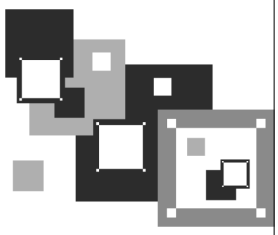
Глава 4



WCF

Microsoft Windows Communication Framework (WCF) является достаточно сложной платформой. Причиной подобной сложности служит тот факт, что WCF на абстрактном уровне является платформой обмена сообщениями, которая должна оставаться релевантной и годной к использованию относительно постоянно эволюционирующих стандартов индустрии. На фазе проектирования WCF доминирующими в будущем структурами и протоколами сообщений считались SOAP и WS-*. Вряд ли кто-либо из архитекторов WCF знал, что JavaScript Object Notation (JSON) станет такой значимой, какой она является на сегодняшний день. Конечно же, они понимали, что WCF обязана легко принимать и адаптировать структуры и транспорты сообщений, которые появляются как грибы. В итоге Microsoft спроектировала WCF существенно расширяемой и адаптируемой, чтобы удовлетворить требованиям сегодняшнего дня и непредсказуемым требованиям дня завтрашнего. Результатом этих усилий стала многогранная платформа, которую легко использовать, но достаточно сложно понять.

Тот, кто уже набил руку в создании мощных платформ, подтвердит, что проектирование, создание, тестирование и сопровождение являются весьма утомительными задачами. Имея опыт проектирования, консультирования и создания нескольких платформ, я понимаю, насколько это может быть трудно. При создании платформы основным правилом проектирования должен быть афоризм, приписываемый Алану Кею (Alan Kay): "Простые вещи должны быть простыми, сложные вещи должны быть возможными". Когда я смотрю на WCF в том виде, в каком она существует сегодня, мне кажется, что Microsoft успешно реализовала этот афоризм, и даже пошла дальше, сделав многие сложные "вещи" простыми. Этим я не хочу сказать, что считаю WCF совершенной и не содержащей ошибок, но данный продукт в целом хорошо продуман и умело спроектирован.

Одним из основных требований WCF является предоставление разработчику объектной модели, которая не противоречит всем транспортам и протоколам. Например, архитекторы команды WCF хотели, чтобы код отправки сообщения посредством транспорта TCP/IP был таким же, как код отправки сообщения посредством транспорта MSMQ. Подобные возможности обладают несколькими преимуществами. Во-первых, разработчикам не нужно изучать все разнообразие объектных моделей всевозможных транспортов и протоколов. В результате разработчики, понимающие объектную модель и модель исполнения, могут встроить в свои приложения поддержку различных транспортов и протоколов. Во-вторых, поскольку инфраструктура WCF позволяет включать новые транспорты, протоколы и функциональность, то разработчикам не нужно изучать способы встраивания новых возможностей в свои приложения. Вместо этого, они могут полагаться на уже реализованные в платформе парадигмы.

Результатом этих требований стала архитектура WCF, состоящая из множества перемешанных уровней. С течением времени я понял, что полное понимание какого-либо отдельно взятого уровня инфраструктуры WCF требует для начала неглубокого погружения в каждый из уровней. Целью данной главы является знакомство с основными уровнями приложений WCF и создание контекста для материала всей остальной книги, где будут подробно рассмотрены многие из этих уровней.

WCF: краткое руководство

В этом разделе мы отдадим дань богам теории вычислительных машин и систем, создав приложение HelloWCF. Затем мы проанализируем все части созданного приложения. Чтобы максимально упростить пример, мы объединим отправителя и получателя в единое консольное приложение. Давайте начнем с создания инфраструктуры для консольного приложения:

```
// File: HelloWCFApp.cs
using System;
sealed class HelloWCF {
  static void Main(){
  }
}
```

Определение сервисного контракта

Первым специфичным для WCF шагом на пути построения нашего приложения HelloWCF является создание сервисного контракта. Контракты детально рассмотрены в *главе 9*, а здесь достаточно сказать, что контракты — это первоначальный способ для отображения формы нашего приложения обмена сообщениями. Под *формой* я подразумеваю выполняемые нашим сервисом операции, производимые и потребляемые этими операциями схемы сообщений, а также образцы Message Exchange Patterns (MEPs), которые каждая операция выполняет. Коротко говоря, контракты определяют, что наше приложение обмена сообщениями производит и потребляет. Большинство контрактов — это определения типов, аннотированные атрибутами, заданными API в WCF. В нашем примере сервисный контракт — это интерфейс, аннотированный атрибутами `System.ServiceModel.ServiceContractAttribute` и `System.ServiceModel.OperationContractAttribute`, как показано далее:

```
// File: HelloWCFApp.cs
[ServiceContract]
public interface IHelloWCF {
  [OperationContract]
  void Say(String input);
}
```

На верхнем уровне детализации наш сервисный контракт указывает на то, что наше приложение-получатель содержит одну операцию под названием `Say`, эта операция принимает параметр типа `String` и имеет возвращаемый тип `void`. Приложение-отправитель может использовать этот контракт как средство для создания и отправки сообщений приложению-получателю. Теперь, когда мы определили сервисный контракт, настало время для добавления кода, который определит, где приложение-получатель будет ожидать входящие сообщения, и как это приложение будет осуществлять обмен сообщениями с другими участниками этого процесса.

Определение адреса и компоновки

Для определения местоположения, где наше приложение ожидает входящих сообщений, необходимо использовать тип `System.Uri`, а для определения того, как наше приложение осуществляет обмен сообщениями с другими участниками процесса, необходимо использовать тип `System.ServiceModel.Channels.Binding` или один из его производных типов.

В приведенном далее фрагменте кода показано, как использовать тип `Uri` и тип `Binding` в нашем приложении:

```
// File: HelloWCFApp.cs
static void Main(){
  // определяем, где ожидать сообщения
  Uri address = new Uri("http://localhost:8000/IHelloWCF");
  // определяем, как обмениваться сообщениями
  BasicHttpBinding binding = new BasicHttpBinding();
}
```

Заметьте, что локальная переменная `address` указывает на Uniform Resource Identifier (URI), использующий схему HTTP. Выбор данного адреса заставляет нас воспользоваться компоновкой, также использующей транспорт HTTP. На верхнем уровне детализации компоновка — это первоначальный способ для указания транспорта, хореографии сообщений и кодировщика сообщений, используемых в приложении. Локальная переменная `binding` имеет тип `BasicHttpBinding`. Как можно понять из названия, тип `BasicHttpBinding` создает инфраструктуру сообщений, которая использует транспорт HTTP.

Создание конечных точек и начало ожидания

Далее мы обязаны использовать адрес, компоновку и контракт для создания конечной точки, а затем ожидать в этой конечной точке входящих сообщений. Вообще говоря, приложение-получатель WCF может создавать множество конечных точек и ожидать сообщения в них, и каждая из них требует адреса, компоновки и контракта. Тип `System.ServiceModel.ServiceHost` создает конечные точки и управляет такими частями принимающей инфраструктуры, как организация поточной обработки и жизненный цикл объектов. Приведенный далее фрагмент кода демонстрирует пример использования типа `ServiceHost`, а также того, как добавлять конечную точку и как начать ожидание входящих сообщений:

```
// File: HelloWCFApp.cs
static void Main(){
  // определяем, где ожидать сообщения
  Uri address = new Uri("http://localhost:4000/IHelloWCF");
  // определяем, как обмениваться сообщениями
  BasicHttpBinding binding = new BasicHttpBinding();
```

```
  // создаем экземпляр ServiceHost, опуская тип для создания экземпляра,
  // когда приложение получит сообщение
  ServiceHost svc = new ServiceHost(typeof(HelloWCF));
  // добавляем конечную точку, опускаем адрес, компоновку и контракт
  svc.AddServiceEndpoint(typeof(IHelloWCF), binding, address);
  // начинаем ожидание
  svc.Open();
  // указываем, что приложение-получатель готово, и
  // удерживаем приложение от немедленного завершения
  Console.WriteLine("The HelloWCF receiving application is ready");
  Console.ReadLine();
  // закрываем сервисный узел
  svc.Close();
}
```

Обратите внимание на аргумент в вызове конструктора `ServiceHost`. Конструктор `ServiceHost` несколько раз перегружен, и каждая перегрузка принимает в том или ином виде определение типов того объекта, которому инфраструктура WCF подготавливает к отправке входящее сообщение. Конструктор `ServiceHost`, использованный в приведенном фрагменте кода, указывает на то, что инфраструктура сообщений подготавливает к отправке полученные сообщения экземпляру типа `HelloWCF`.

Также обратите внимание на вызов `svc.AddServiceEndpoint` и `svc.Open` в приведенном выше фрагменте кода. Экземпляр метода `AddServiceEndpoint` типа `ServiceHost` устанавливает состояние объекта `ServiceHost` таким образом, что тот будет ожидать входящих сообщений в соответствии с адресом, компоновкой и параметрами контракта. Важно отметить, что метод `AddServiceEndpoint` не начинает цикл ожидания, а просто изменяет состояние объекта `ServiceHost` (более подробно об этом сказано в *главе 10*). Экземпляр метода `Open` типа `ServiceHost` создает инфраструктуру сообщений, а затем начинает цикл ожидания. Реализация метода `Open` подтверждает состояние объекта `ServiceHost`, создает конечные точки из этого состояния, а затем начинает цикл ожидания для каждой конечной точки.

Отображение полученных сообщений на элемент *HelloWCF*

Наше приложение в текущем виде будет скомпилировано, однако оно выдаст исключение `InvalidOperationException` при попытке создания конечной

точки. Причина достаточно тривиальна: в конструкторе типа `ServiceHost` мы опустили тип `HelloWCF` как аргумент, обозначая таким образом для инфраструктуры сообщений наше намерение подготовить к отправке полученные сообщения объектам `HelloWCF`. Чтобы это произошло, должно существовать отображение, связывающее сообщения с элементами типа `HelloWCF`. Простейший способ создания подобного отображения — это изменить тип `HelloWCF` для реализации сервисного контракта, заданного интерфейсом `IHelloWCF` так, как показано далее:

```
// File: HelloWCFApp.cs
using System;
using System.ServiceModel;
using System.ServiceModel.Channels;
// реализуем сервисный контракт IHelloWCF
sealed class HelloWCF : IHelloWCF {
  // указываем, когда создан объект HelloWCF
  HelloWCF() { Console.WriteLine("HelloWCF object created"); }
  static void Main(){
    // определяем, где ожидать сообщения
    Uri address = new Uri("http://localhost:4000/IHelloWCF");
    // определяем, как обмениваться сообщениями
    BasicHttpBinding binding = new BasicHttpBinding();
    // создаем экземпляр ServiceHost, опуская тип
    // для создания экземпляра, когда приложение получит сообщение
    ServiceHost svc = new ServiceHost(typeof(HelloWCF));
    // добавляем конечную точку, опускаем адрес, компоновку и контракт
    svc.AddServiceEndpoint(typeof(IHelloWCF), binding, address);
    // начинаем ожидание
    svc.Open();
    // указываем, что приложение-получатель готово, и
    // удерживаем приложение от немедленного завершения
    Console.WriteLine("The HelloWCF receiving application is ready");
    // ожидаем входящие сообщения
    Console.ReadLine();
    // закрываем сервисный узел
    svc.Close();
  }
  // полученные сообщения подготавливаются к отправке этому экземпляру
  // метода согласно сервисному контракту
```

```
  public void Say(String input){
    Console.WriteLine("Message received, the body contains: {0}", input);
  }
}
[ServiceContract]
public interface IHelloWCF {
  [OperationContract]
  void Say(String input);
}
```

Подобно изменение определения типа `HelloWCF` заставит инфраструктуру сообщений подготовить к отправке полученные сообщения экземпляру метода `Say` типа `HelloWCF`, выводя, таким образом, простейшее утверждение на консоль.

Компиляция, запуск и проверка получателя

Теперь мы готовы скомпилировать и запустить приложение с помощью следующей командной строки:

```
C:\temp>csc /nologo /r:"c:\WINDOWS\Microsoft.Net\Framework\v3.0\Windows
Communication Foundation\System.ServiceModel.dll" HelloWCFApp.cs
C:\temp>HelloWCFApp.exe
The HelloWCF receiving application is ready
```

С этого момента приложение-получатель пассивно ожидает входящие сообщения. Мы можем удостовериться в том, что приложение действительно ожидает сообщения, запустив netstat.exe, как показано далее:

```
c:\temp>netstat –a –b
TCP    kermit:4000    0.0.0.0:0    LISTENING    1104
[HelloWCFApp.exe]
```

Естественно, в результате этого запуска будет выведено гораздо больше данных, чем показано в этом примере, но среди прочего вы обязательно увидите две строчки, похожие на эти. (Kermit — это имя моего компьютера.)

Отправка сообщения получателю

Отправляющая инфраструктура полагается на адрес, компоновку и контракт точно так же, как и получающая инфраструктура. Обычно использование ад-

реса, компоновки и контракта, совместимых с получателем, находится в зоне ответственности отправителя. Для упрощения нашего приложения отправитель может повторно использовать адрес, компоновку и контракт, задействованные получающей инфраструктурой. Отправляемый код, тем не менее, работает с типами, отличными от используемых получателем. По сути, это имеет смысл, поскольку отправитель и получатель имеют кардинально различные роли в процессе обмена сообщениями. Вместо прямого использования типа `Uri` большинство отправителей полагаются на тип `System.ServiceModel.EndpointAddress` как средство отображения цели сообщения. Как будет показано в *главе 5*, тип `EndpointAddress` в рамках WCF является абстракцией указателя конечной точки WS-Addressing. Более того, отправитель вместо типа `ServiceHost` использует тип `ChannelFactory<T>`, где `T` — это тип сервисного контракта. Тип `ChannelFactory<T>` создает отправляющую инфраструктуру таким же образом, каким тип `ServiceHost` создает получающую инфраструктуру. В приведенном далее фрагменте кода показано, как использовать типы `EndpointAddress` и `ChannelFactory<T>` для создания отправляющей инфраструктуры:

```
// File: HelloWCFApp.cs
using System;
using System.ServiceModel;
using System.ServiceModel.Channels;
// реализуем сервисный контракт IHelloWCF
sealed class HelloWCF : IHelloWCF {
  // указываем, когда создан объект HelloWCF
  HelloWCF() { Console.WriteLine("HelloWCF object created"); }
  static void Main(){
    // определяем, где ожидать сообщения
    Uri address = new Uri("http://localhost:4000/IHelloWCF");
    // определяем, как обмениваться сообщениями
    BasicHttpBinding binding = new BasicHttpBinding();
    // создаем экземпляр ServiceHost, опуская тип
    // для создания экземпляра, когда приложение получит сообщение
    ServiceHost svc = new ServiceHost(typeof(HelloWCF));
    // добавляем конечную точку, опускаем адрес, компоновку и контракт
    svc.AddServiceEndpoint(typeof(IHelloWCF), binding, address);
    // начинаем ожидание
    svc.Open();
    // указываем, что приложение-получатель готово, и
    // удерживаем приложение от немедленного завершения
```

```
    Console.WriteLine("The HelloWCF receiving application is ready");
    // начало кода отправителя
    // создаем channelFactory<T> с компоновкой и адресом
    ChannelFactory<IHelloWCF> factory =
    new ChannelFactory<IHelloWCF>(binding, new EndpointAddress(address));
    // используем factory для создания прокси
    IHelloWCF proxy = factory.CreateChannel();
    // используем прокси для отправки сообщения получателю
    proxy.Say("Hi there WCF");
    // конец кода отправителя
    Console.ReadLine();
    // закрываем сервисный узел
    svc.Close();
  }
  // полученные сообщения подготавливаются к отправке этому экземпляру
  // метода согласно сервисному контракту
  public void Say(String input){
    Console.WriteLine("Message received, the body contains: {0}", input);
  }
}
[ServiceContract]
public interface IHelloWCF {
  [OperationContract]
  void Say(String input);
}
```

Обратите внимание на то, что мы вызываем экземпляр метода `CreateChannel` объекта `ChannelFactory<T>` и используем возвращенный объект для вызова метода интерфейса нашего сервисного контракта. На верхнем уровне детализации объект `ChannelFactory<T>` является типом, который может породить отправляющую инфраструктуру, необходимую для генерирования и отправки сообщения получателю (отсюда и вытекает необходимость опустить компоновку и адрес в конструкторе). Экземпляр метода `CreateChannel` типа `ChannelFactory<T>` на самом деле создает отправляющую инфраструктуру и возвращает указатель на эту инфраструктуру посредством объекта, чей тип реализует интерфейс сервисного контракта. Мы взаимодействуем с этой отправляющей инфраструктурой путем вызова методов интерфейса нашего сервисного контракта. Имейте в виду, что для всего этого существует несколько других способов, мы изучим их позже в этой главе и *главе 6*.

Компиляция, запуск и проверка отправителя

Теперь, когда у нас существуют отправляющая и получающая инфраструктуры, пришло время скомпилировать и запустить приложение:

```
c:\temp>csc /nologo /r:"c:\WINDOWS\Microsoft.Net\Framework\v3.0\Windows
Communication Foundation\System.ServiceModel.dll" HelloWCFApp.cs
c:\temp>HelloWCFApp.exe
The HelloWCF receiving application is ready
HelloWCF object created
Message received, the body contains: HelloWCF!
```

Как и ожидалось, наше приложение во время запуска сделало следующее:

1. Создало инфраструктуру, необходимую для ожидания входящих сообщений в точке **http://localhost:4000/IHelloWCF**.
2. Начало ожидание входящих сообщений в точке **http://localhost: 4000/IHelloWCF**.
3. Создало инфраструктуру, необходимую для отправки сообщения в точку **http://localhost:4000/IHelloWCF**.
4. Сгенерировало и отправило сообщение в точку **http://localhost: 4000/IHelloWCF**.
5. Получило сообщение, создало экземпляр нового объекта `HelloWCF` и подготовило это сообщение к отправке в метод `Say` объекта `HelloWCF`.

Ожидание сообщений

При ближайшем рассмотрении ни один из наших примеров кода HelloWCF не взаимодействует ни с чем, что хотя бы отдаленно напоминает сообщение. С точки зрения разработчика приложения, любое приложение WCF выглядит совершенно так же, как и любое объектно-ориентированное или компонентно-ориентированное приложение. Однако во время запуска приложение WCF полностью погружается в работу, связанную с генерацией, отправкой, получением и другой обработкой сообщений.

Изменив реализацию метода `Say` на нижеприведенную, мы увидим сообщение, генерируемое инфраструктурой WCF:

```
public void Say(String input){
  Console.WriteLine("Message received, the body contains: {0}", input);
```

```
  // отображаем содержание полученного сообщения
  Console.WriteLine(
    OperationContext.Current.RequestContext.RequestMessage.ToString());
}
```

Изменение метода `Say` приведет к следующим изменениям в выводимых приложением данных:

```
The HelloWCF receiving application is ready
HelloWCF object created
Message received, the body contains: HelloWCF!
<s:Envelope xmlns:s="http://schemas.xmlsoap.org/soap/envelope/">
  <s:Header>
    <To s:mustUnderstand="1"
        xmlns="http://schemas.microsoft.com/ws/2005/05/adessing/none">
      http://localhost:8000/IHelloWCF
    </To>
    <Action s:mustUnderstand="1"
      xmlns="http://schemas.microsoft.com/ws/2005/05/addressing/none">
      http://tempuri.org/IHelloWCF/Say
    </Action>
  </s:Header>
  <s:Body>
    <Say xmlns="http://tempuri.org/">
      <input>HelloWCF!</input>
    </Say>
  </s:Body>
</s:Envelope>
```

Обратите внимание, что выведено сообщение SOAP, а тело этого сообщения SOAP содержит строку, переданную методу `Say` через локальную переменную `channel`. На макроскопическом уровне рассмотрения эта отправляющая часть нашего приложения берет эту строку, использует ее для создания сообщения SOAP, а затем отправляет это сообщение SOAP принимающей части нашего приложения. С другой стороны, принимающая часть нашего приложения получает сообщение SOAP, создает объект `HelloWCF`, извлекает содержимое сообщения SOAP и вызывает метод `Say` объекта `HelloWCF`, передавая строку как аргумент.

Небольшое изменение с глобальными последствиями

Инфраструктура WCF делает за нас большую часть работы по обмену сообщениями, и обычная объектная модель не всегда замечает тот факт, что наше приложение WCF на самом деле передает сообщения между отправителем и получателем. С точки зрения разработчика приведенный в нашем примере код очень похож на код приложения, использующего распределенные объекты, чем на код приложения обмена сообщениями. Тем не менее мы можем очень легко убедиться в том, что наше приложение HelloWCF действительно является приложением обмена сообщениями, изменив одну строчку кода и наблюдая за влиянием, которое окажет это изменение на структуру сообщения.

Если мы изменим строку:

```
BasicHttpBinding binding = new BasicHttpBinding();
```

на такую:

```
WSHttpBinding binding = new WSHttpBinding();
```

то увидим вывод следующих данных:

```
The HelloWCF receiving application is ready
Creating and sending a message to the receiver
HelloWCF object created
Message received, the body contains: HelloWCF!
<s:Envelope xmlns:a="http://www.w3.org/2005/08/addressing"
            xmlns:s="http://www.w3.org/2003/05/soap-envelope">
<s:Header>
<a:Action s:mustUnderstand="1" u:Id="_2"
          xmlns:u="http://docs.oasis-open.org/wss/2004/01/oasis-200401-
  wss-wssecurity-utility-1.0.xsd">
  http://tempuri.org/IHelloWCF/Say
</a:Action>
<a:MessageID u:Id="_3"
  xmlns:u="http://docs.oasis-open.org/wss/2004/01/oasis-
  200401-wss-wssecurity-utility-1.0.xsd">
  urn:uuid:2acf3d19-dac6-4f8f-8c5d-b2ca104cd3a0
</a:MessageID>
<a:ReplyTo u:Id="_4"
    xmlns:u="http://docs.oasis-open.org/wss/2004/01/oasis-
    200401-wss-wssecurity-utility-1.0.xsd">
```

```
  <a:Address>http://www.w3.org/2005/08/addressing/anonymous</a:Address>
</a:ReplyTo>
<a:To s:mustUnderstand="1" u:Id="_5"
      xmlns:u="http://docs.oasis-open.org/wss/2004/01/
  oasis-200401-wss-wssecurity-utility-1.0.xsd">
  http://localhost:8000/IHelloWCF
</a:To>
<o:Security s:mustUnderstand="1"
    xmlns:o="http://docs.oasis-open.org/wss/2004/01/oasis-
  200401-wss-wssecurity-secext-1.0.xsd">
  <u:Timestamp u:Id="uuid-a4e930a1-1fc5-4450-8140-754a98690449-12"
      xmlns:u="http://docs.oasis-open.org/wss/2004/01/oasis-
      200401-wss-wssecurity-utility-1.0.xsd">
    <u:Created>2006-08-29T01:57:50.296Z</u:Created>
    <u:Expires>2006-08-29T02:02:50.296Z</u:Expires>
  </u:Timestamp>
  <c:SecurityContextToken
    u:Id="uuid-a4e930a1-1fc5-4450-8140-754a98690449-6"
    xmlns:c="http://schemas.xmlsoap.org/ws/2005/02/sc"
    xmlns:u="http://docs.oasis-open.org/wss/
    2004/01/oasis-200401-wss-wssecurity-utility-1.0.xsd">
    <c:Identifier>
       urn:uuid:9cb35fed-f9cb-47b5-810b-54cd96970695
    </c:Identifier>
  </c:SecurityContextToken>
  <c:DerivedKeyToken
   u:Id="uuid-a4e930a1-1fc5-4450-8140-754a98690449-10"
   xmlns:c="http://schemas.xmlsoap.org/ws/2005/02/sc"
   xmlns:u="http://docs.oasis-open.org/wss/2004/01/oasis-200401-wsswsse-
curity-utility-1.0.xsd">
    <o:SecurityTokenReference>
    <o:Reference
       ValueType="http://schemas.xmlsoap.org/ws/2005/02/sc/sct"
       URI="#uuid-a4e930a1-1fc5-4450-8140-754a98690449-6" />
    </o:SecurityTokenReference>
    <c:Offset>0</c:Offset>
    <c:Length>24</c:Length>
    <c:Nonce>A170b1nKz88AuWmWYONX5Q==</c:Nonce>
  </c:DerivedKeyToken>
```

```
  <c:DerivedKeyToken
   u:Id="uuid-a4e930a1-1fc5-4450-8140-754a98690449-11"
   xmlns:c="http://schemas.xmlsoap.org/ws/2005/02/sc"
   xmlns:u="http://docs.oasis-open.org/wss/2004/01/oasis-200401-wsswsse-
curity-utility-1.0.xsd">
    <o:SecurityTokenReference>
    <o:Reference
     ValueType="http://schemas.xmlsoap.org/ws/2005/02/sc/sct"
     URI="#uuid-a4e930a1-1fc5-4450-8140-754a98690449-6" />
    </o:SecurityTokenReference>
    <c:Nonce>I8M/H2f3vFuGkwZVV1Yw0A==</c:Nonce>
  </c:DerivedKeyToken>
  <e:ReferenceList xmlns:e="http://www.w3.org/2001/04/xmlenc#">
    <e:DataReference URI="#_1" />
    <e:DataReference URI="#_6" />
  </e:ReferenceList>
  <e:EncryptedData Id="_6"
   Type="http://www.w3.org/2001/04/xmlenc#Element"
   xmlns:e="http://www.w3.org/2001/04/xmlenc#">
  <e:EncryptionMethod
   Algorithm="http://www.w3.org/2001/04/xmlenc#aes256-cbc" />
        <KeyInfo xmlns="http://www.w3.org/2000/09/xmldsig#">
          <o:SecurityTokenReference>
          <o:Reference
           ValueType="http://schemas.xmlsoap.org/ws/2005/02/sc/dk"
           URI="#uuid-a4e930a1-1fc5-4450-8140-754a98690449-11" />
          </o:SecurityTokenReference>
        </KeyInfo>
        <e:CipherData>
          <e:CipherValue>
  vQ+AT5gioRS6rRiNhWw2UJmvYYZpA+cc1DgC/K+6Dsd2enF4RUcwOG2
xqfkD/
EZkSFRKDzrJYBz8ItHLZjsva4kqfx3UsEJjYPKbxihl2GFrXdPwTm-
rHWt35Uw0L2rTh8kU9rtj44NfULS59CJbXE6PC7
Af1qWvnobcPXBqmgm4NA8wwSTuR3IKHPfD/Pg/
3WABob534WD4T1DbRr5tXwNr+yQl2nSWN8C0aaP9+LCKymEK7AbeJXAaGoxdGu/
t6l7Bw1lBsJeSJmsd4otXcLxt976kBEIjTl8/
6SVUd2hmudP2TBGDbCCvgOl4c0vsHmUC1SjXE5vXf6ATkMj6P3o0eMqBiWlG26RWiY-
BZ3OxnC1fDs60uSvfHtfF8CD0I
```

```
LYGHLgnUHz5CFY0rPomT73RCkCfmgFuheCgB9zHZGtWedY6ivNrZe2KPx0u-
jQ2Mq4pv4bLns2qoykwK03ma7YGiGExGc
ZBfkZ2YAkYmHWXJ0Xx4PJmQRAWIKfUCqcrR6lwyLjl5Agsrt0xHA5WEk3hapscW3HZ8wOg-
wv0fcHlZ1e3EAm0dZr5Ose
3TAKMXf7FC1tMy5u0763flA6AZk9l7IpAQXcTLYicriH5hzf1416xbTJCtt2rztiItSkYizk-
iJCUMJLanc6ST5i+GVHz
J5oRCEWgfOTcQpHmri8y1P1+6jYe9ELla8Mj
          </e:CipherValue>
        </e:CipherData>
      </e:EncryptedData>
    </o:Security>
  </s:Header>
  <s:Body u:Id="_0"
   xmlns:u="http://docs.oasis-open.org/wss/2004/01/oasis-
            200401-wss-wssecurity-utility-1.0.xsd">
    <Say xmlns="http://tempuri.org/">
      <input>HelloWCF!</input>
    </Say>
  </s:Body>
</s:Envelope>
```

Как видите, одно простое изменение оказывает существенное влияние на структуру сообщений, производимых нашим приложением. Изменение `BasicHttpBinding` на `WSHttpBinding` меняет наше приложение с использующего простейшие сообщения SOAP по HTTP на задействованное в многообразии протоколов WS-* и хореографий по HTTP. Последствие изменения больше, чем просто многословное и описательное сообщение, поскольку наше приложение теперь отправляет и получает многочисленные сообщения, основанные на WS-Security, WS-SecureConversation и других спецификациях.

Примечание

В действительности, программная модель для WCF на макроуровне полностью скрывает все характеристики приложения, действительно занимающегося обменом сообщениями, и дает ощущение распределенных объектов. На мой взгляд, это является потрясающим преимуществом платформы, но в то же время таит в себе определенные опасности. Как разработчики, мы не имеем права внушить себе мысль, что можем работать с WCF так же, как мы делали бы это с распределенной объектной платформой, и вместо этого выбрать концепции обмена сообщениями. Более того, как разработчики приложений и инфраструктуры мы обязаны понимать, как изменения в способе использования типов WCF повлияют на обрабатываемые нашим приложением сообщения.

Предоставление метаданных

Наше приложение HelloWCF использует очень упрощенный подход для обеспечения совместимости между отправителем и получателем. Поскольку и получатель, и отправитель находятся в одном домене `AppDomain`, а используемые получателем объекты видимы для отправителя, то мы просто повторно используем для отправителя адрес, компоновку и контракт. Тем не менее, такой подход осуществим не для всех приложений обмена сообщениями. В большинстве случаев отправитель и получатель находятся в различных доменах `AppDomains` на разных компьютерах. В таких сценариях обычно получатель навязывает требования для обмена сообщениями, а отправитель старается придерживаться этих требований.

Спецификация WS-MetadataExchange определяет условия, на которых отправитель и получатель могут обмениваться подобной информацией приемлемым поставщиком способом. Другими словами, спецификация WS-MetadataExchange навязывает схемы и хореографии сообщений, способствующие осуществлению обмена информацией между конечными точками процесса обмена сообщениями. В большинстве реальных приложений (или хотя бы в более сложных, чем наше приложение HelloWCF) необходимо предоставлять эту информацию в таком виде, чтобы отправитель смог запросить конечную точку получателя, извлечь метаданные и использовать их для построения инфраструктуры, необходимой для отправки совместимого сообщения в эту конечную точку.

По умолчанию наше приложение HelloWCF не предоставляет никаких метаданных, по крайней мере, в наиболее общепринятом виде форматов Web Ser-

vices Description Language (WSDL) и Extensible Schema Definition (XSD). (Не путайте метаданные приложения обмена сообщениями с метаданными компоновок или типов, даже несмотря на то, что одни могут использоваться для создания других.) Приложения WCF действительно не предоставляют метаданные по умолчанию, причиной чему служит забота о безопасности приложений. Представляемая метаданными информация часто содержит требования по безопасности для самого приложения. Защищая подобного рода секреты, команда разработчиков WCF предпочла по умолчанию отключить эту возможность.

Тем не менее, если мы решим предоставить метаданные нашего приложения, то можем создать конечную точку специально для обмена метаданными, а способ создания конечной точки метаданных совершенно аналогичен способу создания любой другой конечной точки и начинается с адреса, компоновки и контракта. Однако в отличие от конечных точек, с которыми вы уже познакомились, для конечной точки метаданных сервисный контракт уже задан в WCF API.

Первым шагом на пути создания конечной точки метаданных является изменение статуса `ServiceHost` таким образом, чтобы он стал ведущим узлом для метаданных. Мы можем сделать это, добавив объект `System.ServiceModel.Description.ServiceMetadataBehavior` в коллекцию образцов действий `ServiceHost`. Модель поведения (behavior) — это специальная информация, которую инфраструктура WCF использует для изменения локальной обработки сообщений. В приведенном далее коде показано, как добавить объект `ServiceMetadataBehavior` к активному `ServiceHost`:

```
// создаем экземпляр ServiceHost, опуская тип для создания экземпляра,
// когда приложение получит сообщение
ServiceHost svc = new ServiceHost(typeof(HelloWCF), address);

// НАЧАЛО НОВОГО КОДА МЕТАДАННЫХ
// создаем ServiceMetadataBehavior
ServiceMetadataBehavior metadata = new ServiceMetadataBehavior();
metadata.HttpGetEnabled = true;
// добавляем его в описание servicehost
svc.Description.Behaviors.Add(metadata);
```

Следующим шагом является определение `Binding` для конечной точки метаданных. Объектная модель для метаданных `Binding` существенно отличается от других компоновок, а именно мы создаем метаданные `Binding`, вызывая

метод типа `System.ServiceModel.Description.MetadataExchangeBindings`, как показано далее (для большей ясности другие части нашего приложения HelloWCF были опущены):

```
// создаем экземпляр ServiceHost, опуская тип для создания экземпляра,
// когда приложение получит сообщение
ServiceHost svc = new ServiceHost(typeof(HelloWCF));

// НАЧАЛО НОВОГО КОДА МЕТАДАННЫХ
// создаем ServiceMetadataBehavior
ServiceMetadataBehavior metadata = new ServiceMetadataBehavior();
// добавляем его в описание servicehost
svc.Description.Behaviors.Add(metadata);
// создаем компоновку метаданных TCP
Binding mexBinding = MetadataExchangeBindings.CreateMexTcpBinding();
```

В результате предыдущих условий по ASMX у вас может возникнуть ощущение, что метаданные могут отправляться только транспортом HTTP. На самом деле, метаданные могут передаваться посредством множества транспортов, и эта гибкость определяется WS-MetadataExchange. Тем не менее в нашем примере мы вызываем метод `CreateMexTcpBinding`, который возвращает указатель на производный тип `Binding`, использующий транспорт TCP. Поскольку мы применяем транспорт TCP, то обязаны также убедиться, что выбранный нами адрес метаданных использует схему TCP, как показано далее:

```
// создаем экземпляр ServiceHost, опуская тип для создания экземпляра,
// когда приложение получит сообщение
ServiceHost svc = new ServiceHost(typeof(HelloWCF));

// НАЧАЛО НОВОГО КОДА МЕТАДАННЫХ
// создаем ServiceMetadataBehavior
ServiceMetadataBehavior metadata = new ServiceMetadataBehavior();
// добавляем его в описание servicehost
svc.Description.Behaviors.Add(metadata);
// создаем компоновку метаданных TCP
Binding mexBinding = MetadataExchangeBindings.CreateMexTcpBinding();
// создаем адрес для ожидания трафика обмена WS-Metadata
Uri mexAddress = new Uri("net.tcp://localhost:5000/IHelloWCF/Mex");
```

Теперь, когда мы определили адрес и компоновку, настало время использовать конечную точку наших метаданных. Мы должны добавить конечную точку в `ServiceHost` таким же образом, как мы делали это для первой конечной точки сообщений. Однако при добавлении конечной точки мы использовали сервисный контракт, уже определенный в WCF API и названный `System.ServiceModel.Description.IMetadataExchange`. Приведенный далее фрагмент кода показывает, как добавить конечную точку метаданных в `ServiceHost`, используя соответствующий адрес, компоновку и контракт:

```
// создаем экземпляр ServiceHost, опуская тип для создания экземпляра,
// когда приложение получит сообщение
ServiceHost svc = new ServiceHost(typeof(HelloWCF));

// НАЧАЛО НОВОГО КОДА МЕТАДАННЫХ
// создаем ServiceMetadataBehavior
ServiceMetadataBehavior metadata = new ServiceMetadataBehavior();
// добавляем его в описание servicehost
svc.Description.Behaviors.Add(metadata);
// создаем компоновку метаданных TCP
Binding mexBinding = MetadataExchangeBindings.CreateMexTcpBinding();
// создаем адрес для ожидания трафика обмена WS-Metadata
Uri mexAddress = new Uri("net.tcp://localhost:5000/IHelloWCF/Mex");
// добавляем конечную точку метаданных
svc.AddServiceEndpoint(typeof(IMetadataExchange),
                       mexBinding,
                       mexAddress);
// КОНЕЦ КОДА МЕТАДАННЫХ
```

Если мы создадим и запустим наше приложение HelloWCF, то увидим, что приложение действительно ожидает сообщения по двум различным адресам. Один адрес предназначен для сервисных запросов метаданных, а другой — для функциональности `IHelloWCF.Say`. Давайте теперь переключим наше внимание с конечных точек метаданных на непосредственное использование их для построения отправляющей инфраструктуры нашего приложения.

Использование метаданных

Microsoft .NET Framework SDK устанавливает весьма универсальное средство под названием svcutil.exe. Одной из его способностей является возмож-

ность опрашивать запущенное приложение обмена сообщениями и создавать прокси, основываясь на полученной информации. Svcutil.exe использует протокол WS-MetadataExchange так же, как WSDL "использует" семантику, распространяемую с ASMX. Поскольку наше получающее приложение теперь порождает конечную точку метаданных, мы можем направить svcutil.exe на эту конечную точку, и svcutil.exe автоматически сгенерирует тип прокси и конфигурационную информацию, совместимую с той конечной точкой, на которую указывает конечная точка метаданных. При подобном использовании svcutil.exe отправляет сообщения получающему приложению согласно WS-MetadataExchange и преобразует подтверждающий ответ в тип .NET Framework, который облегчает разработку отправляющего приложения.

Создание прокси с помощью svcutil.exe

Прежде чем запустить svcutil.exe, убедитесь, что получающее приложение HelloWCFApp.exe запущено и ожидает входящие сообщения. Затем откройте окно Windows SDK Command Prompt и введите следующую команду:

```
C:\temp>svcutil /target:code net.tcp://localhost:5000/IHelloWCF/Mex
```

Svcutil.exe создаст два файла: HelloWCFProxy.cs и output.config. Если вы изучите файл HelloWCFProxy.cs, вы увидите, что svcutil.exe сгенерировал исходный файл, содержащий определения для интерфейса `IHelloWCF`, интерфейса `IHelloWCFChannel` и типа под названием `HelloWCFClient`.

> ***Примечание***
>
> Среди всех типов, автоматически сгенерированных svcutil.exe, тип `HelloWCFClient` предназначен для самого частого использования. На мой взгляд, добавление слова "Client" к имени этого типа является ошибочным в понимании сообщества разработчиков. Без сомнения, "Client" ассоциируется с фразой "Client and Server". Тип `HelloWCFClient` помогает создать инфраструктуру сообщений, а не обычную клиент-серверную архитектуру. Имейте в виду, что даже если название этого типа заканчивается на "Client", мы все же создаем приложение обмена сообщениями.

Все вместе эти определения типов помогают нам написать отправляющий код, совместимый с получателем. Заметьте, что в файле HelloWCF.cs нет информации об адресе, по которому ожидает сообщений приложение-получатель, также в исходном файле HelloWCF.cs нет компоновки, совместимой с приложением-получателем. Эта информация сохранена в другом файле, со-

зданном svcutil.exe (output.config). WCF обладает богатой конфигурационной инфраструктурой, позволяющей нам формировать различные аспекты принимающего и отправляющего приложений посредством конфигурационных файлов XML. Для наглядности использования преимущества созданных svcutil данных давайте создадим другое консольное приложение, отправляющее сообщения получателю. Назовем это приложение HelloWCFSender. Для этого переименуем файл output.config таким образом, чтобы наше новое приложение-отправитель читало конфигурационный файл (изменим на HelloWCFSender.exe.config).

Кодирование HelloWCFSender с использованием типов, сгенерированных svcutil.exe

Инструмент svcutil.exe сгенерировал практически весь исходный код и конфигурационные настройки, которые необходимы для написания нашего приложения-отправителя. Создание этого отправляющего приложения аналогично созданию такого приложения в HelloWCF.exe.

```
using System;
using System.ServiceModel;
sealed class HelloWCFSender {
  static void Main(){
    // ожидаем старта получателя
    Console.WriteLine("Press ENTER when the Receiver is ready");
    Console.ReadLine();
    // выводим на консоль информацию о том, что мы посылаем сообщение
    Console.WriteLine("Sending a message to the Receiver");
    // создаем тип HelloWCFClient, создаваемый svcutil
    HelloWCFClient proxy = new HelloWCFClient();
    // вызываем метод Say
    proxy.Say("Hi there from a new Sender");
    proxy.Close();
    // выводим на консоль информацию о том, что мы отправили сообщение
    Console.WriteLine("Finished sending a message to the Receiver");
  }
}
```

Обратите внимание, что мы должны только создать экземпляр типа `HelloWCFClient` и вызвать метод `Say`. По-настоящему трудную работу сдела-

ли типы, созданные svcutil.exe и конфигурационной инфраструктурой WCF. После написания этого исходного кода мы можем скомпилировать его в сборку с помощью следующей командной строки:

```
C:\temp>csc /r: "C:\WINDOWS\Microsoft.Net\v3.0\Windows Communication
Foundation\System.ServiceModel.dll" HelloWCFProxy.cs HelloWCFSender.cs
```

Затем мы запускаем приложение-получатель (HelloWCFApp.exe), а затем — приложение-отправитель (HelloWCFSender.exe), и видим выводимую получателем информацию:

```
C:\temp>HelloWCFSender.exe
Press ENTER when the Receiver is ready
Sending a message to the Receiver
Finished sending a message to the Receiver
```

По сути, выводимая нашим приложением информация подтверждает, что отправляющая сторона нашего приложения работает так же, как работала до этого, без повторного применения объектов, использованных для построения принимающей стороны нашего приложения. Мы можем проверить приложение-получатель, чтобы убедиться, что получатель на самом деле получил новое сообщение.

Теперь, когда мы имеем два полностью функционирующих приложения WCF, давайте взглянем на архитектуру WCF целиком.

Анатомия WCF на макроуровне: взгляд извне

Несмотря на то, что WCF является достаточно сложной платформой, для стороннего наблюдателя она может показаться в высшей степени тривиальной. Как было показано в примере HelloWCF, создать приложение-получатель на базе WCF очень просто: для этого достаточно создать одну или несколько конечных точек, используя адрес, компоновку и контракт. Создать приложение-отправитель также достаточно несложно: для этого нужно отправить сообщение в эту получающую конечную точку, используя компоновку, контракт и адрес. Если мы захотим изменить локальную обработку сообщений на стороне отправителя или стороне получателя, то вольны сделать это либо путем создания нашего собственного образца действия, либо же используя образцы действий, поставляемые с WCF (как, например, добавление поддержки метадан-

ных). На рис. 4.1 показана связь между конечными точками, адресами, компоновками, контрактами и моделями поведения.

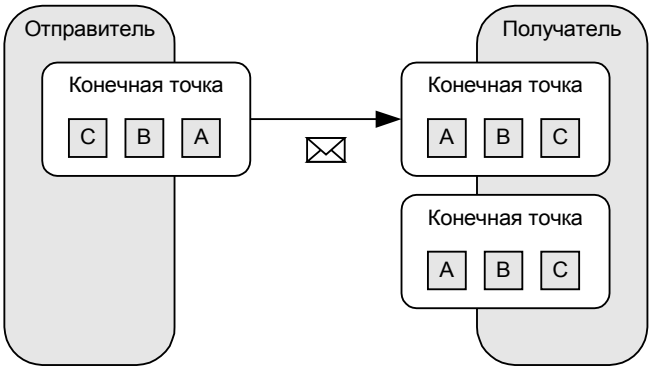


Рис. 4.1. Конечные точки, адреса, компоновки, контракты и модели поведения

Адрес

Все приложения, отправляющие или получающие сообщения, в определенный момент времени должны использовать адрес. Например, приложения-получатели ожидают входящие сообщения по какому-либо адресу, в то время как приложения-отправители направляют сообщения по определенному целевому адресу. Принимающая инфраструктура WCF для создания принимающей конечной точки полагается на тип `System.Uri`. Отправляющая инфраструктура WCF, с другой стороны, для отправки сообщений конечному получателю полагается на тип `System.ServiceModel.EndpointAddress`. Тип `EndpointAddress` является CLR-абстракцией указателя конечной точки WS-Addressing, а отправители используют этот тип как для добавления информации об указателе конечной точки к исходящим сообщениям, так и для установки с этой конечной точкой соединения на транспортном уровне. В *главе 5*, помимо всего прочего, также детально рассмотрен тип `EndpointAddress`.

В контексте WCF адрес является в некотором роде URI (объект `EndpointAddress` является оберткой объекта `System.Uri`). По сути URI — это название схемы. Схема — это абстракция типа идентификатора, представляемого URI, а название схемы — это способ идентифицировать схему. В большинстве случаев название схемы соответствует протоколу, который может быть использован для определения ресурса, что позволяет использовать URI как URL. Например, следующее URI: **http://localhost:5000/**

IHelloWCF определяет **http** как название схемы, а то, что http (Hypertext Transfer Protocol) еще и транспорт, является просто случайным совпадением. Инфраструктура WCF обязана предоставлять возможность использовать URI для создания либо отправляющей, либо принимающей инфраструктуры.

Компоновка

Компоновки являются основным способом, которым мы выражаем, как приложение обмена сообщениями обрабатывает, отправляет и получает сообщения. Более точно — это основной способ, которым мы отображаем используемые конечной точкой транспорт, протоколы WS-*, требования по безопасности и требования по поддержке транзакций. WCF поставляется вместе с девятью компоновками, которые покрывают широкий спектр транспортов, протоколов WS-*, требований по безопасности и требований по поддержке транзакций. Если эти возможности не соответствуют требованиям нашего приложения, то мы можем задать собственную компоновку.

Вообще говоря, компоновка — это тип, определяющий многое для нашей инфраструктуры сообщений; это уровень абстракции транспорта и протоколов, поддерживаемых нашим приложением. Для разработчика такая абстракция означает, что код, требуемый для отправки сообщения посредством транспорта TCP/IP, выглядит совершенно так же, как код, отправляющий сообщение посредством MSMQ, практически не привязывая тем самым наше приложение к конкретному транспорту или конкретному множеству протоколов. Такая слабая привязка в этом смысле позволяет разработчикам приложения гораздо быстрее, чем это было возможно ранее разрабатывать, адаптировать и настраивать приложение под требования заказчика.

Все компоновки являются подклассом типа `System.ServiceModel.Channels.Binding`, что позволяет им разделять общие характеристики. Одной из общих характеристик для всех компоновок является то, что они поддерживают собственный список объектов `System.ServiceModel.Channels.BindingElement.BindingElement`. Список элементов `BindingElement` является некоей абстракцией в процессе обмена сообщениями, такого как транспорт или протокол WS-*. Все компоновки предоставляют метод, называемый `CreateBindingElements`, который создает и возвращает список элементов для данной конкретной компоновки. Приведенное далее простое приложение перечисляет все девять заданных в WCF компоновок и показывает их списки элементов `BindingElement`:

```
using System;
```

```
using System.ServiceModel;
using System.ServiceModel.Channels;
using System.Reflection;
using System.Collections.Generic;
using System.ServiceModel.MsmqIntegration;
sealed class BindingElementsShow
{
  static void Main(){
    List<Binding> bindings = new List<Binding>();
    bindings.Add(new BasicHttpBinding());
    bindings.Add(new NetNamedPipeBinding());
    bindings.Add(new NetTcpBinding());
    bindings.Add(new WSDualHttpBinding());
    bindings.Add(new WSHttpBinding());
    bindings.Add(new NetMsmqBinding());
    bindings.Add(new MsmqIntegrationBinding());
    bindings.Add(new WSFederationHttpBinding());
    // завершаем работу, если не установлен Peer Networking
    bindings.Add(new NetPeerTcpBinding());
    ShowBindingElements(bindings);
  }
  private static void ShowBindingElements(List<Binding> bindings){
    foreach (Binding binding in bindings){
      Console.WriteLine("Showing Binding Elements for {0}",
      binding.GetType().Name);
      foreach (BindingElement element in binding.CreateBindingElements())
{
        Console.WriteLine("\t{0}", element.GetType().Name);
      }
    }
  }
}
```

Данная программа выведет следующую информацию:

```
Showing Binding Elements for BasicHttpBinding
        TextMessageEncodingBindingElement
        HttpTransportBindingElement
Showing Binding Elements for NetNamedPipeBinding
```

```
        TransactionFlowBindingElement
        BinaryMessageEncodingBindingElement
        WindowsStreamSecurityBindingElement
        NamedPipeTransportBindingElement
Showing Binding Elements for NetTcpBinding
        TransactionFlowBindingElement
        BinaryMessageEncodingBindingElement
        WindowsStreamSecurityBindingElement
        TcpTransportBindingElement
Showing Binding Elements for WSDualHttpBinding
        TransactionFlowBindingElement
        ReliableSessionBindingElement
        SymmetricSecurityBindingElement
        CompositeDuplexBindingElement
        OneWayBindingElement
        TextMessageEncodingBindingElement
        HttpTransportBindingElement
Showing Binding Elements for WSHttpBinding
        TransactionFlowBindingElement
        SymmetricSecurityBindingElement
        TextMessageEncodingBindingElement
        HttpTransportBindingElement
Showing Binding Elements for NetMsmqBinding
        BinaryMessageEncodingBindingElement
        MsmqTransportBindingElement
Showing Binding Elements for MsmqIntegrationBinding
        MsmqIntegrationBindingElement
Showing Binding Elements for WSFederationHttpBinding
        TransactionFlowBindingElement
        SymmetricSecurityBindingElement
        TextMessageEncodingBindingElement
        HttpTransportBindingElement
Showing Binding Elements for NetPeerTcpBinding
        PnrpPeerResolverBindingElement
        BinaryMessageEncodingBindingElement
        PeerTransportBindingElement
```

Как показывают выводимые данные, возвращаемый методом `CreateBindingElements` `Binding`-объект является упорядоченным списком элементов

BindingElements. Заметьте, что последний элемент в списке BindingElement всегда является транспортом, и что каждый список содержит BindingElement, представляющий кодировку сообщений. Несколько из поставляемых по умолчанию компоновок создают BindingElement-списки, которые содержат дополнительные элементы BindingElements, но в этом списке всегда должны присутствовать элементы, представляющие транспорт.

В выведенных данных можно заметить, что каждый производный Binding-тип представляет множество характеристик сообщений. Во время запуска содержимое списка элементов BindingElements определяет характеристики сообщений в конечной точке нашего приложения. Другими словами, выбираемая для нашей конечной точки компоновка оказывает прямое влияние на способ, которым наше приложение отправляет и принимает сообщения. В результате понимание характеристик сообщений, а именно Binding, существенно для успешной реализации приложения на базе WCF. В табл. 4.1 показаны самые важные характеристики каждой компоновки, поставляемой с WCF.

Таблица 4.1. *Характеристики компоновок* `Binding`*, поставляемых по умолчанию*

	Interop	Security	Session	Transactions	Duplex	Streaming	Encoder
BasicHttpBinding	BP 1.1	T				❍	TX
WSHttpBinding	WS-*	M	❍	❍	❍	❍	TX/MT
WSDualHttpBinding	WS-*	TM	❍	❍	❍		TX/MT
NetTcpBinding	WCF	TM	❍	❍	❍	❍	B
NetMsmqBInding	WCF	TM	❍	❍			B
MsmqIntegrationBinding	MSMQ	T					TX
NetNamedPipeBinding	WCF	TM	❍	❍	❍	❍	B
NetPeerTcpBinding	WCF	T		B			
WSFederationHttpBinding	WS-*	M	❍	TX			

Обозначения: BP 1.1 — Basic Profile 1.1; T — Transport; M — Message; TX — Text; MT — MTOM; B — Binary.

При первом применении поставляемых с WCF компоновок легко растеряться от того обилия параметров сообщений, которые предоставляют эти компоновки. Имейте в виду, что при выборе компоновки вы на самом деле выбираете компоновку для конкретной конечной точки, а приложение может обладать множеством конечных точек. Если мы создаем и развертываем приложение-получатель, принимающее текстовые сообщения по HTTP, то мы легко можем добавить еще одну конечную точку, позволив тем самым приложению принимать бинарные сообщения по TCP. Вообще говоря, компоновка `Binding`, реализованная в конечной точке, является основным средством для отображения инфраструктуры сообщений конечной точки. Более детально компоновки описаны в *главе 8*.

Контракт

Контракты отображают объектно-ориентированные конструкции на конструкции сообщений. Более точно, конструкции определяют конечные точки в приложении-получателе, образец MEP, используемый этими конечными точками, и структуру сообщений, обрабатываемую конечной точкой. Например, контракт помогает отобразить схему тела сообщения в определение типа .NET Framework, упрощая тем самым код, требуемый для генерации сообщения с соответствующим схеме содержимым. WCF позволяет использовать три типа контрактов: сервисные контракты (service contracts), контракты данных (data contracts) и контракты сообщений (message contracts). Сервисные контракты описывают операции в конечной точке. Такое описание включает название, MEP, относящуюся к сессии информацию, блоки заголовков для сообщений-запросов и сообщений-ответов, а также информацию по безопасности для каждой операции. Контракты данных отображают структуру как тела, так и блоков заголовков сообщения на одну или более операций.

> ***Примечание***
>
> Все контракты являются аннотированными определениями типов и членов типов, а атрибут, используемый в аннотации, указывает на то, что представляет собой определение типа: сервис, данные или контракт сообщений. Важно помнить, что аннотирование определения типа или члена типа — это просто добавление информации в метаданные этого определения типа. Таким образом, все определения атрибутов инертны. Выполнение действий, явившихся следствием присутствия какого-либо специфиче-

ского атрибута, требует другого кода для опроса метаданных определения типа посредством Reflection API. В случае WCF-контрактов, WCF-инфраструктура опрашивает метаданные определения контракта и предпринимает действия в соответствии с содержимым этих метаданных. Поскольку подобную работу можно выполнить вручную, то использование контрактов является необязательным. Иными словами, инфраструктура WCF выполняет достаточно скучную работу на основе определений контрактов, поэтому все приложения на базе WCF должны использовать контракты. Более подробно контракты рассмотрены в *главе 9*.

Конструирование контракта путем аннотирования определения типа может оказаться по сути рискованным. Хотя такой способ и является одним из основных, с помощью которых WCF предоставляет разработчикам возможности расширяемости и адаптируемости, в то же время это означает, что возможная несовместимость не будет выявлена до момента запуска приложения.

Сервисные контракты

Сервисные контракты представляют операции, предоставляемые конечной точкой, и используются в процессе обмена сообщениями как отправителем, так и получателем. Принимающее приложение может использовать сервисный контракт для создания инфраструктуры сообщений, ожидающей входящих сообщений. Отправляющее приложение может использовать сервисный контракт для создания инфраструктуры сообщений, отправляющей сообщения в получающую конечную точку. В сервисном контракте содержится следующая информация: название каждой операции, параметры этой операции, блок заголовков, соответствующий этой операции, и информация об этой операции, относящаяся к сессии.

На самом низком уровне рассмотрения сервисный контракт — это определение класса или интерфейса, аннотированного атрибутом `ServiceContractAttribute` и одним или более атрибутами `OperationContractAttribute`. Атрибут `ServiceContractAttribute` легален как для классов, так и для интерфейсов, тогда как `OperationContractAttribute` легален для методов. Большинство методов, аннотированных атрибутом `OperationContractAttribute`, являются членами типа, аннотированного `ServiceContractAttribute`, с одним существенным исключением: они должны быть дуплексными сервисными контрактами. Повторимся, что эта тема будет детально раскрыта в *главе 9*.

Контракты данных

Контракты данных отображают типы .NET Framework на тело сообщения. Если выбранной структурой сообщений является SOAP, то контракт данных отображает тип .NET Framework на схему тела сообщения SOAP. Как и любой контракт WCF, контракт данных является аннотированным определением типа, а существенными атрибутами являются атрибуты `DataContractAttribute` и `DataMemberAttribute`. В большинстве случаев сервисный контракт ссылается на контракт данных так, как показано в следующем примере:

```
[ServiceContract]
interface ISomeServiceContract {
  [OperationContract]
  void SomeOperation(SomeDataContract info); // отмечаем тип аргумента
}
[DataContract()]
sealed class SomeDataContract {
  [DataMember]
  Int32? number;
  String status;
  [DataMember]
  internal String Status {
    get { return status; }
    set { status = value; }
  }
  internal Int32? Number {
    get { return number; }
  }
  internal SomeDataContract(Int32? number) : this(number, null)
  {
  }
  internal SomeDataContract(Int32? number, String status) {
    this.number = number;
    this.status = status; // учитываем "пустой" вариант
  }
}
```

В этом примере интерфейс `ISomeServiceContract` определяет метод, принимающий аргумент типа `SomeDataContract`. Поскольку тип `SomeDataContract`

аннотирован атрибутом `DataContractAttribute`, то тело сообщения, отправленного операции `SomeOperation`, будет иметь схему, указанную типом `SomeDataContract`.

Контракты сообщений

Контракты сообщений отображают типы .NET Framework на структуру сообщения. Если структурой сообщения является XML, то контракт сообщений отображает тип .NET Framework на схему сообщения. Контракт включает в себя как блоки заголовков, так и тело сообщения, что показано далее:

```
[ServiceContract]
interface ISomeServiceContract {
  [OperationContract]
  void SomeOperation(SomeDataContract info); // отмечаем тип аргумента
  [OperationContract]
  void SomeOtherOperation(SomeMessageContract info);
    // отмечаем тип аргумента
}
[DataContract()]
sealed class SomeDataContract {
  [DataMember]
  Int32? number;
  String status;
  [DataMember]
  internal String Status {
    get { return status; }
    set { status = value; }
  }
  internal Int32? Number {
    get { return number; }
  }
  internal SomeDataContract(Int32 number) : this(number, null)
  {
  }
  internal SomeDataContract(Int32 number, String status) {
    this.number = number;
    this.status = status; // учитываем "пустой" вариант
  }
```

```
}
[MessageContract]
sealed class SomeMessageContract {
  SomeMessageContract() { } // должен иметь конструктор по умолчанию
  [MessageHeader]
  Int32? SomeNumber;
  [MessageBodyMember]
  SomeDataContract messageBody;
  internal SomeMessageContract(Int32? someNumber) {
    SomeNumber = someNumber;
    messageBody = new SomeDataContract(someNumber);
  }
}
```

Заметьте, что в приведенном фрагменте кода метод `SomeOtherOperation` интерфейса `ISomeServiceContract` принимает в качестве аргумента параметр типа `SomeMessageContract`. Это допустимо, поскольку определение типа `SomeMessageContract` имеет аннотацию `MessageContractAttribute`. Итак, мы вкратце рассмотрели контракты, более детально о них будет рассказано в *главе 9*.

Анатомия WCF на макроуровне: взгляд изнутри

При взгляде на приложение WCF со стороны (адрес, компоновка и контракт) кажется удивительным то, как WCF использует адреса, компоновки и контракты для отправки и получения сообщений. В рассмотренных ранее примерах только небольшая часть кода напрямую относится к отправке и получению сообщений. На самом деле адрес, компоновка и контракт сами по себе делают немного. При более пристальном взгляде на приложение WCF мы увидим другую инфраструктуру, использующую адреса, компоновки и контракты для отправки и получения сообщений. По сути оставшаяся часть книги посвящена рассмотрению этой инфраструктуры, поэтому в данной главе я познакомлю вас только с основными частями этой инфраструктуры.

Если посмотреть через призму адресов, компоновок и контрактов на инфраструктуру, то мы увидим, что она имеет два основных архитектурных уровня: уровень ServiceModel и уровень Channel. Уровень ServiceModel является шлюзом между пользовательским кодом и уровнем Channel. Другими слова-

ми, это — часть обычного API. Уровень Channel, с другой стороны, выполняет основную работу по обмену сообщениями. Уровень Channel является тем уровнем, который разбирается в деталях конкретного транспорта и хореографиях сообщений WS-*. WCF поставляется с большими функциональными возможностями уровня Channel. Вообще говоря, уровень Channel — это область знаний разработчиков инфраструктуры, что делает написание полнофункционального приложения WCF без необходимости писать код на уровне Channel вполне возможным.

Примечание

Хотя такой подход может показаться весьма упрощенным, тем не менее я разделяю разработчиков на две категории: разработчиков приложений и разработчиков инфраструктур. Разработчики приложений пишут приложения, а разработчики инфраструктур создают код, который будет использоваться разработчиками приложений. Разработчик приложения может, к примеру, писать приложение для обработки заказов, тогда разработчик инфраструктуры будет писать компонент многократного использования, необходимый для системы обработки заказов. В рамках WCF разработчик приложения пишет приложение обмена сообщениями, а разработчик инфраструктуры — заказной канал.

На рис. 4.2 показана взаимосвязь уровней ServiceModel и Channel.

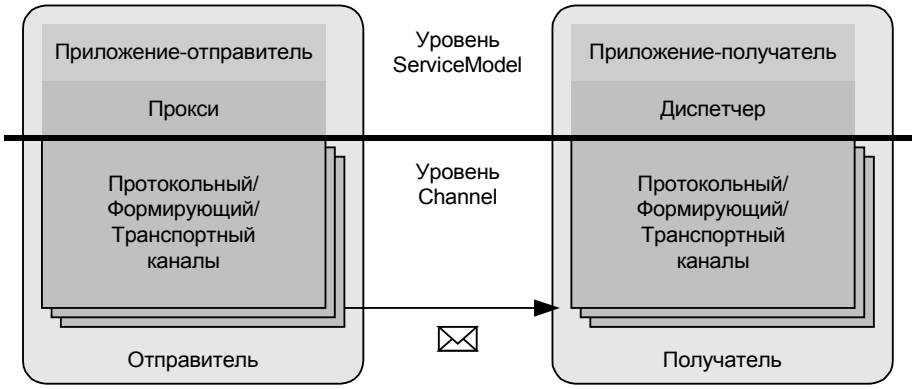


Рис. 4.2. Уровни ServiceModel и Channel

Обратите внимание, что уровень ServiceModel на стороне отправителя называется Прокси (Proxy) или Клиент (Client), а на стороне получателя — Диспетчер (Dispatcher). Несмотря на то, что Прокси и Диспетчер являются частя-

ми одного и того же архитектурного уровня, они играют различные роли. Прокси, помимо всего прочего, отвечает за создание сообщений для отправки на уровень Channel. Диспетчер, с другой стороны, ответственен за десериализацию полученных сообщений, создание экземпляра объекта и диспетчеризацию содержимого десериализованного сообщения к этому объекту. Как Прокси, так и Диспетчер выполняют гораздо больше функций, чем перечислено выше, и они будут подробно рассмотрены в *главе 10*.

Уровни ServiceModel и Channel отделены от простого мира адресов, компоновок и контрактов. На самом деле адрес, компоновка и контракт являются следствием влияния API для разработчиков приложений на создание этих двух уровней. При первом использовании уровней WCF полезно посмотреть, на какие уровни оказывают влияние адрес, компоновка и контракт. На стороне получателя адрес указывает уровню Channel, где ожидать входящих сообщений. На стороне отправителя адрес указывает уровню Channel, где связываться с принимающим приложением. Компоновки, со своей стороны, являются коллекциями объектов, создающих уровень Channel. Контракты используются для сериализации и десериализации сообщений, а также для определения образца MEP принимающего приложения. Вообще говоря, контракт является конструкцией ServiceModel. Образцы действий, с другой стороны, могут оказывать влияние как на уровень SreviceModel, так и на уровень Channel, что иллюстрирует рис. 4.3.

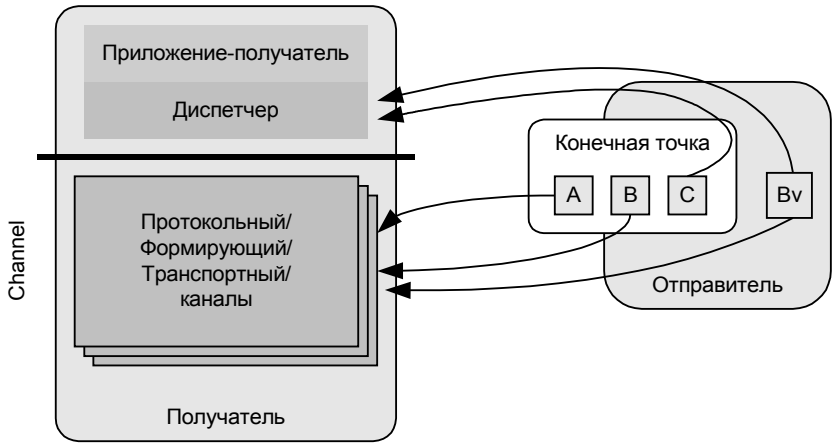


Рис. 4.3. Как конечные точки WCF влияют на уровни ServiceModel и Channel

Выводы

В этой главе мы создали простое приложение WCF и проанализировали во время запуска его основные составные части. Мы увидели, что хотя предоставляемое разработчику WCF API является весьма простым, однако оно позволяет обеспечить достаточную степень гибкости разрабатываемым приложениям. Также мы убедились, что адреса, компоновки, контракты и образцы действий, составляющие простое WCF API, используются двумя основными архитектурными уровнями: уровнем ServiceModel и уровнем Channel. Оставшаяся часть этой книги посвящена детальному рассмотрению этих двух уровней.