на сборку *JobServerLib*, которая содержит не только определения типов *JobServerImpl* и *IJobServer*, но и реализацию типа *JobServerImpl*.

Ниже показан класс-дублер для *JobServerImpl*:

```

public class JobServerImpl : MarshalByRefObject, IJobServer
{
    public event JobEventHandler JobEvent;

    public JobServerImpl()
    { throw new System.NotImplementedException(); }

    private void NotifyClients(JobEventArgs args)
    { throw new System.NotImplementedException(); }

    public void CreateJob( string sDescription )
    { throw new System.NotImplementedException(); }

    public void UpdateJobState( int nJobID, string sUser,
                               string sStatus )
    { throw new System.NotImplementedException(); }

    public ArrayList GetJobs()
    { throw new System.NotImplementedException(); }
}

```

Название типа класса-дублера и его сборки должны совпадать с названиями фактических класса и сборки. Сборка, содержащая класс-дублер, используется клиентом, тогда как сервер исполь-

зует сборку, содержащую фактическую реализацию. В результате клиентское приложение получает метаданные, необходимые инфраструктуре .NET Remoting для создания прокси удаленного объекта, но не имеет доступа к фактической реализации этого объекта. Зависимости между приложениями и сборками *JobServerLib* показаны на рис. 3-9.

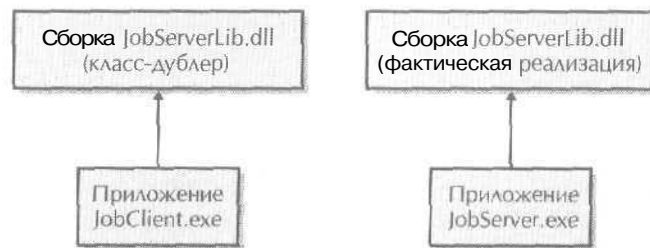


Рис. 3-9. *JobServerLib* и сборка-дублер

Удаленный доступ к интерфейсу *IJobServer*

Другой способ устранения зависимости клиента от реализации удаленного объекта — удаленное взаимодействие посредством интерфейса. Клиент взаимодействует с удаленным объектом через определение интерфейсного типа, а не с помощью определения фактического класса удаленного объекта. Для этого необходимо поместить определение интерфейса и все связанные с ним типы в сборку, которая будет передаваться клиентам. Реализация интерфейса удаленным объектом помещается в отдельную сборку, которая никогда не передается клиенту. Покажем это на примере интерфейса *IJobServer*.

Так как класс *JobServerImpl* реализует интерфейс *IJobServer*, мы можем передать клиенту сборку, содержащую только метаданные интерфейса *IJobServer*. Во-первых, необходимо переместить определение интерфейса *IJobServer*, а также определения типов *JobInfo*, *JobEventArgs*, *JobEvent* и *JobEventHandler* из сборки *JobServerLib* в другую сборку, которую мы назовем *JobLib*. Тогда *JobServerLib* будет содержать только метаданные класса *JobServerImpl* и, таким образом, зависеть от новой сборки *JobLib*. Новые зависимости изображены на рис. 3-10.

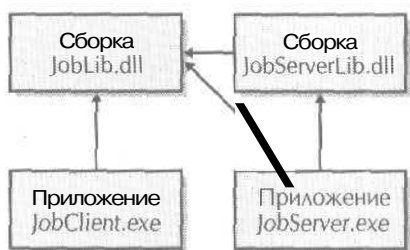


Рис. 3-10. Зависимости сборки JobLib

Теперь приложение *JobClient* зависит только от сборки *JobLib*, которая содержит интерфейс *IJobServer* и вспомогательные типы данных, но не содержит определения *JobServerImpl*. Приложению *JobClient* не требуются метаданные класса *JobServerImpl*, так как оно взаимодействует с этим классом посредством интерфейса *IJobServer*. Из-за того что создать экземпляр интерфейса невозможно, при подобной удаленной работе с интерфейсами вам потребуется метод *Activator.GetObject*. Он позволяет указать тип объекта, который следует активизировать в заданной URL-конечной точке.

Следующий фрагмент кода использует метод *Activator.GetObject* для получения интерфейса *IJobInterface* из конечной точки, указанной в конфигурационном файле:

```

WellKnownClientTypeEntry[] ClientEntries =
    RemotingConfiguration.
        GetRegisteredWellKnownClientTypes();

return (IJobServer)Activator.GetObject( typeof(IJobServer),
                                         ClientEntries[0].
                                         ObjectUrl );

```

Подразумевается, что в конфигурационном файле *JobClient.exe.config* содержится следующая запись:

```

<wellknown type="JobServerLib.IJobServer, JobLib"
  url="http://localhost:4000/JobURI" />

```

Заключение

В этой главе мы продемонстрировали создание простого распределенного приложения на основе .NET Remoting, рассмотрели все основные задачи, которые необходимо решить при создании любого распределенного приложения .NET Remoting. Теперь вы получили четкое представление о том, как использовать средства инфраструктуры .NET Remoting при разработке приложений. Остальные главы книги посвящены более изощренным средствам .NET Remoting, предназначенным для разработки специализированных прокси, каналов и форматировщиков.

SOAP и обмен сообщениями

К настоящему моменту у вас должно уже сложиться вполне законченное представление о том, как использовать .NET Remoting для разработки распределенных приложений. Чтобы дополнить ваши знания в этой области, в этой главе мы рассмотрим сообщения, которыми обмениваются объекты приложений JobClient и JobServer. Изучение формата сообщений даст вам представление о той информации, которую .NET Remoting пересылает между удаленными объектами. Но прежде чем перейти непосредственно к сообщениям, мы кратко познакомим вас с SOAP. Если вы уже с ним знакомы, можете сразу переходить к разделу «Обмен сообщениями».

Протокол SOAP

SOAP (Simple Object Access Protocol) лежит в основе большинства средств .NET для обеспечения открытости. Хотя разработчикам предоставлена возможность использовать более эффективные протоколы, ни один из них не обладает гибкостью SOAP. Так что же такое SOAP?

В общих словах, SOAP — это протокол на основе XML, определяющий механизм, посредством которого распределенные приложения могут обмениваться структурированной и типизированной информацией способом, не зависящим от платформы или языка программирования. Кроме того, SOAP определяет механизм удаленных вызовов процедур с помощью SOAP-сообщений. SOAP определяет формат конверта для пересылки XML-данных, простой адресации, а также представления данных и типов. Осо-

бенно интересна схема представления данных, так как она позволяет описывать типы независимо от платформы или языка программирования. Хотя многие считают, что транспортом для SOAP должен быть HTTP, теоретически SOAP может использовать любой транспорт, например SMTP (Simple Mail Transfer Protocol) или MSMQ (Microsoft Message Queuing). На самом деле сообщение SOAP можно сохранить в файле на дискете и отнести ее на сервер для обработки. Полное описание SOAP 1.1 вы найдете по адресу <http://www.w3.org/TR/SOAP/>.

Огромное влияние, которое оказал SOAP на развитие информационных технологий, объясняется рядом факторов.

- SOAP прост. Так как он использует известные и проверенные технологии, такие, как HTTP и XML, то и его реализация проста.
- SOAP повсеместно распространен, что следует из простоты реализации. На момент написания этой книги известно более 30 реализаций SOAP для различных языков и платформ.
- SOAP создан для Интернета. RFC-спецификация SOAP определяет HTTP-заголовки, необходимые для транспортировки SOAP-сообщений. Благодаря тесной интеграции с HTTP, сообщения SOAP могут проникать через брандмауэры и их легко поддерживать в уже существующих системах. SOAP также определяет и более простую схему работы на основе сообщений, однако мы поговорим здесь о мощном SOAP-RPC, так как именно он используется .NET Remoting.

Одно из самых больших достоинств SOAP одновременно считается и самым значительным его недостатком. Текстовая природа протокола позволяет легко читать сообщения SOAP и обеспечивает переносимость. Однако преобразование структур данных в многословные описания на основе тэгов требует временных затрат и увеличивает объем передаваемых по сети данных. Тем не менее дополнительное время, необходимое на обработку, весьма невелико и не должно представлять проблему в тех случаях, когда вашему приложению нужны те преимущества, которые предоставляет SOAP.

Нужно ли знать SOAP?

Сегодня большинство разработчиков не имеют дело с форматом SOAP напрямую. SOAP теперь настолько стандартизирован, что стал *составной* частью инфраструктур, поставщики которых предоставляют средства для работы с ним. В частности, такие средства предоставляет и .NET Framework. Используя .NET, разработчик может выбрать *SoapFormatter* путем конфигурирования при написании приложений .NET Remoting, писать Web-сервисы XML, которые основаны на SOAP, или применить класс *XmlSerializer* для сериализации классов программы в SOAP-сообщения.

Зачем же нам знать детали, связанные с SOAP, если .NET Framework стремится скрыть их? Одна из основных причин заключается в том, что таким образом удастся понять механизм взаимодействия приложений .NET Remoting. Помимо остальных своих преимуществ (таких, как поддержка брандмауэров и независимость от платформы), SOAP является текстовым, не шифрованным и, значит, читабельным. Настроив приложение .NET Remoting на использование *SoapFormatter*, можно изучить трафик между клиентом и сервером и многое узнать о том, как работает .NET Remoting.

RPC на основе HTTP

В простейшей своей форме спецификация SOAP определяет XML-схему для SOAP-сообщений. Однако спецификация SOAP идет дальше: она определяет HTTP-заголовки, позволяющие использовать HTTP в качестве транспорта. В этой книге в качестве SOAP-транспорта нас интересует только HTTP. Таким образом, мы начнем с примера стандартного HTTP-заголовка для SOAP-сообщения:

```
HTTP/1.1 /JobServer/JobURI POST
Content-Type: text/xml
SOAPAction: MyMethod
```

HTTP URI (Uniform Resource Identifier) задает конечную точку RPC — в данном случае */JobServer/JobURI*. Заголовок *Content-Type*, содержащий *text/xml*, за которым следует заголовок *SOAPAction*, определяет SOAP-заголовок. На основании значения *MyMethod* а заголовке *SOAPAction* можно предположить, что данный HTTP-

запрос содержит SOAP-сообщение, получатель которого вызовет метод *MyMethod* объекта, связанного с */JobServer/JobURL*. В разделе «Обмен сообщениями» вы увидите, что .NET Remoting использует заголовок *SOAPAction* для указания имени вызываемого метода.

RPC на основе SOAP применяет схему «запрос — ответ», Клиент посылает одно сообщение-запрос по определенному адресу, в ответ на которое сервер отвечает также одним сообщением-ответом. Данная схема вполне соответствует схеме «запрос — ответ», используемой HTTP.

Элементы сообщения SOAP

Базовой единицей обмена, определяемой SOAP, считается сообщение. Следующий шаблон показывает порядок следования и уровни вложенности элементов сообщения SOAP. Конкретный экземпляр данного шаблона следует непосредственно после HTTP-заголовка, который мы только что рассмотрели,

```
<SOAP-ENV: Envelope>
  <SOAP-ENV: Header>
    ... Информация заголовка (элемент Header может быть задан)
  </SOAP-ENV: Header>
  <SOAP-ENV: Body>
    ... Информация тела (элемент Body должен быть задан)
  </SOAP-ENV: Body>
</SOAP-ENV: Envelope>
```

Конверт SOAP

Элемент *<Envelope>* является обязательным корневым элементом любого SOAP-сообщения. Кроме того, каждый элемент *<Envelope>*:

- должен ссылаться на пространство имен SOAP *Envelope* (*xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"*);
- может содержать один элемент *<Header>*;
- должен содержать один и только один элемент *<Body>*.

Полностью элемент *<Envelope>*, генерируемый .NET Remoting, обычно выглядит так:

```
<SOAP-ENV:Envelope
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
```

```

xmlns:xsd="http://www.w3.org/2001/XMLSchema"
xmlns:SOAP-ENC="http://schemas.xmlsoap.org/soap/encoding/"
xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
xmlns:clr="http://schemas.microsoft.com/soap/encoding/clr/1.0"
SOAP-ENV:encodingStyle="http://schemas.xmlsoap.org/soap/
encoding/">

```

Задаваемые здесь идентификаторы пространств имен служат для задания областей видимости различных элементов SOAP-сообщения. Первые два атрибута задают удобные псевдонимы *xsi* и *xsd* пространствам имен XML *http://www.w3.org/2001/XMLSchema-instance* и *http://www.w3.org/2001/XMLSchema* соответственно. Атрибут *xmlns:SOAP-ENC* задает псевдоним *SOAP-ENC* для схемы конверта SOAP 1.1. Далее вы увидите, что большинство элементов сообщения SOAP используют псевдоним *SOAP-ENV* для элементов и атрибутов, определенных в элементе *<Envelope>* . Также обратите внимание на атрибут *encodingStyle* , который указывает, что данное сообщение следует правилам представления данных, описанных в разделе 5 спецификации SOAP 1.1.

Заголовок SOAP

Элемент *<Header>* , если он имеется, обязан следовать непосредственно за открывающим тегом *<Envelope>* и может использоваться не более одного раза. Записи заголовка являются дочерними элементами элемента *<Header>* и предоставляют механизм расширения сообщения SOAP, путем помещения в него информации, специфичной для приложения, которая способна влиять на обработку сообщения. Кроме того, запись заголовка может содержать атрибуты заголовка, влияющие на его интерпретацию. Спецификация SOAP 1.1 определяет два атрибута заголовка: *actor* и *mustUnderstand* . Следует иметь в виду, что сообщение SOAP по пути к месту назначения может путешествовать по цепочке обработчиков сообщений SOAP. Атрибут *actor* определяет, какие именно обработчики должны фактически обрабатывать данное сообщение, Атрибут *mustUnderstand* задает, обязан ли каждый получатель сообщения знать, каким образом интерпретируется данная запись заголовка. Если значением *mustUnderstand* является 1, а не 0, то получатель обязан понимать эту запись заголовка либо данное сообщение должно быть им отвергнуто. Ниже показан пример записи заголовка *MessageID* , содержащей атрибут *mustUnderstand* :

```
<SOAP-ENV:Header>
  <z:MessageID
    xmlns:a="My Namespace URI"
    SOAP-ENV:mustUnderstand="1">
    "2EE0E496-73B7-48b4-87A6-2CB2C8D9DBDE"
  </z:MessageID>
</SOAP-ENV:Header>
```

В данном примере для обработки сообщения получатель обязан понимать запись заголовка *MessageID*.

Тело SOAP

Внутри элемента *<Envelope>* может присутствовать один и только один элемент *<Body>*. Он содержит фактическую полезную информацию, отправляемую получателю. Именно здесь находятся данные приложения. В .NET Remoting элемент *<Body>* содержит вызовы методов с параметрами, включающими XML-варианты сложных типов данных, таких, как структуры. Правила сериализации массивов, структур и графов объектов определяются в разделе 5 спецификации 1.1; часто эти правила называют «*форматом раздела 5*» (Section 5 encoding). Ниже показан типичный пример элемента *<Body>* в вызове удаленной процедуры:

```
<SOAP-ENV:Body>
  <myns:GetPopulationOfState xmlns:myns="my-namespace-uri">
    <state>Florida</state>
  </myns:GetPopulationOfState>
</SOAP-ENV:Body>
```

Внутри элемента *<Body>* содержится элемент с именем метода, дочерние элементы которого содержат значения входных параметров метода. В данном примере дочерний элемент *<state>* указывает, что необходимо вернуть число жителей штата Флорида. Полезная информация ответного сообщения содержит результаты вызова. В ответ на приведенное выше сообщение его получатель может вернуть, например, такое сообщение:

```
<SOAP-ENV:Envelope
  xmlns:SOAP-ENV=http://schemas.xmlsoap.org/soap/envelope/
  SOAP-ENV:encodingStyle=
    "http://schemas.xmlsoap.org/soap/encoding/">
  <SOAP-ENV:Body>
    <myns:GetPopulationOfStateResponse
      xmlns:myns="my-namespace-uri">
```

```
<Population>15982378</Population>
</mys:GetLastTradePriceResponse>
</SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

Ошибки в SOAP

Спецификация SOAP определяет тег *<Fault>*, который должен быть дочерним элементом элемента *<Body>*, если на сервере произошла ошибка. Элемент *<Fault>* содержит описание причин ошибки. Конечно, предполагается, что теги *<Fault>* присутствуют только в ответных сообщениях.

Документ-литеральный SOAP

Прежде чем приступить к изучению примеров SOAP-сообщений для приложений, разработанных нами в главе 3, следует отметить, что существует и другая «форма» SOAP. Ранее была рассмотрена *RPC-форматированная (RPC/encoded)* форма SOAP, которую использует .NET Remoting. Другая форма SOAP, известная под названием *документ-литеральная (document/literal)*, используется ASP.NET и Web-сервисами XML.

Документ-литеральная форма SOAP не налагает ограничений на содержимое элемента *<Body>*. Данная схема является более гибкой, так как позволяет использовать любой формат содержимого, о котором договорились отправитель и получатель. При применении документ-литеральной формы сериализация данных выполняется на основе XML-схем, которые работают с данными, как с XML, а не как с объектами или структурами. Естественно, между приверженцами двух форм SOAP-сообщений идет «религиозная война». Мы воздержимся от участия в ней и сосредоточимся на RPC-форматированном SOAP, который позволит нам изучить обмен сообщениями с .NET Remoting.

ПРИМЕЧАНИЕ В спецификации SOAP ничего не говорится о том, что для RPC следует использовать форматированную сериализацию, а для обмена документами — литеральную. Однако практически все известные на сегодня реализации SOAP интерпретируют спецификацию именно таким образом.

Обмен сообщениями

Исследование сообщений, которыми обмениваются приложения `JobClient` и `JobServer` в различных точках взаимодействия друг с другом, может быть очень познавательным. Благодаря выбору канала HTTP, применяющего форматировщик SOAP по умолчанию, мы можем, воспользовавшись утилитой трассировки, просматривать сообщения в читабельной форме. Так как сообщения иногда довольно большие, мы попытаемся разбить некоторые из них на более понятные фрагменты.

При запуске приложение `JobClient` создает прокси для типа `JobServerImpl`. Так как `JobServerImpl` используется этим приложением как общеизвестный объект, то клиенту нет необходимости посылать серверу сообщения до тех пор, пока клиент не выполнит какого-либо обращения к экземпляру удаленного объекта, например, посредством обращения к свойству или вызова метода.

Сообщение-запрос `add JobEvent`

В первый раз клиент обращается к экземпляру удаленного объекта при подписке на событие `JobEvent`, в результате чего происходит следующий обмен сообщениями:

- `JobClient` посылает `JobServer` сообщение-запрос `add JobEvent`;
- `JobServer` посылает `JobClient` сообщение-ответ `add JobEvent`,

Компилятор генерирует в классе `JobServerImpl` методы `add JobEvent` и `remove JobEvent`, потому что этот класс содержит член-событие `JobEvent`. Сообщение-запрос состоит из сообщения-запроса HTTP POST, содержащего данные приложения, которые определяют сообщение SOAP.

HTTP-заголовок сообщения-запроса показан ниже:

```
POST /JobServer/JobURI HTTP/1.1
User-Agent: Mozilla/4.0+(compatible; MSIE 6.0;
Windows 5.0.2195.0; HS .NET Remoting; MS .NET CLR 1.0.3705.0 )
Content-Type: text/xml; charset="utf-8"
SOAPAction: "http://schemas.microsoft.com/clr/nsassem/
JobServerLib.IJobServer/JobServerLib#add_JobEvent"
Content-Length: 6833
Expect: 100-continue
```

Connection: Keep-Alive
Host: localhost

Хотя этого и не видно из показанного заголовка HTTP-запроса, приложение JobClient отправляет сообщение по URL конечной точки, который мы задали при настройке типа *JobServerImpl* как объекта с серверной активизацией. Обратите внимание, что в HTTP-запросе указан URL «*JobServer/JobURL*», что напрямую связано с URL, указанным нами при конфигурировании типа *JobServerImpl*, как типа с серверной активизацией. Инфраструктура .NET Remoting на стороне сервера использует данную информацию для передачи сообщения соответствующему экземпляру объекта.

Заголовок *SOAPAction* свидетельствует о том, что данный HTTP-запрос содержит в своем теле сообщение SOAP, относящееся к методу *addJobEvent*.

Ниже показан элемент *<Envelope>* из тела сообщения, одинаковый для всех сообщений;

```
<SOAP-ENV:Envelope
  xmlns:xsi=http://www.w3.org/2001/XMLSchema-instance
  xmlns:xsd=http://www.w3.org/2001/XMLSchema
  xmlns:SOAP-ENC=http://schemas.xmlsoap.org/soap/encoding/
  xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
  xmlns:clr=http://schemas.microsoft.com/soap/encoding/clr/1.0
  SOAP-ENV:encodingStyle="http://schemas.xmlsoap.org/soap/
    encoding/">
```

Далее следует часть *<Body>* сообщения SOAP:

```
<SOAP-ENV:Body>
  <i2:addJobEvent id="ref-1"
    xmlns:i2="http://schemas.microsoft.com/clr/nsassem/
      JobServerLib.JobServer/JobServerLib">
    <value href="#ref-3"/>
  </i2:addJobEvent>
```

Элемент *<i2:addJobEvent>* указывает, что данное сообщение SOAP содержит удаленный вызов метода *IJobServer.addJobEvent*. Внутри него находится элемент *<value>*, задающий значение параметра, передаваемого методу. Далее показано определение элемента *<a1:DelegateSerializationHolder>*, на который ссылается элемент *<value>*:

```
<a1:DelegateSerializationHolder id="ref-3"
  xmlns:a1="http://schemas.microsoft.com/clr/ns/System">
  <Delegate href="#ref-4"/>
  <target0 href="#ref-5"/>
</a1:DelegateSerializationHolder>
```

Элемент `<a1:DelegateSerializationHolder>` представляет собой сериализованный экземпляр класса `System.DelegateSerializationHolder`. Сериализованный экземпляр содержит два члена, одним из которых является элемент `<Delegate>`, ссылающийся на следующий элемент:

```
<a1:DelegateSerializationHolder_x002B_DelegateEntry id="ref-4"
  xmlns:a1="http://schemas.microsoft.com/clr/ns/System">
  <type id="ref-6">JobServerLib.JobEventHandler</type>
  <assembly id="ref-7">JobServerLib, Version=1.0.807.36861,
    Culture=neutral, PublicKeyToken=null</assembly>
  <target id="ref-8" xsi:type="SOAP-ENC:string">target0
  </target><targetTypeAssembly
    id="ref-9">JobClient, Version=1.0.807.36865,
    Culture=neutral, PublicKeyToken=null</targetTypeAssembly>
  <targetTypeName id="ref-10">JobClient.Form1</targetTypeName>
  <methodName id="ref-11">MyJobEventHandler</methodName>
  <delegateEntryxsi:null="1"/>
</a1:DelegateSerializationHolder_x002B_DelegateEntry>
```

Данный элемент определяет тип делегата и информацию сборки — в данном случае `JobServerLib.JobEventHandler` в сборке `JobServerLib`. Элемент также определяет информацию целевого типа: метод `MyJobEventHandler` в классе `JobClient.Form1`, который определен в сборке `JobClient`.

Вторым членом `<a1:DelegateSerializationHolder>` является элемент `<target0>`, задающий экземпляр целевого объекта делегата:

```
<a2:ObjRef id="ref-5" xmlns:a2="http://schemas.microsoft.com/
  clr/ns/System.Runtime.Remoting">
  <uri id="ref-12">
    /16e04fb4_ad6d_47ec_b584_22624cf6c808/96410280_1.rem
  </uri>
  <objrefFlags>0</objrefFlags>
  <typeInfo href="#ref-13"/>
  <envoyInfo xsi:null="1"/>
  <channelInfo href="#ref-14"/>
</a2:ObjRef>
```

Обратите внимание, что элемент `<target0>` ссылается на элемент `<a2:ObjRef>`. Элемент `<a2:ObjRef>` — это сериализованный экземпляр `System.ObjRef`, представляющий экземпляр целевого объекта делегата. Как вы помните из главы 2, `ObjRef` содержит информацию о каждом типе в иерархии наследования типа, производного от `MarshalByRefObject`. Так как `JobClient.Form1` имеет тип `Windows.Forms.Form`, то сериализованный `ObjRef` описывает множество интерфейсов и конкретных типов. В остальной части сообщения определяются все элементы, содержащиеся в элементе `<a2:ObjRef>`. Сюда входит URI экземпляра сериализованного объекта; идентификатор контекста, идентификатор домена приложения и идентификатор процесса для `MarshalByRefObject`; а также информация о канале: тип транспорта (HTTP), IP адрес и порты:

```
<a2:TypeInfo id="ref-13" xmlns:a2="http://schemas.microsoft.com/
clr/ns/System.Runtime.Remoting">
  <serverType id="ref-15">JobClient.Form1, JobClient,
    Version=1.0.807.36865, Culture=neutral, PublicKeyToken=null
  </serverType>
  <serverHierarchy href="#ref-16"/>
  <interfacesImplemented href="#ref-17"/>
</a2:TypeInfo>

<a2:ChannelInfo id="ref-14" xmlns:a2=
"http://schemas.microsoft.com/clr/ns/System.Runtime.Remoting">
  <channelData href="#ref-18"/>
</a2:ChannelInfo>
```

Следующий элемент — это массив строк, который представляет всю иерархию наследования вплоть до `System.MarshalByRefObject`, но не включая его:

```
<SOAP-ENC:Array id="ref-16" SOAP-ENC:arrayType="xsd:string[5]">
<item id="ref-19">System.Windows.Forms.Form,
  System.Windows.Forms,
  Version=1.0.3300.0, Culture=neutral,
  PublicKeyToken=b77a5c561934e089
</item>
<item id="ref-20">System.Windows.Forms.ContainerControl,
System.Windows.Forms, Version=1.0.3300.0, Culture=neutral,
PublicKeyToken=b77a5c561934e089</item>
<item id="ref-21">System.Windows.Forms.ScrollableControl,
System.Windows.Forms, Version=1.0.3300.0, Culture=neutral,
```

```

PublicKeyToken=b77a5c561934e089</item>
<item id="ref-22">System.Windows.Forms.Control,
    System.Windows.Forms,
    Version=1.0.3300.0, Culture=neutral,
    PublicKeyToken=b77a5c561934e089
</item>
<item id="ref-23">System.ComponentModel.Component, System,
    Version=1.0.3300.0, Culture=neutral,
    PublicKeyToken=b77a5c561934e089
</item>
</SOAP-ENC:Array>

```

Следующий элемент — массив строк, содержащий все интерфейсы, реализованные данным сериализуемым типом:

```

<SOAP-ENC:Array id="ref-17" SOAP-ENC:arrayType="xsd:string[18]">
<item id="ref-24">System.ComponentModel.IComponent, System,
    Version=1.0.3300.0, Culture=neutral,
    PublicKeyToken=b77a5c561934e089
</item>
<item id="ref-25">System.IDisposable, mscorlib,
    Version=1.0.3300.0,
    Culture=neutral, PublicKeyToken=b77a5c561934e089</item>
<item id="ref-26">
    System.Windows.Forms.UnsafeNativeMethods+IOleControl,
    System.Windows.Forms, Version=1.0.3300.0, Culture=neutral,
    PublicKeyToken=b77a5c561934e089</item>
<item id="ref-27">
    System.Windows.Forms.UnsafeNativeMethods+IOleObject,
    System.Windows.Forms, Version=1.0.3300.0, Culture=neutral,
    PublicKeyToken=b77a5c561934e089</item>
<item id="ref-28">
    System.Windows.Forms.UnsafeNativeMethods+IOleInPlaceObject,
    System.Windows.Forms, Version=1.0.3300.0, Culture=neutral,
    PublicKeyToken=b77a5c561934e089</item>
<item id="ref-29">
    System.Windows.Forms.UnsafeNativeMethods+
        IOleInPlaceActiveObject,
    System.Windows.Forms, Version=1.0.3300.0, Culture=neutral,
    PublicKeyToken=b77a5c561934e089</item>
<item id="ref-30">
    System.Windows.Forms.UnsafeNativeMethods+IOleWindow,
    System.Windows.Forms, Version=1.0.3300.0, Culture=neutral,
    PublicKeyToken=b77a5c561934e089</item>
<item id="ref-31">
    System.Windows.Forms.UnsafeNativeMethods+IViewObject,
    System.Windows.Forms, Version=1.0.3300.0, Culture=neutral,

```

```

PublicKeyToken=b77a5c561934e089</item>
<item id="ref-32">
System.Windows.Forms.UnsafeNativeMethods+IViewObject2,
System.Windows.Forms, Version=1.0.3300.0, Culture=neutral,
PublicKeyToken=b77a5c561934e089</item>
<item id="ref-33">System.Windows.Forms.UnsafeNativeMethods+
    IPersist,
System.Windows.Forms, Version=1.0.3300.0, Culture=neutral,
PublicKeyToken=b77a5c561934e089</item>
<item id="ref-34">
System.Windows.Forms.UnsafeNativeMethods+IPersistStreamInit,
System.Windows.Forms, Version=1.0.3300.0, Culture=neutral,
PublicKeyToken=b77a5c561934e089</item>
<item id="ref-35">
System.Windows.Forms.UnsafeNativeMethods+IPersistPropertyBag,
System.Windows.Forms, Version=1.0.3300.0, Culture=neutral,
PublicKeyToken=b77a5c561934e089</item>
<item id="ref-36">
System.Windows.Forms.UnsafeNativeMethods+IPersistStorage,
System.Windows.Forms, Version=1.0.3300.0, Culture=neutral,
PublicKeyToken=b77a5c561934e089</item>
<item id="ref-37">
System.Windows.Forms.UnsafeNativeMethods+IQuickActivate,
System.Windows.Forms, Version=1.0.3300.0, Culture=neutral,
PublicKeyToken=b77a5c561934e089</item>
<item id="ref-38">System.ComponentModel.ISynchronizeInvoke,
    System,
Version=1.0.3300.0, Culture=neutral,
    PublicKeyToken=b77a5c561934e089
</item>
<item id="ref-39">System.Windows.Forms.IWin32Window,
System.Windows.Forms, Version=1.0.3300.0, Culture=neutral,
PublicKeyToken=b77a5c561934e089</item>
<item id="ref-40">System.Windows.Forms.IContainerControl,
System.Windows.Forms, Version=1.0.3300.0, Culture=neutral,
PublicKeyToken=b77a5c561934e089</item>
<item id="ref-41">System.Runtime.Remoting.Lifetime.ISponsor,
mscorlib, Version=1.0.3300.0, Culture=neutral,
PublicKeyToken=b77a5c561934e089</item>
</SOAP-ENC:Array>

```

И, наконец, сообщение содержит информацию о домене, контексте и процессе вызывающего приложения, а также информацию о каналах, публикуемых доменом приложения отправителя. Эти сведения позволяют получателю сообщения при необходимости установить канал обмена данными:

```

<SOAP-ENC:Array id="ref-18" SOAP-ENC:arrayType="xsd:anyType[2]">
  <item href="#ref-42"/>
  <item href="#ref-43"/>
</SOAP-ENC:Array>

<a3:CrossAppDomainData id="ref-42"
  xmlns:a3="http://schemas.microsoft.com/clr/ns/
  System.Runtime.Remoting.Channels">
  <_ContextID>1300872</_ContextID>
  <_DomainID>1</_DomainID>
  <_processGuid id="ref-44">
    20c23b9b_4d09_46a8_bc29_10037f04f46f
  </_processGuid>
</a3:CrossAppDomainData>

<a3:ChannelDataStore id="ref-43"
  xmlns:a3="http://schemas.microsoft.com/clr/ns/
  System.Runtime.Remoting.Channels">
  <_channelURIs href="#ref-45"/>
  <_extraData xsi:null="1"/>
</a3:ChannelDataStore>

<SOAP-ENC:Array id="ref-45" SOAP-ENC:arrayType="xsd:string[1]">
  <item id="ref-46">http://66.156.56.215:1958</item>
</SOAP-ENC:Array>

</SOAP-ENV:Body>
</SOAP-ENV:Envelope>

```

Сообщение-ответ *add_JobEvent*

По завершении работы метода *add_Delegate* инфраструктура .NET Remoting на стороне сервера упакует результат вызова метода в *сообщение-ответ* и вернет его вызывающему приложению. Ниже показано ответное HTTP-сообщение для метода *add_Delegate*:

```

HTTP/1.1 200 OK
Content-Type: text/xml; charset="utf-8"
Server: MS .NET Remoting, MS .NET CLR 1.0.3705.0
Content-Length: 580

```

```

<SOAP-ENV:Envelope ...>
  <SOAP-ENV:Body>
    <i2:add_JobEventResponse id="ref-1"
      xmlns:i2="http://schemas.microsoft.com/clr/nsassem/

```

```

        JobServerLib.IJobServer/JobServerLib">
    </i2:add_JobEventResponse>
</SOAP-ENV:Body>
</SOAP-ENV:Envelope>

```

Сообщение-запрос *GetJobs*

Когда клиент вызывает метод *IJobServer.GetJobs* удаленного экземпляра *IJobServer.GetJobs*, инфраструктура .NET Remoting отправляет серверу сообщение-запрос **GetJobs**:

```

POST /JobServer/JobURI HTTP/1.1
User-Agent: Mozilla/4.0+(compatible; MSIE 6.0;
Windows 5.0.2195.0; MS .NET Remoting; MS .NET CLR 1.0.3705.0 )
Content-Type: text/xml; charset="utf-8"
SOAPAction: "http://schemas.microsoft.com/clr/nsassem/
JobServerLib.IJobServer/JobServerLib#GetJobs"
Content-Length: 554
Expect: 100-continue
Host: localhost

```

```

<SOAP-ENV:Envelope >
  <SOAP-ENV:Body>
    <i2:GetJobs id="ref-1" xmlns:i2=
      "http://schemas.microsoft.com/
      clr/nsassem/JobServerLib.IJobServer/JobServerLib">
    </i2:GetJobs>
  </SOAP-ENV:Body>
</SOAP-ENV:Envelope>

```

Сообщение-ответ *GetJobs*

В ответ на сообщение-запрос *GetJobs* сервер возвращает клиенту сообщение-ответ *GetJobs*. Это сообщение содержит сериализованные результаты: массив *ArrayList*, содержащий экземпляры *JobInfo* для всех, определенных в настоящий момент заданий:

```

HTTP/1.1 200 OK
Content-Type: text/xml; charset="utf-8"
Server: MS .NET Remoting, MS .NET CLR 1.0.3705.0
Content-Length: 1991

```

```

<SOAP-ENV:Envelope ...>

  <SOAP-ENV:Body>

    <i2:GetJobsResponse id="ref-1"

```

```

xmlns:i2="http://schemas.microsoft.com/clr/nsassem/
JobServerLib.JobServer/JobServerLib">
  <return href="#ref-3"/>
</i2:GetJobsResponse>

<a1:ArrayList id="ref-3"
xmlns:a1="http://schemas.microsoft.com/clr/ns/
System.Collections">
  <_items href="#ref-4"/>
  <_size>3</_size>
  <_version>6</_version>
</a1:ArrayList>

<SOAP-ENC:Array id="ref-4"
SOAP-ENC:arrayType="xsd:anyType[16]">

  <item xsi:type="a3:JobInfo"
xmlns:a3="http://schemas.microsoft.com/clr/nsassem/
JobServerLib/JobServerLib%2C%20
Version%3D1.0.819.24637%2C%20Culture%3Dneutral%2C%20
PublicKeyToken%3Dnull">
    <m_nID>0</m_nID>
    <m_sDescription id="ref-6">Wash Windows
      </m_sDescription>
    <m_sAssignedUser id="ref-7">
      Administrator</m_sAssignedUser>
    <m_sStatus id="ref-8">Assigned</m_sStatus>
  </item>

  <item xsi:type="a3:JobInfo"
xmlns:a3="http://schemas.microsoft.com/clr/nsassem/
JobServerLib/JobServerLib%2C%20
Version%3D1.0.819.24637%2C%20Culture%3Dneutral%2C%20
PublicKeyToken%3Dnull">
    <m_nID>1</m_nID>
    <m_sDescription id="ref-9">Fix door</m_sDescription>
    <m_sAssignedUser id="ref-10">
      Administrator</m_sAssignedUser>
    <m_sStatus id="ref-11">Assigned</m_sStatus>
  </item>

  <item xsi:type="a3:JobInfo"
xmlns:a3="http://schemas.microsoft.com/clr/nsassem/
JobServerLib/JobServerLib%2C%20
Version%3D1.0.819.24637%2C%20Culture%3Dneutral%2C%20
PublicKeyToken%3Dnull">

```

```

        <m_nID>2</m_nID>
        <m_sDescription id="ref-12">
          Clean carpets</m_sDescription>
        <m_sAssignedUser id="ref-13">
          Administrator</m_sAssignedUser>
        <m_sStatus id="ref-14">Completed</m_sStatus>
      </item>

    </SOAP-ENC:Array>
  </SOAP-ENV:Body>
</SOAP-ENV:Envelope>

```

Сообщение-запрос *CreateJob*

Клиентское приложение посылает серверу сообщение-запрос *CreateJob* при вызове клиентом метода *IJobServer.CreateJob* удаленного экземпляра *JobServerImpl*. Метод *IJobServer.CreateJob* имеет один параметр — описание нового задания. Следующее сообщение создает задание с описанием «Wash Windows»¹:

```

POST /JobServer/JobURI HTTP/1.1
User-Agent: Mozilla/4.0+(compatible; MSIE 6.0;
Windows 5.0.2195.0; MS .NET Remoting; MS .NET CLR 1.0.3705.0 )
Content-Type: text/xml; charset="utf-8"
SOAPAction: "http://schemas.microsoft.com/clr/nsassem/
  JobServerLib.IJobServer/JobServerLib#CreateJob"
Content-Length: 612
Expect: 100-continue
Host: localhost

```

```

<SOAP-ENV:Envelope ...>
  <SOAP-ENV:Body>
    <i2:CreateJob id="ref-1"
      xmlns:i2="http://schemas.microsoft.com/clr/nsassem/
        JobServerLib.IJobServer/JobServerLib">
      <sDescription id="ref-3">Wash Windows</sDescription>
    </i2:CreateJob>
  </SOAP-ENV:Body>
</SOAP-ENV:Envelope>

```

Сообщение-ответ *CreateJob*

Так как метод *CreateJob* не возвращает значения и не имеет выходных параметров, то сообщение-ответ *CreateJob* по сути дела

¹ «Вымыть окна». — Прим. перев.

представляет собой подтверждение завершения обработки вызова метода:

```
HTTP/1.1 200 OK
Content-Type: text/xml; charset="utf-8"
Server: MS .NET Remoting, MS .NET CLR 1.0.3705.0
Content-Length: 574

<SOAP-ENV:Envelope ...>
  <SOAP-ENV:Body>
    <i2:CreateJobResponse id="ref-1"
      xmlns:i2="http://schemas.microsoft.com/clr/nsassem/
        JobServerLib.IJobServer/JobServerLib">
      </i2:CreateJobResponse>
    </SOAP-ENV:Body>
  </SOAP-ENV:Envelope>
```

Сообщение-запрос *UpdateJobState*

Следующее сообщение генерируется, когда приложение JobClient вызывает метод *UpdateJobState*, передавая в качестве параметров значения 2, Administrator и Completed:

```
POST /JobServer/JobURI HTTP/1.1
User-Agent: Mozilla/4.0+{compatible; MSIE 6.0;
  Windows 5.0.2195.0; MS .NET Remoting; MS .NET CLR 1.0.3705.0 )
Content-Type: text/xml; charset="utf-8"
SOAPAction: "http://schemas.microsoft.com/clr/nsassem/
  JobServerLib.IJobServer/JobServerLib#UpdateJobState"
Content-Length: 670
Expect: 100-continue
Host: localhost

<SOAP-ENV:Envelope ...>
  <SOAP-ENV:Body>

    <i2:UpdateJobState id="ref-1"
      xmlns:i2="http://schemas.microsoft.com/clr/nsassem/
        JobServerLib.IJobServer/JobServerLib">
      <nJobID>2</nJobID>
      <sUser id="ref-3">Administrator</sUser>
      <sStatus id="ref-4">Completed</sStatus>
    </i2:UpdateJobState>

  </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

Сообщение-ответ *UpdateJobState*

Как и сообщение-ответ *CreateJob* сообщение-ответ *UpdateJobState* является просто подтверждением завершения выполнения вызова метода:

```
HTTP/1.1 200 OK
Content-Type: text/xml; charset="utf-8"
Server: MS .NET Remoting, MS .NET CLR 1.0.3705.0
Content-Length: 584

<SOAP-ENV:Envelope ...>
  <SOAP-ENV:Body>
    <i2:UpdateJobStateResponse id="ref-1"
      xmlns:i2="http://schemas.microsoft.com/clr/nsassem/
        JobServerLib.IJobServer/JobServerLib">
      </i2:UpdateJobStateResponse>
    </SOAP-ENV:Body>
  </SOAP-ENV:Envelope>
```

Сообщение-запрос на активизацию *JobNotes*

Когда приложение *JobClient* создает экземпляр класса *JobNotes*, инфраструктура .NET Remoting посылает приложению *JobServer* сообщение-запрос *Activate*:

```
POST /RemoteActivationService.rem HTTP/1.1
User-Agent: Mozilla/4.0+(compatible; MSIE 6.0;
Windows 5.0.2195.0; MS .NET Remoting; MS .NET CLR 1.0.3705.0 )
Content-Type: text/xml; charset="utf-8"
SOAPAction: "http://schemas.microsoft.com/clr/ns/
  System.Runtime.Remoting.Activation.IActivator#Activate"
Content-Length: 2126
Expect: 100-continue
Host: localhost

<SOAP-ENV:Envelope ...>
  <SOAP-ENV:Body>
    <i2:Activate id="ref-1"
      xmlns:i2="http://schemas.microsoft.com/clr/ns/
        System.Runtime.Remoting.Activation.IActivator">
      <msg href="#ref-3"/>
    </i2:Activate>

    <a1:ConstructionCall id="ref-3"
```

```

xmlns:a1="http://schemas.microsoft.com/clr/ns/
System.Runtime.Remoting.Messaging">
  <__Uri xsi:type="xsd:anyType" xsi:null="1"/>
  <__MethodName id="ref-4">.ctor</__MethodName>
  <__MethodSignature href="#ref-5"/>
  <__TypeName id="ref-6">JobServerLib.JobNotes.
    JobServerLib,
    Version=1.0.819.24637, Culture=neutral,
    PublicKeyToken=null</__TypeName>
  <__Args href="#ref-7"/>
  <__CallContext xsi:type="xsd:anyType" xsi:null="1"/>
  <__CallSiteActivationAttributes xsi:type="xsd:anyType"
    xsi:null="1"/>
  <__ActivationType xsi:type="xsd:anyType" xsi:null="1"/>
  <__ContextProperties href="#ref-8"/>
  <__Activator href="#ref-9"/>
  <__ActivationTypeName href="#ref-6"/>
</a1:ConstructionCall>

<SOAP-ENC:Array id="ref-5" SOAP-ENC:arrayType="a2:Type[0]"
  xmlns:a2="http://schemas.microsoft.com/clr/ns/System">
</SOAP-ENC:Array>
<SOAP-ENC:Array id="ref-7"
  SOAP-ENC:arrayType="xsd:anyType[0]">
</SOAP-ENC:Array>
<a3:ArrayList id="ref-8"
  xmlns:a3="http://schemas.microsoft.com/clr/ns/
System.Collections">
  <_items href="#ref-10"/>
  <_size>0</_size>
  <_version>0</_version>
</a3:ArrayList>

<a4:ContextLevelActivator id="ref-9"
  xmlns:a4="http://schemas.microsoft.com/clr/ns/

System.Runtime.Remoting.Activation">
  <m_NextActivator href="#ref-11"/>
</a4:ContextLevelActivator>
<SOAP-ENC:Array id="ref-10"
  SOAP-ENC:arrayType="xsd:anyType[16]">
</SOAP-ENC:Array>
<a4:ConstructionLevelActivator id="ref-11"
  xmlns:a4="http://schemas.microsoft.com/clr/ns/
System.Runtime.Remoting.Activation">
</a4:ConstructionLevelActivator>

```

```
</SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

Сообщение-ответ на активизацию *JobNotes*

После активизации экземпляра **JobNotes** инфраструктура .NET Remoting посылает приложению JobClient сообщение *Activate-Response*. Оно содержит сериализованный объект *System.ObjRef*, который представляет новый экземпляр *JobNotes*, располагающийся в домене приложения JobServer:

```
HTTP/1.1 200 OK
Content-Type: text/xml; charset="utf-8"
Server: MS .NET Remoting, MS .NET CLR 1.0.3705.0
Content-Length: 2723

<SOAP-ENV:Envelope ...>
  <SOAP-ENV:Body>
    <i2:ActivateResponse id="ref-1"
      xmlns:i2="http://schemas.microsoft.com/clr/ns/
        System.Runtime.Remoting.Activation.IActivator">
      <return href="#ref-3"/>
    </i2:ActivateResponse>
    <a1:ConstructionResponse id="ref-3"
      xmlns:a1="http://schemas.microsoft.com/clr/ns/
        System.Runtime.Remoting.Messaging">
      <__Uri xsi:type="xsd:anyType" xsi:null="1"/>
      <__MethodName id="ref-4">.ctor</__MethodName>
      <__TypeName id="ref-5">JobServerLib.JobNotes,
        JobServerLib,
        Version=1.0.830.37588, Culture=neutral,
        PublicKeyToken=null</__TypeName>
      <__Return href="#ref-6"/>
      <__OutArgs href="#ref-7"/>
      <__CallContext xsi:type="xsd:anyType" xsi:null="1"/>
    </a1:ConstructionResponse>
    <a3:ObjRef id="ref-6"
      xmlns:a3="http://schemas.microsoft.com/clr/ns/
        System.Runtime.Remoting">
      <uri id="ref-8">
        /cf2825b9_2974_4f5c_9810_2f96945b529d/
        16921141_1.rem</uri>
      <objrefFlags>0</objrefFlags>
      <typeInfo href="#ref-9"/>
      <envoyInfo xsi:null="1"/>
      <channelInfo href="#ref-10"/>
    </a3:ObjRef>
  </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

```

    <fIsMarshaled>0</fIsMarshaled>
  </a3:ObjRef>
  <SOAP-ENC:Array id="ref-7"
    SOAP-ENC:arrayType="xsd:anyType[0]">
  </SOAP-ENC:Array>
  <a3:TypeInfo id="ref-9"
    xmlns:a3="http://schemas.microsoft.com/clr/ns/
    System.Runtime.Remoting">
    <serverType id="ref-11">JobServerLib.JobNotes,
      JobServerLib,
      Version=1.0.830.37588, Culture=neutral,
      PublicKeyToken=null</serverType>
    <serverHierarchy xsi:null="1"/>
    <interfacesImplemented xsi:null="1"/>
  </a3:TypeInfo>
  <a3:ChannelInfo id="ref-10"
    xmlns:a3="http://schemas.microsoft.com/clr/ns/
    System.Runtime.Remoting">
    <channelData href="#ref-12"/>
  </a3:ChannelInfo>
  <SOAP-ENC:Array id="ref-12"
    SOAP-ENC:arrayType="xsd:anyType[2]">
    <item href="#ref-13"/>
    <item href="#ref-14"/>
  </SOAP-ENC:Array>
  <a4:CrossAppDomainData id="ref-13"
    xmlns:a4="http://schemas.microsoft.com/clr/ns/
    System.Runtime.Remoting.Channels">
    <_ContextID>1299232</_ContextID>
    <_DomainID>1</_DomainID>
    <_processGuid id="ref-15">
      efbc85bf_b165_4953_ab00_f37d49bbffb4</_processGuid>
  </a4:CrossAppDomainData>
  <a4:ChannelDataStore id="ref-14"
    xmlns:a4="http://schemas.microsoft.com/clr/ns/
    System.Runtime.Remoting.Channels">
    <_channelURIs href="#ref-16"/>
    <_extraData xsi:null="1"/>
  </a4:ChannelDataStore>
  <SOAP-ENC:Array id="ref-16"
    SOAP-ENC:arrayType="xsd:string[1]">
    <item id="ref-17">http://66.156.71.188:4000</item>
  </SOAP-ENC:Array>
</SOAP-ENV:Body>
</SOAP-ENV:Envelope>

```

Сообщение-запрос *remove_JobEvent*

Последним взаимодействием между приложением *JobClient* и удаленным экземпляром *JobServerImpl* является отмена подписки экземпляра *JobClient.Form1* на сообщение *JobEvent*. В результате вызывается метод *remove_JobEvent*, который инфраструктура .NET Remoting преобразует в сообщение-запрос *remove_JobEvent*. Фактически данное сообщение идентично сообщению-запросу *add_JobEvent*, так как оба метода имеют одинаковые сигнатуры. Как и в случае сообщения-запроса *add_JobEvent*, сообщение *remove_JobEvent* содержит сериализованный экземпляр делегата. В данном случае целью делегата является экземпляр типа *JobClient.Form1*, производного от *MarshalByRefObject*. Следовательно, сообщение содержит сериализованный экземпляр *System.ObjectRef*, представляющий экземпляр класса *JobClient.Form1*, который отменяет подписку на сообщение. Сообщение-запрос *remove_JobEvent* показано далее:

```
POST /JobServer/JobURI HTTP/1.1
User-Agent: Mozilla/4.0+(compatible; MSIE 6.0;
Windows 5.0.2195.0; MS .NET Remoting; MS .NET CLR 1.0.3705.0 )
Content-Type: text/xml; charset="utf-8"
SOAPAction: "http://schemas.microsoft.com/clr/nsassem/
JobServerLib.IJobServer/JobServerLib#remove_JobEvent"
Content-Length: 6839
Expect: 100-continue
Host: localhost

<SOAP-ENV:Envelope ...>
  <SOAP-ENV:Body>
    <i2:remove_JobEvent id="ref-1"
      xmlns:i2="http://schemas.microsoft.com/clr/nsassem/
JobServerLib.IJobServer/JobServerLib">
      <value href="#ref-3"/>
    </i2:remove_JobEvent>
    <a1:DelegateSerializationHolder id="ref-3"
      xmlns:a1="http://schemas.microsoft.com/clr/ns/System">
      <Delegate href="#ref-4"/>
      <targetO href="#ref-5"/>
    </a1:DelegateSerializationHolder>
    <a1:DelegateSerializationHolder_x002B_DelegateEntry
      id="ref-4"
      xmlns:a1="http://schemas.microsoft.com/clr/ns/System">
      <type id="ref-6">JobServerLib.JobEventHandler</type>
```

```

<assembly id="ref-7">JobServerLib,
    Version=1.0.819.24637,
    Culture=neutral, PublicKeyToken=null</assembly>
<target id="ref-8" xsi:type="SOAP-ENC:string">
    target0</target>
<targetTypeAssembly id="ref-9">JobClient,
    Version=1.0.829.36775, Culture=neutral,
    PublicKeyToken=null</targetTypeAssembly>
<targetTypeName id="ref-10">JobClient.Form1
</targetTypeName>
<methodName id="ref-11">MyJobEventHandler</methodName>
<delegateEntry xsi:null="1"/>
</a1:DelegateSerializationHolder_x002B_DelegateEntry>

<a2:ObjRef id="ref-5"
xmlns:a2="http://schemas.microsoft.com/clr/ns/
System.Runtime.Remoting">
    <uri id="ref-12">
        /295e2d43_876a_4511_a774_12e7a65d96bc/
        13636498_1.rem</uri>
    <objrefFlags>0</objrefFlags>
    <typeInfo href="#ref-13"/>
    <envoyInfo xsi:null="1"/>
    <channelInfo href="#ref-14"/>
</a2:ObjRef>
<a2:TypeInfo id="ref-13"
xmlns:a2="http://schemas.microsoft.com/clr/ns/
System.Runtime.Remoting">
<serverType id="ref-15">JobClient.Form1, JobClient,
    Version=1.0.829.36775, Culture=neutral,
    PublicKeyToken=null</serverType>
    <serverHierarchy href="#ref-16"/>
    <interfacesImplemented href="#ref-17"/>
</a2:TypeInfo>

<a2:ChannelInfo id="ref-14"
xmlns:a2="http://schemas.microsoft.com/clr/ns/
System.Runtime.Remoting">
    <channelData href="#ref-18"/>
</a2:ChannelInfo>

<SOAP-ENC:Array id="ref-16"
    SOAP-ENC:arrayType="xsd:string[5]">
    <item id="ref-19">System.Windows.Forms.Form,
        System.Windows.Forms, Version=1.0.3300.0,
        Culture=neutral,

```

```

        PublicKeyToken=b77a5c561934e089</item>
<item id="ref-20">System.Windows.Forms.
        ContainerControl,
        System.Windows.Forms, Version=1.0.3300.0,
        Culture=neutral,
        PublicKeyToken=b77a5c561934e089</item>
<item id="ref-21">System.Windows.Forms.
        ScrollableControl,
        System.Windows.Forms, Version=1.0.3300.0,
        Culture=neutral,
        PublicKeyToken=b77a5c561934e089</item>
<item id="ref-22">System.Windows.Forms.Control,
        System.Windows.Forms, Version=1.0.3300.0,
        Culture=neutral,
        PublicKeyToken=b77a5c561934e089</item>
<item id="ref-23">System.ComponentModel.Component,
        System,
        Version=1.0.3300.0, Culture=neutral,
        PublicKeyToken=b77a5c561934e089</item>
</SOAP-ENC:Array>

<SOAP-ENC:Array id="ref-17"
SOAP-ENC:arrayType="xsd:string[18]">
<item id="ref-24">System.ComponentModel.IComponent,
        System,
        Version=1.0.3300.0, Culture=neutral,
        PublicKeyToken=b77a5c561934e089</item>
<item id="ref-25">System.IDisposable, mscorlib,
        Version=1.0.3300.0, Culture=neutral,
        PublicKeyToken=b77a5c561934e089</item>
<item id="ref-26">
        System.Windows.Forms.UnsafeNativeMethods+
        IOleControl,
        System.Windows.Forms, Version=1.0.3300.0,
        Culture=neutral, PublicKeyToken=b77a5c561934e089
        </item>
<item id="ref-27">
        System.Windows.Forms.UnsafeNativeMethods+IOleObject,
        System.Windows.Forms, Version=1.0.3300.0,
        Culture=neutral, PublicKeyToken=b77a5c561934e089
        </item>
<item id="ref-28">System.Windows.Forms.
        UnsafeNativeMethods+IOleInPlaceObject,
        System.Windows.Forms, Version=1.0.3300.0,
        Culture=neutral, PublicKeyToken=b77a5c561934e089

```

```

    </item>
<item id="ref-29">System.Windows.Forms.
  UnsafeNativeMethods+IOleInPlaceActiveObject,
  System.Windows.Forms, Version=1.0.3300.0,
  Culture=neutral, PublicKeyToken=b77a5c561934e089
</item>
<item id="ref-30">
  System.Windows.Forms.UnsafeNativeMethods+IOleWindow,
  System.Windows.Forms, Version=1.0.3300.0,
  Culture=neutral, PublicKeyToken=b77a5c561934e089
</item>
<item id="ref-31">
  System.Windows.Forms.UnsafeNativeMethods+
    IViewObject,
  System.Windows.Forms, Version=1.0.3300.0,
  Culture=neutral, PublicKeyToken=b77a5c561934e089
</item>
<item id="ref-32">
  System.Windows.Forms.UnsafeNativeMethods+
    IViewObject2,
  System.Windows.Forms, Version=1.0.3300.0,
  Culture=neutral, PublicKeyToken=b77a5c561934e089
</item>
<item id="ref-33">
  System.Windows.Forms.UnsafeNativeMethods+IPersist,
  System.Windows.Forms, Version=1.0.3300.0,
  Culture=neutral, PublicKeyToken=b77a5c561934e089
</item>
<item id="ref-34">System.Windows.Forms.
  UnsafeNativeMethods+IPersistStreamInit,
  System.Windows.Forms, Version=1.0.3300.0,
  Culture=neutral, PublicKeyToken=b77a5c561934e089
</item>
<item id="ref-35">System.Windows.Forms.
  UnsafeNativeMethods+IPersistPropertyBag,
  System.Windows.Forms, Version=1.0.3300.0,
  Culture=neutral, PublicKeyToken=b77a5c561934e089
</item>
<item id="ref-36">
  System.Windows.Forms.UnsafeNativeMethods+
    IPersistStorage,
  System.Windows.Forms, Version=1.0.3300.0,
  Culture=neutral, PublicKeyToken=b77a5c561934e089
</item>
<item id="ref-37">
  System.Windows.Forms.UnsafeNativeMethods+

```

```

                                IQuickActivate,
System.Windows.Forms, Version=1.0.3300.0,
Culture=neutral, PublicKeyToken=b77a5c561934e089
</item>
<item id="ref-38">System.ComponentModel.
    ISynchronizeInvoke,
System, Version=1.0.3300.0, Culture=neutral,
PublicKeyToken=b77a5c561934e089</item>
<item id="ref-39">System.Windows.Forms.IWin32Window,
System.Windows.Forms, Version=1.0.3300.0,
Culture=neutral,
PublicKeyToken=b77a5c561934e089</item>
<item id="ref-40">System.Windows.Forms.
    IContainerControl,
System.Windows.Forms, Version=1.0.3300.0,
Culture=neutral,
PublicKeyToken=b77a5c561934e089</item>
<item id="ref-41">System.Runtime.Remoting.
    Lifetime.ISponsor,
mscorlib, Version=1.0.3300.0, Culture=neutral,
PublicKeyToken=b77a5c561934e089</item>
</SOAP-ENC:Array>

<SOAP-ENC:Array id="ref-18"
SOAP-ENC:arrayType="xsd:anyType[2]">
  <item href="#ref-42"/>
  <item href="#ref-43"/>
</SOAP-ENC:Array>

<a3:CrossAppDomainData id="ref-42"
xmlns:a3="http://schemas.microsoft.com/clr/ns/
System.Runtime.Remoting.Channels">
  <_ContextID>1300872</_ContextID>
  <_DomainID>1</_DomainID>
  <_processGuid id="ref-44">
    20c23b9b_4d09_46a8_bc29_10037f04f46f
  </_processGuid>
</a3:CrossAppDomainData>

<a3:ChannelDataStore id="ref-43"
xmlns:a3="http://schemas.microsoft.com/clr/ns/
System.Runtime.Remoting.Channels">
  <_channelURIs href="#ref-45"/>
  <_extraData xsi:null="1"/>
</a3:ChannelDataStore>

<SOAP-ENC:Array id="ref-45">

```

```

        SOAP-ENC:arrayType="xsd:string[1]">
        <item id="ref-46">http://66.156.56.215:1958</item>
    </SOAP-ENC:Array>

</SOAP-ENV:Body>
</SOAP-ENV:Envelope>

```

Сообщение-ответ *remove Job Event*

После возврата из метода *remove JobEvent* инфраструктура .NET Remoting отправляет приложению JobClient следующее сообщение, указывающее, что работа метода завершена:

```

HTTP/1.1 200 OK
Content-Type: text/xml; charset="utf-8"
Server: MS .NET Remoting, MS .NET CLR 1.0.3705.0
Content-Length: 586

```

```

<SOAP-ENV:Envelope ...> <SOAP-ENV:Body>
  <i2:remove_JobEventResponse id="ref-1"
    xmlns:i2="http://schemas.microsoft.com/clr/nsassem/
      JobServerLib.IJobServer/JobServerLib">
    </i2:remove_JobEventResponse>
  </SOAP-ENV:Body>
</SOAP-ENV:Envelope>

```

Заключение

В этой главе мы описали особенности протокола SOAP, влияющие на работу приложений .NET Remoting. Используя эти сведения о SOAP, мы познакомили вас с обменами сообщениями между приложениями JobClient и JobServer. Хотя .NET Remoting изолирует программиста, решающего задачи высокого уровня, от деталей SOAP, понимание этого протокола может стать мощным инструментом при разработке приложений .NET Remoting. Наблюдая трафик SOAP, вам удастся проследить за тем, как инфраструктура .NET Remoting посылает сообщения для создания объектов и обновления лицензий, а также просматривать сообщения, которые передаются при вызове ваших методов. Так как теперь вы овладели базовыми познаниями з SOAP, обмен сообщениями на основе этого протокола будет использоваться з дальнейших примерах этой книги.

Глава 5

Сообщения и прокси

До настоящего момента мы пользовались только стандартными возможностями .NET Remoting. Теперь мы займемся настройкой различных элементов инфраструктуры .NET Remoting и начнем с прокси. Из этой главы вы узнаете о создании специализированного прокси, реализующего простую *схему распределения нагрузки* (load-balancing). При разработке другого прокси мы покажем, как применять *ProxyAttribute* для перехвата активизации, а также продемонстрируем использование *контекста вызова* для передачи дополнительной информации при вызове метода. Но прежде чем заняться разработкой специализированных прокси, рассмотрим разновидности сообщений, которые им поступают.

Сообщения

В этой книге мы рассмотрим множество *сообщений*, так как они *представляют собой* базовую единицу передачи данных в приложениях .NET Remoting. Объекты .NET Remoting, такие, как прокси и приемники сообщений, работают с сообщениями очень активно, поэтому прежде, чем приступить к разработке *объектов*, рассмотрим сообщения более подробно.

Как вы помните из *главы 2*, все сообщения .NET Remoting являются производными от */Message*, в котором определено единственное *свойство Properties* типа *IDictionary*. *IDictionary* — это интерфейс для наборов пар «ключ — значение». И ключи, и значения в наборах *IDictionary* имеют тип *Object*, так что в эти наборы можно поместить любой тип .NET. Однако для пересылки через границы .NET Remoting эти объекты должны быть сериализуемыми. Подробнее сериализация рассматривается в *главе 8*.

Сообщения вызовов конструкторов

При вызове метода (включая вызов конструктора) удаленного объекта инфраструктура .NET Remoting создает сообщение, описывающее этот вызов. Рассмотрим следующий фрагмент кода клиента:

```
Object obj = new MyRemoteObject();
```

Создание экземпляра *MyRemoteObject* приводит к созданию экземпляра сообщения .NET Remoting, которое содержит в *IDictionary* записи, показанные в табл. 5-1.

Таблица 5-1. Сообщения вызовов конструкторов

Ключ словаря	Тип данных значения словаря	Значение словаря
<code>__Uri</code>	<i>String</i>	<i>null</i>
<code>__MethodName</code>	<i>String</i>	<i>.ctor</i>
<code>__TypeName</code>	<i>System.String</i>	<i>MyNamespace.MyRemoteObject, MyAssembly, Version=1.0.882.27668, CultureNeutral, PublicKeyToken=null</i>
<code>__MethodSignature</code>	<i>Type[]</i>	<i>null</i>
<code>__Args</code>	<i>Object[]</i>	<i>null</i>
<code>__CallContext</code>	<i>LogicalCallContext</i>	<i>null</i>
<code>__ActivationType</code>	<i>Type</i>	<i>null</i>
<code>__CallSiteActivationAttributes</code>	<i>Object[]</i>	<i>null</i>

Значение при ключе `__MethodName` идентифицирует вызываемый метод как *.ctor*, что соответствует методу-конструктору класса *MyRemoteObject*. Так как конструктор *MyRemoteObject* не имеет параметров, значением `__MethodSignature` и `__Args` является *null*. Вызывающий код клиента может установить контекст вызова, поэтому словарь содержит ключ `__CallContext`. Мы рассмотрим контекст вызова и способы его использования далее. Наконец, сообщение содержит словарные ключи для специализированных свойств активизации, применяемых инфраструктурой .NET Remoting во время активизации.

Сообщения вызовов методов

Рассмотрим следующий вызов метода удаленного объекта:

```
Obj.MyMethod("A string", 14);
```

В результате этого вызова генерируется сообщение, в *IDictionary* которого содержатся записи, приведенные в табл. 5-2.

Таблица 5-2. Сообщения вызовов методов

Ключ словаря	Тип данных значения словаря	Значение словаря
<code>__Uri</code>	<i>String</i>	<i>null</i>
<code>__MethodName</code>	<i>String</i>	<i>MyMethod</i>
<code>__TypeName</code>	<i>System.String</i>	<i>MyNameSpace.MyRemote-Object, MyAssembly, Version= 1.0.882.27668, CultureNeutral, PublicKeyToken=null</i>
<code>__MethodSignature</code>	<i>Type[]</i>	<i>[0] = System.String [1] = System.Int32</i>
<code>__Args</code>	<i>Object[]</i>	<i>[0] = a string [1] = 14</i>
<code>__CallContext</code>	<i>LogicalCallContext</i>	<i>null</i>

Значения для ключей `__MethodName`, `__MethodSignature` и `__Args` задают вызываемый метод удаленного объекта. Ключи активизации отсутствуют, так как объект уже активизирован.

Типы сообщений

Все сообщения .NET Remoting реализуют интерфейс */Message*, поэтому у всех сообщений имеется по крайней мере одно свойство типа *IDictionary*. Инфраструктура .NET Remoting определяет ряд интерфейсов, производных от *IMessage*, которые служат в основном двум целям. Во-первых, каждый из этих интерфейсов предоставляет различные свойства и/или методы для более удобного доступа к внутреннему словарю сообщения. Во-вторых, тип интерфейса служит маркером, позволяющим определить, как следует обрабатывать данное сообщение. Например, иногда требуется отличать сообщение для создания объекта с клиентской активизацией от сообщения, сгенерированного в результате

обычного вызова метода. Хотя оба эти сообщения могут быть доставлены посредством простого класса, реализующего только *IMessage*, наличие у них интерфейсов разных типов позволяет определить назначение сообщения. Сводка распространенных интерфейсов и классов сообщений приведена в табл. 5-3.

Таблица 5-3. Распространенные типы сообщений

Тип сообщения	Имя члена (выборочно)	Описание
<i>IMessage</i>		Реализован всеми сообщениями .NET Remoting
<i>IMethodMessage</i>		Реализован сообщениями, описывающими любые вызовы. Определяет свойства, общие для всех вызовов
	<i>Args</i>	Массив передаваемых аргументов
	<i>MethodName</i>	Имя вызываемого метода
	<i>Uri</i>	URI объекта, которому предназначено данное сообщение
	<i>TypeName</i>	Полное имя типа объекта, которому предназначено данное сообщение
<i>IMethodReturnMessage</i>		Реализован всеми сообщениями, возвращаемыми клиенту
	<i>Exception</i>	Исключение, сгенерированное удаленным объектом (если оно есть)
	<i>OutArgs</i>	Массив аргументов вида <i>out</i>
	<i>ReturnValue</i>	Объект, содержащий возвращаемое значение удаленного метода (если оно есть)
<i>IMethodCallMessage</i>		Реализован всеми сообщениями, генерируемыми при вызове методов. Задает свойства, общие для вызовов всех методов

см. след. стр.

Таблица 5-3. (продолжение)

Тип сообщения	Имя члена (выборочно)	Описание
<i>IConstructionCallMessage</i>	<i>InArgs</i>	Массив аргументов вида [in]
		Сообщение, реализующее данный интерфейс, посылается объекту с клиентской активизацией при вызове <i>new</i> или <i>Activator.CreateInstance</i> . Объекты с серверной активизацией получают это сообщение при первом вызове метода. Как следует из названия, это первое сообщение, посылаемое объекту, которое содержит свойства, общие для создания удаленных объектов
	<i>ActivationType</i>	Возвращает тип активируемого удаленного объекта
	<i>Activator</i>	Возвращает или устанавливает активизатор (activator), используемый для активизации удаленного объекта
<i>IConstructionReturnMessage</i>	<i>ContextProperties</i>	Возвращает список свойств контекста, определяющих контекст создания объекта
		Сообщение, реализующее данный интерфейс, возвращается клиенту в ответ на сообщение <i>IConstructionCallMessage</i>
<i>ReturnMessage</i>		Конкретный класс, реализующий <i>IMethodReturnMessage</i> . Этот класс документирован, что позволяет при перехвате вызова

см. след. стр.

Таблица 5-3. (окончание)

Тип сообщения	Имя члена (выборочно)	Описание
		метода создать сообщение для возврата клиенту без фактического обращения к удаленному объекту

Теперь, познакомившись с видами сообщений, мы можем перейти к объектам .NET Remoting, которые эти сообщения обрабатывают. Начнем с клиентской стороны, откуда приходят вызовы методов удаленных объектов и где инфраструктура .NET Remoting создает сообщения.

Прокси

В теории, *объект-прокси* — это любой объект, подставляемый вместо другого объекта и управляющий доступом к этому второму объекту. Управление другим объектом посредством прокси применяется, например, чтобы отсрочить создание объекта, если его инициализация требует значительных ресурсов, или для контроля прав доступа. Большинство технологий удаленного взаимодействия, включая .NET Remoting, используют прокси как минимум для того, чтобы представить удаленные объекты как локальные. В зависимости от архитектуры, такие технологии удаленного взаимодействия обычно используют прокси и для выполнения других задач.

Фактически слой прокси .NET Remoting состоит из двух объектов-прокси: один из которых реализован, в основном, *неуправляемым* кодом, а другой — управляемым, который вы можете настраивать. Разделение слоя прокси на два разных объекта облегчает настройку прокси для сложных сценариев. Прежде чем мы займемся настройкой прокси, давайте рассмотрим эти два объекта.

TransparentProxy

Предположим, вы создаете экземпляр удаленного объекта вызовом *new* или *Activator.CreateInstance*:

```
MyObj obj = new MyObj();
int Result = obj.MyMethod();
```

После выполнения строки, содержащей `new`, в *MyObj* содержится ссылка на объект типа *TransparentProxy*. Инфраструктура .NET Remoting, используя метаданные реального объекта, динамически создает экземпляр *TransparentProxy* во время выполнения с помощью отражения (reflection). Отражение в .NET Remoting является мощным средством работы с метаданными во время выполнения. С помощью отражения инфраструктура .NET Remoting определяет открытый интерфейс реального объекта и создает *TransparentProxy*, который моделирует этот интерфейс. (Подробнее об отражении — в книге Джеффри Рихтера «Applied Microsoft .NET Framework Programming»¹.) Полученный объект *TransparentProxy* реализует открытые методы, свойства и члены *MyObj*. Код клиента использует локальный объект *TransparentProxy* в качестве дублера реального удаленного объекта, который расположен в другом подразделении .NET Remoting — в другом контексте, домене приложения, процессе или на ином компьютере. С этого момента вы не обнаружите никаких отличий в способах работы с локальным экземпляром *MyObj* и *TransparentProxy*, дублирующим удаленный объект.

Основная задача *TransparentProxy* состоит в перехвате вызовов метода удаленного объекта и передаче их *RealProxy*, который и выполняет основную работу. (Мы рассмотрим *RealProxy* чуть позже.) Используя неуправляемый код, *TransparentProxy* перехватывает вызовы того, что для вызывающего кажется локальным объектом, и создает структуру *MessageData*, состоящую, в основном, из указателей на неуправляемую память и описывающую вызов метода. Эта структура передается *TransparentProxy* методу *PrivateInvoke* объекта *RealProxy*. Метод *PrivateInvoke* по информации из структуры *MessageData* создает сообщения, которые затем отправляются по цепочке приемников сообщений и в конце концов попадают к удаленному объекту.

Хотя клиентский код работает вместо реального объекта именно с *TransparentProxy*, у вас нет возможности как-либо настраивать или расширять *TransparentProxy*. Единственный прокси, который разрешается настраивать — это *RealProxy*.

¹ Рихтер Дж. «Программирование на платформе Microsoft .NET Framework». М.: «Русская Редакция», 2002. — Прим. перев.

RealProxy

RealProxy — это задокументированный и расширяемый управляемый класс, содержащий ссылку на неуправляемый «черный ящик» под названием *TransparentProxy*. Однако *RealProxy* — это абстрактный класс, поэтому непосредственное создание его экземпляров невозможно. Когда экземпляр *TransparentProxy* передает объект *MessageData* объекту *RealProxy*, на самом деле *MessageData* передается конкретному классу, производному от *RealProxy*, который называется *RemotingProxy*. *RemotingProxy* — это недокументированный внутренний класс, нельзя наследовать от него. Вместо этого, чтобы получить возможность настраивать поведение прокси, следует заместить *RemotingProxy* собственным классом, производным от *RealProxy*. Хотя большинство расширений инфраструктуры .NET Remoting выполняется с помощью приемников сообщений, некоторые причины для создания подклассов *RealProxy* все-таки существуют. *RealProxy* предоставляет первую возможность перехвата и настройки как вызовов создания удаленных объектов, так и вызовов их методов. Однако из-за отсутствия настраиваемого прокси на серверной стороне для тех задач, которые требуют обработки и на клиенте, и на сервере, например для шифрования и сжатия, необходимо использовать приемники сообщений.

Расширение RealProxy

Для создания прокси взамен *RemotingProxy* требуется создать класс, производный от *RealProxy*, добавив новый конструктор и закрытый объект *MarshalByRefObject*, как показано ниже:

```
public class MyProxy : RealProxy
{
    MarshalByRefObject _target;
    : base(type)
    public MyProxy(Type type, MarshalByRefObject target)
    :
        _target = target;
}
```

Нам необходимо запомнить ссылку на реальный *MarshalByRefObject*, чтобы прокси мог затем передавать вызовы реальному

удаленному объекту. Мы также вызываем конструктор *RealProxy* и передаем ему тип удаленного объекта, чтобы инфраструктура .NET Remoting сумела сгенерировать экземпляр *TransparentProxy* для этого объекта.

Затем необходимо переопределить абстрактный метод *Invoke*:

```
public override IMessage Invoke(IMessage msg)
{
    // Обработать сообщение.
}
```

Invoke является основной точкой расширения *RealProxy*. Здесь мы можем изменить и передать дальше, отвергнуть или игнорировать сообщения, посланные *TransparentProxy* и предназначенные реальному объекту.

Теперь, когда у нас есть компилирующийся класс, производный от *RealProxy*, следует подключить экземпляр этого класса к некоторому *MarshalByRefObject*, и, таким образом, заменить класс *RemotingProxy*. Известно два приема создания прокси: использование класса *ProxyAttribute* и непосредственное создание прокси. Далее в примерах мы рассмотрим оба эти приема.

Специализированные прокси на практике

Инфраструктура .NET Remoting предоставляет очень много способов расширения архитектуры удаленного взаимодействия, поэтому некоторые задачи настройки можно выполнить более чем одним способом. Хотя наиболее удобными точками расширения считаются приемники сообщений и каналы, по некоторым причинам отдельные настройки выполняются посредством прокси. Так как у прокси отсутствует соответствующий настраиваемый слой на серверной стороне, то лучше всего они подходят для задач, связанных с перехватом вызовов на клиенте. С другой стороны, если вам для настройки требуется выполнение некоторых действий как на клиенте, так и на сервере, необходимо использовать приемники сообщений.

В этом разделе мы рассмотрим три примера специализированных прокси. Сначала мы используем прокси для перехвата активизации удаленного объекта и демонстрации различий между клиентской и серверной активизацией. Затем применим прокси

для переключения между каналами, поддерживающими брандмауэры, и каналами повышенной производительности. И в завершение мы рассмотрим схему распределения нагрузки, использующую контекст вызова.

Пример с активизацией

Одна из интересных особенностей прокси — то, что их можно применять для перехвата активизации объектов как с клиентской, так и с серверной активизацией. Перехватив запрос на активизацию объекта, вы можете активизировать другой объект (иногда даже на другом компьютере), изменить аргументы конструктора объекта с клиентской активизацией или подключить специализированный активизатор. Прежде чем начать, давайте рассмотрим класс *ProxyAttribute*.

ProxyAttribute предоставляет возможность сообщить о том, что .NET Remoting должна использовать наш специализированный прокси вместо стандартного *RemotingProxy*. Единственное ограничение данного способа подключения специализированных прокси состоит в том, что *ProxyAttribute* разрешается применять только к объектам, производным от *ContextBoundObject*, что мы покажем при рассмотрении контекстов в главе 6. Так как *ContextBoundObject* является производным от *MarshalByRefObject*, данное ограничение не станет проблемой. Чтобы использовать *ProxyAttribute*, сначала следует создать класс, производный от *ProxyAttribute*, и переопределить метод *CreateInstance* базового класса:

```
[AttributeUsage(AttributeTargets.Class)]
public class MyProxyAttribute : ProxyAttribute
{
    public override MarshalByRefObject CreateInstance(
        Type serverType)
    {
        MarshalByRefObject target = base.CreateInstance(
            serverType);
        MyProxy myProxy = new MyProxy(serverType, target);
        return (MarshalByRefObject)myProxy.GetTransparentProxy();
    }
}
```

Затем, свяжите *ProxyAttribute* с *ContextBoundObject*:

```
[MyProxyAttribute]
public class MyRemoteObject : ContextBoundObject
{
    I
```

При создании экземпляра *MyRemoteObject* .NET Remoting вызывает переопределенный метод *CreateInstance*, в котором надо создать специализированный прокси. Затем мы создаем наш прокси, преобразуем его тип *TransparentProxy* к *MarshalByRefObject* и возвращаем результат. Данный прием позволяет клиенту вызывать *new* и *Activator.CreateInstance* как обычно и при этом использовать специализированный прокси.

Для перехвата активизации необходимо выполнить следующие действия:

- определить класс *ProxyAttribute*;
- определить класс *RealProxy* и переопределить его метод *Invoke*;
- в методе *Invoke* обработать сообщение *IConstructionCallMessage*;
- применить *ProxyAttribute* к *ContextBoundObject*.

Вот листинг *ProxyAttribute* для нашего примера активизации:

```
[AttributeUsage(AttributeTargets.Class)]
public class SProxyAttribute : ProxyAttribute
{
    public override System.MarshalByRefObject
        CreateInstance(System.Type serverType)
    {
        Console.WriteLine("SProxyAttribute.CreateInstance()");
        // Получить прокси реального объекта.
        MarshalByRefObject mbr = base.CreateInstance(
            serverType);

        // Находимся ли мы на клиентской или на серверной
        // стороне. Это важно так как данный метод вызывается
        // инфраструктурой по обе стороны границы
        // .NET Remoting. На серверной стороне необходимо
        // вернуть объект, созданный выше вызовом
        // CreateInstance из базового класса, а не наш
        // специализированный прокси, Если мы возвратим
        // наш прокси, то при попытке вызова его методов
```

```

// исполняющая среда будет генерировать исключение.
//
if ( RemotingConfiguration.IsActivationAllowed(
    serverType) )
{
    return mbr;
}
else
{
    WellKnownServiceTypeEntry[] wcte =
        RemotingConfiguration
            .GetRegisteredWellKnownServiceTypes();
    if ( wcte.Length > 0 )
    {
        foreach(WellKnownServiceTypeEntry e in wcte)
        {
            if ( e.ObjectType == serverType )
            {
                return mbr;
            }
        }
    }
}

// Если мы попали сюда, то функция выполняется
// на клиентской стороне и мы можем использовать
// наш прокси.
if ( RemotingServices.IsTransparentProxy(mbr) )
{
    // Обернуть прокси нашим простым
    // прокси-перехватчиком.
    SimpleInterceptionProxy p =
        new SimpleInterceptionProxy(serverType, mbr);
    MarshalByRefObject mbr2 =
        (MarshalByRefObject)p.GetTransparentProxy();
    return mbr2;
}
else
{
    // Это реальный объект.
    return mbr;
}
}
}

```

Переопределенный нами метод *ProxyAttribute.CreateInstance* вызывается при создании удаленного объекта, которому присво-

ен данный атрибут. Внутри *CreateInstance* мы сначала вызываем *CreateInstance* из базового класса для создания *TransparentProxy* реального объекта. Далее нам следует иметь в виду, что обращение к *ProxyAttribute* происходит для всех экземпляров удаленных объектов, с которыми связан этот атрибут. Это означает, что наш метод *CreateInstance* будет исполняться и когда клиент создает ссылку на объект, и когда инфраструктура .NET Remoting создает экземпляр объекта на сервере. Мы хотим обрабатывать только вызов на стороне клиента, поэтому необходимо определить, где происходит текущий вызов. Чтобы определить, не исполняется ли наш объект-атрибут на сервере, сначала мы с помощью вызова *RemotingConfiguration.GetRegisteredWellKnownServiceTypes* получаем массив *WellKnownServiceTypeEntry*. Если наш тип найден внутри этого массива, то мы считаем, что код исполняется на сервере и можно просто вернуть *TransparentProxy*. В противном случае мы считаем, что код исполняется на клиенте, создаем наш прокси-перехватчик и возвращаем его *TransparentProxy*.

Что произойдет дальше, зависит от того, применяется ли для удаленного объекта клиентская или серверная активизация. Если это объект с клиентской активизацией, то инфраструктура .NET Remoting вызовет метод *Invoke* нашего прокси-перехватчика до того, как произойдет возврат из клиентского вызова *new* или *Activator.CreateInstance*. Если же это объект с серверной активизацией, то вызов *new* или *Activator.CreateInstance* возвращает управление и метод *Invoke* нашего прокси не вызывается до первого вызова метода удаленного объекта.

Вот код нашего *SimpleInterceptionProxy*:

```
public override Invoke(IMessage msg)
{
    // Сообщения вызовов конструктора обрабатываются отдельно
    // от сообщений вызовов обычных методов.
    if ( msg is IConstructionCallMessage )
    {
        // Необходимо вручную завершить клиентскую активизацию.
        string url = GetUrlForCAO(_type);
        if ( url.Length > 0 )
        {
            ActivateCAO((IConstructionCallMessage)msg, url);
        }
    }
}
```

```

    }

    IConstructionReturnMessage crm =
        EnterpriseServicesHelper.
        CreateConstructionReturnMessage(
            (IConstructionCallMessage)msg,
            (MarshalByRefObject)this.GetTransparentProxy());
    return crm;
}
else
{
    MethodCallMessageWrapper mcm =
        new MethodCallMessageWrapper((IMethodCallMessage)msg);
    mcm.Uri = RemotingServices.GetObjectUri(
        (MarshalByRefObject)_target);
    return RemotingServices.GetEnvoyChainForProxy(
        (MarshalByRefObject)_target).SyncProcessMessage(msg);
}
}

private void ActivateCAO(IConstructionCallMessage ccm,
    string url)
{
    // Подключиться к удаленному сервису активизации.
    string rem = url + @"/RemoteActivationService.rem";
    IActivator remActivator =
        (IActivator)RemotingServices.Connect(typeof(IActivator),
            rem);
    IConstructionReturnMessage crm = remActivator.Activate(ccm);

    //
    // Свойство ReturnValue возвращаемого сообщения -
    // это ObjRef для удаленного объекта.
    // Выполняем его демаршалинг в локальный
    // прокси, которому будем передавать сообщения.
    ObjRef oRef = (ObjRef)crm.ReturnValue;
    _target = RemotingServices.Unmarshal(oRef);
}

string GetUrlForCAO(Type type)
{
    string s = "";
    ActivatedClientTypeEntry[] act =
        RemotingConfiguration.GetRegisteredActivatedClientTypes();

```

```

foreach( ActivatedClientTypeEntry acte in act )
{
    if ( acte.ObjectType == type )
    {
        s = acte.ApplicationUrl;
        break;
    }
}
return s;
}

```

Независимо от того, использует ли удаленный объект клиентскую или серверную активизацию, первое сообщение, посылаемое методу *Invoke* прокси, — *IConstructionCallMessage*. Как вы узнаете далее, клиентскую активизацию объектов необходимо обрабатывать явно. Сначала с использованием нашего метода *GetUrlForСЛО* мы определяем тип активизации. Эта процедура аналогична тому, как мы определяли, выполняется ли *ProxyAttribute* на клиенте или на сервере. Но вместо поиска зарегистрированных объектов с серверной активизацией мы просматриваем список зарегистрированных типов с клиентской активизацией, полученный при помощи *RemotingConfiguration.GetRegisteredActivatedClientTypes*. Если мы находим тип удаленного объекта в массиве объектов *ActivatedClientTypeEntry*, то возвращаем *ApplicationUrl* соответствующего элемента массива, который понадобится нам для выполнения клиентской активизации.

Если URL является пустой строкой, мы считаем, что используется серверная активизация. Обработка сообщений создания объектов с серверной активизацией тривиальна. Как вы помните, объекты с серверной активизацией создаются серверным приложением автоматически, когда какой-нибудь клиент первый раз вызовет метод такого объекта. Если же строка URL не пуста, мы вызываем *ActivateCAO* для выполнения клиентской активизации объекта. Данный процесс несколько сложнее, чем серверная активизация, так как нам необходимо создать новый экземпляр удаленного объекта. При регистрации объекта с клиентской активизацией на сервере, .NET Remoting создает объект с серверной активизацией с общеизвестным URI *RemoteActivation.rem*. Затем этот объект с серверной активизацией используется средой для создания экземпляров зарегистрированных объектов с

клиентской активизацией. Так как мы взяли на себя управление процессом активизации, нам необходимо напрямую обратиться к созданному .NET Remoting *RemoteActivation.rem* URL. Сначала мы добавляем *RemoteActivation.rem* к URL *ActivatedClientTypeEntry*. Затем с помощью *RemotingServices.Connect* мы запрашиваем объект-Активатор для нашего объекта с клиентской активизацией. Для создания экземпляра объекта мы передаем сообщение *IConstructionCallMessage* методу *Activate* активатора, который возвращает сообщение *IConstructionReturnMessage*. Мы извлекаем из этого сообщения *ObjRef* и выполняем демаршалинг последнего для получения прокси удаленного объекта.

Наконец, мы присваиваем наш прокси-атрибут объекту, производному от *ContextBoundObject*:

```
[SProxyAttribute()]
public class CBRemoteObject : ContextBoundObject
{
    ...
}
```

Теперь наш специализированный прокси правильно обрабатывает как клиентскую, так и серверную активизацию объектов. Далее мы рассмотрим реальный пример, использующий сообщения *IMethodCallMessage* и *IMethodReturnMessage*.

Пример со сменой канала

Давайте вновь вернемся к нашему приложению распределения заданий из главы 3. Как вы помните, использование *Microsoft Internet Information Services (IIS)* в качестве сервера для удаленного объекта *JobLib* сделало приложение совместимым с брандмауэрами, но при этом пострадала производительность, поскольку IIS поддерживает только *HttpChannel*. Предположим, что некоторыми компьютерами, на которых работает приложение *JobClient*, являются ноутбуки, пользователи которых сегодня работают по ту, а завтра по эту сторону брандмауэра. Неплохо было бы при работе в интрасети применить более скоростной *TcpChannel*, но переключаться на *HttpChannel* при работе по Интернету — и все это без участия пользователя. Для этой цели разработаны технологии туннелирования протокола TCP по HTTP. Однако с .NET Remoting, нам не понадобится туннелирование, так как по-

средством специализированного прокси описанную проблему можно решить гораздо проще, Вот основные этапы:

1. зарегистрировать на сервере оба канала: TCP и HTTP;
2. в качестве сервера удаленного объекта использовать управляемую исполняемую программу, так как IIS не поддерживает *TcpChannel*;
3. определить специализированный прокси;
4. внутри прокси зарегистрировать канал HTTP;
5. в клиентском коде создать экземпляр прокси вручную;
6. в методе *Invoke* проверить свойство *Exception* возвращаемого сообщения *IMethodReturnMessage* и при необходимости повторить посылку сообщения по HTTP.

Изменения в *JobServer*

Нам нужно изменить *JobServer* так, чтобы регистрировались оба канала: TCP и HTTP:

```
// Зарегистрировать канал сервера.
TcpChannel oTcpJobChannel = new TcpChannel( 5556 );
ChannelServices.RegisterChannel( oTcpJobChannel );
HttpChannel oHttpJobChannel = new HttpChannel( 5555 );
ChannelServices.RegisterChannel( oHttpJobChannel );

// Зарегистрировать общеизвестный тип.
RemotingConfiguration.RegisterWellKnownServiceType(
    typeof( JobServerImpl ), "JobURI",
    WellKnownObjectMode.Singleton );
```

Наиболее интересная деталь здесь — отсутствие связи между объектом и каналом, по которому к нему идет доступ. Единственный объект *JobServerImpl* доступен по всем зарегистрированным каналам. Таким образом, в данном случае клиент может выполнить один вызов метода по TCP, а другой — по HTTP.

Изменения в *JobClient*

Вместо того чтобы использовать для создания нашего прокси *ProxyAttribute*, мы создаем прокси в клиентском коде непосредственно, как показано ниже:

```
MyProxy myProxy =
    new MyProxy(typeof(RemoteObject), new RemoteObject());
RemoteObject foo = (RemoteObject)myProxy.GetTransparentProxy();
```

Хотя непосредственное создание прокси представляет собой самый простой способ подключения нестандартного прокси, у него отмечается ряд недостатков:

- такой прокси не способен перехватывать активизацию удаленного объекта. Возможно, вам никогда не потребуется вмешиваться в процесс активизации, так что это ограничение обычно не является проблемой;
- такой прокси не прозрачен для клиентского кода. Так как клиент создает экземпляр прокси напрямую, эта процедура должна быть «защита» в код клиента.

Реализация нестандартного прокси

Все клиентские алгоритмы, позволяющие пересылать сообщения либо по *TcpChannel*, либо по *HttpChannel*, содержатся внутри специализированного прокси. Прокси регистрирует *HttpChannel*, пытается подключиться по *TcpChannel* и в случае ошибки выполняет повторную попытку подключения по *HttpChannel*.

```
public class ChannelChangerProxy : RealProxy
{
    string _TcpUrl;
    string _HttpUrl;
    IMessageSink[] _messageSinks;

    public ChannelChangerProxy (Type type, string url)
        : base(type)
    {
        _TcpUrl = url;
        BuildHttpURL( url );
        _messageSinks = GetMessageSinks();
    }

    private void BuildHttpURL(string _TcpUrl)
    {
        UriBuilder uBuilder = new UriBuilder(_TcpUrl);
        uBuilder.Port = 5555;
        uBuilder.Scheme = "http";
        _HttpUrl = uBuilder.ToString();
    }

    public override IMessage Invoke(IMessage msg)
    {
        Exception InnerException;
```

```

msg.Properties["__Uri"] = _TopUrl;
IMessage retMsg =
    _messageSinks[0].SyncProcessMessage( msg );
if (retMsg is IMethodReturnMessage)
{
    IMethodReturnMessage mrm = (
        IMethodReturnMessage)retMsg;
    if (mrm.Exception == null)
    {
        return retMsg;
    }
    else
    {
        InnerException = mrm.Exception;
    }
}
// При ошибке повторить с помощью канала HTTP.
msg.Properties["__Uri"] = _HttpUrl;
retMsg = _messageSinks[1].SyncProcessMessage( msg );
if (retMsg is IMethodReturnMessage)
{
    IMethodReturnMessage mrm = {
        IMethodReturnMessage)retMsg;
}
return retMsg;
}

private IMessageSink[] GetMessageSinks()
{
    IChannel[] registeredChannels =
        ChannelServices.RegisteredChannels;
    IMessageSink MessageSink;
    string ObjectURI;
    ArrayList MessageSinks = new ArrayList();
    foreach (IChannel channel in registeredChannels )
    {
        if (channel is IChannelSender)
        {
            IChannelSender channelSender =
                (IChannelSender)channel;
            MessageSink = channelSender.CreateMessageSink(
                _TcpUrl, null, out ObjectURI);
            if (MessageSink != null)
            {
                MessageSinks.Add( MessageSink );
            }
        }
    }
}

```

```

    }
    string objectURI;
    HttpChannel HttpChannel = new HttpChannel();
    ChannelServices.RegisterChannel(HttpChannel);
    MessageSinks.Add(HttpChannel.CreateMessageSink(
        _HttpUrl, HttpChannel, out objectURI);
    if (MessageSinks.Count > 0)
    {
        return (IMessageSink[])MessageSinks.ToArray(
            typeof(IMessageSink));
    }
    // В блоке foreach не был найден MessageSink
    // для заданного URL.
    throw new
        Exception("Unable to find a MessageSink for
            the URL:" + _TcpUrl);
}
}

```

Конструктор *ChannelChangerProxy* ожидает получения *G* в качестве параметра URL, в котором задан протокол TCP. Затем вызывается *BuildHttpUrl*, в котором порт и протокол URL изменяются на общеизвестный канал HTTP приложения *JobServer*. В данном случае используется фиксированный HTTP-порт 80, хотя для большей гибкости номер порта можно читать из конфигурационного файла. Когда метод *Invoke* получает очередное сообщение, оно в конечном счете переправляется первому приемнику сообщений в цепочке. Мы подробно рассмотрим настройку приемников сообщений в главе 6, здесь же просто создадим и используем стандартные приемники.

В методе *GetMessageSinks* просматривается список зарегистрированных каналов. (В данном случае мы зарегистрировали единственный канал TCP.) Найдя зарегистрированный канал, мы создаем приемник сообщений, который будет пересылать их по заданному URL (в данном случае — *tcp://JobMachine:5555/JobURL*), и запоминаем приемник в *ArrayList*. Так как клиенту ничего не известно о намерении прокси использовать HTTP, нам необходимо зарегистрировать HTTP-канал, после чего создать и запомнить приемник сообщений и для него.

Так как `JobServer` предоставляет объект с серверной активизацией, до первого вызова метода не будет ни вызова конструктора, ни какого-либо иного трафика на удаленный сервер. Наш метод `Invoke` получит первое сообщение, когда клиент вызовет первый метод `jobServerLib`. Когда это случится, `TransparentProxy` вызовет метод базового класса нашего прокси `RealProxy.PrivateInvoke`, который создаст сообщение и затем передаст его нашему методу `Invoke`. С помощью `SyncProcessMessage` мы передаем это сообщение непосредственно приемнику сообщений канала `TcpChannel`. `SyncProcessMessage` возвращает объект `IMethodReturnMessage`, по содержимому которого мы определяем успех или неудачу вызова удаленного метода. В случае ошибки инфраструктура .NET Remoting не генерирует исключение, но заносит информацию в свойство `Exception` объекта `IMethodReturnMessage`. Если сервер работает, если отсутствует брандмауэр, отсекающий TCP-трафик, и если нет никаких других проблем с доставкой сообщения, то значение свойства `Exception` равно `null`, и мы просто возвращаем `IMethodReturnMessage` из `Invoke`.

В случае возникновения исключения мы повторяем операцию с использованием `HttpChannel` и URL, в котором задан протокол HTTP. Но прежде чем отослать сообщение, необходимо изменить свойство `Uri` сообщения на HTTP URL. Если снова происходит ошибка, мы возвращаем `IMethodReturnMessage`, чтобы инфраструктура .NET Remoting могла сгенерировать исключение.

В данном примере с помощью специализированного прокси нам удалось поддержать в приложении распределения заданий работу как со скоростными каналами, так и через брандмауэры, не заставляя делать этот выбор в клиентском коде. Более того инфраструктура .NET Remoting позволила нам для решения этой задачи использовать высокоуровневые средства, не прибегая к программированию на уровне протоколов HTTP и TCP.

Теперь мы рассмотрим более сложный пример специализированных прокси: использование `ProxyAttribute` и контекста вызова.

Пример с распределением нагрузки

Предположим, необходимо обеспечить масштабируемость `JobServer`, чтобы он поддерживал сотни или тысячи пользователей и оставался доступным в любое время суток. Текущая архитектура с

единственным *JobServer* не удовлетворяет данным требованиям. Нам потребуются избыточные приложения *JobServer*, исполняющиеся на разных машинах и имеющие доступ к текущим данным каждого *JobClient*, а также средство распределения нагрузки между серверами. Хотя нам придется внести ряд изменений в *JobServer*, используя специализированный прокси, мы сможем сделать прозрачным для кода *JobClient* тот факт, что клиент будет работать с несколькими серверами. В реальном многосерверном приложении потребовался бы некоторый способ доступа к общим данным, возможно, посредством базы данных. Вы можете попробовать реализовать такой вариант в качестве упражнения. Здесь же мы поговорим о деталях, связанных с прокси.

Вот что необходимо сделать:

- создать *сервер поиска* (discovery server) в конечной точке, известной клиенту;
- создать прокси, перехватывающий вызовы клиента и вызывающий сервер поиска для получения списка серверов, которым могут быть направлены клиентские вызовы;
- для всех удаленных объектов на избыточных серверах создать отдельные прокси;
- в прокси перехватывать клиентские вызовы методов и перенаправлять их прокси удаленных объектов по кругу;
- добавить контекст вызова для прозрачного распределения нагрузки.

Изменения в *LoadBalancingServer*

Для поддержки обнаружения избыточных серверов мы добавим общеизвестные URL этих серверов в конфигурационный файл *LoadBalancingServer*:

```
<configuration>
<appSettings>
  Odd key="PeerUrl1" value="tcp://localhost:5556/JobURI"/>
  <add key="PeerUrl2" value="tcp://localhost:5557/JobURI"/>
  <add key="PeerUrl3" value="tcp://localhost:5558/JobURI"/>
</appSettings>
```

Изменения в RemoteObject

Для поддержки сервера поиска *JobServerImpl* мы реализуем следующий новый интерфейс:

```
public interface IDiscoveryServer
{
    string[] GetPeerServerUrls();
}

public string[] GetPeerServerUrls()
{
    ArrayList PeerUrls = new ArrayList();
    bool UrlsFound = true;
    string BaseKeyString = "PeerUrl";
    string KeyString;
    int Count = 0;
    while(UrlsFound)
    {
        KeyString = BaseKeyString + (++Count).ToString();
        string PeerUrl = ConfigurationSettings.AppSettings[
            KeyString];
        if (PeerUrl != null)
        {
            PeerUrls.Add(PeerUrl);
        }
        else
        {
            UrlsFound = false;
        }
    }
    return (string[])PeerUrls.ToArray( typeof(string) );
}
```

Задача данного интерфейса — возврат списка сконфигурированных серверов по запросу. *JobServer* не выполняет каких-либо длительных операций, поэтому для моделирования загрузки мы добавим метод, в котором выполняется задержка с интервалом от 1 до 6 секунд:

```
public bool DoExpensiveOperation()
{
    Random Rand = new Random();
    int RandomNumber = Rand.Next(1000, 6000);
    System.Threading.Thread.Sleep( RandomNumber );
}
```

Реализация специализированного прокси

Вместо того чтобы требовать от клиента знания местонахождения *всех* избыточных серверов, мы используем сервер поиска, который будет *возвращать* их адреса. Таким образом, нам достаточно опубликовать единственный общеизвестный URL сервера поиска. Единственный сервер поиска снижает отказоустойчивость в момент поиска, однако данную проблему легко устранить добавлением дополнительных серверов поиска,

Использование контекста вызова

Допустим, нам нужно добавить информацию к вызову метода, не изменяя списка его аргументов. Это требуется, если вы, например:

- не хотите изменять интерфейс *объекта* из-за проблем совместимости версий;
- не хотите, чтобы вызывающий знал *об* этой информации по причинам, *связанным* с управлением доступом;
- хотите пересылать данные при вызове множества различных методов, и добавление избыточных аргументов засорило бы сигнатуры методов. Web-серверы посылают такие данные с помощью cookie, которые браузеры сохраняют и незаметно подставляют в HTTP-запрос к соответствующему Web-серверу. Большинство *приложений* не на основе браузеров вынуждены отправлять данные cookie как параметры при каждом вызове метода.

Все описанные проблемы удалось бы решить, если бы нам удалось передавать дополнительные параметры незаметно, как *браузеры* делают это с cookie. Именно этой цели и служит контекст вызова.

В нашем примере распределение нагрузки на серверы можно выполнять по примитивной круговой схеме, но *лучшее*, чтобы серверы сообщали нам о своей текущей загрузке, а мы отправляли бы следующее сообщение наименее загруженному серверу. Контекст вызова очень удобен для реализации *такой* схемы.

Что такое контекст вызова

CallContext — это объект, который пересылается между доменами приложений и содержимое которого может при этом иссле-

говаться различными перехватывающими объектами (такими, как прокси и приемники сообщений). *CallContext* также доступен из клиентского и серверного кода, и поэтому его можно использовать для передачи дополнительных данных при вызове метода, не изменяя список параметров. Достаточно просто поместить объект в контекст вызова методом *SetData* и извлечь его оттуда методом *CetData*. Для поддержки *CallContext* объект, помещаемый в него, должен быть помечен атрибутом *Serializable* и наследовать от интерфейса *ILogicalThreadAffinative*. *ILogicalThreadAffinative* — это просто маркер, не требующий от объекта реализации каких-либо методов.

Вот объект, который мы поместим в *CallContext*:

```

[Serializable]
public class CallContextData : ILogicalThreadAffinative
{
    int _CurrentConnections;
    public CallContextData(int CurrentConnections)
    {
        _CurrentConnections = CurrentConnections;
    }

    public int CurrentConnections
    {
        get
        {
            return _CurrentConnections;
        }
    }
}

```

CallContextData — это просто обертка вокруг *int*, позволяющая пересылать эту переменную посредством *CallContext*. С помощью *CallContextData* сервер возвращает общее число подключенных к нему клиентов. Наш прокси использует это значение для определения приоритетов прокси избыточных серверов. Конечно, в этом примере не обеспечивается работоспособное распределение нагрузки, так как число соединений, вероятно, изменится к моменту следующего вызова метода. Тем не менее нам удалось показать мощь совместного использования прокси и контекстов вызова — клиентскому коду нет необходимости знать о том, что его вызовы направляются на разные серверы, а вместе с вызо-

вом между сервером и клиентом пересылается дополнительная информация, не входящая в список параметров ни одного из методов.

Контекст вызова на сервере

Мы добавили к *DoExpensiveWork* следующие строки, записывающие в контекст вызова текущее число клиентов:

```
_CurrentConnections++;
...
CallContextData ccd = new CallContextData(_CurrentConnections-);
CallContext.SetData("CurrentConnections", ccd);
```

Затем это значение считывается прокси на клиентской стороне:

```
object oCurrentConnections =
    CallContext.GetData("CurrentConnections");
```

Изменения в прокси

Ниже приведен полный листинг класса *LoadBalancingProxy*:

```
public class LoadBalancingManager
{
    ArrayList _LoadBalancedServers = new ArrayList();
    public class LoadBalancedServer
    {
        public MarshalByRefObject Proxy;
        public int CurrentConnections;
        public bool IsActive;
    }

    public void AddProxy(MarshalByRefObject Proxy)
    {
        LoadBalancedServer lbs = new LoadBalancedServer();
        lbs.Proxy = Proxy;
        lbs.CurrentConnections = 0;
        lbs.IsActive = true;
        _LoadBalancedServers.Add(lbs);
    }

    public void SetCurrentConnections(MarshalByRefObject Proxy,
                                     int CurrentConnections)
    {
        foreach (LoadBalancedServer lbs in _LoadBalancedServers)
        {
            if (lbs.Proxy == Proxy)
```

```

        {
            lbs.CurrentConnections = CurrentConnections;
            return;
        }
    }

    public MarshalByRefObject GetLeastLoadedServer()
    {
        LoadBalancedServer LeastLoadedServer = null;
        foreach (LoadBalancedServer CurrentServer
            in _LoadBalancedServers)
        {
            if (LeastLoadedServer == null)
            {
                LeastLoadedServer = CurrentServer;
                continue;
            }
            if (CurrentServer.CurrentConnections <
                LeastLoadedServer.CurrentConnections)
            {
                LeastLoadedServer = CurrentServer;
            }
        }

        if (LeastLoadedServer == null)
        {
            return null;
        }
        else
        {
            return LeastLoadedServer.Proxy;
        }
    }
}

public class LoadBalancingProxy : RealProxy
{
    LoadBalancingManager LoadManager;
    int MaxProxies;

    public LoadBalancingProxy (Type type, string DiscoveryUrl)
        : base(type)
    {
        MarshalByRefObject ftp =
            (MarshalByRefObject)RemotingServices.Connect(type,

```

```

DiscoveryUrl,
null);

IDiscoveryServer discoveryServer = (
    IDiscoveryServer)ftp;
string[] PeerUrls = discoveryServer.GetPeerServerUrls();
LoadManager = new LoadBalancingManager();
foreach( string Url in PeerUrls)
{
    MarshalByRefObject PeerServer =
        (MarshalByRefObject)RemotingServices.Connect(
            type, Url, null);
    LoadManager.AddProxy(PeerServer);
}

public override IMessage Invoke(IMessage msg)
{
    MarshalByRefObject CurrentServer =
        LoadManager.GetLeastLoadedServer();
    if (CurrentServer == null)
    {
        throw new Exception(
            "No remote servers are responding at this time.");
    }

    RealProxy rp = RemotingServices.GetRealProxy(
        CurrentServer);
    IMessage retMsg = rp.Invoke(msg);
    if (retMsg is IMessageReturnMessage)
    {
        IMessageReturnMessage mrm = (
            IMessageReturnMessage)retMsg;
        if (mrm.Exception == null)
        {
            // Проверить контекст вызова.
            object oCurrentConnections =
                CallContext.GetData("CurrentConnections");
            if (oCurrentConnections == null)
            {
                // Так как сервер не возвратил никакой
                // информации о загрузке, считаем его
                // наименее загруженным.
                LoadManager.SetCurrentConnections(
                    CurrentServer, 0);
            }
            else if (oCurrentConnections is CallContextData)

```

```

        {
            LoadManager.SetCurrentConnections(
                CurrentServer,
                ((CallContextData)oCurrentConnections)
                    .CurrentConnections);
        }
        return retMsg;
    }
}
return retMsg;
}
}

```

Мы добавили класс *LoadBalancingManager*, реализующий наш алгоритм распределения нагрузки. Основная задача этого класса — поддержание приоритетного списка прокси в зависимости от числа клиентских соединений. Наш специализированный прокси для распределения каждого нового вызова метода будет вызывать *GetLeastLoadedServer*.

Контекст вызова в потоке сообщений

Данные контекста вызова добавляются к вызову метода, но не обязательно связаны с конкретным методом. Соответственно инфраструктура .NET Remoting пересылает наш объект *CallContextData* в заголовке, а не теле SOAP-сообщения. Вот ответное SOAP-сообщение для вызова *DoExpensiveWork*:

```

<SOAP-ENV:Header>
  <h4:__CallContext href="#ref-3"
    xmlns:h4="http://schemas.microsoft.com/
      clr/soap/messageProperties" SOAP-ENC:root="1">
    <a1:LogicalCallContext id="ref-3"
      xmlns:a1="http://schemas.microsoft.com/clr/ns/
        System.Runtime.Remoting.Messaging">
      <CurrentConnections href="#ref-6"/>
    </a1:LogicalCallContext>
    <a2:CallContextData id="ref-6"
      xmlns:a2="http://schemas.microsoft.com/clr/nsassem/
        LoadBalancing/DiscoveryServerLib%2C%20
        Version%3D1.0.871.14847%2C%20Culture%3Dneutral%2C%20
        PublicKeyToken%3Dnull">
      <_CurrentConnections>1</_CurrentConnections>
    </a2:CallContextData>
  </SOAP-ENV:Header>

```

```
<SOAP-ENV:Body>
  <i7:DoExpensiveOperationResponse id="ref-1"
    xmlns:i7="http://schemas.microsoft.com/clr/nsassem/
      LoadBalancing.IJobServer/LoadBalancingLib">
    <return>true</return>
  </i7:DoExpensiveOperationResponse>
</SOAP-ENV:Body>
```

Заключение

На этом мы закончим обсуждение сообщений и прокси. Изучив сообщения, мы использовали различные их типы в примерах реализации специализированных прокси. Так как сообщения являются основной единицей передачи данных в приложениях .NET Remoting, мы предполагаем интенсивно применять их в примерах к главам 6, 7 и 8. Мы рассмотрели специализированные прокси и их использование на клиентской стороне инфраструктуры .NET Remoting. Понимание работы прокси важно, так как они генерируют сообщения и представляют собой первый уровень, доступный клиентскому коду. Мы рассмотрели три класса, составляющих слой прокси в .NET Remoting, и показали, как создавать собственные прокси для перехвата активизации и обычных вызовов методов. Так как прокси представляют собой первый настраиваемый объект инфраструктуры .NET Remoting, они идеальны для перехвата процесса активизации и управления распределением вызовов удаленных методов. Контексты подробно рассматриваются в главе 6, после того как мы расскажем о приемниках сообщений.

Приемники сообщений и контексты

В этой главе мы продолжим рассказ о возможностях настройки .NET Remoting и рассмотрим архитектуру контекстов и приемников сообщений. Глубокое понимание особенностей работы контекстов и приемников сообщений позволит вам создавать более эффективные и мощные приложения .NET Remoting. Мы используем приемники сообщений и контексты для того, чтобы не допустить удаленного вызова метода в случае передачи клиентом неверных параметров и, таким образом, избежать ненужного трафика в сети. Также в этой главе мы коснемся трассировки сообщений и ведения журнала исключений, передаваемых через границы контекстов.

Приемники сообщений

Приемники сообщений являются одним из важнейших элементов, обеспечивающих удивительную гибкость архитектуры .NET Remoting. В главе 2 мы рассказывали, как инфраструктура .NET Remoting объединяет приемники сообщений в цепочки приемников, которые обрабатывают сообщения .NET Remoting и перемещают их через границы контекстов и доменов приложений. Приемники сообщений применяются в различных функциональных областях инфраструктуры .NET Remoting, включая следующие:

- контексты — о них пойдет речь в следующем разделе этой главы;
- каналы — им посвящена глава 7;
- форматировщики — о них рассказано в главе 8,

IMessageSink

Любой тип, реализующий интерфейс *IMessageSink*, можно использовать в архитектуре .NET Remoting в качестве приемника сообщений. Члены этого интерфейса перечислены в табл. 6-1.

Таблица 6-1, Члены

System.Runtime.Remoting.Messaging.IMessageSink

Имя	Тип	Описание
<i>NextSink</i>	Свойство только для чтения	Следующий приемник сообщений в цепочке или <i>null</i> , если это последний приемник
<i>AsyncProcessMessage</i>	Метод	Обрабатывает сообщение асинхронно
<i>SyncProcessMessage</i>	Метод	Обрабатывает сообщение синхронно

Ниже показан пример класса, реализующего *IMessageSink*:

```
public class PassThruMessageSink : IMessageSink
{
    IMessageSink _NextSink;

    public IMessageSink NextSink
    {
        get
        {
            return _NextSink;
        }
    }

    public PassThruMessageSink( IMessageSink next )
    {
        _NextSink = next;
    }

    public virtual IMessage SyncProcessMessage( IMessage msg )
    {
        try
        {
            return _NextSink.SyncProcessMessage( msg );
        }
        catch( System.Exception e )
        {
        }
    }
}
```

```

        return new ReturnMessage(e,
                                (IMethodCallMessage) msg);
    }

    public virtual IMessageCtrl AsyncProcessMessage(
        IMessage msg, IMessageSink replySink )
    {
        try
        {
            AsyncReplyHelperSink.AsyncReplyHelperSinkDelegate
            rsd = new AsyncReplyHelperSink.
                AsyncReplyHelperSinkDelegate(
                    this.AsyncProcessReplyMessage );

            replySink = (IMessageSink)newAsyncReplyHelperSink(
                replySink,
                rsd );

            // Передать сообщение следующему приемнику.
            return _NextSink.AsyncProcessMessage( msg,
                replySink );
        }
        catch(System.Exception e)
        {
            return null;
        }
    }

    // По завершении асинхронного вызова этот метод будет вызван
    // нашим вспомогательным классом и мы сможем обработать
    // возвращаемое сообщение.
    public IMessage AsyncProcessReplyMessage( IMessage msg )
    {
        // Обработать возвращаемое сообщение и вернуть его.
        return msg;
    }
} // Конец class PassThruMessageSink

```

Класс *PassThruMessageSink* представляет собой допустимую реализацию *IMessageSink* несмотря на то, что он просто передает *IMessage* следующему приемнику в цепочке. Тем не менее этот класс демонстрирует общий подход к реализации *IMessageSink*, поэтому рассмотрим его наиболее интересные методы подробнее.

Синхронная обработка сообщения

IMessageSink.SyncProcessMessage выглядит весьма просто. Как следует из названия, этот метод обрабатывает сообщение-запрос синхронно, передавая его методу *SyncProcessMessage* следующего приемника. Синхронная обработка завершается только тогда, когда инфраструктура .NET Remoting получает и возвращает сообщение-ответ, являющееся результатом вызова метода удаленного объекта, при этом *SyncProcessMessage* возвращает *IMessage*, инкапсулирующий возвращаемое значение и выходные параметры метода.

Если во время обработки сообщения возникает исключение, мы помещаем его в объект *ReturnMessage*, который и возвращаем. Данный случай отличается от возникновения исключения во время исполнения метода удаленного объекта, когда исключение, сгенерированное на серверной стороне, содержится в сообщении-ответе, возвращаемом инфраструктурой .NET Remoting,

Асинхронная обработка сообщения

Для обработки асинхронных вызовов методов инфраструктура .NET Remoting вызывает метод *IMessageSink.AsyncProcessMessage*. В отличие от *SyncProcessMessage*, этот метод не обрабатывает и сообщение-запрос, и сообщение-ответ. Вместо этого *AsyncProcessMessage* обрабатывает сообщение-запрос и возвращает управление. Позднее, когда выполнение асинхронной операции завершится, инфраструктура .NET Remoting передаст сообщение-ответ методу *IMessageSink.SyncProcessMessage* приемника, указанного вторым параметром метода *AsyncProcessMessage* — *replySink*. Если вам нужно обрабатывать сообщение-ответ при асинхронном вызове, то прежде чем передать сообщение-запрос дальше по цепочке, необходимо в начало цепочки *replySink* добавить свой приемник сообщений.

Для реализации *AsyncProcessMessage* мы создали вспомогательный класс *AsyncReplyHelperSink*, принимающий делегат, ссылающийся на метод обратного вызова, который будет вызван из *SyncProcessMessage* по получении сообщения-ответа:

```
//
// Универсальный класс AsyncReplyHelperSink - передает вызовы
// SyncProcessMessage делегату, заданному при вызове
```

```
// конструктора.
public class AsyncReplyHelperSink : IMessageSink
{
    // Определение делегата метода обратного вызова.
    public delegate IMessage AsyncReplyHelperSinkDelegate(
        IMessage msg);

    IMessageSink _NextSink;
    AsyncReplyHelperSinkDelegate _delegate;

    public System.Runtime.Remoting.Messaging.IMessageSink
        NextSink
    {
        get
        {
            return _NextSink;
        }
    }

    public AsyncReplyHelperSink( IMessageSink next,
        AsyncReplyHelperSinkDelegate d )
    {
        _NextSink = next;
        _delegate = d;
    }

    public virtual IMessage SyncProcessMessage (IMessage msg )
    {
        if ( _delegate != null )
        {
            // Уведомить делегат о поступлении ответа. Делегат
            // может изменить сообщение, поэтому далее по цепочке
            // передается результат работы делегата.
            IMessage msg2 = _delegate(msg);
            return NextSink.SyncProcessMessage(msg2);
        }
        else
        {
            return new ReturnMessage(
                new System.Exception(
                    "AsyncProcessMessage _delegate member is null!"),
                    (IMethodCallMessage)msg );
        }
    }

    public virtual IMessageCtrl AsyncProcessMessage (
```

```

        System.Runtime.Remoting.Messaging.IMessage msg ,
        System.Runtime.Remoting.Messaging.IMessageSink
                                replySink )
    {
        // Этот метод не должен вызываться при обработке
        // сообщения-ответа. Инфраструктура обрабатывает ответы
        // на асинхронные сообщения синхронно.
        // Видимо, кто-то попытался использовать данный класс
        // в цепочке приемников другого типа!
        return null;
    }
}

```

Класс *PassThruMessageSink* использует вспомогательный класс путем создания экземпляра делегата, ссылающегося на его метод *AsyncProcessReplyMessage*, передачи делегата новому экземпляру вспомогательного класса и добавления последнего в цепочку приемников ответа. Класс *AsyncReplyHelperSink* применяется в ряде примеров следующего раздела.

Контексты

В главе 2 мы ввели понятие контекста. Теперь пора рассмотреть роль контекстов в инфраструктуре .NET Remoting. Понимание контекстов поможет вам создавать более эффективные приложения .NET Remoting. В этом разделе мы расскажем, как настроить средства архитектуры контекстов, чтобы не допустить удаленного вызова метода в случае передачи клиентом неверных параметров и, таким образом, избежать ненужного обмена по сети. Также коснемся трассировки сообщений и ведения журнала исключений, передаваемых через границы контекстов.

Установление контекста

Архитектура контекстов в .NET Remoting включает в себя *контекстно-связанные* (context-bound) *объекты*, атрибуты контекста, свойства контекста и приемники сообщений. Если тип является производным от *System.ContextBoundObject*, то он контекстно-связанный, что требует от .NET Remoting создания особой среды или контекста, внутри которого будут исполняться экземпляры данного типа. В процессе активизации контекстно-связанного типа исполняющая среда выполняет следующую последовательность действий:

1. вызывает метод *IContextAttribute.IsContextOK* для каждого атрибута типа;
2. если какой-либо из атрибутов сообщает, что данный контекст не подходит для активизации, то для каждого атрибута вызывается *IContextAttribute.GetPropertiesForNewContext*, которому передается сообщение вызова конструктора активизируемого типа;
3. создает контекст;
4. с помощью вызова метода *IContextProperty.Freeze* уведомляет каждое свойство контекста о том, что он заморожен;
5. проверяет допустимость нового контекста, вызывая метод *IContextProperty.IsNewContextOK* для каждого свойства контекста;
6. активизирует объект в новом контексте, передавая сообщение вызова конструктора методу *RealProxy.Invoke* объекта-прокси.

Атрибуты и свойства контекста

Для определения и установления контекста контекстно-связанному типу присваивается один или несколько атрибутов, реализующих интерфейс *IContextAttribute*. Методы этого интерфейса перечислены в табл. 6-2.

Таблица 6-2. Члены

System.Runtime.Remoting.Contexts.IContextAttribute

Имя	Описание
<i>IsContextOK</i>	Исполняющая среда вызывает этот метод для того, чтобы определить, подходит ли текущий контекст для активизации типа, снабженного атрибутом
<i>GetPropertiesForNewContext</i>	Исполняющая среда вызывает этот метод после того, как атрибут подтвердил годность текущего контекста для активизации типа, снабженного этим атрибутом

Альтернативно можно создать тип, производный от *System.ContextAttribute*, который в свою очередь является производным от *System.Attribute* и предоставляет стандартную реализацию */Con-*

textAttribute. *System.ContextAttribute* также реализует *IContextProperty*, о котором мы поговорим далее.

Атрибут контекста участвует в процессе активизации и выполняет следующие функции:

- сообщает инфраструктуре .NET Remoting о том, соответствует ли текущий контекст условиям, требуемым для исполнения типа, снабженного атрибутом;
- * добавляет в контекст одно или несколько свойств, предоставляющих сервисы и/или поддерживающие условия для исполнения типа, снабженного атрибутом.

В процессе активизации типа, производного от *ContextBoundObject*, исполняющая среда вызывает метод *IContextAttribute.IsContextOK* всех атрибутов контекста, связанных с данным типом, для того чтобы определить, поддерживает ли текущий контекст условия, требуемые для исполнения типа. Атрибут указывает на неприемлемость контекста, возвращая *false* из *IsContextOK*.

Если *IContextAttribute.IsContextOK* какого-либо из атрибутов типа, производного от *ContextBoundObject*, возвращает *false*, то исполняющая среда создаст новый контекст для данного объекта и для всех атрибутов вызовет *IContextAttribute.GetPropertiesForNewContext*. Это позволяет атрибуту добавить одно или несколько свойств контекста, обеспечивающих условия, требуемые для исполнения объектов типа. Свойства контекста реализуют интерфейс *IContextProperty*, члены которого перечислены в табл. 6-3.

Таблица 6-3. Члены
System.Runtime.Remoting.Contexts.IContextProperty

Имя	Тип	Описание
<i>Name</i>	Свойство только для чтения	Название свойства. Используется в качестве ключа в наборе свойств контекста
<i>Freeze</i>	Метод	Инфраструктура .NET Remoting вызывает этот метод после того, как все атрибуты добавили свойства в контекст. После замораживания контекста инфраструктура .NET Remoting не позволяет добавлять в него новые свойства

см. след. стр.

Таблица fr-3. (окончание)

Имя	Тип	Описание
<i>IsNewContextOK</i>	Метод	Исполняющая среда вызывает этот метод после замораживания контекста, давая каждому свойству контекста возможность прервать создание контекста

Предположим, что нам нужно создать контекст, предоставляющий всем своим объектам средство ведения журнала. То есть любой код, исполняемый внутри такого контекста, сможет получить из свойств контекста объект-регистратор и с помощью метода этого объекта записать сообщение в журнал. Для реализации контекста с журналом нам понадобится реализовать атрибут контекста, добавляющий в контекст свойство, которое представляет сервис ведения журнала. Следующий фрагмент кода определяет атрибут контекста под названием *ContextLogAttribute*:

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

[AttributeUsage(AttributeTargets.Class)]
class ContextLogAttribute : Attribute,
                           IContextAttribute
{
    public bool IsContextOK(
        System.Runtime.Remoting.Contexts.Context ctx,
        System.Runtime.Remoting.Activation.
        IConstructionCallMessage msg)
    {
        // Требуется создания нового контекста.
        return false;
    }

    public void GetPropertiesForNewContext(
        System.Runtime.Remoting.Activation.
        IConstructionCallMessage msg)
    {
        // Добавить наше свойство в контекст.
        msg.ContextProperties.Add(new ContextLogProperty());
    }
}
```

В процессе активизации типа, производного от *ContextBoundObject* и снабженного атрибутом *ContextLogAttribute*, инфраструктура .NET Remoting вызовет *ContextLogAttribute*, передавая ему

IConstructionCallMessage. В этот момент объект еще не создан. Следовательно, конструктор объекта еще не вызывался. *GetPropertiesForNewContext* добавляет к члену *ContextProperties* интерфейса *IConstructionCallMessage* новый экземпляр класса *ContextLogProperty*, определенного ниже:

```
[AttributeUsage(AttributeTargets.Class)]
class ContextLogProperty : Attribute,
                          IContextProperty
{
    public void Freeze(
        System.Runtime.Remoting.Contexts.Context newContext)
    {
        // Добавление новых свойств с этого
        // момента не допускается.
    }

    public bool IsNewContextOK(
        System.Runtime.Remoting.Contexts.Context newCtx)
    {
        // Здесь также можно проверить другие свойства контекста
        // на отсутствие конфликтов, но в этом примере мы
        // просто возвращаем OK.
        return true;
    }

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

    public void LogMessage(string msg)
    {
        Console.WriteLine(msg);
    }
}
```

Класс *ContextLogProperty* реализует средства ведения журнала. Экземпляр данного класса — это свойство контекста, поэтому все объекты данного контекста могут получить экземпляр этого свойства и использовать его для вывода сообщений:

```
[ContextLogAttribute()]
class MyContextBoundObject : System.ContextBoundObject
{
    public void Foo()
    {
        Context ctx = Thread.CurrentContext;
        ContextLogProperty lg =
            (ContextLogProperty)ctx.GetProperty(
                "ContextLogProperty");
        lg.LogMessage("In MyContextBoundObject.Foo()");
    }
}
```

Контексты и удаленное взаимодействие

Как вы помните, контекст образует границу .NET Remoting вокруг находящихся в нем объектов. На рис. 6-1 показано, как инфраструктура .NET Remoting изолирует объекты внутри контекста от объектов за его пределами с помощью особого типа канала, называемого *межконтекстным каналом* (cross-context channel), и четырех цепочек приемников сообщений, которые отделяют обработку входящих сообщений от обработки исходящих сообщений.

Все вызовы методов поступают внутрь контекста как сообщения .NET Remoting, которые проходят через межконтекстный канал, *общеконтекстную цепочку приемников серверного контекста* (server context sink chain), *цепочку приемников серверного объекта* (server object sink chain) и *приемник стекопостроителя* (stackbuilder sink). Все вызовы методов объектов, находящихся за пределами контекста, выходят из него как сообщения .NET Remoting, которые проходят через прокси и связанную с ним *цепочку агентских приемников* (envoy sink chain), *общеконтекстную цепочку приемников клиентского контекста* (client context sink chain) и канал (межконтекстный или *междоменный*). Каждая цепочка приемников состоит из нуля или нескольких пользовательских приемников сообщений, за которыми следует особый *терминаторный* приемник сообщений, реализуемый .NET Remoting.

Как видно из табл. 6-4, выбор цепочки для нового приемника сообщений во многом определяется тем, в какой момент и в каком месте должна обеспечиваться реализация необходимого поведения контекста для входящих и исходящих сообщений вызовов методов.

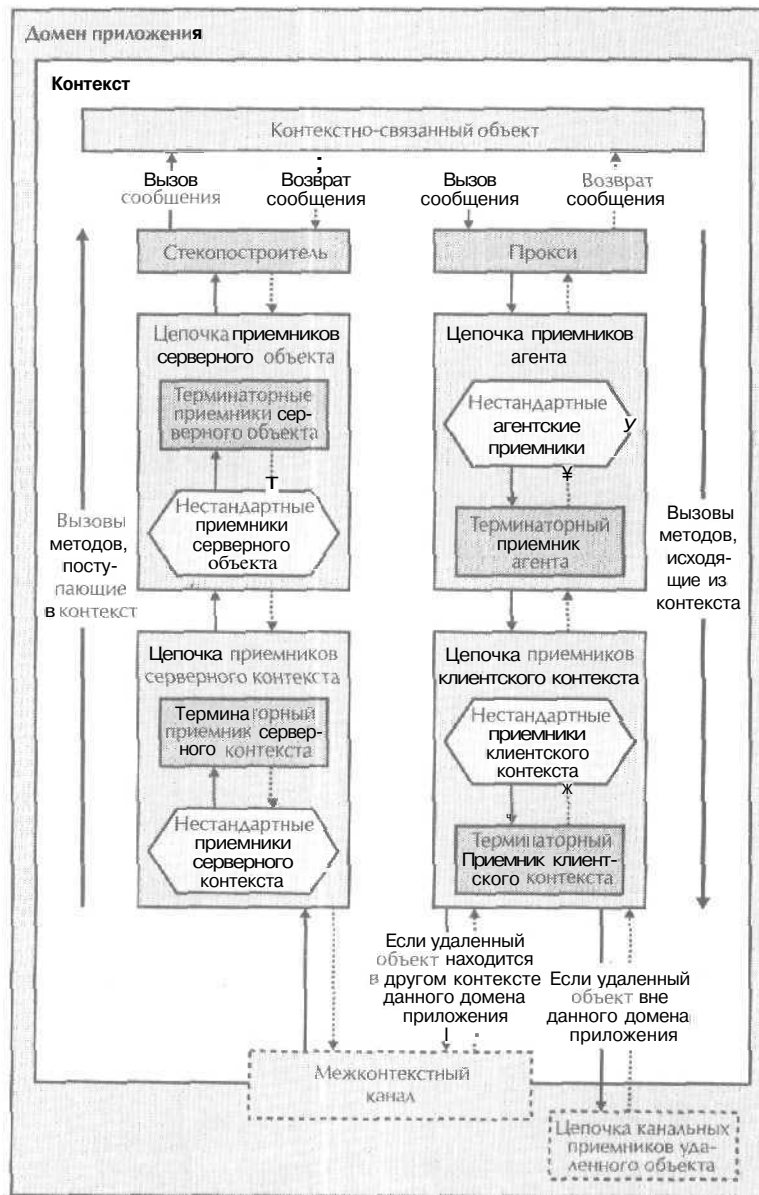


Рис. 6-1. Цепочки приемников сообщений, изолирующие экземпляр *ContextBoundObject* от объектов за пределами контекста

Таблица. 6-4. Цепочки приемников контекста

Цепочка	Перехватывает	Использование
Клиентского контекста	Вызовы методов любых объектов вне контекста, выполняемые любым объектом внутри контекста	<ul style="list-style-type: none"> Ведение журнала всех исходящих из контекста сообщений вызовов методов Синхронизация Защита Транзакции
Серверного контекста	Вызовы методов любых объектов внутри контекста, выполняемые любыми объектами вне контекста	<ul style="list-style-type: none"> Ведение журнала всех поступающих извне контекста сообщений вызовов методов Синхронизация Защита Транзакции
Серверного объекта	Вызовы методов конкретного контекстно-связанного объекта	<ul style="list-style-type: none"> Перехват сообщений вызовов методов конкретного экземпляра объекта Может использоваться совместно с агентским приемником для доставки информации из клиентского в серверный контекст
Агентская	Вызовы со стороны клиента контекстно-связанного объекта. Цепочка агентских приемников исполняется в контексте клиента	<ul style="list-style-type: none"> Проверка аргументов метода перед отправкой сообщения вызова метода от клиента серверу Другие виды оптимизации методов, которые удобнее реализовать в контексте клиента

Прежде чем мы подробно обсудим цепочки приемников контекста, рассмотрим еще один тип приемников, используемых вместе с контекстами, но не входящих в цепочки приемников: динамические контекстные приемники.

Динамические контекстные приемники

Инфраструктура .NET Remoting позволяет во время исполнения программно вводить в архитектуру обработки сообщений контекста экземпляры типов, реализующих *IDynamicMessageSink*. Дина-

мические приемники не считаются «настоящими» приемниками сообщений, так как они не реализуют *IMessageSink* и не объединяются в цепочки. С помощью динамических приемников вы можете перехватывать сообщения вызовов методов в различных точках их обработки.

Создание динамического приемника

Для создания собственного динамического контекстного приемника необходимо выполнить следующие действия:

- определить класс динамического приемника сообщений, реализующий интерфейс *IDynamicMessageSink*;
- определить класс динамического свойства, реализующий интерфейсы *IDynamicProperty* и *IContributeDynamicSink*;
- реализовать *IContributeDynamicSink.GetDynamicSink* таким образом, чтобы возвращать экземпляр класса динамических контекстных приемников;
- программно регистрировать и отменять регистрацию динамического свойства, используя методы *Context.RegisterDynamicProperty* и *Context.UnregisterDynamicProperty* соответственно.

В табл. 6-5 показано, как параметры метода *Context.RegisterDynamicProperty* влияют на набор перехватываемых вызовов. Прежде чем выполнить вызов метода, исполняющая среда вызывает метод *IDynamicMessageSink.ProcessMessageStart* каждого зарегистрированного динамического приемника на данном уровне перехвата, передавая *IMessage*, представляющий текущий вызов. Аналогично после окончания обработки вызова метода исполняющая среда вызывает *IDynamicMessageSink.ProcessMessageFinish*, которому передается *IMessage*, представляющий сообщение-ответ.

Таблица 6-5. *Context.RegisterDynamicProperty* и перехватываемые вызовы

Параметр <i>ContextBoundObject</i>	Параметр <i>Context</i>	Уровень перехвата
Ссылка на прокси	<i>null</i>	Перехватывает все вызовы методов через данный прокси

см, след. стр.

Таблица 6-5. (окончание)

Параметр <i>ContextBoundObject</i>	Параметр <i>Context</i>	Уровень перехвата
Ссылка на реальный объект	<i>null</i>	Перехватывает все вызовы методов реального объекта
<i>null</i>	Ссылка на контекст	Перехватывает всех входящие и исходящие вызовы методов для контекста
<i>null</i>	<i>null</i>	Перехватывает всех входящие и исходящие вызовы методов всех контекстов домена приложения

Клиентская контекстная цепочка

Как видно из рис. 6-1, клиентская контекстная цепочка — это последняя цепочка приемников, через которую проходят сообщения, исходящие из контекста. Однако на рисунке не показано, что она является общеконтекстной и через нее проходят все исходящие вызовы методов, выполняемые любыми объектами данного контекста. Добавив приемник в эту цепочку, можно реализовать некоторую обработку всех исходящих из контекста вызовов методов.

Инфраструктура .NET Remoting добавляет пользовательские приемники в начало клиентской контекстной цепочки, так чтобы последним приемником в ней всегда был *System.Runtime.Remoting.Contexts.ClientContextTerminatorSink*. Любой домен приложения содержит единственный экземпляр этого класса, выполняющий следующие функции для всех контекстов домена:

1. уведомление динамических приемников о начале вызова метода, выходящего за пределы контекста;
2. передачу сообщений *IConstructionCallMessage* всем свойствам контекста, реализующим *IContextActivatorProperty*, перед вызовом метода *Activator.Activate* при обработке сообщения активизации экземпляра удаленного объекта, а также передачу всем таким свойствам контекста *IConstructionReturnMessage* после возврата из *Activator.Activate*. Это позволяет свойствам контекста модифицировать сообщения вызова и возврата конструктора;

3. передачу сообщений, отличных от *IConstructorCallMessage*, либо по *CrossContextChannel*, либо по каналу, выходящему за пределы домена приложения, такому, как *HttpChannel* или *TcpChannel*;
4. уведомление динамических приемников о завершении вызова метода, выходящего за пределы контекста (при асинхронных вызовах данный этап выполняется *AsyncReplySink* в момент фактического завершения вызова).

Создание приемника для клиентской контекстной цепочки

При создании приемника для клиентской контекстной цепочки необходимо выполнить следующие действия:

- создать приемник сообщения, реализующий обработку, которая выполняется для вызовов методов, исходящих из контекста;
- определить свойство контекста, реализующее интерфейс *IContributeClientContextSink*;
- определить атрибут контекста, который при вызове *IContextAttribute.GetPropertiesForNewContext* во время обработки сообщения вызова конструктора добавляет созданное ранее свойство к свойствам контекста;
- добавить атрибут контекста к объявлению класса.

Серверная контекстная цепочка приемников

Дополнением клиентской контекстной цепочки является серверная контекстная цепочка. Как показано на рис. 6-1, серверная контекстная цепочка — это первая цепочка приемников, которую проходят поступающие в контекст сообщения. Аналогично клиентской контекстной цепочке, эта цепочка является общеконтекстной, и через нее проходят все вызовы объектов контекста, поступающие из-за его пределов. Добавив приемник в эту цепочку, можно реализовать некоторую обработку всех входящих вызовов методов.

Инфраструктура .NET Remoting строит серверную контекстную цепочку, просматривая свойства контекста в порядке, обратном построению клиентской контекстной цепочки. Таким образом обеспечивается симметричный порядок выполнения операций в

обеих цепочках для тех свойств контекста, которые добавляют приемники в обе цепочки.

Инфраструктура .NET Remoting добавляет пользовательские приемники в начало серверной контекстной цепочки так, чтобы последним приемником в ней всегда был *System.Runtime.Remoting.Contexts.ServerContextTerminatorSink*. Любой домен приложения содержит единственный экземпляр этого класса, выполняющий следующие функции для всех контекстов домена:

- передачу сообщений *IConstructionCallMessage* всем свойствам контекста, реализующим *IContextActivatorProperty*, перед вызовом метода *Activator.Activate* для сообщения, активизирующего экземпляр серверного объекта, а также передачу таким свойствам сообщения *IConstructionReturnMessage* после возврата из *Activator.Activate*. Это позволяет свойствам контекста модифицировать сообщения вызова и возврата конструктора;
- передачу сообщений, отличных от *IConstructionCallMessage*, цепочке серверных приемников для целевого объекта, указанного в сообщении.

Создание приемника для серверной контекстной цепочки

При создании приемника для серверной контекстной цепочки необходимо выполнить следующие действия:

- создать приемник сообщений, реализующий операции, которые должны выполняться для всех вызовов методов из-за пределов контекста;
- определить свойство контекста, реализующее интерфейс *IContributeServerContextSink*;
- определить атрибут контекста, который при вызове *IContextAttribute.GetPropertiesForNewContext* во время обработки сообщения вызова конструктора добавляет созданное выше свойство к свойствам контекста;
- добавить атрибут контекста к объявлению класса.

Пример: контекст с регистрацией исключений

Для демонстрации использования клиентской и серверной контекстной цепочек реализуем контекст с регистрацией исключений, который выполняет следующие действия:

- регистрирует все исключения, сгенерированные в результате вызовов методов объектов контекста из-за его пределов;
- регистрирует все исключения, сгенерированные в результате вызовов объектами контекста методов объектов, находящихся за его пределами.

Сначала определим класс приемников сообщений, проверяющий все поступающие к нему сообщения на наличие в них исключений:

```
public class ExceptionLoggingMessageSink : IMessageSink
{
    IMessageSink _NextSink;
    static FileStream _stream;

    public IMessageSink NextSink
    {
        get
        {
            return _NextSink;
        }
    }

    public ExceptionLoggingMessageSink( IMessageSink next,
                                       string filename )
    {
        Trace.WriteLine("ExceptionLoggingMessageSink ctor");
        _NextSink = next;
        try
        {
            lock(this)
            {
                if ( _stream == null )
                {
                    _stream = new FileStream( filename,
                                             FileMode.Append );
                }
            }
        }
        catch( System.IO.IOException e )
        {
        }
    }
}
```

```

        Trace.WriteLine(e.Message);
    }
}

```

Итак, мы приступили к созданию класса *ExceptionLoggingMessageSink*. Конструктор обеспечивает подключение экземпляра к цепочке приемников, принимая ссылку на следующий приемник в цепочке. Конструктору также передается имя файла, которое используется для открытия *FileStream*, куда будет записываться информация об исключениях.

Далее необходимо реализовать метод *IMessageSink.SyncProcessMessage*, который передает сообщение следующему приемнику в цепочке и использует вспомогательную функцию для проверки возвращаемого сообщения:

```

public IMessage SyncProcessMessage(IMessage msg)
{
    try
    {
        // Передать следующему приемнику.
        IMessage retmsg = _NextSink.SyncProcessMessage(msg);

        // Проверить возвращаемое сообщение и зарегистрировать
        // исключение.
        InspectReturnMessageAndLogException(retmsg);

        return retmsg;
    }
    catch(System.Exception e)
    {
        return null;
    }
}

void InspectReturnMessageAndLogException(IMessage retmsg)
{
    MethodReturnMessageWrapper mrm =
        new MethodReturnMessageWrapper((IMethodReturnMessage)
                                         retmsg);

    if (mrm.Exception != null )
    {
        lock(_stream)
        {
            Exception e = mrm.Exception;

```

```

StreamWriter w = new StreamWriter(_stream,
                                   Encoding.ASCII);

w.WriteLine();
w.WriteLine("=====");
w.WriteLine();
w.WriteLine(String.Format("Exception: {0}",
                           DateTime.Now.ToString() ));
w.WriteLine(String.Format("Application Name: {0}",
                           e.Source));
w.WriteLine(String.Format("Method Name: {0}",
                           e.TargetSite.ToString()));
w.WriteLine(String.Format("Description: {0}",
                           e.Message));
w.WriteLine(String.Format("More Info: {0}",
                           e.ToString()));

w.Flush();
}
}

```

Метод *InspectReturnMessageAndLogException* помещает сообщение в экземпляр класса *System.Runtime.Remoting.Messaging.MethodReturnMessageWrapper*, который упрощает доступ к свойствам возвращаемого сообщения. Далее проверяется свойство *Exception* сообщения, которое будет иметь значение отличное от *null*, если в сообщении содержится исключение. Так как данный приемник задействован в общеконтекстной цепочке и его код может исполняться параллельно несколькими потоками, то перед записью в файл мы его блокируем.

Теперь реализуем метод *IMessageSink.AsyncProcessMessage*:

```

public IMessageCtrl AsyncProcessMessage(IMessage msg,
                                       IMessageSink replySink )
{
    try
    {
        //
        // Создать приемник ответа с делегатом,
        // ссылающимся на наш метод обратного вызова.
        AsyncReplyHelperSink.AsyncReplyHelperSinkDelegate
            rsd = new AsyncReplyHelperSink,
            AsyncReplyHelperSinkDelegate(
                this.AsyncProcessReplyMessage);

        // Нам необходимо трассировать ответ, когда он будет
    }
}

```

```

        // получен, поэтому добавим приемник к цепочке
        // ответа.
        replySink =
            (IMessageSink) new AsyncReplyHelperSink(
                replySink, rsd );

        return _NextSink.AsyncProcessMessage( msg,
                                                replySink );
    }
    catch(System.Exception e)
    {
        return null;
    }
}

//
// Трассировка и возврат ответного сообщения,
public IMessage AsyncProcessReplyMessage( IMessage msg )
{
    // Проверить ответное сообщение и зарегистрировать
    // исключение.
    Inspect ReturnMessageAndLogException(msg);
    return msg;
}
} // Конец class ExceptionLoggingMessageSink

```

Для асинхронной обработки сообщений *AsyncProcessMessage* использует определенный нами ранее *AsyncReplyHelperSink*. Добавив к цепочке приемников ответа экземпляр *AsyncReplyHelperSink* с делегатом, ссылающимся на метод *ExceptionLoggingMessageSink.AsyncProcessReplyMessage*, мы реализуем проверку ответных сообщений при асинхронных вызовах,

Теперь, когда мы закончили создание приемника, необходимо подключить его экземпляры к клиентской и серверной контекстным цепочкам приемников. Для этого создадим свойство контекста, реализующее интерфейсы *IContributeClientContextSink* и *IContributeServerContextSink*:

```

[Serializable]
public class ExceptionLoggingProperty : IContextProperty,
                                       IContributeClientContextSink,
                                       IContributeServerContextSink
{
    private string _Name;

```

```

IMessageSink _ServerSink;
IMessageSink _ClientSink;
string _FileName;

public ExceptionLoggingProperty(string name,
                                string FileName)
{
    _Name = name;
    _FileName = filename;
}

public void Freeze ( System.Runtime.Remoting.Contexts.
                    Context newContext )
{
    // Добавление новых свойств с этого момента
    // не допускается,
}

public System.Boolean IsNewContextOK (Context newCtx )
{
    return true;
}

public string Name
{
    get
    {
        return _Name;
    }
}

public IMessageSink GetClientContextSink (IMessageSink
                                           nextSink )
{
    Console.WriteLine( "GetClientContextSink()");
    lock(this)
    {
        if ( _ClientSink == null )
        {
            _ClientSink =
                new ExceptionLoggingMessageSink(nextSink,
                                                _FileName);
        }
    }
    return _ClientSink;
}

```

```

public IMessageSink GetServerContextSink (IMessageSink
                                         nextSink )
{
    Console.WriteLine("GetServerContextSink()");
    lock(this)
    {
        if ( _ServerSink == null )
        {
            _ServerSink =
                new ExceptionLoggingMessageSink(nextSink,
                                                _FileName);
        }
        return _ServerSink;
    }
} // Конец class ExceptionLoggingProperty

```

Класс *ExceptionLoggingProperty* реализует интерфейс *IContextProperty*, который мы рассматривали с начала этого раздела. Наиболее интересны здесь методы *IContributeServerContextSink.GetServerContextSink* и *IContributeClientContextSink.GetClientContextSink*, каждый из которых создает и возвращает экземпляр **класса** *ExceptionLoggingMessageSink*, определенный нами ранее. Исполняющая **среда** вызывает метод **GetServerContextSink** в процессе активизации объекта внутри контекста. Аналогично **GetClientContextSink** вызывается исполняющей средой при первом вызове метода **объекта**, находящегося за пределами **контекста**.

ПРЕДУПРЕЖДЕНИЕ Эти методы возвращают разные экземпляры *ExceptionLoggingMessageSink*. Если вы попытаетесь использовать один и тот же экземпляр приемника и в серверной, и в клиентской контекстной цепочке, то столкнетесь с проблемой, которая возникает, когда в результате обработки входящего вызова метода происходит *исходящий* вызов. Если экземпляр *ExceptionLoggingMessageSink*, включенный в серверную контекстную цепочку, включен также и в клиентскую, то возникает *цикл*, не дающий сообщению покинуть контекст в результате *бесконечной* рекурсии. Следовательно, для каж-

гой цепочки необходимо создать собственный экземпляр приемника, как, собственно, мы и поступили.

ПРИМЕЧАНИЕ Все свойства контекста для типа создаются в контексте клиента во время активизации до того, как сообщение-конструктор передано удаленному объекту. Поэтому следует избегать в конструкторах свойств контекста кода, для исполнения которого требуется контекст удаленного объекта.

В нашем примере можно создать *FileStream* в *ExceptionLoggingProperty* и затем передать этот поток конструктору приемника сообщений. Однако в распределенном приложении, когда клиент и сервер находятся в разных приложениях, это вызвало бы проблемы. В этом случае файлом журнала исключений владело бы клиентское приложение, так как поток на этом файле открывался бы в *ExceptionLoggingProperty*. Однако приемники сообщений в серверном приложении не получили бы доступа к потоку в удаленном приложении.

Теперь определим атрибут контекста, добавляющий к нему свойство *ExceptionLoggingProperty*:

```
[AttributeUsage(AttributeTargets.Class)]
public class ExceptionLoggingContextAttribute :
    ContextAttribute
{
    string _FileName;

    public ExceptionLoggingContextAttribute(string filepath) :
        base("ExceptionLoggingContextAttribute")
    {
        _FileName = filepath;
    }

    public override void GetPropertiesForNewContext (
        IConstructionCallMessage msg )
    {
        // Добавить наше свойство к свойствам контекста.
    }
}
```

```

        msg.ContextProperties.Add(
            new ExceptionLoggingProperty( this.AttributeName,
                                          _FileName));
    }

    public override System.Boolean IsContextOK (
        Context ctx, IConstructionCallMessage msg )
    {
        // Есть ли уже в контексте
        // данное свойство?
        return (ctx.GetProperty( this.AttributeName ) != null);
    }
} // Конец class ExceptionLoggingContextAttribute

```

Класс *ExceptionLoggingContextAttribute* следует общему шаблону реализации атрибутов контекста, будучи производным от *ContextAttribute* и переопределяя методы *IsContextOK* и *GetPropertiesForNewContext*, которые соответственно проверяют наличие свойства *ExceptionLoggingContextProperty* и добавляют его.

Теперь мы можем добавить атрибут *ExceptionLoggingContextAttribute* к любому классу, производному от *ContextBoundObject*, например, так:

```

[ ExceptionLoggingContextAttribute(@"C:\exceptions.log") ]
public class SomeCBO : ContextBoundObject
{

```

!

ExceptionLoggingMessageSink будет записывать в файл C:\exceptions.log информацию обо всех исключениях, сгенерированных в результате вызовов, сделанных объектами за пределами контекста экземпляра *SomeCBO*, а также в результате вызовов объектов за пределами контекста, выполненных экземпляром этого класса. Пример фрагмента журнала показан на рис. 6-2.

Серверная объектная цепочка приемников

Как говорилось ранее, и серверная, и клиентская контекстные цепочки перехватывают сообщения на уровне всего контекста. Если поведение контекста изменяется в зависимости от конкретного экземпляра объекта, то необходимо добавить приемник сообщений в серверную объектную цепочку, что позволит перехватывать сообщения, представляющие вызовы методов контекст-

```

exceptions.txt - Notepad
File Edit Format View Help

-----
Exception: 7/27/2001 10:08:16 PM
Application Name: Sink
Method Name: Sink.MyDiv(Int32, Int32)
Description: Attempted to divide by zero.
More Info: System.DivideByZeroException: Attempted to divide by zero.
   at Sink.MyDiv(Int32, Int32) in c:\dev\remotingbook\src\sink\MessageSink.cs:line 124
   at System.Runtime.Remoting.Messaging.StackBuilderSink.PrivateProcessMessage(MethodBase mb,
Object[] args, Object server, Int32 methodPtr, Boolean fExecuteInContext, Object[]& outArgs)
   at System.Runtime.Remoting.Messaging.StackBuilderSink.SyncProcessMessage(IMessage msg, Int32
methodPtr, Boolean fExecuteInContext)

-----
Exception: 7/28/2001 1:43:26 PM
Application Name: Sink
Method Name: Sink.Foo()
Description: Object reference not set to an instance of an object.
More Info: System.NullReferenceException: Object reference not set to an instance of an object.
   at Sink.Foo() in c:\dev\remotingbook\src\sink\Sink.cs:line 149
   at System.Runtime.Remoting.Messaging.StackBuilderSink.PrivateProcessMessage(MethodBase mb,
Object[] args, Object server, Int32 methodPtr, Boolean fExecuteInContext, Object[]& outArgs)
   at System.Runtime.Remoting.Messaging.StackBuilderSink.SyncProcessMessage(IMessage msg, Int32
methodPtr, Boolean fExecuteInContext)

```

Рис. 6-2. Результаты работы класса *ExceptionLoggingMessageSink*

но-связанного объекта из-за пределов контекста. Однако цепочки приемников конкретного экземпляра объекта для перехвата вызовов за пределы контекста, выполняемых объектом внутри контекста, не существует.

Инфраструктура .NET Remoting добавляет пользовательские приемники в начало серверной объектной цепочки, так чтобы последним приемником в ней всегда оказывался *System.Runtime.Remoting.Contexts.ServerObjectTerminatorSink*. Любой домен приложения содержит единственный экземпляр этого класса, выполняющий следующие функции для всех контекстов домена:

- уведомляет связанные с данным экземпляром объекта динамические приемники о начале обработки вызова метода объекта;
- передает сообщение приемнику *StackBuilderSink*, в результате чего происходит вызов метода объекта;
- уведомляет динамические приемники о завершении обработки вызова метода объекта (при асинхронных вызовах данный этап выполняется *AsyncReplySink* в момент фактического завершения вызова).

Создание серверного объектного приемника

Для создания серверного объектного приемника необходимо выполнить следующие действия:

- реализовать приемник сообщений, выполняющий определяемые контекстом действия для всех вызовов методов экземпляра объекта, находящегося внутри контекста;
- задать свойство контекста, реализующее интерфейс *IContributeObjectSink*;
- определить атрибут контекста, который при вызове *ContextAttribute.GetPropertiesForNewContext* во время обработки сообщения вызова конструктора добавляет созданное выше свойство к свойствам контекста;
- добавить атрибут контекста к объявлению класса.

Пример: трассировка всех вызовов методов объекта

Ранее мы разработали свойство контекста, предоставляющее всем объектам контекста возможность выводить сообщения в журнал. В том примере экземпляр объекта мог получить из контекста это свойство и вызвать для регистрации сообщения его метод *LogMessage*. Недостаток такого решения — необходимость добавления кода записи в журнал в каждый метод класса. В противоположность этому возможности по перехвату, предоставляемые приемниками контекста, позволяют реализовать функциональность регистрации вызовов методов в виде сервиса трассировки, обрабатывающего все вызовы методов экземпляра объекта и не требующего явного использования функций трассировки в этих методах. Создание приемника сообщений, выполняющего трассировку, и добавление его к серверной объектной цепочке приемников представляет собой более удобный способ реализации трассировки вызовов методов.

Сначала нам нужно создать приемник сообщений, реализующий трассировку. Вот так реализован класс, выводящий диагностическую информацию для всех получаемых им сообщений:

```
public class TraceMessageSink : IMessageSink
{
    IMessageSink _NextSink;
```

```

public IMessageSink NextSink
{
    get
    {
        return _NextSink;
    }
}

public TraceMessageSink( IMessageSink next )
{
    _NextSink = next;
}

public virtual IMessage SyncProcessMessage ( IMessage msg )
{
    try
    {
        TraceMessage(msg);
        IMessage msgRet = _NextSink.SyncProcessMessage(msg);
        TraceMessage(msgRet);
        return msgRet;
    }
    catch(System.Exception e)
    {
        return new ReturnMessage(e, (IMethodCallMessage)
                                msg);
    }
}

```

Теперь мы реализуем класс *TraceMessageSink*. Метод *SyncProcessMessage* использует следующую вспомогательную функцию, выполняющую фактическую трассировку сообщения:

```

void TraceMessage(IMessage msg)
{
    Trace.WriteLine
    ( "-----" );

    Trace.WriteLine( String.Format(
        "{0} :: TraceMessage() - {1}",
        DateTime.Now.Ticks.ToString(),
        msg.GetType().ToString() );

    Trace.WriteLine( String.Format("\tDomainID={0},
        ContextID={1}",
        Thread.GetDomainID(),

```

```

Thread. CurrentContext.ContextID.ToString()));

IDictionaryEnumerator ie =
    msg.Properties.GetEnumerator();
while(ie.MoveNext())
{
    Trace.WriteLine( String.Format("\tMsg[{0}] = {1}",
                                   ie.Key, ie.Value));

    if ( ie.Value == null )
    {
        continue;
    }

    // Если это массив, то вывести его элементы,
    if C ! ie.Value.GetType().IsArray )
    {
        continue;
    }

    object[] ar = (object[])ie.Value;
    for(int i = 0; i<ar.Length; ++i )
    {
        Trace.WriteLine( String.Format("\t\t[{0}] =
                                   {1}", i, ar.GetValue(i)));
    }
}
}
}

```

Метод *TraceMessage* просто проходит по всем свойствам сообщения, выводя каждую пару «ключ — значение» с помощью метода *Trace.WriteLine*, который отображает результаты в окне Output отладчика.

В завершение реализации *TraceMessageSink* добавим следующее определение метода *AsyncProcessMessage*:

```

public virtual IMessageCtrl AsyncProcessMessage (
    IMessage msg, IMessageSink replySink )
{
    try
    {
        // Трассировать сообщение перед передачей далее
        // по цепочке.
        TraceMessage(msg);
    }
}

```



```

{
    // Реализация IContextProperty для краткости не показана,
    public IMessageSink GetObjectSink ( MarshalByRefObject obj,
                                         IMessageSink nextSink )
    {
        return new TraceMessageSink(nextSink);
    }
}
!

```

Здесь нет ничего нового: метод *GetObjectSink* просто возвращает экземпляр *TraceMessageSink*. Следует отметить, что одним из параметров *GetObjectSink* является *MarshalByRefObject*. Инфраструктура .NET Remoting связывает приемник сообщений, возвращаемый *GetObjectSink*, с объектом, на который ссылается параметр *obj*. Хотя здесь этот параметр игнорируется, вы можете использовать его для получения дополнительной информации.

Теперь нам осталось только определить атрибут, добавляющий *TraceMessageSinkProperty* к свойствам контекста сообщения вызова конструктора в методе *GetPropertiesForNewContext*:

```

[AttributeUsage(AttributeTargets.Class)]
public class TraceMessageSinkAttribute : ContextAttribute
{
    public TraceMessageSinkAttribute() :
        base("TraceMessageSinkAttribute")
    {
    }

    public override void GetPropertiesForNewContext(
        IConstructionCallMessage msg )
    {
        msg.ContextProperties.Add(
            new TraceMessageSinkProperty(this.AttributeName));
    }

    public override System.Boolean IsContextOK (
        Context ctx , IConstructionCallMessage msg )
    {
        return (ctx.GetProperty(this.AttributeName) != null);
    }
}

```

Теперь к любому классу, производному от *ContextBoundObject*, можно добавить атрибут *TraceMessageSinkAttribute*:

```
[TraceMessageSinkAttribute()]
public class C : ContextBoundObject
{
    void Foo(){ ... }
}
```

Класс *TraceMessageSink* будет перехватывать все вызовы методов объектов класса *C* из-за пределов контекста. Ниже показан пример результатов трассировки вызова метода *C.Foo*:

```
631580307740445920 :: TraceMessage() - System.Runtime.Remoting.
                          Messaging.MethodCall
    DomainID=1, ContextID=1
    Msg[__Uri] = /fd062ced_1cc4_423b_949c_75acee5498bc/
                  23509144_1.rem
    Msg[__MethodName] = Foo
    Msg[__MethodSignature] = System.Type[]
    Msg[__TypeName] = clr:CAO.C, Sinks
    Msg[__Args] = System.Object[]
    Msg[__CallContext] = System.Runtime.Remoting.Messaging.
                          LogicalCallContext
    Msg[__ActivationTypeName] = CAO.C, Sinks,
    Version=1.0.876.29571,
    Culture=neutral, PublicKeyToken=null
```

Цепочка агентских приемников

Все обсуждавшиеся ранее приемники контекста перехватывают сообщения **вызовов** методов в контексте экземпляра серверного объекта. Цепочка агентских приемников отличается от других цепочек приемников контекста тем, что она выполняется в контексте *клиентского* объекта, выполняющего вызовы методов удаленного объекта. На рис. 6-3 показана взаимосвязь между экземпляром клиентского объекта, прокси и *цепочкой* агентских приемников в одном домене приложения и экземпляром удаленного объекта.

Инфраструктура .NET Remoting строит цепочку агентских приемников, *просматривая свойства* контекста в порядке, обратном построению серверной объектной цепочки. Это обеспечивает симметричный порядок выполнения *операций* с обеих цепочках для тех свойств контекста, которые добавляют приемники в обе цепочки. Инфраструктура .NET Remoting добавляет пользователь-

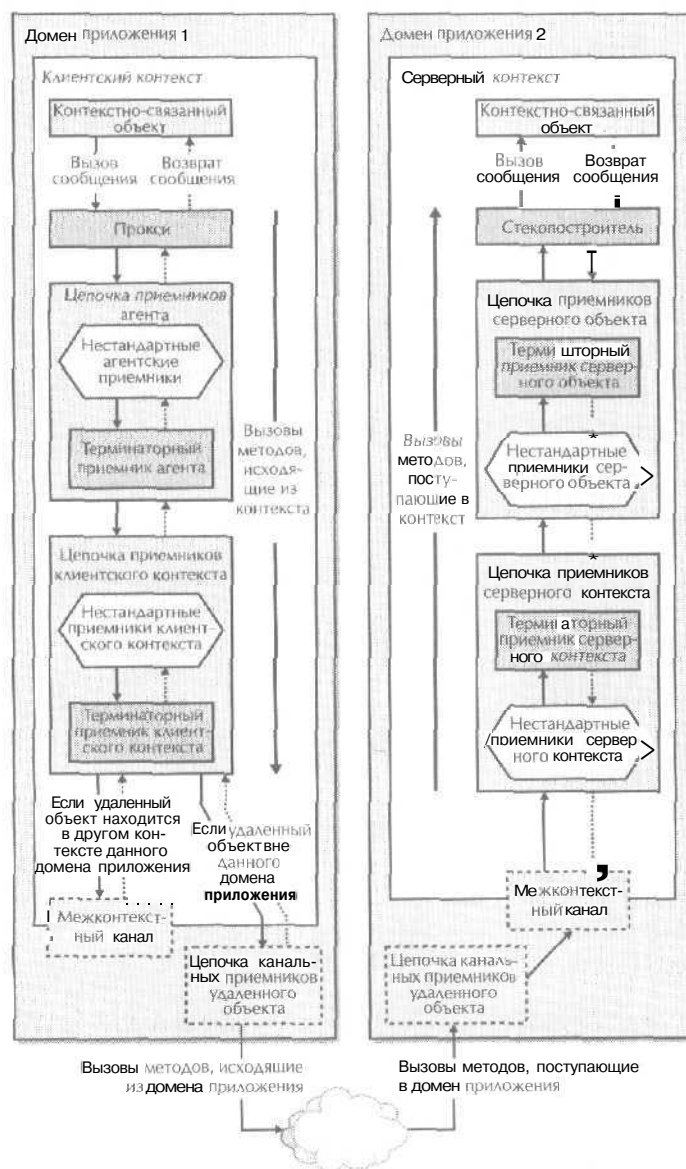


Рис. 6-3. Цепочка агентских приемников, исполняемая в контексте клиента и перехватывающая вызовы методов удаленного объекта

ские приемники в начало цепочки агентских приемников так, чтобы последним приемником в ней всегда был *System.Runtime.Remoting.Contexts.EnvoyTerminatorSink*. Для каждого прокси удаленного объекта существует своя цепочка агентских приемников, содержащая по крайней мере экземпляр приемника *EnvoyTerminatorSink* общий для всего домена приложения.

По получении сообщения вызова метода от прозрачного прокси, реальный прокси (или специализированный, производный от *RealProxy*) передает сообщение цепочке агентских приемников. В главе 5 для получения ссылки на первый приемник этой цепочки мы использовали метод *RemotingServices.GetEnvoyChainForProxy*.

Так как агентские приемники исполняются в контексте клиента, они предоставляют идеальную возможность для проверки аргументов вызова метода или выполнения других оптимизаций вызова метода на клиентской стороне. Проверяя аргументы на клиентской стороне, можно избежать выхода сообщений вызовов методов за пределы контекста и сэкономить сетевые ресурсы в тех случаях, когда известно, что вызов метода все равно завершится с ошибкой из-за некорректных значений аргументов.

Как и в случае с другими терминаторными приемниками, домен приложения содержит единственный экземпляр класса *EnvoyTerminatorSink*, предоставляющий всем контекстам домена следующие функции:

- возможность проверки на неравенство *null* ссылки на (*Message*, передаваемой *SyncProcessMessage* и *AsyncProcessMessage*;
- возможность передачи допустимого сообщения клиентской контекстной цепочке приемников.

Создание агентского приемника

Для создания агентского приемника необходимо выполнить следующие действия:

- создать приемник сообщений, реализующий действия, которые должны выполняться в контексте клиента всякий раз, когда клиент вызывает метод серверного объекта;
- определить свойство контекста, реализующее интерфейс *IContributeEnvoySink*;

- определить атрибут контекста, который при вызове *IContextAttribute.GetPropertiesForNewContext* во время обработки сообщения вызова конструктора добавляет созданное выше свойство к свойствам контекста;
- добавить атрибут контекста к объявлению класса.

ПРЕДУПРЕЖДЕНИЕ Свойство *ObjRef.EnvoyInfo* переносит цепочку агентских приемников через границу .NET Remoting при маршалинге *ObjRef*, что всегда имеет место при клиентской активизации объекта. Однако для общеизвестных объектов инфраструктура .NET Remoting во время активизации не выполняет маршалинг *ObjRef* на клиент. Из-за этого клиент не получает цепочку агентских приемников. Для получения ссылки на общеизвестный объект и связанную с ним цепочку агентских приемников следует принудительно обеспечить маршалинг *ObjRef* общеизвестного объекта, возвратив ссылку на него либо в качестве возвращаемого значения метода, либо в качестве значения параметра метода вида *out*.

Пример: проверка параметров метода

Для демонстрации использования агентских приемников мы реализуем приемник сообщений, проверяющий параметры метода. Чтобы сделать пример более полным, мы создадим механизм проверки на основе классов, реализующих интерфейс *IParamValidator*:

```
public interface IParamValidator
{
    bool Validate(object[] o);
}
```

Метод *IParamValidator.Validate* принимает массив объектов, представляющий подлежащие проверке параметры метода. Первый элемент массива — это параметр 1, второй — параметр 2 и т. д. Например, следующий метод принимает два параметра:

```
public void Foo(int x, int y)
{
```

```
}
```

Соответствующая реализация *IParamValidator.Validate* для *Foo* получала бы массив из двух объектов. В первом элементе массива (по индексу 0) содержалось бы значение *x*, а во втором — значение *y*.

Ниже показано определение приемника сообщений, использующего интерфейс *IParamValidator* для проверки параметров вызова метода в сообщении, полученном *SyncProcessMessage* и *AsyncProcessMessage*:

```
[Serializable]
public class ParamValidatorMessageSink : IMessageSink
{
    Hashtable _htValidators;
    IMessageSink _NextSink;

    public IMessageSink NextSink
    {
        get
        {
            return _NextSink;
        }
    }

    //
    // Конструктор принимает Hashtable, в котором имя метода
    // является ключом поиска ссылки на IParamValidator,
    // выполняющего проверку параметров данного метода.
    public ParamValidatorMessageSink( IMessageSink next,
                                     Hashtable htValidators)
    {
        !
        Trace.WriteLine("ParamValidatorMessageSink ctor");
        _htValidators = htValidators;
        _NextSink = next;
    }

    public IMessage SyncProcessMessage (IMessage msg )
    {
        try
        {
            ValidateParams(msg);
```

```

    }
    catch ( System.Exception e )
    {
        return new ReturnMessage(e,
                                (IMethodCallMessage)msg);
    }
    return _NextSink.SyncProcessMessage(msg);
}

```

Обратите внимание на наличие у приемника сообщений атрибута *Serializable*. Это необходимо для того, чтобы исполняющая среда в процессе активизации могла выполнить маршalling экземпляра приемника сообщений в составе свойства *EnvoyInfo* объекта *ObjRef*. Любой приемник, используемый в цепочке агентских приемников, должен быть сериализуемым, чтобы его удалось переместить в контекст клиента через границу .NET Remoting. Также обратите внимание, что конструктор принимает в качестве второго параметра экземпляр *Hashtable*, связывающий имена методов с *IParamValidator*, который выполняет проверку параметров соответствующего метода. Для проверки параметров, содержащихся в сообщении, метод *SyncProcessMessage* использует следующий вспомогательный метод:

```

//
// Проверка параметров, содержащихся в сообщении.
void ValidateParams(IMessage msg)
{
    // Убедиться, что msg - это сообщение вызова метода.
    if ( msg is IMethodCallMessage )
    {
        // Создать обертку для удобства работы с сообщением.
        MethodCallMessageWrapper mcm =
            new MethodCallMessageWrapper((
                IMethodCallMessage)msg);

        // Найти интерфейс для проверки
        // вызываемого метода.
        IParamValidator v =
            (IParamValidator)_htValidators[mcm.MethodName];
        if ( v != null )
        {
            // Выполнить проверку параметров посредством
            // найденного интерфейса.
            if ( ! v.Validate( mcm.Args ) )

```

```

        throw new ArgumentException(
            "IParmValidator.Validate() returned " +
            "false indicating invalid parameter
            values");
    }
}
}
}

```

Указав содержащееся в сообщении имя метода, *ValidateParams* отыскивает ссылку на соответствующий *IParmValidator*, который затем используется для проверки параметров, содержащихся в сообщении, путем вызова *IParmValidator.Validate*. Если обнаружены недопустимые параметры, то *ValidateParams* генерирует исключение.

AsyncProcessMessage также использует *ValidateParams*:

```

public IMessageCtrl AsyncProcessMessage (
    IMessage msg, IMessageSink replySink )
{
    try
    {
        ValidateParams(msg);
        return _NextSink.AsyncProcessMessage( msg,
                                                replySink );
    }
    catch(System.Exception e)
    {
        replySink.SyncProcessMessage(
            new ReturnMessage( e,
                              (IMethodCallMessage)msg ) );
        return null;
    }
}
>
} // Конец class ParamValidatorMessageSink

```

Ответное сообщение нам неинтересно, поэтому мы не будем добавлять приемник сообщения в цепочку приемников ответа. Вместо этого мы просто проверим параметры и передадим сообщение следующему приемнику в цепочке. При обнаружении неверных параметров мы посылаем экземпляр *ReturnMessage*, инкапсулирующий исключение, первому приемнику в цепочке приемников ответа.

Так определяется свойство контекста, реализующее *IContributeEnvoySink*:

```
[Serializable]
public class ParameterValidatorProperty : IContextProperty,
                                         IContributeEnvoySink
{
    private string _Name;
    Hashtable _htValidators;

    public string Name
    {
        get
        {
            return _Name;
        }
    }

    public ParameterValidatorProperty( string name,
                                       Hashtable htValidator )
    {
        _Name = name;
        _htValidators = htValidators;
    }

    public IMessageSink GetEnvoySink( MarshalByRefObject obj,
                                       IMessageSink nextSink )
    {
        return new ParamValidatorMessageSink(nextSink,
                                             _htValidators);
    }

    public void Freeze ( Context newContext )
    {
        // Добавление новых свойств с этого момента
        // не допускается.
    }

    public Boolean IsNewContextOK ( Context newCtx )
    {
        return true;
    }
} // Конец class ParameterValidatorProperty
```

Конструктор *ParameterValidatorProperty* принимает *Hashtable* ссылок на *IParamValidator*, который *IContributeEnvoySink.GetEnvoySink* передает конструктору *ParameterValidatorMessageSink*.

Ниже приведено определение атрибута, добавляющего свойство *ParameterValidatorProperty* в контекст:

```
[AttributeUsage(AttributeTargets.Class)]
public class ParameterValidatorContextAttribute :
    ContextAttribute
{
    string[] _methodNames;
    Hashtable _htValidators;

    public ParameterValidatorContextAttribute(
        string[] method_names, params Type[] validators)
        : base("ParameterValidatorContextAttribute")
    {
        if ( method_names.Length != validators.Length )
        {
            throw new ArgumentException(
                "ParameterValidatorContextAttribute ctor",
                "Length of method_names and validators " +
                "must be equal");
        }

        _methodNames = method_names;
        _htValidators = new Hashtable(method_names.Length);

        int i = 0;
        foreachC Type t in validators )
        {
            _htValidators.Add(method_names[i++],
                Activator.CreateInstance(t) );
        }
    }

    public override void
        GetPropertiesForNewContext(
            IConstructionCallMessage msg )
    {
        msg.ContextProperties.Add(
            new ParameterValidatorProperty( this.AttributeName,
                _htValidators));
    }

    public override Boolean
        IsContextOK ( Context ctx ,
            IConstructionCallMessage msg )
    {

```

```

        return (ctx.GetProperty(this.AttributeName) != null);
    }
}

```

Здесь интересен конструктор *ParameterValidatorContextAttribute*, принимающий два параметра: массив строк с именами методов и массив переменной длины *params*, содержащий экземпляры *Type*, каждый из которых должен реализовывать интерфейс *IParamValidator*. Элементы в массивах упорядочены таким образом, что метод, имя которого находится в элементе 0 массива строк, соответствует типу, на который ссылается элемент 0 второго массива. Проверив, что длины массивов одинаковы, конструктор заполняет *Hashtable*, отображая имена методов на новые экземпляры соответствующих типов.

Атрибут *ParameterValidatorContextAttribute* можно использовать, например, так:

```

[ ParameterValidatorContextAttribute(
    new string[] { "Bar", "Foo" },
    typeof(BarValidator), typeof(FooValidator))
]
public class SomeObject : ContextBoundObject
{
    public void Bar()
    {
        Console.WriteLine("ContextID={0}, SomeObject.Bar()",
            Thread.CurrentContext.ContextID);
    }

    public void Bar(int x)
    {
        Console.WriteLine("ContextID={0},
            SomeObject.Bar(x={1})",
            Thread.CurrentContext.ContextID, x);
        return this;
    }

    public int Foo( int x, int y )
    {
        return x + y;
    }
}

```

Ниже показана реализация *BarValidator*, учитывающая, что метод *Bar* имеет перегружаемые версии, и ограничивающая значения параметра диапазоном от 0 до 5000:

```
[Serializable]
class BarValidator : IParamValidator
{
    public bool Validate(object[] args)
    {
        // Bar имеет перегруженные версии.
        // Нас интересует только версия, имеющая аргументы.
        if ( args.Length != 0 )
        {
            // Первый параметр - x. Проверка на диапазон
            // от 0 до 5000.
            int x = (int)args[0];
            if (!( 0 <= x && x <= 5000 ))
            {
                string err = String.Format(
                    "BarValidator detected illegal 'x'
                    parameter " +
                    "value of '{0}'.\nLegal values:
                    0 <= x <= 5000.", x);
                throw new ArgumentException(err);
            }
        }
        return true;
    }
}
```

На рис. 6-4 показан снимок экрана, содержащий сообщение об ошибке, произошедшей из-за вызова метода *Bar* с параметром 6500.

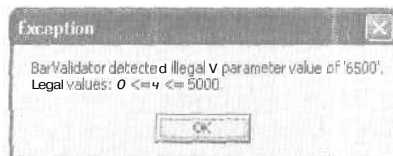


Рис. 6-4. Сообщение, выводимое в результате передачи неверного значения параметра *x* методу *SomeObject.Bar*

Далее показан листинг класса *FooValidator*, проверяющего оба параметра метода *SomeObject.Foo*:

```

[Serializable]
class FooValidator : IParamValidator
{
    public bool Validate(object[] args)
    {
        string e = "";

        // Первый параметр - x. Проверка на диапазон
        // от 0 до 9999.
        int x = (int)args[0];
        if ( x > 9999 )
        {
            e = "parameter 'x' exceeds maximum allowed value " +
                "of '9999'";
        }
        // Второй параметр - y, Проверка, что
        // его значение не больше 1000,
        int y = (int)args[1];
        if ( y > 1000 )
        {
            e = "parameter 'y' exceeds maximum allowed value " +
                "of '1000'";
        }

        if ( e.Length != 0 )
        {
            throw new ArgumentException
                ("FooValidator detected illegal parameter
                values\n\n" + e);
        }

        return true;
    }
}

```

Обратите внимание: классы, выполняющие проверки, объявлены с атрибутом *Serializable*. Это связано с тем, что приемник сообщения содержит ссылки на эти классы в *_htValidators*. Так как цепочка агентских приемников пересылается клиенту в составе *ObjRef* как свойство *EnvoyInfo*, то и все члены агентского приемника должны быть сериализуемы. Кроме того, они должны иметь небольшой размер, чтобы снизить накладные расходы при пересылке через границы .NET Remoting.

Заключение

Приемники сообщений и контексты представляют собой ключевые элементы инфраструктуры .NET Remoting. Как вы узнали из этой главы, приемники сообщений позволяют перехватывать сообщения .NET Remoting в различных точках как клиентского, так и серверного контекста, а контекст позволяет разработчику определять различные сервисы, доступные всем исполняющимся с ним объектам. Примерами таких сервисов могут служить трассировка сообщений и ведение журнала ошибок. Благодаря тому, что цепочка агентских приемников выполняется в контексте клиента, она является удобным местом проверки параметров методов перед их передачей по сети. Дальнейшие возможности настройки архитектуры .NET Remoting связаны с каналами и канальными приемниками, которые и станут темой следующей главы.

Каналы и каналные приемники

В этой главе мы продолжим рассказ о различных средствах настройки .NET Remoting. Чтобы дать вам более детальное представление об архитектуре каналов, мы подробно рассмотрим *HttpChannel*. Затем мы разработаем нестандартный канал с использованием нашего собственного транспорта. В заключение — построим пользовательский каналный приемник (channel sink).

.NET Remoting предоставляет два типа каналов: *HttpChannel* и *TcpChannel*. Хотя их общая структура похожа, они отличаются транспортом, используемым для передачи сообщений. HTTP и TCP способны обеспечить транспорт в большинстве случаев, однако иногда может возникать необходимость применения других транспортов.

Например, когда требуется доступ к удаленным объектам с беспроводного устройства, использующего протокол WAP (Wireless Application Protocol). Для решения этой проблемы потребуется создать нестандартный канал, пересылающий сообщения по WAP. Если мы подробнее рассмотрим структуру каналов, станет ясно, что после того, как сообщение воссоздано в надлежащем формате, выбранный для его получения метод не имеет значения.

Построение каналов

Чтобы облегчить вам понимание общей архитектуры каналов, рассмотрим ее на примере канала HTTP. Изучение структуры существующего канала полезно по ряду причин. Во-первых, это даст более глубокое понимание работы канала HTTP. Во-вторых, мы сможем объяснить концепции на более высоком уровне, что

облегчит понимание общей архитектуры канала, когда мы займемся созданием собственного канала. Мы рассмотрим те особенности канала HTTP, которые окажутся нам полезными при создании своего канала. В структуре канала HTTP нас интересуют шесть основных классов:

- *HttpChannel*;
- *HttpServerChannel*;
- *HttpServerTransportSink*;
- *HttpClientChannel*;
- *HttpClientTransportSinkProvider*;
- *HttpClientTransportSink*.

Терминология каналов

В табл. 7-1 перечислены некоторые понятия, необходимые для понимания работы каналов.

Таблица 7-1. Терминология каналов

Термин	Описание
URI объекта	URI объекта идентифицирует некоторый общеизвестный объект, зарегистрированный на сервере
URI канала	URI канала — это строка, задающая информацию для подключения к серверу
URL серверной активизации	URL серверной активизации — это уникальная строка, используемая клиентом для подключения к соответствующему объекту на сервере. В URL <i>http://localhost:4000/SomeObjectUri</i> подстрока <i>SomeObjectUri</i> является URI объекта, а <i>http://localhost.4000</i> — URI канала
URL клиентской активизации	URL клиентской активизации — это строка, используемая клиентом для подключения к соответствующему объекту на сервере. Для объектов с клиентской активизацией нет необходимости использовать уникальный URL, так как он будет сгенерирован инфраструктурой .NET Remoting

HttpChannel

HttpChannel выполняет небольшой объем работы. Это оболочка, которая предоставляет единый интерфейс к функциональности, реализованной классами *HttpServerChannel* и *HttpClientChannel*. Большинство методов *HttpChannel* просто переадресуют вызовы соответствующим методам *HttpServerChannel* или *HttpClientChannel*. *HttpChannel* реализует интерфейсы *System.Runtime.Remoting.Channels.IChannel*, *System.Runtime.Remoting.Channels.IChannelSender* и *System.Runtime.Remoting.Channels.IChannelReceiver*. Если интерфейс *IChannel* обязателен для каналов, то *IChannelSender* и *IChannelReceiver* нужны не всегда. Для канала, который только принимает сообщения, необходим *IChannelReceiver*; аналогично для канала, который только отправляет сообщения, требуется лишь *IChannelSender*. Нет необходимости наследовать от интерфейса *IChannel* непосредственно, так как от него наследуют и *IChannelSender*, и *IChannelReceiver*. Интерфейс *IChannel* описан в табл. 7-2.

Таблица 7-2. Члены интерфейса
System.Runtime.Remoting.Channels.IChannel

Член	Тип члена	Описание
<i>ChannelName</i>	Свойство	Возвращает имя канала
<i>ChannelPriority</i>	Свойство	Возвращает приоритет канала. По умолчанию он равен 1
<i>Parse</i>	Метод	Извлекает URI канала и URI объекта из URL

Обычно *ChannelName* соответствует названию транспорта. Например, для *HttpChannel* — это свойство возвращает строку «http» (в нижнем регистре). Свойство *ChannelPriority* позволяет управлять порядком выбора каналов при попытке подключения к удаленному объекту. Например, если на сервере зарегистрировано два канала с разными приоритетами и оба эти канала зарегистрированы клиентом, то инфраструктура удаленного взаимодействия выберет канал с наивысшим приоритетом. Свойства *ChannelName* и *ChannelPriority* доступны только для чтения. В *HttpChannel* имеется три конструктора.

```
public HttpChannel();
public HttpChannel( int );
```

```
public HttpChannel( IDictionary,
                  IClientChannelSinkProvider,
                  IServerChannelSinkProvider );
```

Первый конструктор инициализирует канал и для отправки, и для приема сообщений, тогда как второй — для приема сообщений. Параметр второго конструктора задает порт, на котором сервер будет ожидать вызовы. Последний конструктор наиболее интересен. Все каналы, для которых можно использовать файлы конфигурации, должны реализовать конструктор с такой сигнатурой. Исполняющая среда .NET Remoting генерирует исключение, если такой конструктор отсутствует. Так как *HttpChannel* поддерживает множество необязательных параметров, то создание конструкторов для их всевозможных комбинаций непрактично. Чтобы преодолеть данную проблему, каналы используют параметр *IDictionary* для передачи конфигурационной информации конструктору. Конфигурационные параметры, поддерживаемые *HttpChannel*, перечислены в табл. 7-3.

Таблица 7-3. Конфигурационные параметры *HttpChannel*

Название	Где применимо	Описание
<i>path</i>	Сервер и клиент	Используется, когда нужно зарегистрировать несколько экземпляров <i>HttpChannel</i> , так как имя каждого канала должно быть уникальным
<i>priority</i>	Сервер и клиент	Задаёт приоритет канала
<i>clientConnectionLimit</i>	Клиент	Задаёт число клиентов, которые могут быть подключены к серверу одновременно. По умолчанию — 2
<i>proxyName</i>	Клиент	Позволяет указать имя компьютера-прокси
<i>proxyPort</i>	Клиент	Позволяет указать порт прокси
<i>port</i>	Сервер	Устанавливает порт ожидания вызовов
<i>suppressChannelData</i>	Сервер	Если это свойство установлено, то свойство <i>ChannelData</i> будет возвращать <i>null</i>

см. след. стр.

Таблица 7-3. (окончание)

Название	Где применимо	Описание
<i>useIpAddress</i>	Сервер	Определяет, должен ли канал использовать IP-адрес. Если значение равно <i>false</i> , то канал будет использовать имя компьютера, полученное вызовом статического метода <i>Dns.GetHostName</i>
<i>bindTo</i>	Сервер	Позволяет указать IP-адрес сетевой платы, с которой должен работать сервер. Полезно в тех случаях, когда на компьютере установлено несколько сетевых адаптеров
<i>machineName</i>	Сервер	Позволяет переопределить имя компьютера.

Свойства канала можно задать двумя способами. Первый способ — программный. В следующем фрагменте кода задается номер порта и IP-адрес для *bindTo*:

```
IDictionary ChannelProperties = new Hashtable();
ChannelProperties ["port"] = 4001;
ChannelProperties ["bindTo"] = "192.168.0.1";
HttpChannel channel = new HttpChannel( props,
                                     null,
                                     null );
```

Второй способ подразумевает применение конфигурационного файла:

```
<configuration>
  <system.runtime.remoting>
    <application>
      <service>
        :
      </service>
      <channels>
        <channel ref="http" port="4001" bindTo="192.268.0.1"/>
      </channels>
    </application>
  </system.runtime.remoting>
</configuration>
```

Два последних параметра конструктора *IClientChannelSinkProvider* и *IServerChannelSinkProvider* позволяют задать нестандартные провайдеры форматировщиков (formatter provider). Стандартными провайдерами для *HttpChannel* считаются *SoapClientFormatterSinkProvider* и *SoapServerFormatterSinkProvider*.

Ранее уже говорилось, что *HttpChannel* реализует *IChannelReceiver*. Этот интерфейс должен быть реализован любым каналом, по которому будут приниматься сообщения. Члены интерфейса перечислены в табл. 7-4. Подробно мы обсудим их далее.

Таблица 7-4. Члены
System.Runtime.Remoting.Channels.IChannelReceiver

Член	Тип члена	Описание
<i>ChannelData</i>	Свойство	Используется для хранения канальной информации на уровне экземпляра
<i>GetUrlsForUri</i>	Метод	Возвращает массив URL, необходимых для уникальной идентификации удаленных объектов
<i>StartListening</i>	Метод	Запускает ожидание входящих сообщений серверным каналом
<i>StopListening</i>	Метод	Прекращает ожидание входящих сообщений серверным каналом

Как говорилось ранее, *HttpChannel* наследует от *IChannelSender*. *IChannelSender* предоставляет интерфейс для взаимодействия с каналом на стороне клиента. Как видно из табл. 7-5, у него имеется единственный метод.

Таблица 7-5. Член
System.Runtime.Remoting.Channels.IChannelSender

Член	Тип члена	Описание
<i>CreateMessageSink</i>	Метод	Возвращает канальный приемник сообщений

HttpServerChannel

HttpServerChannel реализует средства приема сообщений на сервере. В процессе создания своего экземпляра, класс *HttpChannel* создает экземпляр *HttpServerChannel*. Как и в случае *HttpChannel*,

HttpServerChannel имеет конструктор, принимающий в качестве первого параметра объект *IDictionary*:

```
public HttpServerChannel( IDictionary properties,
                          IServerChannelSinkProvider provider );
```

Перед передачей объекта *IDictionary* конструктору *HttpServerChannel* *HttpChannel* удаляет из него свойства, относящиеся к клиенту. В момент создания *HttpServerChannel* выполняет следующие действия:

- получает значения свойств и устанавливает соответствующие члены-переменные;
- инициализирует данные канала;
- извлекает данные канала из провайдера серверных приемников;
- создает серверную цепочку каналных приемников,

Мы подробно рассмотрим эти операции в разделе «Создание нестандартного канала *FileChannel*» этой главы.

Как уже говорилось, *HttpServerChannel* обрабатывает получение сообщений-запросов; таким образом, он должен быть производным от *IChannelReceiver*. Давайте подробнее рассмотрим члены *HttpServerChannel*, реализующие *IChannelReceiver*. Свойство *ChannelData* возвращает объект типа *ChannelDataStore*. Он хранится в закрытой переменной-члене *_channelData*. Основное назначение *ChannelDataStore* — хранение URI соответствующего канала. URI канала для *HttpChannel* имеет вид *http://<имя_компьютера>:<порт>*. Функция-член *GetUrlsForUri* использует *_channelData* при генерации своего возвращаемого значения. *GetUrlsForUri* добавляет URI объекта к URI канала и возвращает полученное значение по индексу 0 массива строк.

Наиболее интересный метод в этой группе — *StartListening*. Он отвечает за запуск фонового потока, ожидающего поступления входящих сообщений в заданный порт. Реализация *StartListening* аналогична следующему псевдокоду:

```
public void StartListening( Object data )
{
    ThreadStart ListeningThreadStart =
        new ThreadStart( this.Listen );
    _listenerThread = new Thread( ListeningThreadStart );
}
```

```

        _listenerThread.IsBackground = true;
        _listenerThread.Start();
    }

```

В данном фрагменте кода *_listenerThread* — это переменная-член типа *Thread*. Метод *Listen* выполняет цикл ожидания входящих сообщений. При получении очередного сообщения *Listen* отправляет его на обработку вызовом метода *ServiceRequest* класса *HttpServerTransportSink*. Затем *Listen* вновь возвращается в состояние ожидания.

HttpServerTransportSink

Основная задача *HttpServerTransportSink* — обработка сообщений-запросов. *HttpServerTransportSink*, как и все серверные каналные приемники, должен быть производным от интерфейса *System.Runtime.Remoting.Channels.IServerChannelSink*. Открытые члены этого интерфейса перечислены в табл. 7-6.

Таблица 7-6. Члены

System.Runtime.Remoting.Channels.IServerChannelSink

Член	Тип члена	Описание
<i>NextChannelSink</i>	Свойство	Возвращает ссылку на следующий каналный приемник в цепочке
<i>AsyncProcessResponse</i>	Метод	Возвращает данные ответа при обработке асинхронного сообщения
<i>GetResponseStream</i>	Метод	Создает объект <i>Stream</i> , содержащий объект <i>IMessage</i> , а также нужные пары «ключ — значение» из объекта <i>ITransportHeaders</i>
<i>ProcessMessage</i>	Метод	Используется цепочкой приемников для обработки входящих сообщений. Так как транспортный приемник является первым приемником в цепочке, никаких действий не требуется

В процессе своего создания *HttpServerTransportSink* получает ссылку на следующий приемник в цепочке приемников. *HttpServerTransportSink* помещает эту ссылку в переменную-член *_nextSink*. Каждый приемник отвечает за хранение ссылки на следующий приемник. По получении сообщения каждый приемник передает его далее по цепочке вызовом *_nextSink.ProcessMessage*.

Возвращаемое значение метода *ProcessMessage* является перечислением типа *ServerProcessing*, значения которого указаны в табл. 7-7.

Таблица 7-7. Константы перечисления
System.Runtime.Remoting.Channels.ServerProcessing

Константа	Описание
<i>Complete</i>	Сообщение запроса обработано синхронно
<i>Async</i>	Сообщение запроса отправлено на асинхронную обработку. Данные ответа должны быть сохранены для дальнейшей обработки
<i>OneWay</i>	Сообщение запроса обработано, и ответ не ожидается

Сообщение запроса содержит два ключевых элемента: *ITransportHeaders* и *Stream*. *ITransportHeaders* — это словарь, позволяющий передавать пары «ключ — значение» между клиентом и сервером. .NET Framework предоставляет класс под названием *System.Runtime.Remoting.Channels.CommonTransportKeys*, определяющий строковые ключи для информации, наиболее часто встречающейся в объектах *ITransportHeaders*. *CommonTransportKeys* содержит три открытых строковых поля:

- *ConnectionId*
- *IPAddress*
- *RequestUri*

Объект *Stream* содержит сериализованное сообщение .NET Remoting, например *ConstructionCallMessage* или *MethodCallMessage*.

HttpClientChannel

HttpClientChannel обрабатывает отправку сообщений-запросов на сервер. Как и *HttpServerChannel*, экземпляр *HttpClientChannel* создается в конструкторе *HttpChannel*. *HttpClientChannel* имеет множество конструкторов, наиболее гибким из которых является следующий:

```
public HttpClientChannel( IDictionary properties,
                        IClientChannelSinkProvider
                        sinkProvider );
```

Данный конструктор позволяет задать для канала клиентские свойства, описанные в табл. 7-3, а также альтернативный провайдер приемников. Провайдеры приемников мы рассмотрим чуть позже.

Основная задача *HttpClientChannel* — создание объекта *IMessageSink*. Он создается и возвращается методом *CreateMessageSink*. Как вы помните, *CreateMessageSink* входит в интерфейс *IChannelSender*. В разделе «Создание нестандартных каналов» далее в этой главе, мы рассмотрим *CreateMessageSink* подробно, пока же достаточно отметить, что он возвращает *HttpClientTransportSink*. *CreateMessageSink* не создает новый транспортный приемник напрямую. Для этой цели используется класс *HttpClientTransportSinkProvider*.

HttpClientTransportSinkProvider

Единственная задача *HttpClientTransportSinkProvider* — создание экземпляра *HttpClientTransportSink*. Как и все провайдеры клиентских приемников *HttpClientTransportSinkProvider* наследует от интерфейса *IClientChannelSinkProvider*, члены которого показаны в табл. 7-8.

Таблица 7-8. Члены

System.Runtime.Remoting.Channels.IClientChannelSinkProvider

Член	Тип члена	Описание
<i>Next</i>	Свойство	Возвращает или устанавливает следующий <i>IClientChannelSinkProvider</i> в цепочке
<i>CreateSink</i>	Метод	Создает и возвращает новый объект, производный от <i>IClientChannelSink</i>

Свойство *Next* всегда возвращает *null*, так как *HttpClientTransportSinkProvider* — это последний провайдер в цепочке. *CreateSink* возвращает вновь созданный *HttpClientTransportSink*.

HttpClientTransportSink

HttpClientTransportSink — это класс, отправляющий сообщения на сервер. Он реализует интерфейс *System.Runtime.Remoting.Channels.IClientChannelSink*, члены которого показаны в табл. 7-9.

Таблица 7-9. Члены
System.Runtime.Remoting.Channels.IClientChannelSink

Член	Тип члена	Описание
<i>NextChannelSink</i>	Свойство	Хранит ссылку на следующий приемник в цепочке
<i>AsyncProcessRequest</i>	Метод	Обрабатывает асинхронные вызовы методов
<i>AsyncProcessResponse</i>	Метод	Обрабатывает возвращаемые сервером результаты асинхронных вызовов
<i>GetRequestStream</i>	Метод	Создает объект <i>Stream</i> , содержащий объект <i>IMessage</i> и необходимые пары «ключ — значение» из объекта <i>ITransportHeaders</i>
<i>ProcessMessage</i>	Метод	Обрабатывает синхронные вызовы методов

Так как *HttpClientTransportSink* является последним приемником в цепочке, функциональны в нем только два метода: *ProcessMessage* и *AsyncProcessRequest*. Их задача — упаковка и отправка сообщения. Основное различие между ними в том, что *ProcessMessage* ожидает ответа от сервера, тогда как *AsyncProcessRequest* устанавливает метод обратного вызова, который ожидает получения от сервера возвращаемого сообщения, когда исполнение продолжается.

В этом разделе мы обсудили высокоуровневую структуру каналов. В качестве модели взяли *HttpChannel*. Далее мы используем полученные здесь знания для построения нестандартного канала.

Создание нестандартных каналов

Инфраструктура .NET Remoting предоставляет интерфейсы и архитектуру, позволяющую вам подключать свои собственные нестандартные каналы. Для создания нестандартного канала имеется множество причин. Например, необходимость реализовать удаленные вызовы между компьютерами, которые могут взаимодействовать только по телефонной линии. В этом случае нестандартный канал будет отличным решением: серверные и клиентские каналные приемники удастся настроить таким обра-

зом, чтобы при пересылке сообщения они автоматически устанавливали и разрывали соединение. Вот некоторые другие примеры возможных нестандартных каналов:

- канал для протокола UDP (User Datagram Protocol);
- канал для протокола *именованных каналов* (named pipe);
- файловый канал;
- канал для *<подставьте здесь имя нестандартного протокола>*

Вероятно, вы подумали: «Зачем мне понадобятся эти каналы, когда уже есть *HttpChannel* и *TcpChannel*?» Рассмотрим ситуацию, когда клиент или сервер не поддерживают TCP или HTTP. В системе, где отсутствует общезыковая исполняющая среда .NET, вам придется генерировать сообщения вручную. При соблюдении надлежащего формата сообщений принимающий или отправляющий канал не заметит никаких отличий. Это классная возможность!

Этапы создания нестандартного канала .NET Remoting

Сейчас мы поговорим об этапах создания нестандартного канала .NET Remoting. Они не зависят от транспорта, следовательно, вы можете применить их при создании любого канала. Предположим, что транспортом, используемым для описания основных этапов, будет гипотетический транспорт Widget. Следуя соглашению об именах, используемому для стандартных каналов .NET Remoting, мы получим название нашего канала: *WidgetChannel*. Этапы создания канала описаны далее.

1. Создайте классы для клиентской стороны канала. Клиентский канал состоит из трех классов, производных соответственно от *IChannelSender*, *IClientChannelSinkProvider* и *IClientChannelSink*:

```
public class WidgetClientChannel : IChannelSender
{
    :
}
internal class
    WidgetClientChannelSinkProvider :
        IClientChannelSinkProvider
{
    :
}
```

```

}
public class WidgetClientChannelSink : IClientChannelSink
{
    ...
}

```

2. Создайте классы для серверной стороны канала. Серверная сторона содержит два класса, производные соответственно от *IChannelReceiver* и *IServerChannelSink*:

```

public class WidgetServerChannel : IChannelReceiver
{
    ...
}

public class WidgetServerChannelSink : IServerChannelSink
{
    ...
}

```

3. Создайте вспомогательный класс *WidgetChannelHelper*. Он содержит общую функциональность для классов серверного и клиентского канала.
4. Создайте главный класс канала, объединяющий в себе функциональность клиентских и серверных классов:

```

public class WidgetChannel : IChannelSender, IChannelReceiver
{
    private WidgetClientChannel = null;
    private WidgetServerChannel = null;
    ...
}

```

Теперь, когда мы познакомили вас с основными этапами создания нестандартного канала .NET Remoting, попробуем создать такой канал!

Создание нестандартного канала *FileChannel*

FileChannel — это канал .NET Remoting, использующий для пересылки сообщений между сервером и клиентом файлы и каталоги. Причина применения столь примитивного средства, как файлы, проста: мы хотим показать гибкость .NET Remoting. Можно, конечно, передавать объект между двумя компьютерами на диске! Но гибкость — не единственное достоинство *FileChannel*. Так как операции с файлами всем хорошо знакомы, мы сосредото-

точимся на деталях создания канала, не отвлекаясь на реализацию сложного транспортного механизма. Наконец, сообщения запросов и ответов, передаваемые по каналу *FileChannel*, остаются после обработки в каталоге на сервере. Это позволяет сохранить хронологию обмена сообщениями между сервером и клиентом, которая пригодится при диагностике и отладке.

Проекты *FileChannel*

Код примера *FileChannel* состоит из следующих проектов:

- * **FileChannel** — содержит реализацию *FileChannel*;
- » **DemonstrationObjects** — содержит класс *Demo*. Клиент и сервер в нашем примере используют удаленный экземпляр класса *Demo*;
- * **Server** — регистрирует и использует *FileChannel*. Затем данный модуль просто ожидает завершения своей работы пользователем по нажатию на Ввод. В конфигурационном файле сервер настраивается на работу с каталогом C:\File;
- » : **Client** — как и проект сервера, данный модуль регистрирует *FileChannel*. Проект демонстрирует различные типы вызовов методов.

Реализация класса *FileClientChannel*

Так как *FileClientChannel* наследует от интерфейса *IChannelSender*, мы обязаны реализовать метод *CreateSink*. Помимо *CreateSink*, нам необходимо добавить свойства *ChannelName*, *ChannelPriority* и метод *Parse*. Последние три члена обязательны потому, что *IChannelSender* наследует от *IChannel*. Создание их станет отправной точкой для создания *FileClientChannel*.

```
public class FileClientChannel : IChannelSender
{
    private String m_ChannelName = "file";
    private int m_ChannelPriority = 1;

    private IClientChannelSinkProvider m_ClientProvider = null;

    IClientChannelSinkProvider m_ClientProvider = null;
    public IMessageSink CreateMessageSink( String url,
```

```

        object remoteChannelData,
        out String objectURI )
    {
        :
    }

    public String Parse( String url, out String objectURI )
    {
        return FileChannelHelper.Parse( url, out objectURI );
    }

    public String ChannelName
    {
        get { return m_ChannelName; }
    }

    public int ChannelPriority
    {
        get { return m_ChannelPriority; }
    }
}

```

Обратите внимание: мы добавили в новый класс два метода и два свойства. Все эти четыре члена являются обязательными для каналов, используемых для отправки сообщений-запросов. Для хранения значений, которые будут возвращаться свойствами *ChannelName* и *ChannelPriority*, добавлены три новых закрытых поля: *m_ChannelName*, *m_ChannelPriority* и *m_ClientProvider*. Первые два инициализируются значениями по умолчанию, которые можно переопределить при использовании одного из конструкторов. Переменная *m_ClientProvider* содержит ссылку на поставщик приемников. Метод *Parse* вызывает одноименный статический метод класса *FileChannelHelper*. Реализация метода вынесена в отдельный класс для того, чтобы ее могли использовать и *FileClientChannel*, и *FileServerChannel*. Метод *Parse* мы подробно рассмотрим немного позже.

Затем нам надо добавить к *FileClientChannel* два конструктора. Первый из них создает базовый канал, использующий в качестве форматировщика SOAP:

```

public FileClientChannel()
{
    :
}

```

Второй конструктор позволяет настраивать канал с помощью объекта *IDictionary* и создавать канал посредством конфигурационного файла. Чтобы соответствовать требованиям по использованию конфигурационного файла, конструктор должен содержать параметр, принимающий экземпляр объекта *IClientChannelSinkProvider*. Этим требованиям удовлетворяет следующий конструктор:

```
public FileClientChannel( IDictionary properties,
                        IClientChannelSinkProvider
                        sinkProvider )
{
    if( properties != null )
    {
        foreach (DictionaryEntry entry in properties)
        {
            switch ((String)entry.Key)
            {
                case "name":
                    m_ChannelName = ( String ) entry.Value;
                    break;
                case "priority":
                    m_ChannelPriority = ( int ) entry.Value;
                    break;
            }
        }

        m_ClientProvider = sinkProvider;
    }
}
```

Конструктор извлекает информацию из объекта *IDictionary*. Единственными настраиваемыми параметрами *FileClientChannel* являются *ChannelName* и *ChannelPriority*. Так как эти свойства доступны только для чтения, данный конструктор — единственное место, где их значения разрешается корректировать.

Класс *FileClientChannel* отвечает за создание цепочки поставщиков канала. Затем эта цепочка провайдеров будет создавать цепочку канальных приемников. *FileClientChannel* также обязан предоставить форматировщик по умолчанию в тех случаях, когда он

явно не указан пользователем класса. Рассмотренная функциональность реализуется таким образом:

```
private void SetupClientChannel()
{
    if( m_ClientProvider == null )
    {
        m_ClientProvider = new
            SoapClientFormatterSinkProvider();
        m_ClientProvider.Next = new
            FileClientChannelSinkProvider();
    }
    else
    {
        AddClientProviderToChain(
            m_ClientProvider,
            new FileClientChannelSinkProvider( ));
    }
}
```

Сначала *SetupClientChannel* проверяет, не задан ли провайдер явно, сравнивая *m_ClientProvider* с *null*. Если *m_ClientProvider* равен *null*, то необходимо построить цепочку *IClientChannelSinkProvider*. При этом следует выбрать для канала провайдер форматировщика. *FileChannel* использует *System.Runtime.Remoting.Channels.SoapClientFormatterSinkProvider*. Затем мы добавим наш провайдер в конец цепочки. Если же *m_ClientProvider* не *null*, то следует просто добавить наш провайдер в конец цепочки. Для этого используется метод *AddClientProviderToChain*, который является копией одноименного внутреннего метода .NET Remoting из класса *CoreChannel*.

```
private static void AddClientProviderToChain(
    IClientChannelSinkProvider clientChain,
    IClientChannelSinkProvider clientProvider )
{
    while( clientChain.Next != null )
    {
        clientChain = clientChain.Next;
    }

    clientChain.Next = clientProvider;
}
```

Так как наш *провайдер* должен оказаться последним в цепочке, *AddClientProviderToChain* сначала следует найти ее конец. Для этого в цикле вызывается свойство *Next* до тех пор, пока не будет возвращен *null*, после чего новый провайдер добавляется к цепочке.

Нам осталось реализовать метод *CreateMessageSink*. Он возвращает ссылку на объект *IMessageSink*. Этот объект содержит, как минимум, приемник форматировщика и наш *FileClientChannelSink*. Прокси удаленного объекта использует эту цепочку для отправки вызовов методов. *CreateMessageSink* должен поддерживать объекты как с серверной, так и с клиентской активизацией. При вызове метода объекта с серверной активизацией параметр *url* содержит URL, сконфигурированный для канала. Однако при вызове объекта с клиентской активизацией URL передается в параметре *remoteChannelData*.

```
public IMessageSink CreateMessageSink( String url,
                                     object remoteChannelData,
                                     out String objectURI )
{
    objectURI = null;
    String ChannelURI = null;

    if( url != null )
    {
        ChannelURI = Parse( url, out objectURI );
    }
    else
    {
        if( ( remoteChannelData != null ) &&
            C remoteChannelData is IChannelDataStore )
        {
            IChannelDataStore DataStore =
                ( IChannelDataStore ) remoteChannelData;

            ChannelURI = Parse( DataStore.ChannelUris[0],
                               out objectURI );
            if( ChannelURI != null )
            {
                url = DataStore.ChannelUris[0];
            }
        }
    }
}
```

```

    if( ChannelURI != null )
    {
        return ( IMessageSink ) m_ClientProvider.CreateSink(
            this, url, remoteChannelData );
    }

    objectURI = "";
    return null;
}

```

Большая часть кода *CreateMessageSink* посвящена определению URL, который следует передать *CreateSink*. Сначала проверяется на равенство *null* значение параметра *url*. Если параметр не равен *null*, то мы просто извлекаем из него URI канала и URI объекта с помощью функции *Parse*. Вызов *Parse* также считается проверкой на допустимость переданного URL. Если и возвращаемое значение, и *objectURI* равны *null*, то URL не подходит для данного канала. Если параметр *url* равен *null*, то необходимо получить нужную информацию из параметра *remoteChannelData*. Инфраструктура .NET Remoting получает *remoteChannelData* с серверной стороны канала. Данный объект должен иметь тип *System.Runtime.Remoting.Channels.IChannelDataStore*, что проверяется перед обращением к объекту с помощью ключевого слова *is*. Если объект имеет правильный тип, то после приведения типа он сохраняется в локальной переменной типа *IChannelDataStore*. Во время обсуждения *HttpChannel* говорилось, что объект *IChannelDataStore* содержит массив URI, поддерживаемых сервером для данного канала. Как и в случае общеизвестного объекта, значение, возвращаемое *Parse*, сравнивается с *null* для проверки правильности полученного нами URL. Теперь мы можем безопасно вызывать метод *CreateSink* нашего *FileClientChannelSinkProvider*.

Реализация класса *FileClientChannelSinkProvider*

Основная задача класса *FileClientChannelSinkProvider* — создание нашего транспортного приемника *FileClientChannelSink*. Так как *FileClientChannelSinkProvider* наследует от интерфейса *IChannelSinkProvider*, необходимо реализовать члены *CreateSink* и *Next*. Так как это последний провайдер в цепочке, *Next* всегда возвра-

щает *null*, *CreateSink* же всегда возвращает ссылку на новый *FileClientChannelSink*.

```
internal class
    FileClientChannelSinkProvider : IClientChannelSinkProvider
    {
        public IClientChannelSink CreateSink( IChannelSender
                                              channel, String url,
                                              Object
                                              remoteChannelData )
        {
            return new FileClientChannelSink( url );
        }

        public IClientChannelSinkProvider Next
        {
            get
            {
                return null;
            }
            set
            {
                throw new NotSupportedException();
            }
        }
    }
}
```

Как видите, *CreateSink* просто создает новый *FileClientChannelSink*, передавая ему URL.

Реализация класса *FileClientChannelSink*

FileClientChannelSink обрабатывает исполнение вызовов методов. На него возлагаются следующие обязанности:

- обработка синхронных вызовов методов;
- обработка асинхронных вызовов методов;
- запись и считывание из файла содержимого и *ITransportHeaders* и данных *Stream*;
- обработка сообщений, возвращаемых сервером.

FileClientChannelSink наследует от интерфейса *IClientChannelSink*. Так как это последний приемник в цепочке, функциональными окажутся только реализации *IClientChannelSink.ProcessMessage* и *IClientChannelSink.AsyncProcessRequest*. *ProcessMessage* и *AsyncPro-*

cessRequest отвечают за первые два пункта списка. Базовая структура нового класса показана ниже:

```
internal class FileClientChannelSink : IClientChannelSink
{
    private String m_PathToServer = null;

    public delegate void AsyncDelegate( String fileName,
                                        IClientChannelSinkStack sinkStack );

    public FileClientChannelSink( String url )
    {
        String ObjectURI;
        m_PathToServer = FileChannelHelper.Parse( url,
                                                    out ObjectURI );
    }

    public void AsyncProcessRequest(
        IClientChannelSinkStack sinkStack,
        IMessage msg,
        ITransportHeaders requestHeaders,
        Stream requestStream )
    {
    }

    public void AsyncProcessResponseC
        IClientResponseChannelSinkStack sinkStack,
        object state,
        ITransportHeaders headers,
        Stream stream )
    {
        throw new NotSupportedException();
    }

    public Stream GetRequestStream( IMessage msg,
                                    ITransportHeaders headers )
    {
        return null;
    }

    public void ProcessMessage( IMessage msg,
                                ITransportHeaders requestHeaders,
                                Stream requestStream,
                                out ITransportHeaders
                                responseHeaders,
```

```

        out Stream responseStream )
    {
        :
    }

    public IClientChannelSink NextChannelSink
    {
        get
        {
            return null;
        }
    }

    public IDictionary Properties
    {
        get
        {
            return null;
        }
    }
}

```

ProcessMessage обрабатывает все синхронные вызовы методов. Для этого он сначала должен подготовить данные, необходимые серверу для выполнения вызова метода. Затем метод отправляет на сервер сообщение-запрос и ожидает ответное сообщение. По получении от сервера ответного сообщения *ProcessMessage* помещает его данные в соответствующие переменные.

```

public void ProcessMessage( IMessage msg,
                           ITransportHeaders requestHeaders,
                           Stream requestStream,
                           out ITransportHeaders
                           responseHeaders,
                           out Stream responseStream )
{
    String uri = ExtractURI( msg );

    ChannelFileData data = new ChannelFileData( uri,
                                                requestHeaders,
                                                requestStream );

    String FileName = ChannelFileTransportWriter.Write(
        data,
        m_PathToServer,
        null );
}

```

```

FileChannelHelper.WriteSOAPStream( requestStream,
                                   FileName + "_SOAP" );

FileName = ChangeFileExtension.
            ChangeFileNameToClientExtension(
                                   FileName );

WaitForFile.Wait( FileName );

ChannelFileData ReturnData =
    ChannelFileTransportReader.Read(
                                   FileName );

responseHeaders = ReturnData.header;
responseStream = ReturnData.stream;
}

```

ProcessMessage — это первый из написанных нами методов, специально для нашего транспорта. Позднее мы поговорим о специфике этого метода, связанной с транспортом. Это позволит нам добиться максимальной абстракции от конкретного транспорта.

AsyncProcessRequest обрабатывает асинхронные вызовы методов удаленного объекта двух типов. Оба типа запросов форматируют и отправляют сообщение аналогично *ProcessMessage*. Отличие состоит в действиях, выполняемых после доставки сообщения. Первый тип запросов — *односторонний* (*OneWay*) асинхронный запрос. Они помечены атрибутом *OneWayAttribute*, означающим, что поступление ответа или подтверждения успеха не ожидается. Для выделения односторонних методов необходимо проверить содержимое объекта */Message*. Второй тип запросов — обычные асинхронные. Для них мы будем вызывать делегат, ожидающий ответного сообщения.

```

public void AsyncProcessRequest( IClientChannelSinkStack
                                sinkStack,
                                IMessage msg,
                                ITransportHeaders
                                requestHeaders,
                                Stream requestStream )
{
    String uri = ExtractURI( msg );

    ChannelFileData data = new ChannelFileData( uri,

```

```

                                requestHeaders,
                                requestStream };
String FileName = ChannelFileTransportWriter.Write( data,
                                                    m_PathToServer, null );
FileChannelHelper.WriteSOAPStream( requestStream,
                                    FileName + "_SOAP" );

if( !IsOneWayMethod( (IMethodCallMessage)msg ) )
{
    FileName =
        ChangeFileExtension.
            ChangeFileNameToClientExtension( FileName );
    AsyncDelegate Del = new AsyncDelegate(
                                                this.AsyncHandler );
    Del.BeginInvoke( FileName, sinkStack, null, null );
}
}

```

Наша реализация *AsyncProcessRequest* упаковывает сообщение и отправляет его серверу. Далее с помощью закрытого метода *FileClientChannelSink.IsOneWayMethod* проверяется, нужно ли ожидать ответное сообщение;

```

private bool IsOneWayMethod( IMethodCallMessage
                             methodCallMessage )
{
    MethodBase methodBase = methodCallMessage.MethodBase;
    return RemotingServices.IsOneWay(methodBase);
}

```

Статический метод *System.Runtime.RemotingServices.IsOneWay* возвращает булево значение, указывающее, помечен ли метод, описываемый параметром *MethodBase*, атрибутом *OneWayAttribute*.

Если метод не является односторонним, мы вызываем *AsyncDelegate*. При создании *AsyncDelegate* мы передаем ему метод *FileClientChannelSink.AsyncHandler*. Одного лишь этого метода недостаточно для завершения асинхронного вызова. Следует также передать *BeginInvoke* стек приемников: он содержит все приемники, которые должны обработать ответное сообщение.

Задача *AsyncHandler* — ожидание ответного сообщения от сервера, восстановление из него объектов *ITransportHeader* и *Stream*, а также передача их далее по цепочке приемников:

```

private void AsyncHandler( String FileName,
                           IClientChannelSinkStack sinkStack )
{
    try
    {
        if( WaitForFile.Wait( FileName ))
        {
            ChannelFileData ReturnData =
                ChannelFileTransportReader.Read(
                    FileName );

            sinkStack.AsyncProcessResponse( ReturnData.header,
                                           ReturnData.stream );
        }
    }
    catch (Exception e)
    {
        if (sinkStack != null)
        {
            sinkStack.DispatchException(e)
        }
    }
}

```

После вызова делегата текущий поток управления продолжает исполнение. Это позволяет приложению продолжить работу, не ожидая ответа от сервера.

Теперь нам необходимо создать классы для серверной стороны *FileChannel*. Для этого мы разработаем два класса: *FileServerChannel* и *FileServerChannelSink*, которые соответствуют серверным классам *HttpServerChannel* и *HttpServerTransportSink*, обсуждавшимся в разделе «Построение каналов».

Реализация класса *FileServerChannel*

FileServerChannel выполняет основную обработку на серверной стороне. Он отвечает за:

- ожидание входящих сообщений;
- создание приемника форматировщика;
- создание *FileServerChannelSink*;
- построение цепочки приемников.

При обсуждении разработки *FileServerChannel* решению этих задач мы уделим особое внимание.

FileServerChannel будет принимать сообщения; следовательно, он должен реализовать интерфейс *IChannelReceiver*. Так как *IChannelReceiver* наследует от *IChannel*, необходимо добавить к *FileServerChannel* три новых члена: *ChannelName*, *ChannelPriority* и *Parse*. Для поддержки функциональности *IChannelReceiver* нужно добавить члены *ChannelData*, *GetUrlsForUri*, *StartListening* и *StopListening*. Для начала мы рассмотрим открытый интерфейс *FileServerChannel*:

```
public class FileServerChannel : IChannelReceiver
{
    public FileServerChannel( String serverPath ){ ... }

    public FileServerChannel( IDictionary properties,
                             IServerChannelSinkProvider
                             provider )
    { ... }

    // Члены IChannel.
    public String Parse( String url,
                        out String objectURI ){ ... }

    public String ChannelName{ ... }

    public int ChannelPriority{ ... }

    // Члены IChannelReceiver.
    public void StartListening( Object data ){ ... }

    public void StopListening( Object data ){ ... }

    public String[] GetUrlsForUri( String objectURI ){ ... }

    public Object ChannelData{ ... }
}
```

Как и *FileClientChannel*, *FileServerChannel* имеет два конструктора. Первый — это простой конструктор, помещающий в закрытую переменную-член *m_ServerPath* значение, переданное в качестве параметра, и вызывающий закрытый метод *Init*. Переменная *m_ServerPath* задает каталог, за появлением файлов в котором будет следить сервер.

Второй конструктор позволяет более гибко настраивать параметры *FileServerChannel*. Как и первый конструктор, он также обязан вызвать *Init*.

```
public FileServerChannel( IDictionary properties,
                        IServerChannelSinkProvider provider )
{
    m_SinkProvider = provider;

    if( properties != null )
    {
        foreach (DictionaryEntry entry in properties)
        {
            switch ((String)entry.Key)
            {
                case "name":
                    m_ChannelName = ( String ) entry.Value;
                    break;
                case "priority":
                    m_ChannelPriority = ( int ) entry.Value;
                    break;
                case "serverpath":
                    m_ServerPath = ( String ) entry.Value;
                    break;
            }
        }

        // Так как конструктор FileChannel, из которого вызывается
        // этот конструктор, создает оба объекта FileClientChannel
        // и FileServerChannel, то для определения необходимости
        // ожидания входящих сообщений нам следует проверить
        // значение m_ServerPath.
        if( m_ServerPath != null )
        {
            Init();
        }
    }
}
```

Использование данного конструктора представляет собой единственный способ изменения значений свойств *ChannelName* и *ChannelPriority*, доступных только для чтения. Оба конструктора вызывают метод *Init*, который выполняет следующие действия:

1. создает форматировщик;
2. инициализирует объект *ChannelDataStore*;
3. помещает в объект *ChannelDataStore* данные из цепочки провайдера;
4. создает цепочку приемников;
5. вызывает метод *StartListening*.

При разработке *FileClientChannel* мы выбрали в качестве провайдера форматировщика *SoapClientFormatterSinkProvider*, следовательно, для сервера необходимо использовать соответствующий серверный провайдер — *SoapServerFormatterSinkProvider*.

```
private void Init()
{
    // Создать провайдер, если он не задан явно.
    if( m_SinkProvider == null )
    {
        // FileChannel по умолчанию использует
        // форматировщик SOAP.
        m_SinkProvider = new SoapServerFormatterSinkProvider();
    }

    // Инициализировать объект ChannelDataStore URI нашего канала.
    m_DataStore = new ChannelDataStore( null );
    m_DataStore.ChannelUris = new String[1];
    m_DataStore.ChannelUris[0] = "file://" + m_ServerPath;

    PopulateChannelData( m_DataStore, m_SinkProvider );

    IServerChannelSink sink =
        ChannelServices.CreateServerChannelSinkChain(
            m_SinkProvider, this );

    // Добавить к цепочке наш транспортный приемник.
    m_Sink = new FileServerChannelSink( sink, m_ServerPath );

    StartListening( null );
}
```

В предыдущем фрагменте кода в *m_DataStore* помещается URI нашего канала. Для прохода по цепочке провайдеров и сбора данных применяется закрытый метод *PopulateChannelData*:

```
private void PopulateChannelData( ChannelDataStore channelData,
                                IServerChannelSinkProvider
                                provider)
{
    while (provider != null)
    {
        provider.GetChannelData(channelData);

        provider = provider.Next;
    }
}
```

Метод *GetChannelData* извлекает информацию из члена *ChannelDataStore* провайдера. Так как *GetChannelData* является членом интерфейса *IServerChannelSinkProvider*, этот метод поддерживается всеми провайдерами каналных приемников.

В реализации *Init* интересен вызов *ChannelServices.CreateServerChannelSinkChain*. Этот статический метод строит серверную цепочку приемников. Получив ее, мы должны добавить к ней свой приемник *FileServerChannelSink*. Последнее, что делает *Init* — вызывает метод *StartListening*.

StartListening — первый из четырех методов *ChannelReceiver*, которые нам предстоит создать. Он должен создать и запустить поток для контроля за появлением в заданном каталоге входящих сообщений. После создания потока функция не должна блокировать исполнение главного потока, а вернуть управление. Это позволит серверу продолжать работу одновременно с ожиданием входящих сообщений;

```
public void StartListening( Object data )
{
    ThreadStart ListeningThreadStart = new ThreadStart(
        m_Sink.ListenAndProcessMessage );
    m_ListeningThread = new Thread( ListeningThreadStart );
    m_ListeningThread.IsBackground = true;
    m_ListeningThread.Start();
}
```

Делегату *ThreadStart* следует передать метод, исполняемый при старте нового потока. Данный метод — *ListenAndProcessMessage* — реализован в качестве открытого метода нашего приемника. После того как поток запущен, сервер готов к приему входящих

сообщений. Метод *StopListening* просто завершает работу потока ожидания, вызывая *m_ListeningThread.Abort*.

```
public void StopListening( Object data )
{
    if( m_ListeningThread != null )
    {
        m_ListeningThread.Abort();
        m_ListeningThread = null;
    }
}
```

Нам осталось реализовать еще два члена *IChannelReceiver* — *ChannelData* и *GetUrlsForUri*. *ChannelData* — это просто свойство, доступное только для чтения, возвращающее член *IChannelDataStore*. *GetUrlsForUri* принимает URI объекта и возвращает массив URL. Например, если *FileServerChannel.GetUrlsForUri* будет передан URI объекта *Demo.uri*, то в ответ должно быть получено *file://<m_ServerPath>/Demo.uri*.

```
public String[] GetUrlsForUri( String objectURI )
{
    String[] URL = new String[1];

    if ( !objectURI.StartsWith("/") )
    {
        objectURI = "/" + objectURI;
    }

    URL[0] = "file://" + m_ServerPath + objectURI;

    return URL;
}
```

Реализация членов *IChannel* очевидна. *ChannelName* и *ChannelPriority* — свойства, доступные только для чтения, которые возвращают соответствующие переменные-члены *m_ChannelName* и *m_ChannelPriority*. *Parse* вызывает общую реализацию *Parse* из класса *FileChannelHelper*.

Реализация класса *FileServerChannelSink*

Задача *FileServerChannelSink* состоит в передаче клиентских сообщений-запросов инфраструктуре .NET Remoting. *FileServerChannelSink* реализует интерфейс *IServerChannelSink*, поэтому мы должны реализовать члены *NextChannelSink*, *AsyncProcessRespon-*

се, *GetResponseStream* и *ProcessMessage*. Так как никакая обработка сообщения-запроса не требуется, то нам не нужна какая-либо функциональность в *ProcessMessage*. *GetResponseStream* генерирует *NotSupportedException*, так как нам не надо создавать поток. Конструктор *FileServerChannelSink* должен принимать ссылку на следующий приемник в цепочке. Доступ к ней осуществляется через свойство *NextChannelSink*, поддерживающее только чтение. Базовая структура класса *FileServerChannelSink* показана ниже:

```
internal class FileServerChannelSink : IServerChannelSink
{
    private IServerChannelSink m_NextSink = null;
    private string m_DirectoryToWatch;

    public FileServerChannelSink( IServerChannelSink nextSink,
                                  String directoryToWatch )
    {
        m_NextSink = nextSink;
        m_DirectoryToWatch = directoryToWatch;
    }

    public ServerProcessing ProcessMessage(
        IServerChannelSinkStack
        sinkStack,
        IMessage requestMsg,
        ITransportHeaders requestHeaders,
        Stream requestStream,
        out IMessage responseMsg,
        out ITransportHeaders
        responseHeaders,
        out Stream responseStream )
    {
        throw new NotSupportedException();
    }

    public void AsyncProcessResponse(
        IServerResponseChannelSinkStack
        sinkStack,
        Object state,
        IMessage msg,
        ITransportHeaders headers,
        Stream stream )
    {
    }
}
```

```

public Stream GetResponseStream(
    IServerResponseChannelSinkStack
    sinkStack,
    Object state,
    IMessage msg,
    ITransportHeaders headers )
{
    throw new NotSupportedException();
    return null;
}

public IServerChannelSink NextChannelSink
{
    get
    {
        return m_NextSink;
    }
}

public IDictionary Properties
{
    get
    {
        return null;
    }
}

internal void ListenAndProcessMessage()
{
    :
}
}

```

Основная часть реализации *FileServerChannelSink* заключена в *ListenAndProcessMessage*, который отвечает за выполнение следующих задач:

- ожидание сообщений;
- извлечение из сообщения данных для *ITransportHeaders* и *Stream*;
- передача запроса цепочке приемников.
- отправку ответа клиенту.

```

internal void ListenAndProcessMessage()
{

```

```

while( true )
{
    // Ждать сообщение от клиента.
    String FileName = WaitForFile.InfiniteWait();

    // Сервер получил сообщение; извлечь из
    // него данные.
    ChannelFileData Data = ChannelFileTransportReader.Read(
                                                FileName );

    ITransportHeaders MessageHeader = Data.header;
    MessageHeader[ CommonTransportKeys.RequestUri ] =
        Data.URI;
    Stream MessageStream = Data.stream;

    // Добавить себя к стеку приемников.
    ServerChannelSinkStack Stack =
        new ServerChannelSinkStack();
    Stack.Push(this, null);

    IMessage ResponseMsg;
    ITransportHeaders ResponseHeaders;
    Stream ResponseStream;

    // Начать обработку запроса в цепочке приемников.
    ServerProcessing Operation = m_NextSink.ProcessMessage(
        Stack,
        null,
        MessageHeader,
        MessageStream,
        out ResponseMsg,
        out ResponseHeaders,
        out ResponseStream );

    // Возвратить ответ клиенту.
    switch( Operation )
    {
        case ServerProcessing.Complete:
            Stack.Pop( this );
            ChannelFileData data = new ChannelFileData(
                null,
                ResponseHeaders,
                ResponseStream );

            String ClientFileName = ChangeFileExtension.
                ChangeFileNameToClientExtension(

```

```

        FileName );
ChannelFileTransportWriter.Write(
    data,
    null,
    ClientFileName );
FileChannelHelper.WriteSOAPMessageToFile(
    ResponseStream,
    ClientFileName +
    "_SOAP" );

break;
case ServerProcessing.Async:
    Stack.StoreAndDispatch(m_NextSink, null);
    break;
case ServerProcessing.OneWay:
    break;

```

О транспорте мы поговорим в конце раздела, поэтому сейчас пропустим эти детали. Первое, что делает *ListenAndProcessMessage*, — бесконечно ожидает сообщение. По получении сообщения-запроса из него извлекаются данные. Прежде чем передать данные цепочке приемников, мы создаем стек приемников. Для этого используется класс *System.Runtime.Remoting.Channel.ServerChainSinkStack*. Создав новый объект *ServerChainSinkStack*, мы вызываем его метод *Push*. Первый параметр — объект *IServerChannelSink*, второй параметр — произвольный объект, который позволяет связать с приемником некоторые данные состояния. Данные состояния используется только каналами, обрабатывающими *AsyncProcessResponse*, *ProcessMessage* и *GetResponseStream*, к которым наш канал не относится. *ProcessMessage* начинает обработку клиентского запроса и возвращает объект *ServerProcessing*. С его помощью удастся определить, что делать дальше. В случае *ServerProcessing.Complete* мы удаляем наш приемник из стека приемников средствами метода *ServerChainSinkStack.Pop*. *Pop* удаляет не только наш приемник, но и все остальные, добавленные после него.

Итак, у нас есть пять классов, реализующих основную часть функциональности нашего канала. Теперь необходимо реализовать класс, который объединит серверные и клиентские классы вместе.

Реализация класса *FileChannel*

Класс *FileChannel* очень прост. Его единственное назначение — предоставить унифицированный интерфейс для клиента и для сервера, поэтому он должен реализовать как *IChannelSender*, так и *IChannelReceiver*. У *FileChannel* имеется три конструктора:

```
public FileChannel();
public FileChannel( String serverPath );
public FileChannel( IDictionary properties,
                  IClientChannelSinkProvider
                  clientProviderChain,
                  IServerChannelSinkProvider
                  serverProviderChain );
```

Первый конструктор создает объект *FileClientChannel* и сохраняет его в закрытом члене *m_ClientChannel*. При использовании данного конструктора *FileChannel* может только отправлять сообщения-запросы. Второй конструктор помещает в закрытый член *m_ServerChannel* вновь создаваемый *FileServerChannel*. Этому конструктору необходимо указать каталог, который будет передан *FileServerChannel*. Третий конструктор создает и *FileServerChannel*, и *FileClientChannel*.

```
public FileChannel( IDictionary properties,
                  IClientChannelSinkProvider
                  clientProviderChain,
                  IServerChannelSinkProvider
                  serverProviderChain )
{
    m_ClientChannel = new FileClientChannel(
                                properties,
                                clientProviderChain );
    m_ServerChannel = new FileServerChannel(
                                properties,
                                serverProviderChain );
}
```

При настройке *FileChannel* с помощью конфигурационного файла инфраструктура .NET Remoting использует третий конструктор. Оставшиеся открытые члены *FileChannel* просто вызывают соответствующие члены, реализованные либо в *FileServerChannel*, либо в *FileClientChannel*.

```

public class FileChannel : IChannelSender, IChannelReceiver
{
    private FileClientChannel m_ClientChannel = null;
    private FileServerChannel m_ServerChannel = null;

    // Здесь конструкторы не показаны.

    public String Parse( String url, out String objectURI )
    {
        return FileChannelHelper.Parse( url, out objectURI );
    }

    public String ChannelName
    {
        get
        {
            if( m_ServerChannel != null )
            {
                return m_ServerChannel.ChannelName;
            }
            else
            {
                return m_ClientChannel.ChannelName;
            }
        }
    }

    public int ChannelPriority
    {
        get
        {
            if( m_ServerChannel != null )
            {
                return m_ServerChannel.ChannelPriority;
            }
            else
            {
                return m_ClientChannel.ChannelPriority;
            }
        }
    }

    public Object ChannelData
    {
        get
        {

```

```

        if( m_ServerChannel != null )
        {
            return m_ServerChannel.ChannelData;
        }

        return null;
    }

    public String[] GetUrlsForUri( String objectURI )
    {
        return m_ServerChannel.GetUrlsForUri( objectURI );
    }

    public void StartListening( Object data )
    {
        m_ServerChannel.StartListening( data );
    }

    public void StopListening( Object data )
    {
        m_ServerChannel.StopListening( data );
    }

    public IMessageSink CreateMessageSink(
        String url,
        object remoteChannelData,
        out String objectURI )
    {
        return m_ClientChannel.CreateMessageSink(
            url,
            remoteChannelData,
            out objectURI );
    }
}

```

Реализация класса *FileChannelHelper*

Следующий этап — создание вспомогательного класса, методы которого реализуют общую функциональность для классов *FileClientChannel* и *FileServerChannel*. Так как все эти методы статические, сделаем конструктор класса закрытым. В приведенных ранее фрагментах кода мы несколько раз вызывали *FileChannelHelper.Parse*. Теперь нам предстоит создать этот метод. Как уже


```

{
    StreamWriter Writer = new StreamWriter( FileName );
    StreamReader sr = new StreamReader(stream);
    String line;
    while ((line = sr.ReadLine()) != null)
    {
        Writer.WriteLine(line);
    }
    Writer.Flush();
    Writer.Close();

    stream.Position = 0;
}

```

Создание класса транспорта *FileChannel*

До настоящего момента мы избегали обсуждения вопросов, затрагивающих специфику данного транспорта, а именно обеспечение четкого разделения процедур создания канала и реализации транспорта. За исключением конфигурационной информации код, зависящий от транспорта, присутствует только в следующих методах:

- *FileClientChannelSink.ProcessMessage;*
- *FileClientChannelSink.AsyncProcessRequest;*
- *FileClientChannelSink.AsyncHandler;*
- *FileServerChannelSink.ProcessMessage.*

Как чтение, так и запись сообщений нашим транспортом представляет собой двухэтапный процесс. Первый — загрузка в класс *ChannelFileData* данных, которые нужно отправить серверу. После того как они помещены в этот класс, мы сериализуем его в заданный файл с помощью *FileStream*. При чтении данных этапы выполняются в обратном порядке.

Данные, содержащиеся в *ChannelFileData*, состоят из URI запроса, *ITransportHeaders* и *Stream*:

```

[Serializable]
public class ChannelFileData
{
    private String m_URI = null;
    private ITransportHeaders m_Header = null;
    private byte[] m_StreamBytes = null;
}

```

```

public ChannelFileData( String URI,
                        ITransportHeaders headers,
                        Stream stream )
{
    String objectURI;
    String ChannelURI = FileChannelHelper.Parse(
                                                URL,
                                                out objectURI );

    if( ChannelURI == null )
    {
        objectURI = URL;
    }
    m_URI = objectURI;
    m_Header = headers;
    m_StreamBytes = new Byte[ (int)stream.Length ];
    stream.Read( m_StreamBytes, 0, (int) stream.Length );
    stream.Position = 0;
}

public String URI
{
    get
    {
        return m_URI;
    }
}

public ITransportHeaders header
{
    get
    {
        return m_Header;
    }
}

public Stream stream
{
    get
    {
        return new MemoryStream( m_StreamBytes, false );
    }
}
}

```

Во-первых, обратите внимание на наличие атрибута *Serializable* у класса *ChannelFileData*. Он необходим для реализации следующей

го этапа. Единственный способ задания данных для *ChannelFileData* считается конструктор. Для доступа к этим данным имеются свойства, доступные только для чтения.

Теперь, когда у нас имеется класс с данными, подлежащими сериализации, нам нужен класс, который записывает файл на диск. Этот класс *ChannelFileTransportWriter* будет иметь единственный метод *Write*. В качестве параметров последний принимает объект *ChannelFileData*, каталог, в котором должен быть создан файл, и имя файла.

```
public class ChannelFileTransportWriter
{
    private ChannelFileTransportWriter()
    {
    }

    public static String Write( ChannelFileData data,
                               String ServerPath,
                               String FileName )
    {
        // Если FileName равно null,
        // сгенерировать имя с помощью Guid.NewGuid.
        if( FileName == null )
        {
            FileName = Path.Combine( ServerPath,
                                     Guid.NewGuid().ToString() +
                                     ".chn.server" );
        }

        // Добавить _Temp к имени файла, чтобы исключить
        // обращение к файлу со стороны сервера или клиента,
        // пока мы не закончим запись в него. После окончания
        // записи файл будет переименован.
        String TempFileName = FileName + "_Temp";
        IFormatter DataFormatter = new SoapFormatter();
        Stream DataStream = new FileStream( TempFileName,
                                           FileMode.Create,
                                           FileAccess.Write,
                                           FileShare.None );

        DataFormatter.Serialize( DataStream, data);
        DataStream.Close();

        File.Move( TempFileName, FileName );
        return FileName;
    }
}
```

Для *ChannelFileTransportWriter* следует ряд ключевых моментов. Во-первых, *ChannelFileTransportWriter* не имеет состояния, так что з нем не нужны экземпляры данного класса. Это позволяет сделать его конструктор закрытым, а метод *Write* — статическим. Обратите внимание на использование *Guid.NewGuid* для генерации уникальных имен. Таким образом удастся избежать конфликта имен при использовании сервером нескольких каналов.

Теперь, когда мы можем создавать файлы сообщений запросов и ответов, следует научиться читать их. Для этого мы создадим класс *ChannelFileTransportReader* с единственным методом *Read*, который заполняет объект *ChannelFileData*:

```
public class ChannelFileTransportReader
{
    private ChannelFileTransportReader()
    {
    }

    public static ChannelFileData Read( String FileName )
    {
        IFormatter DataFormatter = new SoapFormatter();
        Stream DataStream = new FileStream( FileName,
                                           FileMode.Open,
                                           FileAccess.Read,
                                           FileShare.Read);

        ChannelFileData data =
            (ChannelFileData) DataFormatter.Deserialize(
                                                         DataStream );

        DataStream.Close();
        File.Move( FileName, FileName + "_processed" );

        return data;
    }
}
```

Обратите внимание: после завершения обработки файла методом *Read* он переименовывается с добавлением к его имени строки *_processed*. Это позволит нам проследить историю пересылки сообщений между клиентом и сервером.

Последним классом транспорта, который следует реализовать, является *WaitForFile*. В этом классе будет два метода: *Wait* и *InfiniteWait*. Первый ожидает появления файла в течение определенного периода времени.

```

public static bool Wait( String filename )
{
    int RetryCount = 120;

    while( RetryCount > 0 )
    {
        Thread.Sleep( 500 );

        if( File.Exists( filename ))
        {
            return true;
        }

        RetryCount--;
    }

    return false;
}

```

В течение 1 минуты *Wait* выполняет проверки через каждые полсекунды. Более интеллектуальная версия *FileChannel* позволила бы задавать число повторений и время ожидания в конфигурационном файле. *InfiniteWait* каждые полсекунды проверяет наличие в заданном каталоге файла с расширением *.server*.

```

public static String InfiniteWait( String DirectoryToWatch )
{
    // Бесконечный цикл пока файл не будет найден.
    while( true )
    {
        String[] File = Directory.GetFiles( DirectoryToWatch,
                                             "*.server" );

        if( File.Length > 0 )
        {
            return File[0];
        }

        Thread.Sleep( 500 );
    }
}

```

Нестандартные каналные приемники

Сообщения запросов и ответов проходят от одного конца цепочки приемников до другого. В это время приемники имеют воз-

возможность выполнять некоторые действия на основании информации, *содержащейся в сообщении*. Создав нестандартный приемник, мы можем подключиться к цепи *приемников* и получить доступ к каждому сообщению перед отправкой и после приема. Однако приемники умеют не только манипулировать сообщениями. На основании содержимого сообщения приемник способен выполнять определенные *действия*. Наконец, нестандартные приемники не должны быть *симметричными*, то есть они не обязаны иметь приемник и на *сервере*, и на клиенте. В реальных системах приемники применяются при:

- **шифровании** Приемник может шифровать все сообщения, пересылаемые между *отправителем* и получателем. Приемник шифрования должен быть как на отправителе, так и на получателе;
- **регистрации** Приемник может помещать в некую базу данных сообщение каждый раз при создании объекта или вызове метода. Такой приемник может располагаться только на сервере, только на клиенте или и там, и там;
- **контроле времени доступа** Приемник способен блокировать вызовы удаленных *объектов* в течение определенного периода времени. Такой приемник может располагаться только на *сервере*, только на клиенте или и там, и там;
- **аутентификации** Приемник может запрещать доступ некоторым пользователям на основании информации, *содержащейся в сообщении*, посылаемом клиентом. В этом случае приемники размещаются и на клиенте, и на *сервере*, но выполняют разные операции. Клиент добавляет к сообщению *аутентификационную* информацию, а сервер на основании этой информации определяет возможность вызова метода.

В первой части главы мы создали канал для нестандартного транспорта. При этом нам пришлось создать нестандартный приемник для клиента и сервера. В этом разделе мы разработаем приемник, который позволит изменить поведение цепи приемников. При создании классов для серверной стороны канала мы не создали класс, реализующий *IServerChannelSinkProvider*. Это было единственное место, где нарушалась симметрия между серверными и клиентскими классами. Нам не пришлось создавать про-

вайдер серверного приемника, так как мы оказались первым приемником в цепочке. Реализация *IServerChannelSinkProvider* потребует, чтобы поместить нестандартный приемник в цепочку приемников. Члены этого интерфейса показаны в табл. 7-10.

Таблица 7-10. Члены

System.Runtime.Remoting.Channel.IServerChannelSinkProvider

Член	Тип члена	Описание
<i>Next</i>	Свойство	Позволяет задавать и получать следующий объект <i>IServerChannelSinkProvider</i>
<i>CreateSink</i>	Метод	Возвращает <i>IServerChannelSink</i> для начала новой цепочки приемников
<i>GetChannelData</i>	Метод	Возвращает объект <i>IChannelDataStore</i> для текущего объекта <i>IServerChannelSinkProvider</i>

Создание приемника контроля времени доступа

Теперь мы поговорим о создании нестандартного приемника, который позволит блокировать вызовы методов удаленного объекта в течение некоторого периода времени. Процесс создания состоит из трех этапов:

- создания приемника *AccessTimeServerChannelSink*, реализующего *IServerChannelSink*;
- создания провайдера *AccessTimeServerChannelSinkProvider*, реализующего *IServerChannelSinkProvider*;
- реализации функциональности нестандартного приемника.

Назначение нестандартного приемника *Access Time* — управление временем, когда возможны вызовы удаленного объекта. В процессе создания канала можно задать начальное и конечное время. При попытке вызова метода в интервале между этими моментами времени вызов не будет обработан и возвратит ошибку.

Проекты *AccessTime*

Пример пользовательского приемника *AccessTime* содержит следующие проекты :

- **AccessTimeSinkLib** реализация приемника и провайдера;
- » • **DemonstrationObjects** класс *Demo*. Клиент и сервер в нашем примере применяют удаленный экземпляр класса *Demo*;
- » **Server** регистрирует *AccessTimeServerChannelSinkProvider* и *AccessTimeServerChannelSink* для использования канальными сервисами. Затем данный модуль просто ожидает завершения своей работы пользователем по нажатию Enter;
- » **Client** как и проект сервера, регистрирует *AccessTimeServerChannelSinkProvider* и *AccessTimeServerChannelSink* для использования канальными сервисами. Затем проект демонстрирует вызов метода удаленного объекта.

Реализация класса *AccessTimeServerChannelSink*

Начнем с общего обзора открытого интерфейса *AccessTimeServerChannelSink*:

```
public class AccessTimeServerChannelSink : IServerChannelSink
{
    public AccessTimeServerChannelSink( IServerChannelSink nextSink,
                                        String blockStartTime,
                                        String blockStopTime,
                                        bool isHttpChannel )
    {
    }

    public ServerProcessing ProcessMessage(
        IServerChannelSinkStack sinkStack,
        IMessage requestMsg,
        ITransportHeaders requestHeaders,
        Stream requestStream,
        out IMessage responseMsg,
        out ITransportHeaders responseHeaders,
        out Stream responseStream )
    {
    }
}
```

```

    }

    public void AsyncProcessResponse(
        IServerResponseChannelSinkStack sinkStack,
        Object state,
        IMessage msg,
        ITransportHeaders headers,
        Stream stream )
    {
    }

    public Stream GetResponseStream(
        IServerResponseChannelSinkStack sinkStack,
        Object state,
        IMessage msg,
        ITransportHeaders headers )
    {
        return null;
    }

    public IServerChannelSink NextChannelSink
    {
        get
        {
            return m_NextSink;
        }
    }

    public IDictionary Properties
    {
        get
        {
            return null;
        }
    }
}

```

Как видите, приемник *AccessTimeServerChannel* очень прост. Среди его открытых членов фактическую работу выполняют конструктор и *ProcessMessage*. Конструктор выглядит так:

```

public AccessTimeServerChannelSink( IServerChannelSink nextSink,
                                    String blockStartTime,
                                    String blockStopTime,
                                    bool isHttpChannel )
{

```



```

        out responseMsg,
        out responseHeaders,
        out responseStream );
    }
    else
    {
        ifC m_IsHttpChannel )
        {
            responseMsg = null;
            responseStream = null;

            responseHeaders = new TransportHeaders();
            responseHeaders["__HttpStatusCode"] = "403";

            return ServerProcessing.Complete;
        }
        else
        {
            throw new RemotingException( "Attempt made to
            call a " + "method during a blocked time" );
        }
    }
}

```

Во-первых, обратите внимание на вызов закрытого члена *IsBlockTimePeriod*. Этот метод сначала проверяет, содержат ли переменные-члены ненулевые значения времени. Если так, то выполняется сравнение времен и возвращается *true*, если запрос должен быть заблокирован, либо *false*, если запрос следует обработать. В том случае если блокирование не требуется, мы пересылаем те же самые параметры *m_NextSink.ProcessMessage*. В случае блокирования ответ посылается клиенту одним из двух способов. Если транспортный канал, используемый данной цепочкой приемников, — это *HttpChannel*, то возвращается код статуса 403 протокола HTTP. По получении этого кода состояния клиентом генерируется исключение. Если же используется канал, отличный от *HttpChannel*, мы генерируем *RemotingException*. Это исключение помещается в сообщение-ответ и снова генерируется на клиентской стороне. Но прежде, чем что-либо из описанного выше произойдет, *AccessTimeServerChannelSinkProvider* должен создать объект *AccessTimeServerChannelSink*.

Реализация класса *AccessTimeServerChannelSinkProvider*

Основная задача *AccessTimeServerChannelSinkProvider* — вставка нашего приемника в цепочку приемников. Его вторая задача состоит в сборе данных необходимых приемнику, *AccessTimeServerChannelSinkProvider* реализует интерфейс *IServerChannelSinkProvider*:

```
public class AccessTimeServerChannelSinkProvider :
    IServerChannelSinkProvider
{
    private IServerChannelSinkProvider m_NextProvider;

    String m_BlockStartTime;
    String m_BlockStopTime;

    public AccessTimeServerChannelSinkProvider(
        IDictionary properties,
        ICollection providerData )
    {
        // Получить начальное и конечное время.
        // Если эти ключи не будут найдены среди свойств,
        // то будет сгенерировано исключение.
        m_BlockStartTime = ( String )properties["BlockStartTime"];
        m_BlockStopTime = ( String )properties["BlockStopTime"];
    }

    public IServerChannelSink CreateSink( IChannelReceiver
                                         channel )
    {
        bool IsHttpChannel = channel is HttpServerChannel;

        IServerChannelSink nextSink = null;
        if( m_NextProvider != null )
        {
            nextSink = m_NextProvider.CreateSink( channel );
        }

        return new AccessTimeServerChannelSink(
            nextSink,
            m_BlockStartTime,
            m_BlockStopTime,
            IsHttpChannel );
    }

    public void GetChannelData( IChannelDataStore channelData )
    {
    }
}
```

```

    {
    }

    public IServerChannelSinkProvider Next
    {
        get
        {
            return m_NextProvider;
        }
        set
        {
            m_NextProvider = value;
        }
    }
}

```

Самым важным методом *AccessTimeServerChannelSinkProvider* считается *CreateSink*. Он отвечает за:

- создание оставшейся части цепочки приемников;
- создание *AccessTimeServerChannelSink*;
- проверку, не является ли *IChannelReceiver* типом *HttpServerChannel*.

Для создания остатка цепочки приемников мы вызываем *CreateSink* следующего провайдера, ссылка на который хранится в *m_NextProvider*. Этот член устанавливается инфраструктурой .NET Remoting посредством свойства *Next*. После возврата из *m_NextProvider.CreateSink* мы получим ссылку на первый приемник в цепочке, который передается затем конструктору *AccessTimeServerChannelSink*, что позволяет нашему приемнику подключиться к цепочке.

Добавление *AccessTimeServerChannelSink* в конфигурационный файл

Следующий конфигурационный файл демонстрирует, как добавляется приемник;

```

<configuration>
  <system.runtime.remoting>
    <application>
      <service>
        <wellknown mode="Singleton"
                    type="DemonstrationObjects.Demo,

```

```

        DemonstrationObjects"
        objectUri="DemoURI" />
    </service>
    <channels>
        <channel ref="http" port="4000">
            <serverProviders>
                <provider type="AccessTimeSyncLib.
                    AccessTimeServerChannelSinkProvider,
                    AccessTimeSinkLib"
                    BlockStartTime="10:00"
                    BlockStopTime="16:00"/>
                <formatter ref="soap"/>
            </serverProviders>
        </channel>
    </channels>
</application>
</system.runtime.remoting>
</configuration>

```

Обратите внимание на добавленный к элементу *channel* элемент *serverProviders*. Он содержит два элемента. Первый из них *provider*, в котором с помощью атрибута *type* добавляется наш приемник. Здесь имеются два атрибута, определяющие интервал запрета вызовов. Вторым элементом внутри *serverProviders* является *formatter*. Как вы помните, конструктор *FileServerChannel* автоматически создает форматировщик только тогда, когда параметр *provider* равен *null*; таким образом, при добавлении приемника мы обязаны задать форматировщик.

Заключение

В этой главе мы на примере *HttpChannel* рассмотрели структуру каналов. Далее с помощью этих знаний мы создали собственный канал, использующий в качестве транспорта файловую систему. Независимо от того, потребуется ли вам когда-либо создавать свой канал, вы теперь должны отлично представлять себе, что же происходит «за кулисами», когда вы используете *HttpChannel* и *TcpChannel*. В заключение главы мы разработали нестандартный приемник. В главе 8 вы расширите свои познания о приемниках сообщений, создав приемник форматировщика.

Глава 8

Форматировщики сериализации

В этой главе мы разработаем форматировщик сериализации, подключаемый к инфраструктуре .NET Remoting. Вы узнаете об архитектуре сериализации объектов, используемой .NET Framework, а также о ряде классов .NET Framework, применяемых при создании форматировщиков сериализации. В заключение мы разработаем серверный и клиентский приемники сериализации, которые будут использоваться для сериализации сообщений .NET Remoting.

Сериализация объектов

Сериализация — это процесс преобразования экземпляра объекта в последовательность бит, которую можно затем вывести в поток или на носитель. Десериализация — это процесс преобразования сериализованного потока бит в экземпляр некоторого объектного типа. Как отмечалось в главе 2, инфраструктура .NET Remoting использует сериализацию для переноса передаваемых по значению объектных типов через границы .NET Remoting. В последующих разделах мы расскажем о том, как работает сериализация в .NET Framework, они понадобятся вам, когда мы приступим к разработке собственного форматировщика.

ПРИМЕЧАНИЕ Подробнее о сериализации объектов — в документации MSDN и превосходной серии статей Джеффри Рихтера в его колонке MSDN Magazine за апрель и июль 2002 года.

Атрибут *Serializable*

.NET Framework предоставляет разработчикам объектов удобный механизм сериализации. Чтобы сделать класс, структуру, делегат или перечисление сериализуемыми, достаточно просто снабдить их атрибутом *SerializableAttribute*, как показано ниже;

```
[Serializable]
class SomeClass
{
    public int m_public = 5000;
    private int m_private = 5001;
}
```

Так как класс *SomeClass* имеет атрибут *SerializableAttribute*, то общезыковая исполняющая среда будет выполнять сериализацию автоматически. Чтобы не допустить сериализацию поля, являющегося членом типа с атрибутом *SerializableAttribute*, этот член можно снабдить атрибутом *NonSerializedAttribute*.

Для сериализации экземпляров типа *SomeClass* нам понадобится форматировщик, который будет выполнять сериализацию, и поток для хранения полученных бит. Как упоминалось в главе 2, .NET Framework предоставляет для сериализации графов объектов в потоки классы *SoapFormatter* и *BinaryFormatter*. В следующем примере экземпляр типа *SomeClass* сериализуется в *MemoryStream* с помощью *SoapFormatter*:

```
// Создать поток в памяти и сериализовать в него
// новый экземпляр SomeClass с помощью форматировщика.
MemoryStream s = new MemoryStream();
SoapFormatter fm = new SoapFormatter();
fm.Serialize( s, new SomeClass() );

// Вывести содержимое потока на консоль.
StreamReader r = new StreamReader(s);
s.Position = 0;
Console.WriteLine( r.ReadToEnd() );
s.Position = 0;

// Десериализовать поток в экземпляр SomeClass.
SomeClass sc = (SomeClass)fm.Deserialize(s);
```

В результате исполнения этого кода экземпляр *SomeClass* сериализуется в экземпляр *MemoryStream* в формате SOAP. Далее по-

казано содержимое потока в памяти после сериализации в него экземпляра *SomeClass*. (Для лучшей читабельности добавлены пробелы и переносы строк.)

```
<SOAP-ENV:Envelope
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  xmlns:SOAP-ENC="http://schemas.xmlsoap.org/soap/encoding/"
  xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
  xmlns:clr="http://schemas.microsoft.com/soap/encoding/clr/1.0"
  SOAP-ENV:encodingStyle="http://schemas.xmlsoap.org/soap/
    encoding/">
  <SOAP-ENV:Body>
    <a1:SomeClass id="ref-1" xmlns:
      a1="http://schemas.microsoft.com/clr/nsassem/
        RemoteObjects/
        RemoteObjects%2C%20Version%3D1.0.904.25890%2C%20
        Culture%3Dneutral%2C%20PublicKeyToken%3Dnull">
      <m_public>5000</m_public>
      <m_private>5001</m_private>
    </a1:SomeClass>
  </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

Как видите, внутри элемента *<SOAP-ENV:Envelope>* *SoapFormatter* сериализовал экземпляр *SomeClass* как дочерний элемент элемента *<SOAP-ENV:Body>*. Обратите внимание, что элемент *<a1:SomeClass>* содержит атрибут *id* равный *ref-1*. Как вы узнаете далее из этой главы, во время сериализации графа объектов форматировщики присваивают каждому сериализованному объекту идентификатор, используемый во время сериализации и десериализации. После атрибута *id* элемент *<a1:SomeClass>* содержит псевдоним пространства имен *xml*, задающий полное имя, версию, региональные стандарты и маркер открытого ключа сборки, содержащей тип *SomeClass*. Каждый дочерний элемент элемента *<a1:SomeClass>* соответствует члену класса и содержит его значение. Значение члена *m_public* равно 5000, а члена *m_private* — 5001.

Расширение механизма сериализации объекта

Хотя атрибут *SerializableAttribute* очень прост в использовании, иногда он не соответствует требованиям сериализации. .NET Framework позволяет типу с атрибутом *SerializableAttribute* выпол-

нять свою сериализацию самостоятельно путем реализации интерфейса *ISerializable*. Этот интерфейс содержит единственный метод *GetObjectData*:

```
void GetObjectData(
    SerializationInfo info,
    StreamingContext context
);
```

Если в процессе сериализации форматировщик встречается экземпляр типа, поддерживающего интерфейс *ISerializable*, то он вызывает метод *GetObjectData*, передавая ему ссылку на экземпляр *SerializationInfo* и ссылку на экземпляр *StreamingContext*. Класс *SerializationInfo* по сути представляет собой специализированный словарь, содержащий пары «ключ/значение», которые форматировщик будет выводить в поток. Некоторые наиболее важные открытые члены класса *SerializationInfo* показаны в табл. 8-1.

Таблица 8-1. Важнейшие открытые члены *System.Runtime.Serialization.SerializationInfo*

Член	Тип члена	Описание
<i>AssemblyName</i>	Свойство для чтения и записи	Полное имя сборки, содержащей сериализуемый тип. Включает информацию о версии, региональных стандартах и маркер открытого ключа. В своей реализации <i>GetObjectData</i> вы можете изменить значение этого свойства, чтобы задать сборку, которая будет использоваться при десериализации
<i>FullTypeName</i>	Свойство для чтения и записи	Эквивалентно <i>Type.FullName</i> для сериализуемого типа. В своей реализации <i>GetObjectData</i> вы можете изменить значение этого свойства, и тогда тип будет десериализован как другой тип
<i>MemberCount</i>	Свойство только для чтения	Возвращает число членов, добавленных к данному экземпляру <i>SerializationInfo</i>
<i>AddValue</i>	Метод	Добавляет поименованное значение к экземпляру <i>SerializationInfo</i> . Этот метод имеет ряд перегружаемых вариантов в зависимости от типов добавляемых объектов

см. след. стр.

Таблица 8-1. (окончание)

Член	Тип члена	Описание
<i>GetValue</i>	Метод	Возвращает поименованное значение из экземпляра <i>SerializationInfo</i> . Имеется ряд перегруженных вариантов для разных типов

Объект, реализующий *GetObjectData*, использует экземпляр *SerializationInfo*, на который указывает первый параметр, *info*, для сериализации информации, которая потребуется затем для десериализации. Вторым параметром *GetObjectData* — *context*, ссылается на экземпляр *StreamingContext* и соответствует объекту, на который ссылается свойство *Context* форматировщика. Экземпляр *StreamingContext* имеет два свойства — *Context* и *State*, несущие дополнительную информацию, которая влияет на процесс сериализации. Оба свойства доступны только для чтения и задаются как параметры конструктора *StreamingContext*. Свойство *State* представляет собой произвольную побитовую комбинацию членов перечислимого типа *StreamingContextStates*: *All*, *Clone*, *CrossAppDomain*, *CrossMachine*, *CrossProcess*, *File*, *Other*, *Persistence* и *Remoting*. Свойство *Context* может ссылаться на любой объект и используется для передачи алгоритму сериализации и десериализации данных, определяемых пользователем.

Ниже показан класс, реализующий интерфейс *ISerializable*:

```
[Serializable]
public class SomeClass2 : ISerializable
{
    public int m_public = 5000;
    private int m_private = 5001;

    public void GetObjectData( SerializationInfo info,
                               StreamingContext context)
    {
        //
        // Добавить текущее время к данным сериализации.
        info.AddValue( "TimeStamp", DateTime.Now );

        //
        // Сериализовать члены объекта,
        info.AddValue( "m1", m_public );
        info.AddValueC "m2", m_public};
    }
}
```

```

!
// Специальный конструктор сериализации.
protected SomeClass2( SerializationInfo info,
                      StreamingContext context)
{
    // Получить члены объекта из
    // экземпляра SerializationInfo .
    m_public = info.GetInt32("m1");
    m_private = info.GetInt32("m2");

    // Получить отметку времени.
    DateTime ts = info.GetDateTime("TimeStamp");
}
}

```

Тут *SomeClass2* реализует метод *ISerializable.GetObjectData*. Посредством параметра *info* добавляется новое значение с именем *TimeStamp*, содержащее текущее время и дату. Обратите внимание, что в дополнение к реализации *GetObjectData* *SomeClass2* определяет особый конструктор, принимающий такие же параметры, что и *GetObjectData*. Если конструктор такого вида отсутствует, то исполняющая среда сгенерирует исключение во время десериализации экземпляров данного типа. Ниже показано, как выглядит сериализованный экземпляр *SomeClass2* в формате SOAP:

```

<SOAP-ENV:Envelope
  xmlns:xsi=http://www.w3.org/2001/XMLSchema-instance
  xmlns:xsd=http://www.w3.org/2001/XMLSchema
  xmlns:SOAP-ENC=http://schemas.xmlsoap.org/soap/encoding/
  xmlns:SOAP-ENV=http://schemas.xmlsoap.org/soap/envelope/
  xmlns:clr=http://schemas.microsoft.com/soap/encoding/clr/1.0
  SOAP-ENV:encodingStyle="http://schemas.xmlsoap.org/soap/
    encoding/">
  <SOAP-ENV:Body>
    <a1:SomeClass2 id="ref-1" xmlns:
      a1="http://schemas.microsoft.com/clr/nsassem/
        RemoteObjects/
        RemoteObjects%2C%20Version%3D1.0.904.32400%2C%20
        Culture%3Dneutral%2C%20PublicKeyToken%3Dnull">
      <TimeStamp xsi:type="xsd:dateTime">
        2002-06-23T19:00:22.7003264-04:00
      </TimeStamp>
      <m1>5000</m1>
    </a1>
  </SOAP-ENV:Body>
</SOAP-ENV:Envelope>

```

```

        <m2>5001</m2>
      </a1:SomeClass2>
    </SOAP-ENV:Body>
  </SOAP-ENV:Envelope>

```

Обратите внимание, что элемент `<a1:SomeClass2>` содержит три дочерних элемента, соответствующих трем значениям, добавленным в экземпляр *SerializationInfo* о *GetObjectData*. Имя каждого дочернего элемента соответствует имени, переданному методу *SerializationInfo.AddData*. Эти имена должны быть уникальными в пределах данного экземпляра *SerializationInfo*.

Сериализация графов объектов

В предыдущих примерах целочисленные члены отображались как дочерние элементы соответствующего родительского элемента. Это связано с тем, что *SoapFormatter* сериализует элементарные типы подстановкой в тело объекта (*inline*). Давайте рассмотрим пример сериализации объекта, член которого ссылается на экземпляр другого объекта. В данном случае форматировщик вместо подстановки указываемого ссылкой объекта подставляет его идентификатор. Сам указываемый объект будет сериализован форматировщиком в другой части потока.

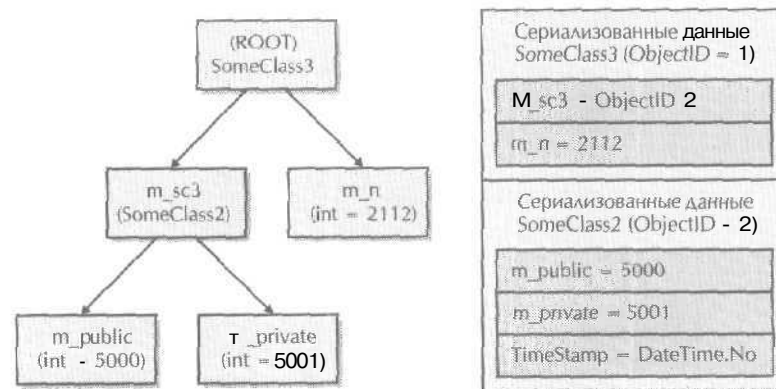


Рис. 8-1. Граф объектов

На рис. 8-1 показан граф объектов, полученный в результате создания экземпляра класса *SomeClass3*, который определен так:

```
[Serializable]
public class SomeClass3
{
    public SomeClass2 m_sc3 = new SomeClass2();
    public int m_n = 2112;
}
```

Ниже приведен пример сериализации экземпляра типа *SomeClass3* в формате **SOAP**:

```
<SOAP-ENV:Envelope
  xmlns:xsi=http://www.w3.org/2001/XMLSchema-instance
  xmlns:xsd=http://www.w3.org/2001/XMLSchema
  xmlns:SOAP-ENC=http://schemas.xmlsoap.org/soap/encoding/
  xmlns:SOAP-ENV=http://schemas.xmlsoap.org/soap/envelope/
  xmlns:clr=http://schemas.microsoft.com/soap/encoding/clr/1.0
  SOAP-ENV:encodingStyle="http://schemas.xmlsoap.org/soap/
    encoding/">
  <SOAP-ENV:Body>
    <a1:SomeClass3 id="ref-1" xmlns:
      a1="http://schemas.microsoft.com/clr/nsassem/
        BasicSerialization/BasicSerialization%2C%20
        Version%3D1.0.905.37158%2C%20Culture%3Dneutral%2C%20
        PublicKeyToken%3Dnull">
      <m_sc2 href="#ref-3"/>
      <m_n>2112</m_n>
    </a1:SomeClass3>
    <a1:SomeClass2 id="ref-3" xmlns:
      a1="http://schemas.microsoft.com/clr/nsassem/
        BasicSerialization/BasicSerialization%2C%20
        Version%3D1.0.905.37158%2C%20Culture%3Dneutral%2C%20
        PublicKeyToken%3Dnull">
      <TimeStamp xsi:type="xsd:dateTime">
        2002-06-24T21:38:42.8411136-04:00
      </TimeStamp>
      <m1>5000</m1>
      <m2>5001</m2>
    </a1:SomeClass2>
  </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

Обратите внимание, что элемент *<SOAP-ENV:Body>* содержит два дочерних элемента, каждый из которых соответствует сериализованному экземпляру некоторого объекта. Экземпляр *SomeClass3* является корнем графа объектов и выступает первым дочерним элементом элемента *<SOAP-ENV:Body>*. Дочерний эле-

мент `<m_sc2>` элемента `<a1:SomeClass3>` содержит атрибут `href`, ссылающийся на элемент с идентификатором `ref-3`, который соответствует экземпляру `SomeClass2`, сериализованному в элементе `<a1:SomeClass2>`.

Десериализация графа объектов

В процессе десериализации графа объектов форматировщик создает и инициализирует объекты графа по мере чтения сериализованных данных из потока. В процессе десериализации форматировщик отслеживает все объекты и их идентификаторы. Каждый член объекта сериализованного графа может ссылаться на:

- неинициализированный объект, который еще не был десериализован;
- частично инициализированный десериализованный объект;
- полностью инициализированный десериализованный объект,

Форматировщик может встретить объектный член, ссылающийся на другой, еще не десериализованный объект, такой, как дочерний элемент `<m_sc2>` элемента `<a1:SomeClass3>` из предыдущего примера. В этом случае форматировщик обновляет внутреннюю структуру, связывающую объектный член с идентификатором объекта, на который он ссылается. Все объектные члены, ссылающиеся на данный объект, будут инициализированы позднее, когда форматировщик десериализует этот объект. Если объектный член ссылается на уже десериализованный объект, то данный член будет проинициализирован ссылкой на экземпляр этого объекта.

Уведомление о завершении десериализации

Независимо от того, используете вы атрибут `SerializableAttribute` и предоставляете все детали сериализации общезыковой исполняющей среде или используете расширенный механизм сериализации, реализуя интерфейс `ISerializable`, вам может понадобиться получать уведомление о том, что форматировщик завершил десериализацию всего графа объектов. В этом случае следует реализовать интерфейс `IDeserializationCallback`, который содержит единственный член `OnDeserializationCallback`. По окончании

десериализации всего графа объектов форматировщик вызывает этот метод для всех объектов графа, реализующих данный интерфейс. Когда форматировщик вызывает *OnDeserializationCallback* некоторого объекта, данный объект может быть уверен в том, что все объекты, на которые ссылаются его члены, уже полностью инициализированы. Однако, объект не может быть уверен в том, что уже вызваны методы *OnDeserializationCallback* объектов, поддерживающих *IDeserializationCallback*, на которые ссылаются его члены.

Суррогаты сериализации и селекторы суррогатов

Иногда требуется сериализовать тип, который не поддерживает сериализацию или добавить к сериализованным данным типа дополнительную информацию. Для обеспечения дополнительной гибкости архитектуры сериализации .NET Framework использует *суррогаты* и *селекторы суррогатов*. Суррогат — это класс, который выполняет сериализацию экземпляров других типов. Селектор суррогатов по существу представляет собой набор суррогатов, который по запросу возвращает подходящий суррогат для данного типа.

Суррогаты

Суррогат реализует интерфейс *ISerializationSurrogate*. Данный интерфейс содержит два метода: *GetObjectData* и *SetObjectData*. Сигнатура метода *ISerializationSurrogate.GetObjectData* почти идентична сигнатуре метода *ISerializable.GetObjectData*. Однако метод *ISerializationSurrogate.GetObjectData* принимает еще один параметр типа *object* — сериализуемый объект. Реализация *GetObjectData*, предоставляемая суррогатом, может добавить члены к экземпляру *SerializationInfo* перед сериализацией членов объекта или полностью заместить собой функциональность сериализации для данного объекта.

Сигнатура *SetObjectData* аналогична сигнатуре *ISerializationSurrogate.GetObjectData* за исключением того, что *SetObjectData* возвращает заполненный экземпляр объекта и принимает дополнительный параметр типа *ISurrogateSelector*, который мы обсудим немного позже. Реализация *SetObjectData*, предоставляемая суррогатом, способна считать из экземпляра *SerializationInfo* члены, которые были помещены туда во время сериализации экземпляра

ра объекта реализацией *GetObjectData*, предоставляемой суррогатом.

Селекторы суррогатов

Как вы узнаете из раздела «Форматировщики сериализации», одно из свойств, поддерживаемых форматировщиком, — *SurrogateSelector*. В качестве значения этого свойства можно установить любой объект, реализующий интерфейс *ISurrogateSelector*. Форматировщики используют селектор суррогатов для того, чтобы определить, имеется ли для сериализуемого или десериализуемого типа суррогат. Если да, то форматировщик использует данный суррогат для сериализации и десериализации экземпляров данного типа. Подробности процесса сериализации и десериализации обсуждаются далее в этой главе.

.NET Framework определяет класс *System.Runtime.Serialization.SurrogateSelector*, реализующий интерфейс *ISurrogateSelector*. Данный класс также предоставляет метод *Add*, позволяющий добавить суррогат к внутреннему набору, и метод *Remove*, позволяющий удалить его оттуда.

Суррогат *TimeStamper*

Следующий пример демонстрирует использование суррогатов сериализации и селекторов суррогатов. Мы реализуем суррогат, добавляющий член-метку времени к данным *SerializationInfo* объекта, и затем сериализующий экземпляр объекта. Класс *TimeStamperSurrogate* определен так:

```
class TimeStamperSurrogate : ISerializationSurrogate
{
    public void GetObjectData ( object obj,
                               SerializationInfo info,
                               StreamingContext context )
    {
        //
        // Добавить метку времени к данным сериализации.
        info.AddValue(
            "TimeStamperSurrogate.SerializedDateTime",
            DateTime.Now );

        //
        // Сериализовать объект.
        if ( obj is ISerializable )
```

```

    {
        ((ISerializable)obj).GetObjectData(info, context);
    }
    else
    {
        MemberInfo[] mi =
            FormatterServices.GetSerializableMembers(
                obj.GetType());

        object[] od = FormatterServices.GetObjectData(
            obj,
            mi );

        for( int i = 0; i < mi.Length; ++i)
        {
            info.AddValue( mi[i].Name, od[i] );
        }
    }
}

public System.Object SetObjectData (
    object obj,
    SerializationInfo info,
    StreamingContext context,
    ISurrogateSelector selector )
{
    // Получить значения, добавленные этим суррогатом
    // в GetObjectData.
    DateTime dt = (DateTime)info.GetValue(
        "TimeStamperSurrogate.SerializedDateTime",
        typeof(DateTime) );

    if ( obj is ISerializable )
    {
        ObjectManager om = new ObjectManager(selector,
            context);

        om.RegisterObject(obj, 1, info);
        om.DoFixups();
        obj = om.GetObject(1);
    }
    else
    {
        MemberInfo[] mi =
            FormatterServices.GetSerializableMembers(
                obj.GetType());
    }
}

```

```

        object[] od = FormatterServices.GetObjectData(
                                                    obj,
                                                    mi );

        int i = 0;
        SerializationInfoEnumerator ie =
            info.GetEnumerator();
        while(ie.MoveNext())
        {
            if ( mi[i].Name == ie.Name )
            {
                od[i] = Convert.ChangeType( ie.Value,
                                              ((FieldInfo)mi[i]).FieldType);
                ++i;
            }
        }

        FormatterServices.PopulateObjectMembers( obj,
                                                    mi, od );
    }
    return obj;
}

```

Метод *TimeStamperSurrogate.GetObjectData* добавляет текущее время и дату к экземпляру *SerializationInfo* — *info*. Далее этот метод позволяет объекту, поддерживающему интерфейс *ISerializable*, добавить свои данные в экземпляр *SerializationInfo*. Если же объект не поддерживает *ISerializable*, то метод с помощью класса *FormatterServices* получает значения сериализуемых членов объекта и добавляет их к экземпляру *SerializationInfo*. Класс *FormatterServices* обсуждается в разделе «Форматировщики сериализации» этой главы.

Метод *TimeStamperSurrogate.SetObjectData* выполняет действия, обратные *GetObjectData*, считывая сначала метку времени из *SerializationInfo*. Если объект поддерживает *ISerializable*, то *SetObjectData* использует класс *ObjectManager*, о котором также речь пойдет далее в разделе «Форматировщики сериализации». На данный момент вам достаточно знать, что в результате исполнения этих операторов происходит вызовов специального конструктора объекта, поддерживающего *ISerializable*, которому передается экземпляр *SerializationInfo*, *info* и экземпляр *Streaming-*

Context — *context*. Если интерфейс *ISerializable* не реализован объектом, то *SetObjectData* использует класс *FormatterServices* для получения информации о сериализуемых членах объекта и считывает их значения из экземпляра *Serialization Info*.

RemotingSurrogateSelector

Помимо класса *SurrogateSelector* .NET Framework определяет еще один селектор суррогатов — *RemotingSurrogateSelector*, который необходим во время сериализации типов, связанных с .NET Remoting. В табл. 8-2 перечислены различные классы суррогатов, используемые инфраструктурой .NET Remoting.

Таблица 8-2. Суррогаты .NET Remoting

Класс	Описание
<i>RemotingSurrogate</i>	Выполняет сериализацию экземпляров объектов, передаваемых по ссылке
<i>ObjRefSurrogate</i>	Выполняет сериализацию экземпляров <i>ObjRef</i> путем добавления к экземпляру <i>SerializationInfo</i> значения <i>IsMarshaled</i> , указывающего, что в качестве параметра был передан сериализуемый экземпляр <i>ObjRef</i> , а не представляемый им объект, передаваемый по ссылке
<i>MessageSurrogate</i>	Выполняет сериализацию сообщений <i>MethodCall</i> , <i>Method Response</i> , <i>ConstructionCall</i> и <i>ConstructionCall</i> путем перебора содержимого набора <i>IMessage.Properties</i> и добавления каждого свойства к <i>SerializationInfo</i>
<i>SoapMessageSurrogate</i>	Реализует специальные требования сериализации сообщений SOAP

Форматировщики сериализации

Как было показано выше, форматировщики сериализуют объекты в потоки. Однако мы до сих пор ничего не говорили о том, как написать свой форматировщик, подключаемый к .NET Remoting. Написание форматировщика включает в себя выполнение следующих основных действий без строго определенного порядка:

- получение списка сериализуемых членов объектного типа;
- обход графа объектов, начиная с сериализуемого объекта;

- сериализацию полного имени типа объекта, его сборки и значений его сериализуемых членов;
- сериализацию ссылок на объекты внутри графа, чтобы граф удалось восстанавливать во время десериализации.

К счастью, .NET Framework предоставляет ряд классов, которые можно использовать при решении каждой из этих задач. Их перечень и назначение приводятся в табл. 8-3. Примеры их использования мы рассмотрим, когда будем говорить о способах решения перечисленных выше задач.

Таблица 8-3. Классы, необходимые при написании форматировщиков сериализации

Класс	Пространство имен	Описание
<i>FormatterServices</i>	<i>System.Runtime.Serialization</i>	Сериализация/десериализация
<i>ObjectIDGenerator</i>	<i>System.Runtime.Serialization</i>	Сериализация
<i>ObjectManager</i>	<i>System.Runtime.Serialization</i>	Десериализация
<i>Formatter</i>	<i>System.Runtime.Serialization</i>	Можно использовать в качестве базового класса для форматировщика; предоставляет члены для выполнения обхода графа объектов и каталогизации (scheduling) объектов

Давайте сначала рассмотрим эти классы по отдельности и изучим их возможности. Мы соберем их вместе, когда займемся написанием форматировщика далее в этом разделе.

Получение сериализуемых членов типа

В табл. 8-4 перечислены некоторые из методов, предоставляемых классом *FormatterServices* для поддержки написания форматировщиков.

Таблица 8-4. Основные открытые методы
System.Runtime.Serialization.FormatterServices

Метод	Описание
<i>GetSerializableMembers</i>	Возвращает сериализуемые члены заданного типа
<i>GetObjectData</i>	Возвращает значения одного или нескольких сериализуемых членов экземпляра объекта
<i>GetUninitializedObject</i>	Возвращает неинициализированный экземпляр объекта во время десериализации
<i>PopulateObjectMembers</i>	Инициализирует члены неинициализированного экземпляра объекта заданными значениями

В примере из предыдущего раздела был определен тип *SomeClass2*. Следующий фрагмент кода демонстрирует использование каждого из перечисленных в предыдущей таблице членов *FormatterServices* для получения сериализуемых членов и их значений для экземпляра типа *SomeClass2*:

```
SomeClass2 sc = new SomeClass2();

// Получить сериализуемые члены и их значения.
MemberInfo[] mi =
    FormatterServices.GetSerializableMembers(sc.GetType());

object[] vals = FormatterServices.GetObjectData(sc, mi);

// Получить неинициализированный экземпляр
// и заполнить его члены.
SomeClass2 sc2 =
    (SomeClass2)FormatterServices.GetUninitializedObject(typeof(
        SomeClass2));

FormatterServices.PopulateObjectMembers(sc2, mi, vals);
```

Метод *GetSerializableMembers* возвращает для заданного типа массив экземпляров класса *System.Reflection.MemberInfo*. Если передать методу *GetSerializableMembers* несериализуемый тип, то общезыковая исполняющая среда сгенерирует исключение *System.Runtime.Serialization.SerializationException*. Обратите внимание, что передача методу *GetSerializableMembers* типа, поддерживающего интерфейс *ISerializable*, не приводит к вызову *ISeriali-*

zable.GetObjectData из *GetSerializableMembers*. Это означает, что вам не удастся получить все сериализуемые данные, определенные создателем типа.

Чтобы получить значение каждого сериализуемого члена, массив *MemberInfo* передается методу *GetObjectData*, возвращающему массив объектов, элементы которого соответствуют значениям сериализуемых членов. Эти два массива заполняются таким образом, что в *n*-ом элементе массива объектов хранится значение члена, заданного *n*-ым элементом массива *MemberInfo*.

Для выполнения процесса, обратного описанному выше, с помощью *FormatterServices.GetUninitializedObject* создается неинициализированный экземпляр типа *SomeClass2*. Главное слово здесь — *неинициализированный*: конструктор не вызывается, а члены, которые ссылаются на другие объекты, устанавливаются в *null* или 0. Для инициализации такого экземпляра объекта используется метод *PopulateObjectMembers*, которому передается неинициализированный экземпляр и соответствующий массив объектов, содержащий значения членов. Значение, возвращаемое *PopulateObjectMembers*, — это заполненный объект.

Обход графа объектов

Сериализуемый объект соответствует корневому узлу в графе объектов. Все остальные узлы графа соответствуют объектам, доступным от сериализуемого объекта либо непосредственно через его член, либо косвенно через член объекта, на который ссылается исходный объект. Базовая процедура обхода графа объектов для сериализации начинается с корневого объекта и получает его сериализуемые члены. Затем выполняется обход графа объектов для каждого сериализуемого члена и так далее, до тех пор пока не будут обойдены все узлы графа. Для ациклического графа объектов, такого, как показанный на рис. 8-2, это очень просто.

Однако в графах объектов могут быть и циклы. Они возникают тогда, когда объект ссылается на другой объект, который ссылается на первый объект непосредственно или косвенно. Проблема, вызываемая циклом, состоит в том, что если вы не обнаружите его, то будете ходить по нему вечно. На рис. 8-3 показано абстрактное представление графа объектов, содержащего цикл.

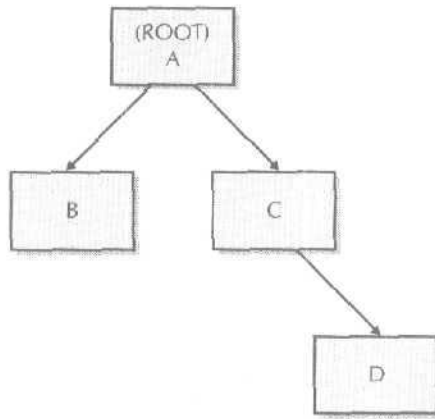


Рис. 8-2. Ациклический граф объектов

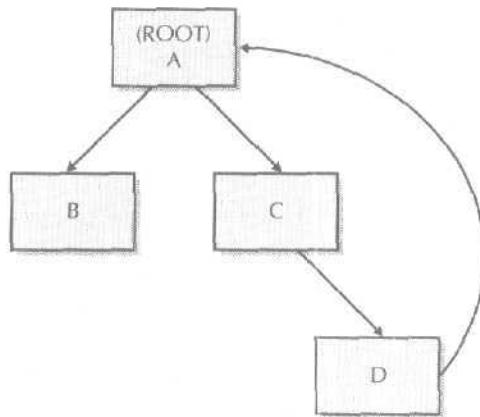


Рис. 8-3. Граф объектов, содержащий цикл

Идентификация объектов с использованием класса *ObjectIDGenerator*

В процессе обхода графа объектов форматировщик назначает каждому из них идентификатор, который используется при сериализации объектов, ссылающихся на этот объект. При этом форматировщик использует класс *ObjectIDGenerator*, который отслеживает все объекты, обнаруженные в процессе обхода. Метод *ObjectIDGenerator.GetID* требуется для получения значения типа

long, которое идентифицирует экземпляр объекта, переданный как параметр метода. Метод *GetID* принимает два параметра. Первый — это экземпляр объекта, для которого необходимо получить идентификатор. Второй — это параметр булевского типа, указывающий, является ли этот вызов *GetID* для данного объекта первым. Использование класса *ObjectIDGenerator* демонстрирует следующий фрагмент кода:

```
ObjectIDGenerator idgen = new ObjectIDGenerator();
SomeClass sc = new SomeClass();
bool firstTime = false;

// id_sc == 0, firstTime == true после возврата
long id_sc = idgen.HasId(sc, out firstTime);

// id_sc == 1, firstTime == true после возврата
id_sc = idgen.GetId(sc, out firstTime);

// id_sc == 1, firstTime == false после возврата
id_sc = idgen.HasId(sc, out firstTime);

// id_sc == 1, firstTime == false после возврата
id_sc = idgen.GetId(sc, out firstTime);
```

Каталогизация объектов для сериализации

Для поддержки сериализации графа объектов .NET Framework использует прием, называемый *каталогизацией* (*scheduling*). Во время обхода графа объектов для каждого обнаруженного объекта форматировщик выполняет следующие действия:

1. с помощью класса *ObjectIDGenerator* получает идентификатор этого объекта;
2. сериализует ссылку на экземпляр объекта, используя идентификатор объекта, вместо того чтобы сериализовать сам экземпляр объекта;
3. каталогизирует экземпляр объекта для дальнейшей сериализации, помещая его в очередь объектов, ожидающих сериализации.

Итак, мы рассмотрели использование класса *FormatterServices* для получения сериализуемых членов экземпляра объекта и обход графа объектов, а также класса *ObjectIDGenerator*. Чуть позже

мы применим эти приемы при реализации метода *IFormatter.Serialize* нашего собственного форматировщика. Но сначала рассмотрим класс *ObjectManager*, который необходим при десериализации.

Использование класса *ObjectManager*

Класс *ObjectManager* позволяет конструировать новые графы объектов. Если у вас есть экземпляры всех объектов графа и вы знаете, как они связаны друг с другом, то с помощью класса *ObjectManager* сможете построить этот граф. Основные члены класса *ObjectManager* перечислены в табл. 8-5.

Таблица 8-5. Основные открытые методы класса *System.Runtime.Serialization.ObjectManager*

Метод	Описание
<i>DoFixups</i>	Вызывается после того, как все объекты графа зарегистрированы. После возврата из метода все неразрешенные ссылки на объекты восстанавливаются
<i>GetObject</i>	Возвращает зарегистрированный экземпляр объекта по его идентификатору
<i>RaiseDeserializationEvent</i>	Генерирует событие десериализации
<i>RecordArrayElementFixup</i>	Записывает поправку (fixup) для элемента массива, который ссылается на экземпляр другого объекта в графе
<i>RecordDelayedFixup</i>	Записывает поправку для членов экземпляра <i>SerializationInfo</i> , связанного с зарегистрированным объектом
<i>RecordFixup</i>	Записывает поправку для членов экземпляра объекта
<i>RegisterObject</i>	Регистрирует экземпляр объекта по его идентификатором

В общем случае, чтобы воссоздать граф объектов при помощи *ObjectManager*, нужно выполнить следующие действия:

1. зарегистрировать экземпляры объектов и их идентификаторы с помощью метода *RegisterObject* класса *ObjectManager*;
2. воспользовавшись методами *RecordFixup*, *RecordArrayElementFixup* и *RecordDelayedFixup*, записать поправки, отображающие члены экземпляров объектов на другие экземпляры объектов графа при помощи идентификаторов объектов;

3. выдать *ObjectManager* команду на выполнение записанных поправок, вызвав метод *DoFixups*;
4. запросить у *ObjectManager* корневой объект графа с помощью метода *GetObject*.

Следующий фрагмент кода использует класс *ObjectManager* для воссоздания графа объектов, изображенного на рис. 8-1:

```
// Создать экземпляр ObjectManager.
ObjectManager om =
    new ObjectManager(new SurrogateSelector(),
        new StreamingContext(StreamingContextStates.All));

// Подготовить объект 1 - корневой объект.
object sc3 =
    FormatterServices.GetUninitializedObject(
        typeof(SomeClass3));

// Зарегистрировать 1,
om.RegisterObject(sc3, 1);

// Подготовиться к инициализации членов объекта 1.
System.Reflection.MemberInfo[] mi =
    FormatterServices.GetSerializableMembers(
        typeof(SomeClass3));

// Записать поправку для первого члена объекта 1,
// ссылающегося на объекта 2,
om.RecordFixup(1, mi[0], 2);

// Инициализировать второй член объекта 1
// с помощью FormatterServices.
FormatterServices.PopulateObjectMembers(
    sc3,
    new MemberInfo[]{ mi[1] },
    new Object[]{ 2112 });

// Подготовить объект 2 - экземпляр SomeClass2.
SerializationInfo info =
    new SerializationInfo( typeof(SomeClass2),
        new FormatterConverter());
info.AddValue("m1", 4000);
info.AddValue("m2", 4001);

// Записать отложенную поправку для члена TimeStamp
```

```
// объекта 2, ссылающегося на объект 3.
om.RecordDelayedFixup(2, "TimeStamp", 3);

// Зарегистрировать объект 2 и связанный с ним экземпляр.
object sc2 =
    FormatterServices.GetUninitializedObject(
        typeof(SomeClass2));
om.RegisterObject(sc2, 2, info);

// Зарегистрировать объект 3 - значение TimeStamp,
// формально не входящее в граф объектов, но необходимое,
// так как SomeClass2 ожидает присутствия
// значения TimeStamp в SerializationInfo.
om.RegisterObject(DateTime.Now, 3);

// У ObjectManager теперь достаточно информации для
// создания графа объектов - выполнения поправок.
om.DoFixups();

// Получить корневой объект.
SomeClass3 _sc3 = (SomeClass3)om.GetObject(1);
```

Сначала создается экземпляр класса *ObjectManager*. Корневым объектом в графе является экземпляр класса *SomeClass3*. С помощью метода *FormatterServices.GetUninitializedObject* создается неинициализированный экземпляр класса *SomeClass3*, который затем передается *ObjectManager* с идентификатором 1. Следующий шаг — инициализация членов объекта 1. Член *m_sc2* класса *SomeClass3* ссылается на экземпляр класса *SomeClass2*. Так как у нас еще нет экземпляра этого типа, мы не можем инициализировать член *m_sc2* немедленно. Следовательно, нам необходимо записать поправку для члена *m_sc2* объекта 1, который ссылается на объект с идентификатором 2. Эта информация сообщает *ObjectManager* о том, что на этапе выполнения поправок, когда мы вызовем метод *DoFixups*, в член *m_sc2* должна быть помещена ссылка на экземпляр объекта, идентификатор которого равен 2. Вторым членом класса *SomeClass3* является целое значение, которое можно инициализировать немедленно средствами метода *FormatterServices.PopulateObjectMembers*.

Далее мы подготавливаем второй объект графа — экземпляр класса *SomeClass2*. Так как *SomeClass2* поддерживает *ISerializable*, мы создаем новый экземпляр класса *SerializationInfo*, который

немедленно заполняем значениями двух членов целого типа: *m1* и *m2*. Реализация *ISerializable.GetObjectData* в классе *SomeClass2* ожидает наличия в составе *SerializationInfo* третьего члена — *TimeStamp*. Для демонстрации использования отложенных поправок мы вызываем метод *RecordDelayedFixup* для записи отложенной поправки, связывающей член *TimeStamp* объекта 1 с объектом 3, который еще не зарегистрирован. Метод *RecordDelayedFixup* откладывает инициализацию члена *SerializationInfo*, ссылающегося на еще незарегистрированный объект, до тех пор пока этот объект не будет передан *ObjectManager*. После записи отложенных поправок мы регистрируем неинициализированный экземпляр *SomeClass2* и связанный с ним *SerializationInfo*, назначая этому экземпляру идентификатор 2. В этот момент *SerializationInfo* содержит два члена — *m1* и *m2* — и их значения. Из-за наличия отложенной поправки *ObjectManager* добавит к *SerializationInfo* объекта 2 член с именем *TimeStamp*, когда мы зарегистрируем экземпляр *DateTime* как объект 3. Этот член *TimeStamp* ссылается на экземпляр объекта с идентификатором 3.

Затем вызывается метод *DoFixups* экземпляра *ObjectManager*. В результате *ObjectManager* просматривает свои внутренние структуры, выполняя все записанные поправки. Для поправок членов *ObjectManager* инициализирует член ссылкой на фактический экземпляр объекта. Для отложенных поправок объектов, у которых нет суррогатов сериализации, *ObjectManager* вызывает метод *ISerializable.GetObjectData* объекта, передавая ему экземпляр *SerializationInfo*, указанный при регистрации данного объекта. Если у объекта есть суррогат сериализации, то *ObjectManager* вызывает метод суррогата *ISerializationSurrogate.SetObjectData*, передавая ему экземпляр *SerializationInfo*, указанный при регистрации данного объекта. Для поправок массива *ObjectManager* инициализирует элемент массива ссылкой на фактический экземпляр объекта. После выполнения поправок мы запрашиваем экземпляры объектов по номерам их идентификаторов.

Использование класса *Formatter*

.NET Framework содержит класс *System.Runtime.Serialization.Formatter*, который годится в качестве базового при написании собственного форматировщика. В табл. 8-6 перечислены основные открытые члены класса *Formatter*. Методы *Schedule* и *GetNext*

реализуют обсуждавшуюся ранее технику каталогизации при сериализации объектов. Чтобы каталогизировать объект для дальнейшей сериализации, следует вызывать метод *Schedule*, передав экземпляр объекта в качестве параметра. Метод *Schedule* получает и возвращает идентификатор объекта, назначенный с помощью *ObjectIDGenerator*, на который ссылается член *m_idGenerator* данного экземпляра *Formatter*. Перед возвратом управления метод *Schedule* помещает экземпляр объекта в очередь, на которую ссылается член *m_objectQueue*. Метод *GetNext* извлекает и возвращает следующий объект из этой очереди.

Таблица 8-6. Основные члены *System.Runtime.Serialization.Formatter*

Член	Тип члена	Описание
<i>m_idGenerator</i>	Защищенное поле	Ссылка на экземпляр <i>ObjectIDGenerator</i> , используемый для идентификации объектов во время обхода графа объектов
<i>m_objectQueue</i>	Защищенное поле	Очередь объектов, ожидающих сериализации
<i>Schedule</i>	Метод	Каталогизирует экземпляр объекта для последующей сериализации. Возвращаемое значение соответствует идентификатору, присвоенному этому экземпляру объекта
<i>GetNext</i>	Метод	Возвращает следующий экземпляр объекта, подлежащий сериализации
<i>WriteMember</i>	Метод	Сериализует член экземпляра объекта. Данный метод вызывает один из членов <i>WriteXXX</i> в зависимости от типа объекта
<i>WriteObjectRef</i>	Метод	Переопределите этот виртуальный метод для записи ссылки на объект вместо собственно экземпляра объекта. Большинство форматировщиков просто выводят в поток идентификатор объекта
<i>WriteValueType</i>	Метод	Переопределите этот виртуальный метод для вывода в поток значения размерного типа. Для элементарных типов переопределите методы

см. след. стр.

Таблица 8-6. (окончание)

Член	Тип члена	Описание
<i>WriteXXXX</i>	Метод	<i>WriteXXXX</i> . Данный метод вызывает- ся <i>WriteMember</i> , если размерный тип не относится к элементарным типам Группа методов, названных по типу, значение которого они выводят— <i>WriteByte</i> , <i>WriteInt32</i> , <i>WriteDateTime</i> и т. д.

Реализация собственного форматировщика

Теперь, когда вы познакомились с рядом классов, предоставляемых .NET Framework для поддержки разработки пользовательских форматировщиков сериализации, пора применить полученные знания на практике. Вот чем мы займемся далее:

1. определим формат сериализации;
2. реализуем *IFormatter.Serialize*;
3. реализуем *IFormatter.Deserialize*.

Определение формата сериализации

Как неоднократно отмечалось в этой книге, *SoapFormatter* сериализует графы объектов с использованием формата SOAP. Аналогично *BinaryFormatter* сериализует графы объектов с использованием эффективного двоичного формата. Прежде чем разрабатывать форматировщик, необходимо выбрать формат, который будет реализован. Для целей обучения и демонстрации принципов реализации пользовательского форматировщика мы возьмем читабельный формат, состоящий из тегов полей, за которыми следуют строковые представления значений. Каждый тег поля определяет некоторый тип информации, необходимой для воссоздания графа объектов. Теги применяемых нами в нашем форматировщике полей перечислены в табл. 8-7,

Таблица 8-7. Пользовательский формат сериализации

Тег поля	Смысл значения
<i>o_id:</i>	Идентификатор объекта, назначенный с помощью <i>ObjectIDGenerator</i>

см. след. стр.

Таблица 8-7. (окончание)

Тег поля	Смысл значения
<i>o_assembly:</i>	Полное имя сборки объекта, включая версию, региональные стандарты и маркер открытого ключа
<i>o_type:</i>	Полностью квалифицированное имя типа объекта
<i>t_count:</i>	Используется, когда сериализуемые члены являются членами <i>SerializationInfo</i> . Число членов, сериализованных для текущего объекта
<i>m_name:</i>	Имя члена
<i>m_type:</i>	Полностью квалифицированное имя типа члена
<i>m_value:</i>	Значение члена в строковой форме
<i>m_value_type:</i>	Указывает, что значением данного члена фактически является тип. Мы сериализуем имя типа, а не экземпляр <i>Type</i>
<i>m_value_refid:</i>	Указывает, что значение ссылается на другой объект графа по его идентификатору
<i>array_rank:</i>	Число измерений массива
<i>array_length:</i>	Длина измерения массива
<i>arrayLowerbound:</i>	Нижняя граница измерения массива

Ниже показан пример сериализованного графа объектов, полученного с помощью пользовательского форматировщика, который мы создадим в этом разделе:

```

o_id:1
o_assembly:Serialization, Version=1.0.912.37506,
Culture=neutral, PublicKeyToken=null
o_type:Serialization.TestClassA
m_count:2
m_name: __v1
m_type:System.DateTime
m_value: 7/1/2002 9:50:24 PM
m_name: __v2
m_type:Serialization.TestClassB
m_value_refid:2
o_id:2
o_assembly:Serialization, Version=1.0.912.37506,
Culture=neutral, PublicKeyToken=null
o_type:Serialization.TestClassB
m_name:m_a
m_type:Serialization.TestClassA
m_value_refid:1

```

Как видите, каждый тег поля и связанное с ним значение размещаются на одной строке. Это имеет одно нежелательное следствие: значения не могут содержать символы возврата каретки и перевода строки. Если бы мы собирались использовать этот форматировщик в реальных программах, то данную проблему, конечно, следовало бы решить, но в учебном примере мы ее игнорируем.

Первые три строки сериализованной информации объекта содержат его идентификатор, полное имя сборки и название типа. Следующая строка начинается с *m_count* и говорит о том, что для этого объекта сериализовано два члена. Наличие тега поля *m_count* также сообщает, что следующие за ним члены должны быть во время десериализации помещены в *SerializationInfo*. Далее следует название, тип и значение каждого члена. Обратите внимание, что второй член с именем *v2* — это ссылка на объект с идентификатором равным 2. Следующие три строки начинают новый объект с идентификатором 2. У этого объекта имеется только один член, для которого далее следуют имя, тип и значение указывающее, что он ссылается на объект с идентификатором 1.

Следующий код определяет класс *FieldNames*, который будет использоваться при реализации нашего форматировщика:

```
class FieldNames
{
    // Идентификатор объекта (для ObjectManager).
    public static string OBJECT_ID           = "o_id: ";

    // Имя сборки.
    public static string OBJECT_ASSEMBLY     = "o_assembly: ";

    // Тип объекта.
    public static string OBJECT_TYPE         = "o_type: ";

    // Число членов данного объекта.
    public static string MEMBER_COUNT        = "m_count: ";

    // Имя члена.
    public static string MEMBER_NAME         = "m_name: ";

    // Тип члена.
}
```

```

public static string MEMBER_TYPE           = "m_type:";

// Значение члена.
public static string MEMBER_VALUE         = "m_value:";

// Значением члена является тип.
public static string MEMBER_VALUE_TYPE    = "m_value_type:";

// Идентификатор объекта (для ObjectManager).
public static string OBJECT_REFID         = "m_value_refid:";

// Число измерений.
public static string ARRAY_RANK           = "array_rank:";

// Длина измерения.
public static string ARRAY_LENGTH         = "array_length:";

// Нижняя граница измерения.
public static String ARRAY_LOWERBOUND     =
    "array_lowerbound:";

// Специальный индикатор null-значения.
public static string MEMBER_VALUE_NULL    = "m_value_null";

public static long ParseObjectRefID(string s)
{ return Convert.ToInt64(s.Substring(
    OBJECT_REFID.Length)); }

public static long ParseObjectID(string s)
{ return Convert.ToInt64(s.Substring(
    OBJECT_ID.Length)); }

public static string ParseObjectAssembly(string s)
{ return s.Substring(OBJECT_ASSEMBLY.Length); }

public static string ParseObjectType(string s)
{ return s.Substring(OBJECT_TYPE.Length); }

public static long ParseMemberCount(string s)
{ return Convert.ToInt64(s.Substring(
    MEMBER_COUNT.Length)); }

public static string ParseMemberName(string s)
{ return s.Substring(MEMBER_NAME.Length); }

public static string ParseMemberType(string s)

```

```

        { return s.Substring(MEMBER_TYPE.Length); }

    public static string ParseMemberValue(string s)
    { return s.Substring(MEMBER_VALUE.Length); }

    public static string ParseMemberValueType(string s)
    { return s.Substring(MEMBER_VALUE_TYPE.Length); }

    public static long ParseArrayRank(string s)
    { return Convert.ToInt64(s.Substring(
        ARRAY_RANK.Length)); }

    public static long ParseArrayLength(string s)
    { return Convert.ToInt64(s.Substring(
        ARRAY_LENGTH.Length)); }

    public static long ParseArrayLowerBound(string s)
    { return Convert.ToInt64(s.Substring(
        ARRAY_LOWERBOUND.Length)); }
}

```

Класс *FieldNames* просто определяет статический член для каждого из тегов полей, перечисленных в табл. 8-7. Кроме того, в нем определены методы *ParseXXXX*, выполняющие разбор каждого из тегов полей. Мы применим методы *ParseXXXX* при реализации метода *IFormatter.Deserialize*.

Реализация интерфейса *IFormatter*

Форматировщики сериализации — это классы, поддерживающие интерфейс *IFormatter*. В табл. 8-8 перечислены члены этого интерфейса:

Таблица 8-8. Члены интерфейса
System.Runtime.Serialization.IFormatter

Член	Тип члена	Описание
<i>Binder</i>	Свойство для чтения и записи	Позволяет снабдить экземпляр форматировщика объектом <i>SerializationBinder</i> для десериализации сериализованного типа в другой тип
<i>Context</i>	Свойство для чтения и записи	Может ссылаться на экземпляр <i>StreamingContext</i> , содержащий информацию о текущей операции сериализации или десериализации

см. след. стр.

Таблица 8-8. (окончание)

Член	Тип члена	Описание
<i>SurrogateSelector</i>	Свойство для чтения и записи	Может ссылаться на экземпляр класса селектора суррогатов
<i>Deserialize</i>	Метод	Десериализует граф объектов из потока
<i>Serialize</i>	Метод	Сериализует граф объектов в поток

Следующий листинг начинает реализацию класса *MyFormatter*, о полной реализации которого рассказывается в нескольких следующих разделах:

```
public class MyFormatter : Formatter
{
    SerializationBinder _binder;
    StreamingContext _streamingcontext;
    ISurrogateSelector _surrogateselector;

    StreamWriter _writer;
    StreamReader _reader;
    ObjectManager _om;

    public MyFormatter()
    {
    }

    public override SerializationBinder Binder
    {
        get
        { return _binder; }
        set
        { _binder = value; }
    }

    public override StreamingContext Context
    {
        get
        { return _streamingcontext; }
        set
        { streamingcontext = value; }
    }

    public override ISurrogateSelector SurrogateSelector
    {
```

```

    get
    { return _surrogateselector; }
    set
    { _surrogateselector = value; }
}

```

Класс *MyFormatter* — производный от *System.Runtime.Serialization.Formatter*, это позволяет задействовать функциональность обхода графа объектов, о которой мы говорили ранее. Вместе с членами, реализующими свойства *IFormatter*, мы определили член *_writer* типа *StreamWriter*, который будет использоваться для сериализации. Для десериализации определен член типа *StreamReader* с именем *_reader* и член типа *ObjectManager* с именем *_om*.

Реализация метода *IFormatter.Serialize*

Задача метода *IFormatter.Serialize* — сериализовать граф объектов в поток, используя некоторый формат представления сериализованной информации. Наша реализация *IFormatter.Serialize* такова:

```

public override void Serialize(Stream serializationStream,
                               object graph)
{
    _writer = new StreamWriter(serializationStream);

    // Каталогизировать корневой объект.
    Schedule(graph);

    // Получить следующий объект и сериализовать его.
    object oTop;
    long topId;
    while( (oTop = GetNext(out topId) != null )
    {
        // Исполнение метода WriteObject, вероятно, вызовет
        // каталогизацию дополнительных объектов для сериализации.
        WriteObject(oTop, topId);
    }

    _writer.Flush();
}

```

Для запуска процесса сериализации метод *IFormatter.Serialize* передает корневой объект графа объектов в вызове *Formatter.Schedule*. *Schedule* создаст идентификатор корневого объек-

та, который должен быть равен 1, и поместит объект в очередь для дальнейшей сериализации. Для продолжения сериализации мы вызываем метод *Formatter.GetNext*, возвращающий следующий объект из очереди сериализации. Если *GetNext* возвращает экземпляр объекта, а не *null*, то мы передаем этот экземпляр вместе с его идентификатором методу *MyFormatter.WriteObject*. Описанный процесс продолжается до тех пор, пока *GetNext* не возвратит *null*, то есть пока в очереди не останется объектов. Реализация метода *WriteObject* показана ниже:

```
private void WriteObject(object obj, long objId)
{
    // Вывод преамбулы объекта.
    WriteField(FieldNames.OBJECT_ID, objId);
    WriteField(FieldNames.OBJECT_ASSEMBLY, obj.GetType().Assembly);
    WriteField(FieldNames.OBJECT_TYPE, obj.GetType().FullName);

    // Не выводит члены-массивы и строки; они будут
    // обрабатываться особым способом.
    if ( obj.GetType().IsArray )
    {
        WriteArray(obj, "", obj.GetType());
    }
    else if ( obj.GetType() == typeof(string) )
    {
        WriteField(FieldNames.MEMBER_VALUE, obj);
    }
    else
    {
        // Вывод членов объекта.
        WriteObjectMembers(obj, objId);
    }
}
```

Метод *WriteObject* обрабатывает массив и строки особым образом. Для массивов мы не хотим выводить в поток все члены класса *Array*. Нам нужно вывести только размерность массива, нижние границы и длины, а также все элементы массива, что и делает метод *WriteArray* (о нем речь пойдет далее). Для строк мы выводим в поток только значение строки с помощью метода *WriteField*. Для всех остальных типов *WriteObject* сериализует экземпляр объекта в поток, выводя в поток идентификатор объекта, информацию о сборке и полное имя типа с помощью метода

WriteField. Затем метод *WriteObject* выводит сериализуемые члены экземпляра объекта с помощью метода *WriteObjectMembers*. Реализация метода *WriteField* просто выводит в одну строку имя поля и строковое представление его значения:

```
//
// Вывод поля в поток.
void WriteField(string field_name, object oValue)
{
    _writer.WriteLine("{0}{1}", field_name, oValue);
}
```

Метод *WriteObjectMembers* немного сложнее:

```
private void WriteObjectMembers(object obj, long objId)
{
    // Проверить необходимость использования
    // для объекта суррогата.
    ISerializationSurrogate surrogate = null;
    if ( _surrogateselector != null )
    {
        // Зарегистрирован ли в селекторе суррогатов
        // суррогат для данного типа?
        ISurrogateSelector selector;
        surrogate = _surrogateselector.GetSurrogate(
                                                    obj.GetType(),
                                                    _streamingcontext,
                                                    out selector );
    }

    if ( surrogate != null )
    {
        // Да, суррогат зарегистрирован;
        // вызвать его метод GetObjectData.
        SerializationInfo info =
            new SerializationInfo( obj.GetType(),
                                  new FormatterConverter());

        surrogate.GetObjectData(obj, info,
                                this._streamingcontext);

        // Вывести в поток члены информации сериализации.
        WriteSerializationInfo(info);
    }
    else if ( IsMarkedSerializable(obj) &&
              ( obj is ISerializable ) )
```

```

// Тип объекта поддерживает ISerializable.

// Позволить объекту сериализовать себя
// посредством его метода GetObjectData.
SerializationInfo info =
    new SerializationInfo( obj.GetType(),
                          new FormatterConverter());

((ISerializable)obj).GetObjectData(
    info,
    this._streamingcontext);

// Вывести в поток члены SerializationInfo.
WriteSerializationInfo(info);
!
else if ( IsMarkedSerializable(obj) )
{
    // Тип объекта не поддерживает
    // ISerializable и для него нет суррогата.
    WriteSerializableMembers(obj, objId);
}
else
{
    // Тип не может быть сериализован;
    // сгенерировать исключение.
    throw new SerializationException();
}
}

```

Метод *WriteObjectMembers* использует ряд обсуждавшихся ранее приемов сериализации объектов и следует алгоритму, которому должны следовать все форматировщики при сериализации экземпляров объекта. Сначала метод определяет доступность селектора суррогатов. Если он имеется, метод запрашивает у селектора суррогатов суррогат для текущего объекта. При наличии суррогата метод *WriteObjectMembers* использует его для сериализации членов объекта в новый экземпляр *SerializationInfo*, который затем выводится в поток путем вызова метода *WriteSerializationInfo*. Если у форматировщика нет селектора суррогатов или если суррогат для данного типа отсутствует, то определяется, имеется ли у типа объекта атрибут *SerializableAttribute* и поддерживает ли этот тип интерфейс *ISerializable*. Если тип объекта соответствует этим критериям, то мы даем объекту возможность сериализовать себя

з новый экземпляр *SerializationInfo*, который затем выводится в поток путем вызова метода *WriteSerializationInfo*. Если у типа объекта есть атрибут *SerializableAttribute*, но интерфейс *ISerializable* не поддерживается, то члены объекта сериализуются вручную с помощью вызова метода *WriteSerializableMembers*. Если не выполнено ни одно из этих условий, то объект не может быть сериализован, и мы генерируем исключение *SerializationException*. Метод *IsMarkedSerializable* проверяет атрибуты типа объекта по маске *TypeAttributes.Serializable*:

```
// Есть ли у типа атрибут [Serializable]?
private bool IsMarkedSerializable(object o)
{
    Type t = o.GetType();

    TypeAttributes taSerializableMask =
        (t.Attributes & TypeAttributes.Serializable);

    return ( taSerializableMask ==
        TypeAttributes.Serializable );
}
```

Для обеспечения десериализации членов *SerializationInfo* метод *WriteSerializationInfo* сначала выводит в поток число членов с помощью метода *WriteField*. Затем с помощью метода *Formatter.WriteMember* в поток выводится каждый член экземпляра *SerializationInfo*, как показано ниже:

```
// Вывести все члены SerializationInfo.
private void WriteSerializationInfo(SerializationInfo info)
{
    // Вывести в поток число членов.
    WriteField(FieldNames.MEMBER_COUNT, info.MemberCount);

    // Вывести в поток все члены SerializationInfo.
    SerializationInfoEnumerator sie = info.GetEnumerator();
    while(sie.MoveNext())
    {
        WriteMember(sie.Name, sie.Value);
    }
}
```

Для получения сериализуемых членов и их значений метод *WriteSerializableMembers* использует класс *FormatterServices*, как мы уже делали ранее в этой главе. Получив массив *MemberInfo*

и массив объектов, содержащий значения членов, метод выводит каждый член в поток путем вызова *WriteMember*:

```
private void WriteSerializableMembers(object obj, long objId)
{
    System.Reflection.MemberInfo[] mi =
        FormatterServices.GetSerializableMembers(obj.GetType());

    if ( mi.Length > 0 )
    {
        object[] od = FormatterServices.GetObjectData(obj, mi);
        for( int i = 0; i < mi.Length; ++i )
        {
            WriteMember(mi[i].Name, od[i]);
        }
    }
}
```

Formatter.WriteMember— это защищенный виртуальный метод. Как показано в табл. 8-6, метод *WriteMember* определяет тип объекта, переданный как параметр *data*, и в зависимости от типа вызывает один из методов *WriteXXXX*, которые необходимо переопределить в производных классах. Метод *Formatter.WriteMember* не делает исключения для типа *string*. Объекты данного типа передаются методу *WriteObjectRef*. Чтобы упростить реализацию метода *WriteObjectRef* (обсуждается далее), мы решили переопределить *WriteMember* и обрабатывать строки как отдельный случай. Экземпляры типа *Type* также обрабатываются особо (поясняется далее). Для всех остальных типов и случая, когда объект равен *null*, мы вызываем реализацию *WriteMember* из базового класса:

```
protected override void WriteMember(string name, object data)
{
    if ( data == null )
    {
        base.WriteMember(name, data);
    }
    else if ( data.GetType() == typeof(string) )
    {
        WriteString(data.ToString(), name);
    }
    else if ( data.GetType().IsSubclassOf(typeof(System.Type)) )
    {
        WriteType(data, name);
    }
}
```

```

    }
    else
    {
        base.WriteMember(name, data);
    }
}

```

Метод *WriteString* выводит в поток экземпляр класса *string*. В общем случае возможны два варианта сериализации экземпляров данного, да и любого типа. Первый — сериализация подстановкой вместе с остальными членами объекта, второй — сериализация ссылки на экземпляр и отсрочка сериализации самого экземпляра. Мы решили сериализовать экземпляры *string* подстановкой. Метод *MyFormatter.WriteString* реализован так:

```

private void WriteString(string val, string name)
{
    // Строковые типы:
    // Так как класс Formatter игнорирует строковый тип и мы
    // переопределили метод WriteMember для обработки строк
    // особым образом, вызов WriteMember для строкового типа
    // в конечном счете обрабатывается здесь.
    // Возможно два варианта обработки:
    // (1) записать непосредственно саму строку или
    // (2) записать ссылку на объект и каталогизировать
    // объект-строку для дальнейшей сериализации.
    //

    // В данном примере мы просто выводим строку,
    if ( name != "" )
    {
        WriteField(FieldNames.MEMBER_NAME, name);
    }

    // Вывести тип члена.
    WriteField( FieldNames.MEMBER_TYPE,
                typeof(string).AssemblyQualifiedName);

    // Вывести значение члена.
    WriteField(FieldNames.MEMBER_VALUE, val);
}

```

Метод *MyFormatter.WriteMember* также особым образом обрабатывает экземпляры класса *Type*. Реализация метода *WriteType* показана далее:

```

private void WriteType(object data, string name)
{
    // Имя члена.
    if ( name != "" )
    {
        WriteField(FieldNames.MEMBER_NAME, name);
    }

    Type t = (System.Type)data;
    if ( t.FullName != "System.RuntimeType" )
    {
        // Вместо сериализации самого типа мы сериализуем лишь
        // его полное имя как строку, помечая ее, чтобы
        // при десериализации она интерпретировалась как тип,
        // а не как строка.
        data = t.AssemblyQualifiedName;
    }
    else
    {
        throw new SerializationException("Unexpected type");
    }

    // Тип члена.
    WriteField(FieldNames.MEMBER_TYPE,
               typeof(string).AssemblyQualifiedName);

    // Во время десериализации значение должно
    // интерпретироваться как тип.
    WriteField(FieldNames.MEMBER_VALUE_TYPE, data);
}

```

Общезыко́вая исполня́ющая среда рассматривает экземпляры *Type* как экземпляры *System.RuntimeType*. Тип *RuntimeType* поддерживает интерфейс *ISerializable*, но не поддерживает специальный конструктор, необходимый для десериализации. Чтобы обойти эту проблему, мы выводим в поток полное квалифицированное имя сборки для типа, представленного данным экземпляром *Type*, и помечаем значение с помощью флага *FieldNames.MEMBER_VALUE_TYPE*. Например, если экземпляр *Type* — это *typeof(SomeClass)*, то вместо сериализации экземпляра *RuntimeType* мы выводим квалифицированное имя сборки. Во время десериализации по имени сборки мы создадим экземпляр *Type* с помощью метода *Type.GetType*.

Оставшиеся методы, необходимые для завершения поддержки сериализации, — это виртуальные члены класса *Formatter*, вызываемые из метода *WriteMember*. Это методы *WriteArray*, *WriteObjectRef*, *WriteValueType* и типизированные методы *WriteXXXX* для элементарных типов. Ниже показана реализация *WriteArray*:

```
protected override void WriteArray( object obj,
                                     string name,
                                     Type  memberType)
{
    if ( name != "" )
    {
        //
        // Если имя члена не "", то данный объект является
        // членом другого объекта. Вместо сериализации такого
        // массива подстановкой в родительский объект,
        // мы сериализуем только ссылку на него и каталогизируем
        // объект для дальнейшей сериализации.
        WriteObjectRef(obj, name, memberType);
    }
    else
    {
        //
        // Непосредственная сериализация массива.

        // Для создания массива требуется его тип, длины и
        // нижние границы по всем измерениям.
        System.Array a = (Array)obj;

        // Пока что этот форматировщик поддерживает лишь
        // одномерные массивы.
        if ( a.Rank != 1 )
        {
            throw new NotSupportedException(
                "This formatter supports only
                1-dimensional arrays");
        }

        WriteField(FieldNames.ARRAY_RANK, a.Rank);

        for(int i = 0; i < a.Rank; ++i)
        {
            WriteField(FieldNames.ARRAY_LENGTH, a.GetLength(i));
            WriteField(FieldNames.ARRAY_LOWERBOUND,
                a.GetLowerBound(i));
        }
    }
}
```



```

string name,
Type  memberType )

{
    // Имя члена.
    if ( name != "" )
    {
        WriteField(FieldNames.MEMBER_NAME, name);
    }

    // Имя типа.
    WriteField(FieldNames.MEMBER_TYPE,
        memberType.AssemblyQualifiedName);

    // Значение члена,
    if ( obj == null )
    {
        // Null:
        // Для значений null используется специальный индикатор.
        WriteField(FieldNames.MEMBER_VALUE_NULL, "");
    }
    else
    {
        // Объект:
        // Объект каталогизируется для сериализации.
        long id = Schedule(obj);

        // Выводим в поток идентификатор объекта,
        // а не сам объект.
        WriteField(FieldNames.OBJECT_REFID, id);
    }
}

```

Для размерных типов, отличных от элементарных, метод *WriteMember* вызывает метод *WriteValueType*. Мы выводим в поток имя члена, если оно задано, за которым следуют тип и значение члена. Класс *System.Void*, представляющий тип *void*, необходимо обрабатывать особо. Экземпляр *System.Void* нельзя создать непосредственно. Следовательно, мы используем тот же прием, что и в методе *WriteType*, и просто выводим квалифицированное имя сборки, содержащей тип *System.Void*, и флаг, обеспечивающий корректную десериализацию:

```

protected override void WriteValueType( object obj,
string name,
Type  memberType )

```

```

{
    // Вывести имя члена, если оно задано.
    if ( name != "" )
    {
        WriteField(FieldNames.MEMBER_NAME, name);
    }

    // Вывести тип члена.
    WriteField(FieldNames.MEMBER_TYPE,
        memberType.AssemblyQualifiedName);

    // Вывести значение члена.
    // Особая обработка типа void.
    if ( memberType.FullName == "System.Void" )
    {
        WriteField( FieldNames.MEMBER_VALUE_TYPE,
            memberType.AssemblyQualifiedName );
    }
    else
    {
        WriteField(FieldNames.MEMBER_VALUE, obj);
    }
}

```

Реализация остальных защищенных виртуальных методов *WriteXXXX* вызывает метод *WriteValueType*. Добавив этот код, мы получаем законченную реализацию метода *IFormatter.Serialize*.

```

protected override void WriteBoolean(bool val, string name)
{
    WriteValueType(val, name, val.GetType());
}

protected override void WriteByte(byte val, string name)
{
    WriteValueType(val, name, val.GetType());
}

protected override void WriteChar(char val, string name)
{
    WriteValueType(val, name, val.GetType());
}

protected override void WriteDateTime(System.DateTime val,
    string name)
{
    WriteValueType(val, name, val.GetType());
}

```

```
}

protected override void WriteDecimal(decimal val, string name)
{
    WriteValueType(val, name, val.GetType());
}

protected override void WriteDouble(double val, string name)
{
    WriteValueType(val, name, val.GetType());
}

protected override void WriteInt16(short val, string name)
{
    WriteValueType(val, name, val.GetType());
}

protected override void WriteInt32(int val, string name)
{
    WriteValueType(val, name, val.GetType());
}

protected override void WriteInt64(long val, string name)
{
    WriteValueType(val, name, val.GetType());
}

protected override void WriteSByte(sbyte val, string name)
{
    WriteValueType(val, name, val.GetType());
}

protected override void WriteSingle(float val, string name)
{
    WriteValueType(val, name, val.GetType());
}

protected override void WriteTimeSpan(System.TimeSpan val,
                                         string name)
{
    WriteValueType(val, name, val.GetType());
}

protected override void WriteUInt16(ushort val, string name)
{
    WriteValueType(val, name, val.GetType());
}
```

```

        WriteValueType(val, name, val.GetType());
    }

    protected override void WriteUInt32(uint val, string name)
    {
        WriteValueType(val, name, val.GetType());
    }

    protected override void WriteUInt64(ulong val, string name)
    {
        WriteValueType(val, name, val.GetType());
    }
}

```

Реализация метода *IFormatter.Deserialize*

Назначение метода *IFormatter.Deserialize* заключается в десериализации графа объектов из потока и возврате корневого объекта графа. Вот реализация метода *IFormatter.Deserialize* для класса *MyFormatter*:

```

public override
    object Deserialize(System.IO.Stream serializationStream)
{
    //
    // Создать ObjectManager для поддержки десериализации.
    _om = new ObjectManager( _surrogateselector,
                            _streamingcontext );

    _reader = new StreamReader(serializationStream);

    // Читать объекты, пока не будет найден конец потока.
    while( _reader.Peek() != -1 )
    {
        ReadObject();
    }

    //
    // Теперь можно выполнить поправки и вернуть
    // корневой объект.
    _om.DoFixups();

    // Возвратить корневой объект.
    return _om.GetObject(1);
}

```

Для воссоздания графа объектов метод *MyFormatter.Deserialize* использует класс *ObjectManager*, который мы рассмотрели ранее.

После создания экземпляра *StreamReader* для входного потока в цикле последовательно считываются объекты, пока не будет найден конец потока, определяемый по возвращаемому значению *StreamReader.Peek*, равному -1 . После десериализации графа из потока мы вызываем *ObjectManager.DoFixup* для выполнения поправок и получения корня графа объектов. Вот реализация метода *ReadObject*:

```
void ReadObject()
{
    // Считать идентификатор объекта.
    long oid = FieldNames.ParseObjectID(_reader.ReadLine());

    // Считать сборку объекта.
    string s_assembly =
        FieldNames.ParseObjectAssembly(_reader.ReadLine());

    // Считать тип объекта.
    string s_otype = FieldNames.ParseObjectType(
        _reader.ReadLine());

    // Считать члены объекта для этого типа.
    Type t = System.Type.GetType( String.Format( "{0},{1}",
                                                s_otype,
                                                s_assembly ) );

    if ( t.IsArray )
    {
        ReadArray(oid, t);
    }
    else if ( t == typeof(string) )
    {
        object o = FieldNames.ParseMemberValue(
            _reader.ReadLine());
        _om.RegisterObject(o, oid);
    }
    else
    {
        SerializationInfo info;

        object o = ReadObjectMembers(oid, t, out info);

        if { info == null }
        {
            _om.RegisterObject(o, oid);
        }
    }
}
```

```

    }
    else
    {
        _om.RegisterObject(o, oid, info);
    }
}

```

По сути метод *ReadObject* — обратный методу *WriteObject*. *ReadObject* считывает из потока идентификатор объекта, имя сборки и тип объекта. Вспомните, что *WriteObject* обрабатывает экземпляры строк и массивов не так, как другие типы. Это означает, что *ReadObject* также должен обрабатывать их особо. Если тип является массивом, то для его считывания вызывается метод *ReadArray*, который мы рассмотрим позже. Если типом объекта является строка, то она считывается из потока с помощью метода *FieldNames.ParseMemberValue*, который был определен ранее. Теперь, когда мы считали объект целиком, он регистрируется в *ObjectManager*. Для всех остальных типов метод *ReadObject* считывает сериализованные члены объекта из потока с помощью метода *ReadObjectMembers*:

```

private void ReadObjectMembers( long oid,
                                Type t,
                                out SerializationInfo info )
{
    info = null;

    // Попытаться найти суррогат для типа.
    ISerializationSurrogate surrogate = null;
    if ( surrogateselector != null )
    {
        ISurrogateSelector selector;
        surrogate =
            _surrogateselector.GetSurrogate( t,
                                              _streamingcontext,
                                              out selector );
    }

    object o = FormatterServices.GetUninitializedObject( t );

    // Считать члены объекта.
    if ( surrogate != null )
    {

```



```

// Считать число членов.
long count = FieldNames.ParseMemberCount(
    _reader.ReadLine());

for( int i = 0; i < count; ++i)
{
    ReadMember(oid, null, o, info);
}

```

Метод *ReadSerializationInfo* соответствует методу *WriteSerializationInfo*. Он считывает из потока число членов и затем считывает оттуда каждый из них, вызывая метод *ReadMember* (о нем речь пойдет далее).

Аналогично *ReadSerializationInfo*, метод *ReadSerializableMembers* получает массив экземпляров *MemberInfo* для сериализуемых членов типа и затем считывает каждый член из потока вызовом *ReadMember*:

```

void ReadSerializableMembers( Object o, long oid)
{
    MemberInfo[] mi =
        FormatterServices.GetSerializableMembers(o.GetType());

    // Считать все члены.
    for( int i = 0; i < mi.Length; ++i )
    {
        ReadMember(oid, mi[i], o, null);
    }
}

```

Вот реализация метода *ReadMember*, который считывает член из потока:

```

void ReadMember( long oid,
                 MemberInfo mi,
                 object o,
                 SerializationInfo info )
{
    // Считать имя члена.
    string sname = FieldNames.ParseMemberName(
        _reader.ReadLine());

    // Считать тип члена.

```

```

string stype = FieldNames.ParseMemberType(
    _reader.ReadLine());

// Считать значение члена.
string svalue = _reader.ReadLine();
long roid = 0;
object ovalue = ReadMemberValue(svalue, stype, ref roid);

if ( roid != 0 )
{
    // Встречался ли уже этот объект?
    if ( ovalue == null )
    {
        // При наличии у объекта SerializationInfo
        // записываем отложенную поправку.
        if ( info != null )
        {
            _om.RecordDelayedFixup(oid, sname, roid);
        }
        else
        {
            _om.RecordFixup(oid, mi, roid);
        }
    }

    return;
}

if ( info != null )
{
    info.AddValue(sname, ovalue);
}
else
{
    FormatterServices.PopulateObjectMembers(
        o,
        new MemberInfo[]{mi},
        new object[]{ovalue});
}
}

```

Метод *ReadMember* считывает из потока имя, тип и значение члена. Чтобы считать значение члена, *ReadMember* вызывает метод *ReadMemberValue*, который мы скоро рассмотрим. Возвращаемым значением последнего является значение члена, а последний параметр с именем *roid* будет ненулевым, если член ссы-

лается на другой объект в графе. Если значение члена ссылается на объект графа, который еще не был десериализован, то необходимо записать для этого члена поправку для помещения в него ссылки на объект по идентификатору объекта. Если у объекта, на который ссылается член, имеется *SerializationInfo*, то с помощью *RecordDelayedFixup* записывается отложенная поправка для имени члена. В противном случае поправка записывается посредством экземпляра *MemberInfo*. Если *roid* равен 0, то значение члена не является ссылкой на другой объект в графе, и мы можем использовать его для инициализации члена объекта. Если у объекта имеется *SerializationInfo*, мы добавляем значение члена в экземпляр *SerializationInfo*. В противном случае для инициализации члена объекта десериализованным значением используется метод *FormatterServices.PopulateObjectMembers*. Ниже показана реализация метода *ReadMemberValue*:

```
private object ReadMemberValue(string svalue,
                               string stype,
                               ref long rroid)
{
    if ( svalue.StartsWith( FieldNames.OBJECT_REFID ) )
    {
        // Этот член ссылается на другой объект.
        rroid = FieldNames.ParseObjectRefID(svalue);
        return _om.GetObject(rroid);
    }
    else if ( svalue.StartsWith(
        FieldNames.MEMBER_VALUE_TYPE ) )
    {
        // Этот член следует интерпретировать как тип.
        string s = FieldNames.ParseMemberValueType(svalue);
        return Type.GetType(s);
    }
    else if ( svalue.StartsWith( FieldNames.MEMBER_VALUE ) )
    {
        // Размерный тип; преобразовать строковое представление
        // в фактический тип.
        string s = FieldNames.ParseMemberValue(svalue);
        return Convert.ChangeType(s, Type.GetType(stype));
    }
    else if ( svalue.StartsWith(
        FieldNames.MEMBER_VALUE_NULL ) )
    {

```

```

        // Значением является null.
        return null;
    }
    else
    {
        throw new SerializationException("Parse error.");
    }
}

```

Метод *ReadMemberValue* проверяет флаг поля, с которого начинается параметр *svalue*. Для нашего форматировщика возможны четыре варианта. Значением члена может быть ссылка на другой объект, тогда мы считываем идентификатор объекта и возвращаем соответствующий объект, если он уже был десериализован. Другой вариант реализуется, когда значение члена должно быть интерпретировано как *Type*. В этом случае мы считываем значение члена и создаем экземпляр *Type* вызовом метода *Type.GetType*. Значением члена также может быть просто строковое представление элементарного или размерного типа, и тогда мы считываем значение члена и с помощью метода *Convert.ChangeType* преобразуем строку в экземпляр сериализованного типа. Последний вариант возможен, когда значение есть *null*, и тогда мы возвращаем *null*.

Последние два метода понадобятся нам для реализации чтения из потока массива объектов. Метод *ReadArray* выглядит так:

```

object ReadArray(long arrayId, Type t)
{
    // Считать размерность массива.
    long rank = FieldNames.ParseArrayRank(_reader.ReadLine());

    // Мы поддерживаем только размерность == 1.
    if ( rank != 1 )
    {
        throw new System.NotSupportedException(
            "This formatter supports only
            1-dimensional arrays");
    }

    long length = FieldNames.ParseArrayLength(
        _reader.ReadLine());

    long lowerbound =

```

```

        FieldNames.ParseArrayLowerBound(_reader.ReadLine());

        // Создать массив с помощью Array.CreateInstance.
        Array oa = Array.CreateInstance(t.GetElementType(),
                                       (int)length);

        // Зарегистрировать массив на тот случай, если потребуются
        // поправки для элементов.

        _om.RegisterObject(oa, arrayId);

        // Считать элементы массива.
        for(int i=0; i < length; ++i)
        {
            ReadArrayElement(oa, arrayId, i, t.GetElementType());
        }

        return oa;
    }
}

```

Метод *ReadArray* считывает из потока число размерностей, нижнюю границу и длину массива. Для простоты в нашем примере поддерживаются только одномерные массивы. Сначала при разработке этого метода мы попытались использовать для создания экземпляра массива метод *FormatterServices.GetUninitializedObject*. В результате генерировалось исключение *ExecutionEngineException*. Мы также пытались просто создать универсальный массив объектов (*object []*), но тогда исключение возникало при преобразовании возвращаемого значения метода *Deserialize* в массив соответствующего типа (например, *int []*). Единственным способом заставить этот метод работать оказался вызов метода *Array.CreateInstance* для создания массива конкретного типа и длины. Создав массив, мы регистрируем его в *ObjectManager*. После этого каждый элемент массива считывается с помощью метода *ReadArrayElement*:

```

void ReadArrayElement(System.Array oa,
                     long oid,
                     int index,
                     Type el type)
{
    // Считать тип.
    string stype = FieldNames.ParseMemberType(
        _reader.ReadLine());
}

```

```

Type t = Type.GetType(stype);

// Считать значение.
string svalue = _reader.ReadLine();
long roid = 0;
object ovalue = ReadMemberValue(svalue, stype, ref roid);

if ( roid != 0 )
{
    // Встречался ли этот объект ранее?
    if ( ovalue == null )
    {
        _om.RecordArrayElementFixup(oid, index, roid);
        return;
    }
}

oa.SetValue(ovalue, index) ;
}

```

Метод *ReadArrayElement* считывает из потока тип и значение. Как и в методе *ReadMember*, после вызова *ReadMemberValue* мы проверяем значение переменной *roid*. Ненулевое значение указывает на то, что элемент массива ссылается на экземпляр другого объекта графа. Если объект, возвращаемый *ReadMemberValue*, есть *null*, то объект, на который ссылается этот член, еще не десериализован. Тогда мы записываем поправку с помощью метода *RecordArrayElementFixup* и возвращаемся из *ReadArrayElement*. Иначе элемент массива не ссылается на другой объект графа или его значение равно *null*. В обоих случаях мы устанавливаем значение элемента массива в объект, возвращенный методом *ReadMemberValue*.

Итак, у нас имеется полностью работоспособный форматировщик. Теперь, когда мы разработали свой форматировщик, давайте рассмотрим процедуру его подключения к архитектуре .NET Remoting.

Создание приемника форматировщика

Как говорилось в главе 2, .NET Remoting использует приемники форматировщиков для записи объектов *IMessage* в поток, который затем передается остальным канальным приемникам для доставки удаленному объекту. Инфраструктура .NET Remoting

разделяет реализацию приемника форматировщика на две части: *клиентский приемник форматировщика* (client formatter sink) и *серверный приемник форматировщика* (server formatter sink),

Клиентский приемник форматировщика

Первым приемником в цепочке канальных приемников на клиентской стороне является экземпляр клиентского приемника форматировщика, который реализует интерфейс *IClientFormatterSink*. Клиентский приемник форматировщика выступает в качестве моста между цепочкой приемников сообщений и цепочкой канальных приемников. Таким образом, клиентский приемник форматировщика является и приемником сообщений, и канальным приемником. Интерфейс *IClientFormatterSink* является композицией интерфейсов *IMessageSink*, *IClientChannelSink* и *IChannelSinkBase*. Ниже определяется класс *MyClientFormatterSink*, использующий форматировщик *MyFormatter*, разработанный нами в предыдущем разделе:

```
public class MyClientFormatterSink : IClientFormatterSink
{
    private IClientChannelSink _NextChannelSink;
    private IMessageSink _NextMessageSink;

    public MyClientFormatterSink(IClientChannelSink next)
    {
        _NextChannelSink = next;
    }

    //
    // IChannelSinkBase
    public IDictionary Properties
    {
        get{ return null; }
    }

    //
    // IClientChannelSink
    public IClientChannelSink NextChannelSink
    {
        get{return _NextChannelSink;}
    }

    public void AsyncProcessRequest(
```

```

        IClientChannelSinkStack sinkStack,
        IMessage msg,
        ITransportHeaders headers,
        Stream stream )
    {
        // Данный приемник должен быть первым в цепочке,
        // так что этот метод не должен никогда вызываться.
        throw new NotSupportedException();
    }

    public void AsyncProcessResponse (
        IClientResponseChannelSinkStack sinkStack,
        object state,
        ITransportHeaders headers,
        Stream stream )
    {
        // Можно было бы реализовать, но не реализован.
        throw new NotImplementedException();
    }

    public System.IO.Stream GetRequestStream ( IMessage msg,
        ITransportHeaders headers )
    {
        // Данный приемник должен быть первым в цепочке,
        // так что этот метод не должен никогда вызываться.
        throw new NotSupportedException();
    }

    public void ProcessMessage ( IMessage msg,
        ITransportHeaders requestHeaders,
        Stream requestStream,
        out ITransportHeaders
        responseHeaders,
        out Stream responseStream )
    {
        // Данный приемник должен быть первым в цепочке,
        // так что этот метод не должен никогда вызываться.
        throw new NotSupportedException();
    }

    //
    // IMessageSink
    public System.Runtime.Remoting.Messaging.IMessageSink
        NextSink
    {
        get{return _NextMessageSink; }
    }

```



```

        out responseHeaders,
        out responseStream );

    // Использовать нашу версию IMessage для десериализации.
    fm.SurrogateSelector = null;

    // Форматировщик обрабатывает объекты типа
    // IMessage особым образом и десериализует их как типы
    // MyMessage.
    MyMessage mr = (MyMessage)fm.Deserialize(
        responseStream);
    return mr.ConvertMyMessagePropertiesToMethodResponse(mc);
}
}

```

Во-первых, необходимо сделать ряд замечаний по поводу реализации класса *MyClientFormatterSink*. Так как клиентский приемник форматировщика является первым приемником в цепочке канальных приемников, то мы не ожидаем вызовов следующих методов *IClientChannelSink*: *AsyncProcessRequest*, *GetRequestStream* и *ProcessMessage*. Таким образом, каждый из них генерирует исключение *NotSupportedException*. Мы также не реализовали поддержку асинхронных вызовов и, следовательно, генерируем исключение *NotImplementedException* в методах *IClientChannelSink.AsyncProcessResponse* и *IMessageSink.AsyncProcessMessage*, реализация которых предлагается вам в качестве упражнения.

Основная работа выполняется в методе *IMessage.SyncProcessMessage*. В общем случае реализация *SyncProcessMessage* в клиентском приемнике форматировщика:

1. получает поток запроса для сериализации сообщения запроса;
2. сериализует сообщение в поток запроса;
3. передает поток запроса методу *ProcessMessage* следующего канального приемника в цепочке;
4. десериализует возвращаемое сообщение из потока ответа и возвращает его.

Реализация *SyncProcessMessage* в *MyClientFormatterSink* создает новый экземпляр класса *TransportHeaders*, который затем вместе с параметром *msg* используется для получения потока запроса вызовом метода *GetRequestStream* следующего приемника в це-

почке канальных приемников. Если *GetRequestStream* не возвращает экземпляр *Stream*, мы создаем поток в памяти.

Зачем нужен класс *MyMessage*?

При реализации приемников форматировщиков мы столкнулись с проблемой при десериализации экземпляров *MethodCall*. В серверном приемнике форматировщика после десериализации экземпляра *MethodCall* мы пытались передать его методу *ProcessMessage* следующего канального приемника. В результате генерировалось исключение *StackOverflowException*.

Единственный способ борьбы с данной проблемой — класс *MyMessage*, который создается форматировщиком во время десериализации вместо типов *IMessage*. Нам пришлось изменить метод *MyFormatter.ReadObjectMembers*, чтобы сразу после создания неинициализированного экземпляра объекта метод проверял тип объекта на предмет поддержки им интерфейса *IMessage*. Если тип объекта поддерживает этот интерфейс, мы создаем вместо неинициализированного экземпляра, возвращаемого *FormatterServices.GetUninitializedObject*, экземпляр класса *MyMessage*.

Следовательно, в десериализованном графе объектов вместо экземпляра *MethodCall* класс создает экземпляр *MyMessage*. Данный класс реализует интерфейс *IMessage* и выступает в качестве временного заместителя, хранящего свойства сообщений для экземпляров типов, поддерживающих интерфейс *IMessage*.

Класс *MyMessage* предоставляет два метода — *ConvertMyMessagePropertiesToMethodCall* и *ConvertMyMessagePropertiesToMethodResponse*, которые преобразуют свойства, содержащиеся в *MyMessage*, в экземпляры *MethodCall* или *MethodResponse* соответственно.

Так как мы будем сериализовать типы инфраструктуры .NET Remoting, экземпляру форматировщика передается экземпляр класса *RemotingSurrogateSelector*. Также обратите внимание на то, что мы сериализуем в поток новый экземпляр сообщения *MethodCall*, а не экземпляр *IMessage*, переданный *SyncProcessMessage*.

Через параметр *msg* методу *SyncProcessMessage* клиентского приемника форматировщика передается экземпляр *System.Runtime.Remoting.Messaging.Message*. Класс *Message* поддерживает *ISerializable*, но не имеет специального конструктора, необходимого для десериализации. *MessageSurrogate* обрабатывает сериализацию типа *Message*, но не поддерживает десериализацию каких-либо типов *IMessage*. Вместо сериализации в поток типа *Message*, мы можем создать новый экземпляр класса *MethodCall*, передав конструктору объект *msg*. В отличие от класса *Message*, класс *MethodCall* поддерживает *ISerializable* и имеет специальный конструктор, необходимый для десериализации. Создав новый экземпляр *MethodCall*, мы сериализуем его в поток, указываемый переменной *requestStream*, который затем передается следующему приемнику в цепочке канальных приемников путем вызова метода *ProcessMessage* члена *_NextChannelSink*. После того как вызов *ProcessMessage* следующего приемника возвращает управление, мы устанавливаем свойство *SurrogateSelector* в *null* и десериализуем поток ответа в тип *MyMessage* (см. врезку «Зачем нужен класс *MyMessage*?»). Наконец, мы преобразуем тип *MyMessage* в экземпляр *Method Response*, который и возвращается.

ClientFormatterSinkProvider

Теперь, когда у нас есть клиентский приемник форматировщика, необходимо создать класс провайдера канального приемника, который годился бы для помещения нашего приемника форматировщика в клиентскую цепочку канальных приемников. Класс *MyFormatterClientSinkProvider* определен ниже:

```
public class MyFormatterClientSinkProvider :
    IClientFormatterSinkProvider
{
    public MyFormatterClientSinkProvider()
    {
    }

    public MyFormatterClientSinkProvider(
        IDictionary properties,
        ICollection providerData)
    {
    }

    // Построить клиентскую канальную цепочку.
```

```

public IClientChannelSink CreateSink (
    IChannelSender channel,
    string url ,
    object remoteChannelData )
{
    // Запросить цепочку у следующего провайдера,
    IClientChannelSink chain =
        _Next.CreateSink(channel,url,remoteChannelData);

    // Добавить наш форматировщик в начало цепочки.
    IClientChannelSink sinkFormatter =
        new MyClientFormatterSink(chain);

    return sinkFormatter;
}

private IClientChannelSinkProvider _Next=null;
public IClientChannelSinkProvider Next
{
    get{return _Next;}
    set{ Next = value;}
}
}

```

Серверный приемник форматировщика

В отличие от клиентского приемника форматировщика, серверный приемник форматировщика не является одновременно и приемником сообщений, и канальным приемником, но *лишь* последним. Серверный приемник форматировщика реализует интерфейс *IServerChannelSink* и является последним в цепочке канальных приемников. Ниже определен класс *MyServerFormatterSink*, использующий разработанный нами ранее форматировщик *MyFormatter*:

```

public class MyServerFormatterSink : IServerChannelSink
{
    private IServerChannelSink _NextChannelSink;

    public MyServerFormatterSink( IServerChannelSink snk)
    {
        _NextChannelSink = snk;
    }

    public IDictionary Properties
    {

```

```

        get
        { return null; }
    }

    public IServerChannelSink NextChannelSink
    {
        get
        { return _NextChannelSink; }
    }

    public Stream GetResponseStream(
        IServerResponseChannelSinkStack sinkStack,
        object state,
        IMessage msg,
        ITransportHeaders headers).
    {
        // Вызов данного метода не ожидается, так как
        // этот приемник не помещается в стек приемников ответа.
        throw new NotSupportedException();
    }

    public void AsyncProcessResponse(
        IServerResponseChannelSinkStack sinkStack,
        object state,
        IMessage msg,
        ITransportHeaders headers,
        Stream stream)
    {
        // Можно было бы реализовать, но не реализован.
        throw new NotImplementedException();
    }

    public ServerProcessing ProcessMessage(
        IServerChannelSinkStack sinkStack,
        IMessage requestMsg,
        ITransportHeaders requestHeaders,
        System.IO.Stream requestStream,
        out IMessage responseMsg,
        out ITransportHeaders responseHeaders,
        out System.IO.Stream responseStream)
    {
        // Инициализация выходных параметров.
        responseMsg = null;
        responseHeaders = null;
        responseStream = null;
    }

```

```

// Подготовка к десериализации потока запроса.
RemotingSurrogateSelector rem_ss =
    new RemotingSurrogateSelector();

MyFormatter fm = new MyFormatter();
fm.SurrogateSelector = null;
fm.Context =
    new StreamingContext(
        StreamingContextStates.Other );

IMessage msg = null;

MyMessage mymsg =
    (MyMessage)fm.Deserialize(requestStream);

//
// Анализ свойства __URI.
string uri = (string)mymsg.Properties["__Uri"];
int n = uri.LastIndexOf("/");
if ( n != -1 )
{
    uri = uri.Substring(n);
    mymsg.Properties[" Uri"] = uri;
}

// Преобразовать MyMessage в MethodCall.
MethodCall mc =
    mymsg.ConvertMyMessagePropertiesToMethodCall();
msg = (IMessage)mc;

// При вызове исполняющего (dispatch) приемника
// (следующего в нашей цепочке) поток запроса
// должен быть null.
ServerProcessing sp =
    this._NextChannelSink.ProcessMessage(
        sinkStack,
        msg,
        requestHeaders,
        null,
        out responseMsg,
        out responseHeaders,
        out responseStream );

if ( sp == ServerProcessing.Complete )
{
    // Сериализовать ответное сообщение в поток ответа.

```

```

        if ( responseMsg != null && responseStream == null )
        {
            responseStream = sinkStack.GetResponseStream(
                responseMsg,
                responseHeaders);

            if ( responseStream == null )
            {
                responseStream = new MemoryStream();
            }

            fm.SurrogateSelector = rem_ss;
            fm.Serialize(responseStream, responseMsg);
        }
    }

    return sp;
}

```

Основная работа выполняется в методе *IServerChannelSink.ProcessMessage*. В общем случае реализация *ProcessMessage* серверного приемника форматировщика должна выполнять следующие действия:

1. десериализовать экземпляр *MethodCall* из потока запроса;
2. передать экземпляр *MethodCall* методу *ProcessMessage* следующего приемника в цепочке — *DispatchSink*;
3. получить поток ответа для сериализации ответного сообщения;
4. сериализовать ответное сообщение в поток ответа,

Реализация *ProcessMessage* в *MyServerFormatterSink* инициализирует свои выходные параметры, создает экземпляр класса *MyFormatter* и подготавливает форматировщик к десериализации *IMessage* из потока запроса. Из-за проблем, которые обсуждались на врезке «Зачем нужен класс *MyMessage*?», объект-сообщение десериализуется как экземпляр класса *MyMessage*. Затем необходимо преобразовать свойство *_Uri* сообщения в URI объекта. Для этого мы оставляем только ту часть строки свойства *_Uri*, которая следует за последним символом «/». Если не модифицировать значение свойства *_Uri*, то при попытке дальнейшей обработки сообщения будет сгенерировано исключение, похожее на изображенное на рис. 8-4.

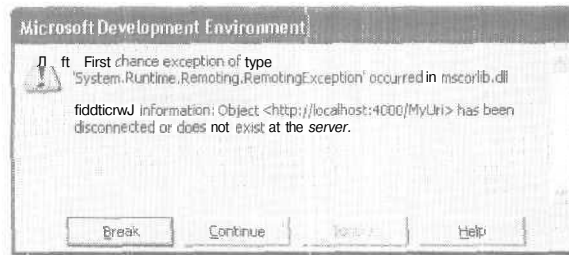


Рис. 8-4. Исключение, генерируемое в том случае, если не модифицировать свойство `_Uri` перед его передачей `DispatchSink`

ВНИМАНИЕ! Инфраструктура .NET Remoting требует, чтобы объект-сообщение, передаваемый `DispatchSink`, был типом, поддерживающим `IMessage` и определенным .NET Framework. Первоначально мы передавали тип `MyMessage`, который поддерживает `IMessage`, но в результате генерировалось исключение с сообщением об ошибке «Permission denied. Cannot call methods on `AppDomain` class remotely.»¹

После модификации свойства `_Uri` мы преобразуем экземпляр `MyMessage` в экземпляр `MethodCall`, который затем передается методу `ProcessMessage` следующего канального приемника - `DispatchSink`. Если возвращаемое значение `ProcessMessage` свидетельствует о том, что вызов метода завершен, мы получаем поток для сериализации ответного сообщения, вызывая метод `GetRequestStream` объекта `IServerChannelSinkStack`. Это стандартное соглашение для получения потока, которое позволяет приемникам в цепочке добавлять информацию в поток ответа до того, как серверный приемник форматировщика сериализует ответное сообщение. Если `GetRequestStream` не возвращает экземпляр `Stream`, то серверный приемник форматировщика создает его. Затем свойство форматировщика `SurrogateSelector` устанавливается в объект класса `RemotingSurrogateSelector` и выполняется сериализация ответного сообщения в поток ответа,

¹ Доступ запрещен. Удаленный вызов методов класса `AppDomain` не допускается. — Прим. перев.

ServerFormatterSinkProvider

Теперь, когда у нас есть серверный приемник форматировщика, необходимо создать класс провайдера канального приемника, с помощью которого удалось бы добавить наш приемник в серверную цепочку канальных приемников. Класс *MyFormatterServerSinkProvider* определен ниже:

```
public class MyFormatterServerSinkProvider :
    IServerFormatterSinkProvider
{
    public MyFormatterServerSinkProvider()
    {
    }

    // Эта форма конструктора служит для инициализации
    // из конфигурационного файла,
    public MyFormatterServerSinkProvider(
        IDictionary properties,
        ICollection providerData )
    {
    }

    private IServerChannelSinkProvider _Next;

    public IServerChannelSink CreateSink(
        IChannelReceiver channel)
    {
        IServerChannelSink chain = _Next.CreateSink(channel);
        IServerChannelSink sinkFormatter =
            new MyServerFormatterSink(chain);
        return sinkFormatter;
    }

    public void GetChannelData(IChannelDataStore channelData)
    {
        if ( _Next != null )
        {
            _Next.GetChannelData(channelData);
        }
    }

    public IServerChannelSinkProvider Next
    {
        get
        { return _Next; }
    }
}
```

```
        set  
        { _Next = value; }  
    }  
}
```

Заключение

В этой главе мы обсудили сериализацию объектов, рассмотрели ряд классов, применяющихся для сериализации графов объектов, и создали свой форматировщик сериализации, который затем использовали при реализации серверного и клиентского приемников форматировщика.

Предметный указатель

- - .NET Code Access Security 103
 - .NET Framework 2
 - .NET Remoting 1, 2, 8, 9, 18, 19, 22
 - граница 28, 43
 - канал 20
 - подключаемый модуль 19
 - форматировщик 20
- A**
 - API COM 17
 - ASP (Active Server Pages) 90
 - ASP.NET 65, 90
 - assembly *см.* сборка
- C**
 - C# 74
 - channel sink *см.* приемник, каналный
 - client context sink chain
 - см.* цепочка, приемников клиентского контекста
 - client formatter sink
 - см.* приемник, форматировщика, клиентский
 - CLR (common language runtime) 21, 22, 25, 28, 90, 93
 - COM 2, 18, 21
 - COM+ 25, 65, 66
 - common language runtime *см.* CLR
 - Common Object Request Broker Architecture *см.* CORBA
 - common type system *см.* CTS
 - context-bound *см.* объект, контекстно-связанный
 - CORBA (Common Object Request Broker Architecture) 2, 8, 9, 15, 22
 - cross-context channel *см.* канал, межконтекстный
 - CTS (common type system) 21, 22
- D**
 - DCE (Distributed Computing Environment) 8
 - DCOM 1, 2, 8, 9, 15, 17, 18, 21, 22, 66
 - Dcomcnfg 17
 - direct principal access *см.* прямой опрос пользователя
 - discovery server *см.* сервер, поиска
 - Distributed Computing Environment *см.* DCE
 - document/literal *см.* SOAP, документ-литеральный
- E**
 - endpoint *см.* конечная точка
 - envoy sink chain *см.* цепочка, агентских приемников
- F**
 - formatter provider *см.* провайдер форматировщика
- G**
 - GAC (global assembly cache) 95

global assembly cache *см.* CAC
Gnutella 6
GUID 45

H

HTML 90
HTTP 16, 20, 23, 49, 50, 53,
89, 91, 93, 124, 125, 166,
170, 225

I

identity *см.* идентифициру-
емость
IDL (Interface Definition
Language) 8, 21, 91, 92
IIS (Internet Information
Services) 24, 25, 65, 66, 67,
94, 98, 166
Interface Definition Language
см. IIS
interprocess communication
см. IPC
IPC (interprocess
communication) 27

J

Java 2
Java RMI 111

L

lease *см.* лицензия
lease manager *см.* диспетчер
лицензий
listening port *см.* порт ожидания
load-balancing
см. распределение нагрузки

M

marshal-by-reference *см.* тип,
дистанцируемый,
передаваемый по ссылке

marshal-by-value *см.* тип,
дистанцируемый,
передаваемый по значению
Microsoft Management Console
см. MMC
Microsoft Message Queuing
см. MSMQ
Microsoft Visual Studio .NET 75
MMC (Microsoft Management
Console) 21, 94
MSMQ (Microsoft Message
Queuing) 124

N

named pipe *см.* канал,
именованный
Napster 6
NAT (Network Address
Translation) 93
Network Address Translation
см. NAT
nonremotable *см.* объект,
недистанцируемый
nonwrapped proxy *см.* прокси,
открытый
NTLM 94, 98
NTML 24

O

Object Management Group
см. OMG
OMG (Object Management
Group) 2

P

pinging *см.* объект, опрос

R

reflection *см.* отражение
remotable *см.* объект,
дистанцируемый

Remote Method Invocation
 см. RMI
 remote procedure call *CM.* RPC
 repeater *см.* повторитель
 RMI (Remote Method
 Invocation) 2, 8, 9
 RPC (remote procedure call) 8, 9

S

scheduling *см.* объект,
 каталогизация
 SCM (Service Control
 Manager) 66
 Secure Sockets Layer *CM.* SSL
 server context sink chain
 см. *общеконтекстная*
 цепочка приемников
 серверного контекста
 server formatter sink *см.*
 приемник, форматировщика,
 серверный
 server object sink chain
 см. цепочка, приемников
 серверного объекта
 Service Control Manager *см.* SCM
 Simple Mail Transfer Protocol
CM. SMTP
 Simple Object Access Protocol
CM. SOAP
 SMTP (Simple Mail Transfer
 Protocol) 124
 snap-in *см.* оснастка
 SOAP (Simple Object Access
 Protocol) 2, 20, 23, 53, 91,
 123, 125
 — документ-литеральный 129
 — заголовок 127
 — конверт 126
 — ошибка 129
 — тело 128
 — форматировщик 130

SOAPSuds 96
 SSL (Secure Sockets Layer) 24,
 100
stackbuilder sink *см.* приемник,
 стекопостроителя
 stand-in *см.* класс, дублер
 stub *см.* заглушка

T

TCP 12, 20, 49, 166, 170, 225
 TCP/IP 52

U

UDDI (Universal Description,
 Discovery, and Integration) 91
 UDP (User Datagram
 Protocol) 235
 Uniform Resource Identifier *CM.*
 URI
 Universal Description, Discovery,
 and Integration *CM.* UDDI
 URI (Uniform Resource
 Identifier) 34, 37, 45, 48, 125
 URL 91
 User Datagram Protocol *CM.* UDP

W

WAP (Wireless Application
 Protocol) 225
 Web Service Description
 Language *CM.* WSDL
 Web-сервер 16
 Web-сервис 23, 67, 90, 91, 93,
 98
 Windows Forms 65, 67, 74, 84,
 93
 Wireless Application Protocol
CM. WAP
 wrapped proxy *см.* прокси,
 закрытый
 WSDL (Web Service Description
 Language) 23, 91, 96, 97

X

XML 2, 20, 23, 91, 123

A

аутентификация 15, 24

— Passport 24

— базовая 24

— на основе сертификатов 24

Б

библиотека типов 22, 92

брандмауэр 16

В

ввод-вывод потоковый 7

Гглобальный кэш сборок
см, GAC**Д**

десериализация 30, 276, 284

диспетчер лицензий 39, 40, 41

домен приложения 28, 41, 43,
50

— клиентский 74

— серверный 65

Ж

журнал 190

З

заглушка 8

И

идентифицируемость 9

изоляция адресных
пространств 27

инкапсуляция 48

интерфейс

— AccessTimeServerChannel-
Sink 269

— FileServerChannel 249

— IChannel 49, 87

— IChannelReceiver 49

— IChannelSender 49, 237

— IChannelSinkBase 328

— IClientChannelSink 328

— IClientChannelSink-
Provider 242

— IClientFormatterSink 328

— IContextAttribute 187

— IContextProperty 188, 203

— IContributeClientContext-
Sink 196, 201

— IContributeDynamicSink 194

— IContributeEnvoySink 214

— IContributeObjectSink 207,
210— IContributeServerContext-
Sink 197, 201

— IDictionary 151

— IDynamicMessageSink 194

— IDynamicProperty 194

— IFormatter 303

— IJobServer 58, 60, 75, 80,
120

— ILease 40, 115

— ILogicalThreadAffinative 175

— IMessage 48, 153

— IMessageSink 182, 328

— IParamValidator 215, 216,
221

— IPrincipal 103

— ISerializable 30, 279, 280,
287

— ISerializationSurrogate 285

— IServerChannelSink 334

— IServerChannelSink-
Provider 273

— ISponsor 42, 114, 115

— ISurrogateSelector 286

— System.Runtime.Remo-
ting.Channels.IChannel 227

- System.Runtime.Remoting.Channels.IChannelReceiver 227
- System.Runtime.Remoting.Channels.IChannelSender 227
- System.Runtime.Remoting.Channels.IClientChannelSink 233
- System.Runtime.Remoting.Channels.IServerChannelSink 232
- System.Runtime.Remoting.Messages.IMessage 48
- подключения 20
- пользовательский 3, 57, 66
- удаленный доступ 120
- исключение 21, 29, 42
- К**
- канал 46, 49, 85, 86, 87, 89, 95, 181, 225
- FileChannel 236
- HttpChannel 23, 24, 49, 50, 71, 97, 99, 225, 227
- TcpChannel 24, 49, 71, 98, 225
- UDP 235
- URI 226
- именованный 235
- межконтекстный 191
- нестандартный 234
- построение 225
- смена 166
- файловый 235
- каталог виртуальный 94, 99
- класс
 - AccessTimeServerChannelSink 269
 - AccessTimeServerChannelSinkProvider 273
 - Array 306
 - AsyncReplyHelperSink 184, 210
 - ChannelFileData 262
 - ChannelFileTransportReader 265
 - ChannelFileTransportWriter 264
 - ChannelServices 71, 87
 - ContextLogProperty 190
 - EnvoyTerminatorSink 214
 - ExceptionLoggingContextAttribute 205
 - ExceptionLoggingMessageSink 199, 203
 - ExceptionLoggingProperty 203
 - FieldNames 301
 - FileChannel 258
 - FileChannelHelper 260
 - FileClientChannel 237, 239, 260
 - FileClientChannelSink 243
 - FileClientChannelSinkProvider 242
 - FileServerChannel 248, 260
 - FileServerChannelSink 248, 253
 - FooValidator 222
 - FormAddNote 107
 - Formatter 289
 - FormatterServices 288, 289
 - FormCreateJob 83
 - HttpChannel 86, 226
 - HttpClientChannel 226, 227
 - HttpClientTransportSink 226
 - HttpClientTransportSinkProvider 226
 - HttpServerChannel 226, 227, 230, 248
 - HttpServerTransportSink 226, 232, 248
 - JobClient.Form1 132, 145

- JobEventArgs 58, 59, 60, 75
- JobServerImpl 58
- JobEventRepeater 117, 118
- JobNotes 104, 105, 106, 109, 111, 112, 114, 116, 141
- JobServerImpl 60, 62, 64, 75, 76, 80, 84, 90, 96, 113, 114, 116, 118, 120, 130
- JobServerLib 96
- JobServerLib.JobServerImpl 84
- LoadBalancingManager 179
- LoadBalancingProxy 176
- Message 333
- MethodCall 333
- MyClientFormatterSink 328
- MyFormatter 305
- MyFormatterClientSinkProvider 333
- MyFormatterServerSinkProvider 339
- MyMessage 332
- MyRemoteObject 152
- MyServerFormatterSink 334
- ObjectIDGenerator 289, 292
- ObjectManager 289, 294, 296, 318
- PassThruMessageSink 183, 186
- ProxyAttribute 160, 161
- RealProxy 158, 161
- RemotingConfiguration 68, 86, 87, 88, 110, 111
- RemotingSurrogateSelector 332
- SomeClass 277
- SomeClass2 296
- SomeClass3 282, 296
- System.DelegateSerializationHolder 132
- System.EnterpriseServices.ServicedComponent 67
- System.MarshalByRefObject 64, 67, 112
- System.Runtime.Remoting.Channel.ServerChainSinkStack 257
- System.Runtime.Remoting.Channels.CommonTransportKeys 233
- System.Runtime.Remoting.Messaging.MethodReturnMessageWrapper 200
- System.Runtime.Remoting.WellKnownClientTypeEntry 88
- System.Runtime.Serialization.Formatter 297
- System.Runtime.Serialization.SurrogateSelector 286
- System.Security.Permissions.PrincipalPermissionAttribute 102
- TimeStamperSurrogate 286
- TraceMessageSink 208, 212
- XmlSerializer 125
- время жизни 112
- дублер 119
- компилируемый 97
- клиент
 - обратный вызов 93
 - тонкий 11
- клиентская контекстная цепочка 195
- код
 - возврата 21
 - защита доступа 103
 - разделение 3
 - управляемый 28
- коммуникации между процессами см. IPC
- компонент 9
- конечная точка 17

контекст 28, 43, 181, 186, 191
 — атрибут 186, 187
 — **вызова** 151, 174
 — по умолчанию 28
 — с регистрацией исключений 198
 — свойство 186-188
 — синхронизации 28
 — в потоке сообщении 179
 — на сервере 176
 контроль типов 28
 конфигурационный файл 20, 25, 73, 86, 88, 95, 99, 110, 112, 172, 274
 — Machine.config 21
 — Web.config 21
 приложения 21
 криптография 15

Λ

лицензия 25, 39, 40
 — инициализация 114
 — окончание действия 41
 — первоначальный срок 41
 — продление 41, 42
 — спонсор 43
 — таблица 42

М

маршalling 8, 43
 масштабируемость 171
 метаданные 22, 45, 72, 83, 116
 метод
 — Activator.Activate 195, 197
 — Activator.GetObject 88, 121
 — add_Delegate 136
 — add_JobEvent 130, 131
 — AddClientProviderToChain 240
 — AddJobToListView 78

— AddNote 105
 — AddValue 279
 — AsyncProcessMessage 184, 209, 210
 — AsyncProcessReplyMessage 186
 — AsyncProcessRequest 234, 331
 — AsyncProcessResponse 232, 234
 — Bar 222
 — buttonCreate_Click 82
 — C.Foo 212
 — ChannelServices.CreateServerChannelSinkChain 252
 — ChannelServices.RegisterChannel 71
 — Configure 68
 — Context.RegisterDynamicProperty 194
 — Convert.ChangeType 325
 — ConvertListViewItemToJobInfo 81
 — ConvertMyMessagePropertiesToMethodCall 332
 — ConvertMyMessagePropertiesToMethodResponse 332
 — CreateInstance 160
 — CreateJob 61, 62, 139
 — CreateMessageSink 230, 233, 241
 — CreateObjRef 64
 — CreateSink 233, 237, 268, 274
 — Demand 102
 — Deserialize 304
 — DoFixups 294, 297
 — ExceptionLoggingMessageSink.AsyncProcessReplyMessage 201
 — FileClientChannelSink.AsyncHandler 247, 262

- `FileClientChannelSink.AsyncProcessRequest` 262
- `FileClientChannelSink.IsOneWayMethod` 247
- `FileClientChannelSink.ProcessMessage` 262
- `FileServerChannelSink.ProcessMessage` 262
- `Form1.MyJobEventHandler` 118
- `Formatter.WriteMember` 310
- `FormatterServices.GetUninitialized` 296
- `FormatterServices.GetUninitializedObject` 326
- `Freeze` 188
- `GetChannel` 71
- `GetChannelData` 252, 268
- `GetData` 175
- `GetEnumerator` 79
- `GetIJobServer` 75, 76, 84, 86, 89, 119
- `GetJobs` 61, 93
- `GetLifetimeService` 64
- `GetMessageSinks` 170
- `GetNext` 298
- `GetNotes` 106
- `GetObject` 294
- `GetObjectData` 279, 285, 290, 291
- `GetObjectSink` 211
- `GetPropertiesForNewContext` 187, 205, 211
- `GetRegisteredActivatedClientTypes` 69
- `GetRegisteredActivatedServiceTypes` 69
- `GetRegisteredWellKnownClientTypes` 69
- `GetRegisteredWellKnownServiceTypes` 69
- `GetRequestStream` 234, 331
- `GetResponseStream` 232
- `GetSelectedJob` 80
- `GetSerializableMembers` 290
- `GetServerContextSink` 203
- `GetUninitializedObject` 290
- `GetUrlsForObject` 72
- `GetUrlsForUri` 230
- `GetValue` 280
- `IClientChannelSink.AsyncProcessResponse` 331
- `IContextAttribute.IsContextOK` 187, 188
- `IContextProperty.Freeze` 187
- `IContextProperty.IsNewContextOK` 187
- `IContributeClientContextSink.GetClientContextSink` 203
- `IContributeServerContextSink.GetServerContextSink` 203
- `IDynamicMessageSink.ProcessMessageStart` 194
- `IFormatter.Deserialize` 318
- `IFormatter.Serialize` 305
- `IJobServer.addJobEvent` 131
- `IJobServer.CreateJob` 59, 82, 139
- `IJobServer.GetJobs` 59, 137
- `IJobServer.UpdateJobState` 59
- `ILease.Register` 115
- `IMessageSink.AsyncProcessMessage` 184, 200, 331
- `IMessageSink.SyncProcessMessage` 184, 199
- `Init` 249
- `InitializeLifetimeService` 64, 112, 113, 114, 116
- `InspectReturnMessageAndLogException` 200
- `Invoke` 78, 94, 161, 163, 167
- `IParamValidator.Validate` 215

- IsActivationAllowed 69
- IsContextOK 187, 205
- ISerializable.GetObjectData 281, 297
- IServerChannelSink.ProcessMessage 337
- IsInRole 103
- IsMarkedSerializable 309
- IsNewContextOK 189
- ISponsor.Renewal 113, 115, 116
- IsRemotelyActivatedClientType 69
- IsWellKnownClientType 70
- JobEventRepeater.RepeatEventHandler 117
- Listen 232
- ListenAndProcessMessage 252
- ListView.Invoke 80
- LogMessage 207
- Main 67, 111
- MyFormatter.Deserialize 318
- MyFormatter.ReadObjectMembers 332
- MyFormatter.WriteString 311
- MyJobEventHandler 77, 132
- MyMethod 126
- MySecureMethod 102
- NotifyClients 61, 62
- OnDeserializationCallback 285
- Parse 227, 237
- PopulateChannelData 251
- PopulateObjectMembers 290, 291
- PrivateInvoke 157
- ProcessMessage 232, 234, 246, 331
- ProxyAttribute.CreateInstance 162
- Push 257
- RaiseDeserializationEvent 294
- ReadArray 325, 326
- ReadArrayElement 327
- ReadMember 322, 323
- ReadMemberValue 323, 325
- ReadObject 319, 320
- ReadObjectMembers 321
- ReadSerializableMembers 321, 322
- ReadSerializationInfo 321, 322
- RealProxy.Invoke 187
- RecordArrayElementFixup 294
- RecordDelayedFixup 294
- RecordFixup 294
- RegisterActivatedClientType 70, 110
- RegisterActivatedServiceType 70, 111
- RegisterChannel 72, 37
- RegisterObject 294
- RegisterWellKnownClientType 70, 37, 88
- RegisterWellKnownServiceType 70
- RemotingServices.GetEnvelopeChainForProxy 214
- remove JobEvent 130, 145
- RepeatEventHandler 117
- Schedule 298
- Serialize 304
- ServerChainSinkStack.Pop 257
- ServiceRequest 232
- SetData 175
- SetObjectData 285
- SomeObject.Foo 222
- StartListening 230, 231, 251, 252
- StopListening 230, 253
- SyncProcessMessage 184, 208, 217

- System.Runtime.Remoting-
Services.IsOneWay 247
- TimeStamperSurrogate.Get-
ObjectData 287
- TimeStamperSurrogate.Set-
ObjectData 287
- Trace.WriteLine 209
- TraceMessage 209
- TraceMessageSink.Async-
ProcessReplyMessage 210
- UnregisterChannel 72
- UpdateJobInListView 80
- UpdateJobState 61, 63, 140
- Write 264
- WriteArray 313
- WriteMember 298
- WriteObject 306
- WriteObjectMembers 308,
321
- WriteObjectRef 298, 310,
313, 314
- WriteSerializableMembers
309
- WriteSerializationInfo 308
- WriteSOAPMessageToFile
261
- WriteString 311
- WriteValueType 298, 313,
315, 316
- WriteXXXX 299, 316
- О**
- общая система типов *см.* CTS
- общеконтекстная цепочка
приемников серверного
контекста 191
- общезыковая исполняющая
среда *см.* CLR
- объект
 - AccessTimeServerChannel-
Sink 272
 - CallContext 174
 - CallContextData 179
 - ChannelFileData 264
 - FileClientChannel 258
 - IDictionary 231, 239
 - IMessageSink 241
 - IMethodReturnMessage 171
 - IServerChannelSink 257
 - IServerChannelSinkStack 338
 - ITransportHeader 247
 - JobLib 166
 - JobNotes 115
 - JobServerImpl 64, 88, 98, 167
 - Principal 101
 - PrincipalPermission 102
 - RealProxy 157, 158
 - ReturnMessage 184
 - ServerChainSinkStack 257
 - ServerProcessing 257
 - Stream 233, 247
 - System.ObjRef 143
 - System.Security.Permis-
sions.PrincipalPermission 102
 - TransparentProxy 156
 - URI 226
 - активизация
 - клиентская 68, 104, 111
 - режим SingleCall 68
 - режим Singleton 68
 - серверная 68
 - время жизни 17, 25, 39, 40
 - граф 282, 291
 - десериализация 284
 - дистанцируемый 29, 41
 - инкапсуляция 20
 - каталогизация 293
 - клиентский 46
 - контекстно-связанный 186
 - локальный 13, 48
 - массив 78
 - недистанцируемый 29
 - опрос 18
 - подсчет ссылок 18

- приемник 50
- транспортный S3
- форматирующий 52
- пул 25, 67
- распределенный 8, 17
- серверный 17, 89
- сериализация 64, 276
- сообщение 46, 48, 49
- состояние 13
- ссылка 44
- удаленный 14, 37, 39, 41, 42, 45, 46, 48, 58, 68, 85, 89, 101, 166
- оснастка 21
- отражение 157

П

- повторитель 117
- подсчет ссылок 25
- порт ожидания 87
- приемник 50
 - динамический 193
 - каналный 50, 225, 266
 - контекстный 193
 - контроля времени доступа 268
 - сообщений 181, 186
 - стекопостроителя 191
- форматировщика 327
 - клиентский 328
 - серверный 328, 334
- приложение
 - .NET Remoting 19
 - DCOM 19
 - JobClient 67, 74, 90, 99, 104, 106, 130
 - JobServer 57, 60, 63, 67, 72, 90, 130
 - JobServer.exe 111
 - аутентификация 94
 - клиентское 57, 85, 106
 - консольное 65, 67

- прототип /6
- серверное 57
- удаленное взаимодействие 72
- провайдер
 - форматировщика 230
- программирование
 - декларативное 102
 - императивное 102
- прокси 44, 84, 156
 - закрытый 98
 - открытый 98
 - прозрачный 46
 - реальный 46, 47, 214
- прямой опрос
 - пользователя 103

Р

- рабочая группа 5
- распределение нагрузки 151
- распределенная архитектура
 - клиент-серверная 3
 - многоуровневая 4
 - модульное программирование 3
 - одноранговая 5
- распределенное приложение 2
 - Xsoru-установка 20
- администрирование 11
 - защита 15
 - конфигурирование 17
 - масштабируемость 11
 - отказоустойчивость 10
 - производительность 12

С

- сборка 22
 - имя 45
 - открытый ключ 45
 - с метаданными 96
 - с реализацией 96
- сборщик мусора 36, 42

СВОЙСТВО

- ApplicationId 68
- ApplicationName 68
- AssemblyName 279
- Binder 303
- ChannelData 230
- ChannelName 227, 237
- ChannelPriority 227, 237
- ClientSponsor.RenewalTime 43
- Context 280, 303
- CurrentLeaseTime 41
- Description 107
- EnvoyInfo 217
- Exception 167, 171
- ExceptionLoggingContextProperty 205
- ExceptionLoggingProperty 204
- FullTypeName 279
- InitialLeaseTime 41, 113, 116
- MemberCount 279
- Name 188
- Next 233, 268
- NextChannelSink 232, 234
- ParameterValidatorProperty 220
- ProcessId 70
- Properties 48, 151
- Reason 77
- ref 89
- RegisteredChannels 72
- RenewOnCallTime 41, 113
- SponsorshipTimeout 41, 113
- SurrogateSelector 304
- useDefaultCredentials 100
- селектор суррогатов 285, 286, 288
- сервер
 - общеизвестный 5
 - поиска 172
 - удаленный 24
 - централизованный 6

- серверная контекстная цепочка
 - приемников 196
- серверная объектная цепочка
 - приемников 205
- сериализация 30, 44, 48, 52, 276
- служба Windows 65, 66
- сокет 7
- сообщение
 - вызова конструктора 152
 - вызова метода 153
 - обработка
 - асинхронная 184
 - синхронная 184
- состояние сессии 57
- спонсор 25, 39, 41, 42, 43
 - лицензии 112
- регистрация 115
- суррогат 285

Т

- тип 29
 - активизация 33
 - клиентская 33, 37
 - режим SingleCall 36
 - режим Singleton 34, 39
 - серверная 33, 34
 - дистанцируемый 29, 64
 - контекстно-связанный 29, 31
 - передаваемый по значению 29, 33
 - передаваемый по ссылке 29, 30, 33
 - недистанцируемый 29
 - общеизвестный 34
 - полное квалифицированное имя 45
 - публикация 34
 - регистрация 34, 38
- транзакция распределенная 25, 67
- туннелирование 16

У

удаленный вызов процедур *см.* RPC

унифицированный идентификатор ресурса *см.* UR1

управление доступом 15

уровень

— клиента 5

— представления 3

— сервера 5

Ф

форматировщик 52, 181, 251, 277

— SOAP 23, 24, 130

— двоичный 24

— сериализации 288

Ц

цепочка

— агентских приемников 191, 212

— приемников клиентского контекста 191

— приемников серверного объекта 191

Ш

широковещательное сообщение 5

Об авторах

Скотт Маклин

Скотт Маклин начинал программировать еще на компьютерах Atari 400. Изучив Atari BASIC, он освоил ассемблер 6502. Через несколько лет он ушел служить в ВМС США, где оттрубил 7 лет на атомной подводной лодке. Демобилизовавшись, Скотт вернулся к учебе и защитил степень бакалавра информатики в университете штата Джорджия.



Сейчас он программист в XcelleNet, Inc., специализируется на архитектурах серверных приложений масштаба предприятия и разработке распределенных систем. Он построил множество приложений с применением многопоточности, сокетов, портов завершения ввода/вывода, COM, ATL и .NET. Он опубликовал статью по .NET Remoting для *.NET Magazine Online*, а также является соавтором книги *Visual C++ + .NET: A Primer for .NET Developers*, выпущенной издательством WROX Press, Ltd. Скотт — соучредитель и автор www.thinkdotnet.com, интерактивного ресурса для разработчиков, ориентированных на .NET.

Джеймс Нафтел

Джеймс Нафтел познакомился с компьютерами в Allied Collection and Credit Bureau, Inc., неподалеку от Атланты, шт. Джорджия. Поначалу он даже не знал, что такое приглашение на ввод команды в DOS. Владелец фирмы, Рекс Геллогли, заставлял его помногу работать с компьютерами, и постепенно Джеймс заинтересовался ими. В то время он слушал в университете штата Джорджия курс по бизнесу, но возникшая любовь к компьютерам заставила его переориентироваться на информатику.

Сменив специализацию, он заодно женился на своей подружке Эприл. Они переехали в Афины, шт. Джорджия и поступили в местный университет. Параллельно с учебой Джеймс служил в консалтинговой компании PICS, где разработал складскую программу на Microsoft Visual FoxPro. Вернувшись в район Атланты, Джордж получил степень бакалавра информатики в университете штата Джорджия.



Закончив учебу, Джеймс поступил в XcelleNet, Inc., где работает и сейчас в должности ведущего программиста. Он занимался созданием приложений для баз данных масштаба предприятия и распределенных систем и сейчас руководит группой, которая занимается технологией синхронизации баз данных. Джеймс живет в пригороде Атланты, шт. Джорджия, со своей женой, двумя дочерьми и двумя собаками. Он соучредитель www.thinkdotnet.com и автор многих статей на этом сайте, кроме того, он пишет статьи о надстройках для Microsoft Visual Studio .NET в *Windows Developer Journal*. Его страсть — ваять всякую всячину на языках программирования, особенно на C++ и C#.

Ким Уильямс

Ким Уильямс начал свою профессиональную карьеру как джазовый пианист. Через несколько лет он, решив превратить свое увлечение программированием из хобби в профессию, всерьез занялся информатикой. Закончив обучение, он писал антивирусные программы и дизассемблировал вирусы, а также конструировал распределенные приложения, связанные с защитой.

Перейдя в XcelleNet, Inc., Ким работал в качестве ведущего разработчика с различными технологиями, такими, как Java RMI, DCOM, ATL, ASP. Сейчас он руководит разработкой масштабных приложений, связанных с ASP.NET и Web-сервисами. Он соучредитель www.thinkdotnet.com и автор статей на этом сайте. Живет в Атланте, шт. Джорджия, с женой Пэтти и сыном Шоном и до сих пор находит время для игры на фортепиано.



Маклин Скотт, Нафтел Джеймс, Уильямс Ким

Microsoft .NET Remoting

Перевод с английского под общей редакцией **В. Г. Вшивцева**

Переводчик **Д. Г. Новоселов**

Редактор **Ю. П. Леонова**

Технический редактор **Л. А. Панчук**

Компьютерная верстка **В. Б. Хильченко**

Дизайнер обложки **Е. В. Козлова**

Оригинал-макет выполнен с использованием
издательской системы Adobe PageMaker 6.0

TypeMarketFontLibrary
легальный пользователь

ПОЛЬЗОВАТЕЛЬ
Para(-)Type
G A T I U S E

Главный редактор **А. И. Козлов**

Подготовлено к печати издательством «Русская Редакция»

121087, Москва, ул. Заречная, д.9

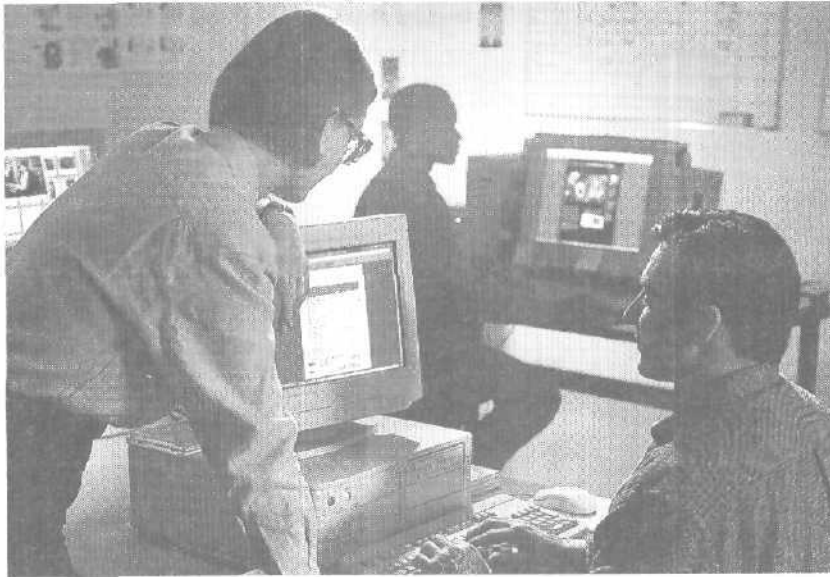
тел.: (095) 142-0571, тел./факс: (095) 145-4519

e-mail: info@rusedit.ru, <http://www.rusedit.ru>

РУССКАЯ РЕДАКЦИЯ

Подписано в печать 27-02.02 г. Тираж 2000 экз.
Формат 60x90/16. Физ. п. л. 24

Отпечатано в ОАО «Типография «Новости»
107105, Москва, ул. Фр. Энгельса, 46



Учебный центр SoftLine

Ваш курс начинается завтра!

Подготовка сертифицированных инженеров
и администраторов Microsoft

Авторизованные и авторские курсы по:

- Windows 2000 / XP
- Sun Solaris 8
- * Visual Studio .NET

* Электронной коммерции

да Безопасности информационных систем

и еще более 40 курсов по самым современным компьютерным технологиям.

Дневные и вечерние занятия.

Опытные преподаватели.

Индивидуальные консультации.

softline
education

Microsoft
CERTIFIED
Technical Education
Center

Учебный центр SoftLine

119991 г. Москва, ул. Губкина, д. 8

тел.: (095) 232 00 23

e-mail: educ@softline.ru

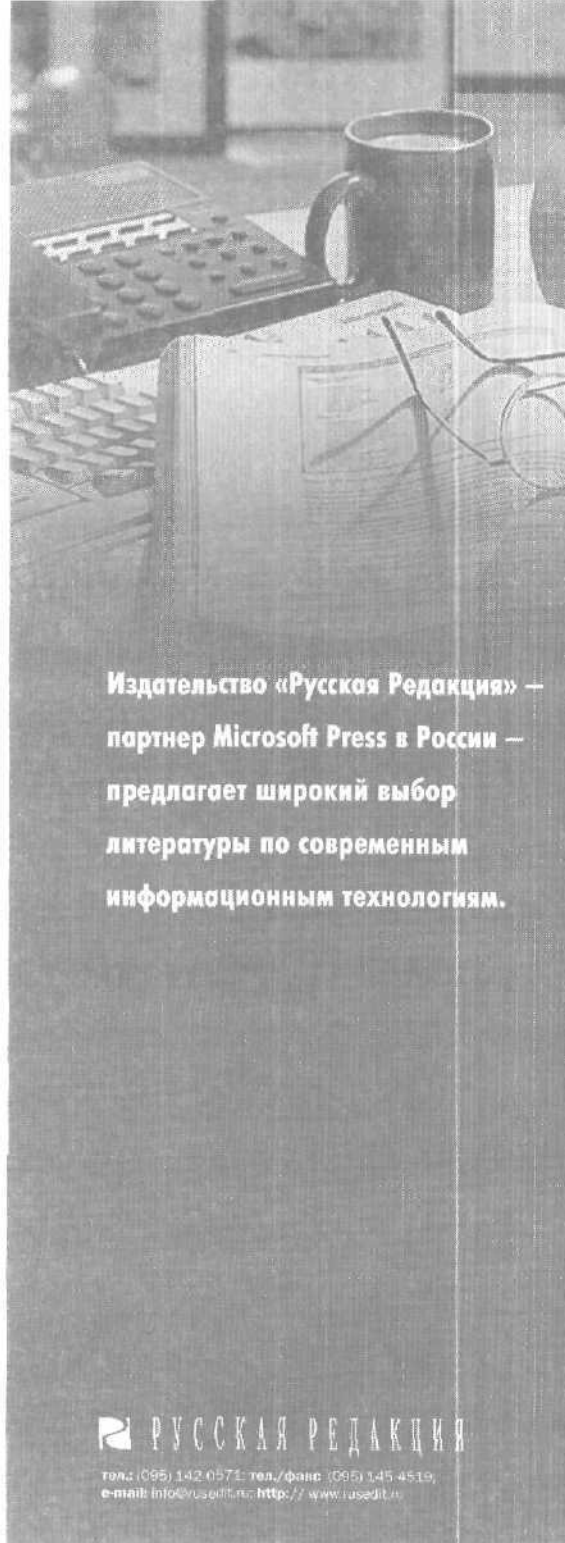
<http://education.softline.ru>

ПРОГРАММНОЕ ОБЕСПЕЧЕНИЕ

softline

ЛИЦЕНЗИРОВАНИЕ • ОБУЧЕНИЕ • КОНСУЛЬТИНГ

www.softline.ru • 232 0023 • info@softline.ru



**Издательство «Русская Редакция» —
партнер Microsoft Press в России —
предлагает широкий выбор
литературы по современным
информационным технологиям.**

 **РУССКАЯ РЕДАКЦИЯ**

телеф. (095) 142-0571; тел./факс (095) 145-4519;
e-mail: info@rusedit.ru; <http://www.rusedit.ru>

Наши книги Вы можете приобрести

• в Москве:

Специализированный магазин
«Компьютерная и деловая книга»
Ленинский проспект, строение 38,
тел.: (095) 778-7269

«Библио-Глобус» ул. Мясницкая, 6.
тел.: (095) 928-3567

«Московский дом книги» ул. Новый Арбат, 8,
тел.: (095) 290-4507

«Дом технической книги» Ленинский пр-т. 40.
тел.: (095) 137-6019

«Молодая гвардия» ул. Большая Полянка, 26,
тел.: (095) 238-5001

«Дом книги на Соколе» Ленинградский пр-т.
7В тел. (095) 152-4511

«Дом книги на Войковской» Ленинградское ш.,
13, стр. 1, тел.: (095) 150-6917

«Мир печати» ул. 2-я Тверская-Ямская, 54,
тел.: (095) 978-5047

Торговый дом книги «Москва» ул. Тверская 8,
тел.: (095) 229-6483

• в Санкт-Петербурге:

СПб Дом книги, Невский пр-т., 28
тел.: (812) 318-6402

СПб Дом военной книги, Невский пр-т., 20
тел.: (812) 312-0563, 314-7184

Магазин «Подписные издания...»
Литейный пр-т., 57, тел.: (812) 273-5053

Магазин «Техническая книга» ул. Пушкинская,
2, тел.: (812) 164-6565, 164-1413

Магазин «Буквоед», Невский пр-т., 13,
тел.: (812) 312-6734

ЗАО «Торговый Дом «Диалект»,
тел.: (812) 247-1483

Оптово-розничный магазин «Наука и техника»,
тел.: (812) 567-7025

• в Екатеринбурге:

Магазин «Дом книги»,
ул. Валека, 12,
тел.: (3432) 59-4200

• в Великом Новгороде:

«Наука и техника»,
ул. Большая Санкт-Петербургская, 44,
Дворец Молодежи, 2-й этаж

• в Новосибирске:

ООО «Топ-книга», тел. (3832) 36-1026

• в Алма-Ате (Казахстан):

ЧП Болат Амреев,
моб. тел.: 8-327-908-28-57, (3272) 76-1404

• в Киеве (Украина):

ООО Издательство «Ирина-Пресс»,
тел.: (+1038044) 269-0423

«Техническая книга на Петровке»,
тел.: (+1038044) 268-5346

Интернет-магазин

ITbook.ru

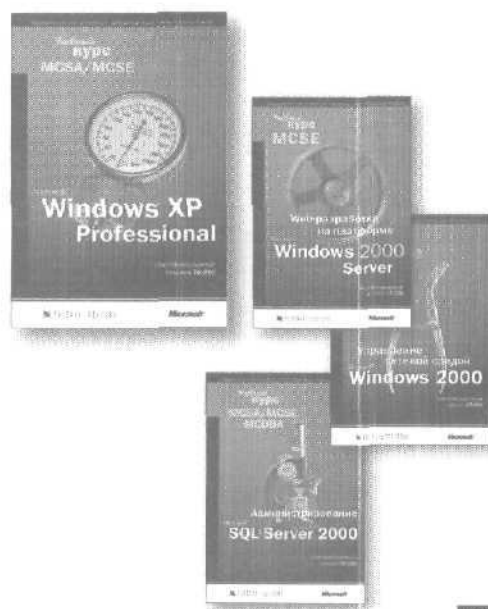
книги и журналы
для профессионалов

The screenshot displays the ITbook.ru website interface. At the top, there's a navigation bar with links like 'Главная', 'Вход', 'Интернет', 'Справка', 'Оплата', 'Назад', 'Попросить', 'Добавить', 'Избранное', 'Избранное', 'Справка', 'Справка'. Below this is a search bar and a list of categories: 'Книги по программированию', 'Информационным технологиям', 'Русская Редакция', 'Издания компьютерной литературы'. A sidebar on the left lists various categories: 'MSDN Magazine/Русская Редакция', 'Книги по программированию', 'Книги на английском языке', 'Интернет и сети', 'Базы данных', 'Программирование', 'Операционные системы', 'Информационная безопасность', 'IT-журналы', 'Средства разработки'. The main content area shows a search result for 'MSDN Magazine/Русская Редакция' with a list of items. The first item is 'MSDN Magazine/Русская Редакция. Спецвыпуск №1. Русская Редакция, 96 стр., с ил. Серия: ISBN: 1662 6841. Издательская цена: 180 руб. (без учета доставки). Оценка читателей: 4.5 из 5 (всего оценок: 22 средняя оценка: 2.9). Подарок! Предназначено профессиональным программистам, разработчикам и IT-специалистам высокого уровня в области системного, прикладного и интернет-программирования: ПИИРСИБЕЕ...'. The second item is 'MSDN Magazine/Русская Редакция. Спецвыпуск №2. Русская Редакция, 96 стр., с ил. Серия: ISBN: 1662 6842...'. At the bottom, there's a contact information block: 'тел.: (095) 145-4519', 'e-mail: sr@rusedit.ru', 'http://www.ITbook.ru'.

тел.: (095) 145-4519
e-mail: sr@rusedit.ru
http://www.ITbook.ru

Официальные учебные пособия Microsoft

для самостоятельной подготовки



Каждая книга серии
«Учебный курс» («Training») **Microsoft Press** — исчерпывающая
справочная информация
от первоисточника, занятия,
упражнения для самостоятельной
работы, видеоролики
и вопросы для самопроверки
и закрепления знаний.
Издания подготовлены с учетом
современных требований
корпорации Microsoft к уровню
профессиональных знаний
специалистов по конкретной теме.

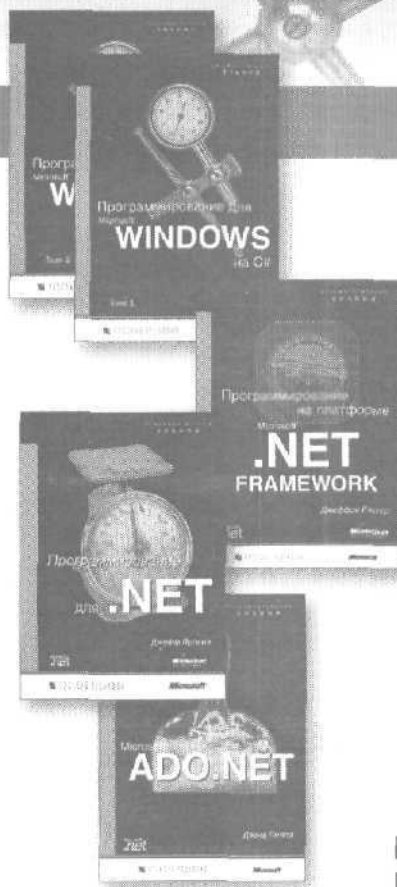
издательство компьютерной литературы

 РУССКАЯ РЕДАКЦИЯ

тел.: (095) 142-0571 тел./факс (095) 145-4519
e-mail: info@rusedit.ru; <http://www.rusedit.ru>

Фундаментальные знания

новая серия книг



Маститые авторы и ведущие специалисты Microsoft в области разработки — Чарльз Петцольд, Джеффри Рихтер, Джеф Проксиз, Дэвид Селпа и др. — познакомят вас с флагманской платформой Microsoft .NET.

Каждая книга серии — это полное, обстоятельное руководство по .NET Framework, .NET Enterprise Servers, Microsoft Visual Studio .NET и др. Основа серии «Фундаментальные знания» — книги Microsoft Press со статусом **Core Reference** — ведущие издания от разработчиков для разработчиков.

издательство компьютерной литературы

РУССКАЯ РЕДАКЦИЯ

тел.: (095) 142-0571; тел./факс (095) 145-4519;
e-mail: info@rusedit.ru; <http://www.rusedit.ru>

ЕЖЕМЕСЯЧНЫЙ
НАУЧНО-ПОПУЛЯРНЫЙ
КОМПЬЮТЕРНЫЙ ЖУРНАЛ



e-mail:
info@hardnsoft.ru

HARD'n'SOFT

МАКСИМАЛЬНО ПОЛНАЯ
И ОБЪЕКТИВНАЯ ИНФОРМАЦИЯ
ДЛЯ ЧИТАТЕЛЕЙ, УВЛЕЧЕННЫХ
КОМПЬЮТЕРНОЙ ТЕХНИКОЙ

В каждом из номеров нашего журнала:

- новости компьютерной индустрии
- подробности о современных и перспективных технологиях
- тесты и обзоры аппаратных и программных продуктов
- интернет и мультимедиа, игры и прикладные программы
- консультации экспертов, встречи с интересными людьми
- CD-приложение с полезными утилитами



WWW.HARDNSOFT.RU

НАШИ ИНДЕКСЫ:

Hard'n'Soft • 73140, Hard'n'Soft + CD - 26067

п р о ф е с с и о н а л ь н ы й ж у р н а л

ПРОГРАММИСТ

Журнал освещает вопросы разработки программного обеспечения. Наши авторы – профессиональные программисты

Статьи на самые разные «программерские» темы любого уровня сложности

Материалы о самых современных технологиях и средствах разработки. Статьи о принципах и методах, теории и практике программирования.

- Постоянные рубрики:
- Система
- Технологии
- Средства разработки
- Мульты
- Алгоритмы
- Жизнь
- новости, обзоры, информация

Журнал «Программист» –

НАСТОЛЬНЫЙ

ЖУРНАЛ

РАЗРАБОТЧИКА



ПОДПИСКА: КАТЕГОРИЯ «ПРОФЕССИОНАЛ» • ИНДЕКС 60467 / КАТАЛОГ ПРЕССА РСО 4114 • 1-й ПЕРИОД 1977/78

Создавайте будущее с нами



Журнал
для разработчиков
программного
обеспечения

www.microsoft.com/rus/msdn/magazine

Подписной индекс по каталогу Агентства «Роспечать» — 81240
Подписной индекс по каталогу Агентства «Книга-сервис» — 43449
Интернет-магазин издательства <http://www.ITbook.ru>, тел.: (095) 142-0571

Microsoft® .NET REMOTING

Создание масштабируемых распределенных Интернет-приложений

Microsoft .NET Framework предлагает гибкую модель для создания отказоустойчивых, масштабируемых, защищенных распределенных приложений, основанных на взаимодействии объектов, — .NET Remoting. О возможностях этой передовой технологии в данном издании рассказывают авторитетные эксперты по .NET. Вы узнаете, как использовать беспрецедентные возможности .NET Remoting для построения быстрых и простых в сопровождении и администрировании распределенных приложений для работы в Интернете. Здесь же вы найдете конкретные примеры и полезные советы по применению .NET Remoting для усовершенствования и настройки распределенных приложений.

Основные темы книги

- Основы распределенных приложений.
- * Архитектура .NET Remoting и возможности ее расширения.
- Создание распределенных приложений на основе .NET Remoting.
- » Протокол SOAP и обмен сообщениями.
- * Сообщения и прокси.
- * Приемники сообщений и контексты.
- * Каналы и каналные приемники.
- » Форматировщики сериализации и приемники форматирования.

Примеры программ на C#
опубликованы на Web-сайте:

www.microsoft.com/mspress/books/6172.asp

Об авторах:

Скотт **Маклин** работает программистом в XcelleNet, Inc. и специализируется на архитектурах серверных приложений масштаба предприятия и разработке распределенных систем.

Его перу принадлежат книга о Microsoft Visual C++, а также статьи по .NET Remoting для .NET Magazine Online.

Джеймс Нафтел — ведущий программист XcelleNet, Inc., занимается базами данных.

Он пишет статьи о надстройках для Microsoft Visual Studio .NET в Windows Developer Journal.

Ким **Уильямс** работает ведущим программистом в XcelleNet, Inc., и возглавляет команду Web-разработчиков, создающих приложения на базе Microsoft ASP.NET.

ISBN 5-7502-0229-1



9 785750 202294

Web-узел издательства: www.rusedit.ru
Интернет-магазин: www.ITbook.ru