转眼微软的WCF已走过十个年头，它是微软通信框架的集大成者，将之前微软所有的通信框架进行了整合，提供了统一的应用方式。记得从自己最开始做MFC时，就使用过Named Pipe命名管道，之后做Winform时，进程会使用Remoting，再之后做B/S架构时，就会经常使用.NET平台下的Web Service，直到使用上WCF。看上去有了一些WCF的使用经验，实则不然，比如对安全、分布式事务、可靠会话等主题仍然接触甚少，因而决定重新回顾学习一下相关知识，尤其是对WCF框架的理解（已于2015年开源，可下载源码，https://github.com/dotnet/wcf/）。很多大公司都构建了自己的SOA框架，不过基本上都是以WCF框架为基础，对其进行了相应的简化和微调。因此学习该框架，可以触类旁通，对应用和搭建自有的SOA架构也有很大的帮助。当然，个人认为WCF已足够强大，并且其管道模式有极强的扩展性，可以通过自定义绑定满足绝大部分的需求。整个学习过程将参考蒋金楠大师的《WCF全面解析》一书，本章主要介绍WCF的基本概念和传说中的“ABC”，Let go。

在介绍WCF之前，不得不提一个称为**SOA**(Service Orientation Architecture)的概念，也就是我们常说的面向服务的架构，这是一个很老的概念了。即使如此，如果要以SOA为题，写一遍2000字的论文，感觉仍然很难下手，说明对概念理解还不够深刻（之后打算专门撰文一篇，为软考做准备）。实际上，其是构建大型软件应用的一种重要理念，并不是什么具体的技术或者平台。这个提法的出现其实有一个过程，就是在过去软件的架构说到底是基于数据库的（至于什么基于组件、基于领域等概念，其实是在应用范畴的，而不是架构范畴的概念），比如不同的两个系统的交互，往往是通过公用同一个数据库，或者通过Job等方式同步两个应用各自的数据，最终都是以数据为中心的。这种架构的优点是开发快速，与数据库紧密相连，事务性很好，适用于中小系统；缺点是因为各个系统都可以直接和数据库连接，层次不清晰，当系统越来越庞大时，运维成本越来越大，此外，其可控性、安全性、扩展性也相对较差。而SOA是以上缺点的一个很合适的解决方案，比如：基于开放的标准，使得可以跨平台调用（.NET, J2EE…）；基于自治的服务，便于安全性的控制和服务限流；基于契约，将各个子系统解耦。

接下来，详细回顾一下微软的所有分布式通信技术，包括如下4种具体技术。

**COM和DCOM**：COM基于组件设计，通过GUID唯一标识、IKnown与其他接口进行互操作，例如ActiveX，DCOM是COM的分布式版本，提供了可靠传输、安全等支持。

**.NET Remoting**：其基于信道栈的“管道式”消息处理和传输机制，支持TCP,UDP等传输协议。

**Web Service**：其提供跨平台的互操作性，构建在ASP.NET平台上，基于一系列开放的标准，包括XML、XSD、SOAP和WSDL等。此外，微软还通过WSE（Web Service Enhancement）组件为Web服务提供**WS-\***规范的支持。

**MSMQ**（Message Queuing）：MSMQ通过异步通信的方式，解耦了服务的提供者和调用者，为系统提供了可观的伸缩性和可用性，并支持可靠信息传输、错误处理和对事务的支持。

Tip:

J2EE架构其实也有相对应的技术，例如官方的Java RPC，WebService，JMS，第三方的Axis，RabbitMQ等。

本节最后通过一个非常简单的自寄宿的WCF示例来熟悉WCF的应用以及引入传说中的三要素”ABC”,Address服务地址、Binding服务绑定、Contract服务契约，之后将分节进行详细介绍

|  |
| --- |
| **Contract:**  [ServiceContract]  public interface IAddService  {  [OperationContract]  CompositeType Add(CompositeType a, CompositeType b);  }  [DataContract]  public class CompositeType  {  [DataMember]  public int PartA { get; set; }  [DataMember]  public string PartB { get; set; }  }  public class AddService : IAddService  {  public CompositeType Add(CompositeType a, CompositeType b)  {  return new CompositeType() { PartA = a.PartA + b.PartA, PartB = a.PartB + b.PartB };  }  }  **Host:**  static void Main(string[] args)  {  using (var host = new ServiceHost(typeof(AddService)))  {  host.Opened += (target, eventArgs) => Console.WriteLine("AddService已经启动，按任意键终止服务！");  host.Open();  Console.Read();  }  }  **Config:**  <system.serviceModel>  <behaviors>  <serviceBehaviors>  <behavior name="metadataBehavior">  <serviceMetadata httpGetEnabled="True" httpGetUrl="http://127.0.0.1:9901/addservice/metadata"/>  </behavior>  </serviceBehaviors>  </behaviors>  <services>  <service name="Sory.Entertainment.WCF.AddService" behaviorConfiguration="metadataBehavior">  <endpoint address="http://127.0.0.1:9901/addservice" binding="wsHttpBinding" contract="Sory.Entertainment.WCF.IAddService"/>  </service>  </services>  </system.serviceModel>  **Client:**  static void Main(string[] args)  {  using (var client = new WcfService.AddServiceClient())  {  var result = client.Add(new WcfService.CompositeType { PartA = 1, PartB = "Hello, " }, new WcfService.CompositeType { PartA = 2, PartB = "World!" });  Console.WriteLine(string.Format("PartA: {0}, PartB: {1}", result.PartA, result.PartB));  }  Console.Read();  } |

本节将介绍URI、端口共享、请求监听和消息分发等概念。正如之前所说的，WCF服务是通过终结点EndPoint发布，而终结点由地址、绑定和契约三要素组成，其中地址用于定位服务，并提供额外的寻址信息和认证信息。既然是服务定位，首先引入URI的概念，URI的全称为Uniform Resource Identifier统一资源标识，其形式是，[Schema传输协议]://[主机名|域名|IP地址]:[端口号]/[资源路经]，其中支持的协议类型如下表所示。

|  |  |
| --- | --- |
| 协议类型 | 解释 |
| HTTP/HTTP | 前者是互联网时代的核心--超文本传输协议，其是建立在TCP/IP协议簇上应用层协议。特点无状态、无连接、提供简单请求-回复消息传输方式；后者是采用了SSL（TLS）的HTTP，提供数据加密，实际上，大部分主流网站已实现全站HTTPS。 |
| Net.TCP | TCP全称传输控制协议，属于传输层协议，基于网络层IP协议，是应用层HTTP协议的基础。其特点是有状态、支持全双工、支持可靠通信，其是基于连接的协议，在数据传输前通过3次“握手”创建连接，在传输结束后，通过4次“握手”终止连接。 |
| Net.Pipe | 命名管道是Windows等操作系统实现跨进程通信(Inter Process Communication, IPC)的标准实现方式，虽然命名管道本身可以跨机器通信，不过WCF中的命名管道专注于同一台机器中的跨进程通信，因此其主机名为localhost，此外由于基于同一台机器，端口变得没有意义。 |
| Net.Msmq | 消息队列提供了支持离线的通信机制，其包括公共消息队列和私有消息队列两种方式，前者需要注册到AD域中。此外，除了存储业务数据消息的普通队列之外，还有存储消息拷贝的日志队列、存储确认消息的管理队列、存储回复消息的回复队列和存储死信消息的死信队列等。  其URI格式为： net.msmq://sory.com/private/xxxservice |

之前提及的核心概念终结点在WCF中，通过System.ServiceModel.Description.ServiceEndpoint类表示，其包括Address、Binding、Contract三个核心属性。其中的Address是EndpointAddress的实现类，其包含**Uri**、**Headers**、Identity三个属性，Uri即是服务的唯一标识，也是服务的目标地址，且这个地址可以使物理的，也可以是逻辑的。这儿的Headers其实就是SOAP消息中的消息头(类似于Http协议的，也包括消息头和消息体，前者主要提供一些控制信息，后者存放数据部分)，它默认通过DataContractSerializer进行序列化和反序列化，最终转化为SOAP消息的MessageHeader，相应配置如下所示，添加了服务端消息头后，在客户端也需要增加相应消息头，否则会被地址过滤器给过滤掉（之后的客户端通过ChannelFactory调用服务的示例中可以看到）。

|  |
| --- |
| <endpoint address="http://127.0.0.1:9901/addservice" binding="wsHttpBinding" contract="Sory.Entertainment.WCF.IAddService">  <headers>  <authentication xmlns="http://www.sory.com/">{12345678}</authentication>  </headers>  </endpoint> |

补充一点的是，可以通过将服务的ServiceBehavior特性中的AddressFilterModel属性设置为Any，跳过消息头的检验。

在基础概念一节的代码示例中，可以看到WCF通过ServiceHost完成服务寄宿，其中通过AddServiceEndpoint实现终结点的添加，当然也可以通过配置文件的方式添加终结点，在配置文件的<system.serviceModel>模块的<service>子节点中添加<endpoint>，并补全address、binding、contract属性，注意在IIS寄宿的情况下，无需提供address，因为.svc文件的地址就是服务的地址。同时，可以通过ServiceHost的Description属性（.NET中习惯使用Description获取元数据相关信息，无论是哪一种框架）获取终结点和服务行为的相关信息。

此外，除了使用绝对地址来指定某个服务的终结点地址外，还可以通过“基地址+相对地址”的方式，其配置形式如下，需要注意一种类型的协议只能有一个基地址，并且当一个服务实现类同时实现了多个服务接口时，该终结点地址可以共享。

|  |
| --- |
| <service name="XXX" behaviorConfiguration="XXX">  <host>  <baseAddresses>  <add baseAddress="net.tcp://127.0.0.1/baseservice"/>  </baseAddresses>  </host>  </service> |

客户端通过服务代理实现对服务的调用，包括两种方式：通过服务引用或者借助SvcUtil.exe工具来生成服务代理类，该生成类继承自ClientBase<TChannel>；直接通过ChannelFactory<TChannel>创建服务代理。前者比较简单，只需要在<system.serviceModel>的子节点<client>中添加对应的<endpoint>节点，然后直接生成的对应的Client类即可，后者如下所示。

|  |
| --- |
| var uri = new Uri("http://127.0.0.1:9901/addservice");  var header = **AddressHeader.CreateAddressHeader**("authentication", "http://www.sory.com/", "{12345678}");  var address = new EndpointAddress(uri, header);  var binding = new WSHttpBinding();  var contract = ContractDescription.GetContract(typeof(IAddService));  var endpoint = new ServiceEndpoint(contract, binding, address);  using (var factory = new ChannelFactory<IAddService>(endpoint))  {  var channel = factory.CreateChannel();  var result = channel.Add(new CompositeType { PartA = 1, PartB = "Hello, " }, new CompositeType { PartA = 2, PartB = "World!" });  Console.WriteLine(string.Format("PartA: {0}, PartB: {1}", result.PartA, result.PartB));  } |

* 端口共享

在Windows系统，为了安全，常常只开发少量端口，当有大量应用需要使用不同端口时，会显得捉襟见肘，因此多个应用共享同一个端口显得很有必要。对于Http/Https协议来说，由于其可以通过IIS来管理应用，其自身通过HTTP.SYS已经实现了80|443端口的共享。而对于TCP协议来说，其通过一个Windows服务(名称为Net.Tcp Port Sharing Service)来管理，可以通过如下方式实现其共享。

|  |
| --- |
| <bindings>  <netTcpBinding>  <binding name="portSharingBinding" portSharingEnabled="true"></binding>  </netTcpBinding>  </bindings> |

* 逻辑地址和物理地址

之前在EndpointAddress中提及的Uri属性表示服务的逻辑地址，而物理地址对于服务端来说是监听地址，对于客户端来说是消息真正发送的目标地址。默认情况下，两个地址是统一的，但在需要中介进行消息转发的场景下，需要将两者分离。

对于服务端，可以设置终结点的ListenUri的属性和ListenUriMode属性（包括Explicit和Unique，前者严格使用ListenUri作为最终的监听地址，后者将通过不同的策略保证监听地址的唯一性，如针对端口共享的情况，将在默认Uri后加GUID以作识别），共同完成该需求，示例如下。

示例如下。

|  |
| --- |
| <endpoint address="http://127.0.0.1:9901/addservice" listenUri="http://127.0.0.1:9900/addservice" listenUriMode="**Unique**" …/> |

对于客户端,需要借助ClientViaBehavior这一终结点行为来实现，示例如下。

|  |
| --- |
| <behaviors>  <endpointBehaviors>  <behavior name="clientViaBehavior">  <clientVia viaUri="http://127.0.0.1:9900/addservice"/>  </behavior>  </endpointBehaviors>  </behaviors>  <client >  <endpoint behaviorConfiguration="clientViaBehavior"></endpoint>  </client> |

补充：**行为**这个概念在WCF中非常的重要，很多的功能都是通过相应的行为实现的，接下来进行简要介绍。如果说契约是客户端和服务端达成的某种共识，是双边协议，而行为则是客户端或服务端在本地实现某个功能的一种方式，是一种单边行为。WCF提供了4种类型的行为，包括服务行为、契约行为、终结点行为和操作行为，它们一般可以通过特性或者配置文件的方式进行设置。

* 请求监听和消息分发

这部分内容涉及到整个WCF服务端的架构，下图展示了一个最简单的请求分发过程。



在整个消息监听和分发体系中，信道分发器和终结点分发器是两个核心的对象，前者负责请求监听、消息接收并通过消息筛选器选择正确的终结点，后者完成消息的处理。终结点分发器具有两个消息消息筛选器，分别是AddressFilter和ContractFilter，均是MessageFilter类型，前者对应的AddressFilterMode包含Exact、Prefix、Any三种枚举类型。WCF提供6种典型的消息筛选器，包括:**ActionMessageFilter**，判断请求消息(SOAP)的<Action>报头是否和终结点契约中任意操作的Action属性相匹配(Match)；**EndpointAddressMessageFilter**判断<To>报头是否和终结点地址相匹配；MatchAllMessageFilter，表示全匹配；以及不常用的XPathMessageFilter、MatchNoneMeesageFilter和PrefixEndpointAddressMessageFilter。

从基础架构的角度上看，WCF可以分为服务模型层和信道层两个层次，服务模型层建立在信道层的基础是上，而信道层就是通过本节即将介绍的binding绑定创建，注意这儿的绑定与.NET很多地方的绑定概念不同（例如最常见的数据绑定），注意理解。那么binding是如何创建信道层的呢？它通过组合不同的信道，将其整合为一个指定的信道栈，这个过程其实就是一个职责链模式的实现，每个信道都只处理自己的一部分内容，最基本的有传输、编码，复杂一些的包括事务流转、安全传输和可靠传输，使得整个框架足够灵活，已于扩展，一个支持WS-\*的信道栈如下图所示。



其中传输信道实现了基于某种协议的消息传输，消息编码信道实现了消息的编码（例如XML、Binary、MTOM），而WS-AT(WS-Atomic Transaction)实现了分布式的事务支持，WS-RM(WS-Reliable Messaging)实现了信息的可靠传输，WS-Security实现了消息的传输安全，他们都可以被称为协议信道。接下来通过一个简单的例子来演示通过绑定进行消息通信，在其中将引入信道、信道监听器、信道工厂等主要对象。

|  |
| --- |
| 服务端：  static void Main(string[] args)  {  var listenUri = new Uri("http://127.0.0.1:9902/listener");  var binding = new BasicHttpBinding();  //创建和开启信道监听器  var channelListener = binding.BuildChannelListener<IReplyChannel>(listenUri);  channelListener.Open();  //创建、开启回复信道  var channel = channelListener.AcceptChannel(TimeSpan.MaxValue);  channel.Open();  //开始监听  while (true)  {  //接受输入请求信息  var requestContext = channel.ReceiveRequest(TimeSpan.MaxValue);  Console.WriteLine(requestContext.RequestMessage);  requestContext.Reply(CreateReplyMessage(binding));  }  }  private static Message CreateReplyMessage(Binding binding)  {  var action = "http://www.sory.com/addservice/AddResponse";  XNamespace ns = "http://www.sory.com";  XElement body = new XElement(new XElement(ns + "AddResponse", new XElement(ns + "AddResult", 3)));  return Message.CreateMessage(binding.MessageVersion, action, body);  }  客户端：  static void Main(string[] args)  {  var listenUri = new Uri("http://127.0.0.1:9902/listener");  var binding = new BasicHttpBinding();  //创建和开启信道工厂  var channelFactory = binding.BuildChannelFactory<IRequestChannel>();  channelFactory.Open();  //创建、开启请求信道  var channel = channelFactory.CreateChannel(new EndpointAddress(listenUri));  channel.Open();  //发送请求消息，接受回复消息  var replyMessage = channel.Request(CreateRequestMessage(binding));  Console.WriteLine(replyMessage);  Console.Read();  }  private static Message CreateRequestMessage(Binding binding)  {  var action = "http://www.sory.com/addservice/Add";  XNamespace ns = "http://www.sory.com";  XElement body = new XElement(new XElement(ns + "Add", new XElement(ns + "x", 1), new XElement(ns + "y", 2)));  return Message.CreateMessage(binding.MessageVersion, action, body);  } |

通过这个例子看起来很像以前的Window网络编程中的Socket编程形式，首先服务端监听，然后客户端请求，服务端接收并绑定Socket(这儿是绑定信道)，之后就可以在此基础上进行通讯了。这部分涉及到的类型很多，接下来通过一个表格简述部分主要类，浏览即可。

|  |  |
| --- | --- |
| 类别 | 介绍 |
| 信道与信道栈 | 最基础的ICommunicationObject接口，提供统一管理通信对象的状态机，可以作为一种设计范例用于实际项目中；DefaultCommunicationTimeouts类负责控制超时时限；IChannel和ChannelBase用于表示信道；ISession和ISessionChannel<TSession>用于表示会话信道。此外，支持3种消息交换模式。  数据报Datagram模式：一般使一部的消息发送方式，支持1或多个接收者，对应IOutputChannel, IInputChannel  请求-回复模式：对应IRequestChannel、IReplyChannel  双工模式：对应IDuplexChannel |
| 信道监听器(Server) | IChannelListener, ChannelListenerBase |
| 信道工厂(Client) | IChannelFactory, ChannelFactoryBase |

最后，进入绑定元素与绑定的介绍，之前提到过，绑定是用于创建信道栈的，而它其中的绑定元素则是用于创建具体的信道的。常见的系统绑定包括：BasicHttpBinding、WSHttpBinding、WS2007HttpBinding、WSDualHttpBinding、NetTcpBinding、NetNamedPipeBinding和NetMsmqBinding。其中BasicHttpBinding最为基础，在构建类似web服务形式的应用中使用最多，所有带Net前缀的绑定将局限于.NET平台，不同的绑定的运行效率有不小差异。一般来说，企业内部的服务推荐使用RPC类型的服务，如NetTcpBinding，而对外服务推荐使用WSHttpBinding，当然实际项目中，对外服务一般不会使用WCF框架，而是使用Restful风格的WebAPI。此外，也可以建立自定义的绑定，将框架提供的绑定元素进行重新组合，更有甚者，可以自定义绑定元素，不过这部分内容使用的场景非常的少。最后，提供一个简单自定义绑定配置作为参考，其组合了传输、编码和安全3个绑定元素，前两者是必选项，且必须按照顺序构建。

|  |
| --- |
| <bindings >  <customBinding>  <binding name="testBinding">  <security></security>  <textMessageEncoding></textMessageEncoding>  <tcpTransport></tcpTransport>  </binding>  </customBinding>  </bindings> |

契约其实就是一个生活中的概念，是一种双边和多边的协议，在WCF中，其保证了无论服务的实现有任何的改变，而服务的消费者始终可以通过契约约定方式来调用服务。由于整个WCF都是基于SOAP以及WS-\*的，因此其XML是数据格式标准，通过XSD控制XML的数据结构，用WSDL（web服务描述语言）来提供跨平台的描述服务。

服务契约的定义通过ServiceContractAttribute和OperationContractAttribute两个特性来定义，前者定义整个服务，后者定义服务中具体的方法，接下来具体介绍一下这两个类。ServiceContractAttribute类，比较重要的属性包括：Name，可以定义服务的名称，默认为接口名；Namespace定义服务的命名空间，可以使用自己的公司名和项目名的组合来设定，其和之前的Name在wsdl文件中均是对<portType>元素的修饰；ConfigurationName实际上就对应配置中的Contract名称；SessionMode表示契约的会话模式，比如Allowed、Required等；ProtectionLevel表示消息的保护级别；CallbackContract在双工通信时指定回调操作的接口类型。OperationContractAttribute类，其属性Name、Namespace、ProtectionLevel与之前相似，值得一提的属性包括：Action/ReplyAction用于控制某个操作请求/回复信息的<Action>头，其默认通过命名空间、服务契约、操作名称组成，后者默认添加Response；IsOneWay控制消息交换的模式。提到消息交换的模式，记得之前提到过主要的三种请求-回复、单向和双工，前两项之前的例子中已有展示，之后的示例将展示双工模式。

|  |
| --- |
| 服务端：  public interface IAddCallback  {  [OperationContract]  void DisplayResult(CompositeType result, CompositeType a, CompositeType b);  }  [ServiceContract(CallbackContract=typeof(IAddCallback))]  public interface IAddService  {  [OperationContract]  void Add(CompositeType a, CompositeType b);  }  [DataContract]  public class CompositeType  {  [DataMember]  public int PartA { get; set; }  [DataMember]  public string PartB { get; set; }  }  public class AddCallbackService : IAddCallback  {  public void DisplayResult(CompositeType result, CompositeType a, CompositeType b)  {  Console.WriteLine("x + y = {2} when x= {0} and y = {1}", a.PartA, b.PartA, result.PartA);  }  }  public class AddService : IAddService  {  public void Add(CompositeType a, CompositeType b)  {  var result = new CompositeType() { PartA = a.PartA + b.PartA, PartB = a.PartB + b.PartB };  IAddCallback callback = OperationContext.Current.GetCallbackChannel<IAddCallback>();  callback.DisplayResult(result, a, b);  }  }  配置：  <system.serviceModel>  <behaviors>  <serviceBehaviors>  <behavior name="metadataBehavior">  <serviceMetadata httpGetEnabled="True" httpGetUrl="http://127.0.0.1:9901/addservice/metadata"/>  <serviceDebug includeExceptionDetailInFaults="true"/>  </behavior>  </serviceBehaviors>  </behaviors>  <services>  <service name="Sory.Entertainment.WCF.AddService" behaviorConfiguration="metadataBehavior">  <endpoint address="net.tcp://127.0.0.1:1001/addservice" binding="netTcpBinding" contract="Sory.Entertainment.WCF.IAddService"/>  </service>  </services>  </system.serviceModel>  客户端：  InstanceContext callback = new InstanceContext(new AddCallbackService());  using (DuplexChannelFactory<IAddService> channelFactory = new DuplexChannelFactory<IAddService>(callback, "addservice"))  {  var addChannel = channelFactory.CreateChannel();  addChannel.Add(new CompositeType { PartA = 1 }, new CompositeType { PartA = 2 });  }  配置：  <system.serviceModel>  <client>  <endpoint name="addservice" address="net.tcp://127.0.0.1:1001/addservice" binding="netTcpBinding" contract="Sory.Entertainment.WCF.IAddService"/>  </client>  </system.serviceModel> |

当调用以上示例的服务时，会抛出一个关于死锁的异常，原因是其在并发场景下会造成回调死锁的情况，可以通过将请求或回调方法设置为单向即可。

此外，服务契约是不支持继承的，而操作契约支持继承，不过这部分也不太常用，而与契约相关的元数据描述类也非常简单，这儿就不展开介绍了。

* 多线程和异步操作

在《CLR via C#》中，将操作分为计算限制的和I/O限制的，一般来说，WCF中主要涉及到I/O限制的操作，这种类型的操作主要是通过**异步模型**来提高其并发性。谈到异步操作，在SOA这类应用中包含3个不同异步场景，这部分知识比较有意思，曾经困到鄙人多年。这3中场景包括：异步的信道调用，客户端可以通过代理对象异步的调用信道；单向消息交换，客户端的信道通过单向的消息交换模式向服务端发送消息，发送立刻返回；异步服务实现，服务端在具体实现服务操作时，采用异步调用的方式。

异步服务代理的创建，可以通过在添加服务引用时通过高级选项添加生成异步操作选项，之后可以通过使用BeginXX/EndXX方法、回调和事件注册等方式使用异步服务代理类。而异步的服务实现可以在服务接口中将原有方法修改为BeginXXX/EndXXX形式的异步方法名，并将OperationContract契约的AsyncPattern属性设置为true即可。

* 操作的选择与执行

之前提及的契约描述类中的Operations列表只包含了被OperationContractAttribute特性修饰的服务操作，而运行时的操作是通过DispatchOperation和ClientOperation两个类型表示。DispatchOperation在服务端的终结点分发器初始化时建立一个DispatchRuntime类，其通过一个SynchronizedKeyedCollection<string, DispatchOperation>集合类型来管理所有的运行时分发操作，OperationSelector用于操作选择,IOperationInvoker用于操作执行。ClientOperation和前者的结构基本一致，只不过它用于客户端而已。

Tip:在实际中，很多公司选用ServiceStack的开源架构来构建的自身的SOA服务，此外，过去也常常以通过WebService搭建企业服务总线ESB的方式构建SOA服务。这部分推荐两位大神的博文，寒江独钓的<http://www.cnblogs.com/yangecnu/p/Introduce-ServiceStack.html>和张善友的<http://www.cnblogs.com/shanyou/p/3348347.html>。

最后，分享一个好玩的东西，就是在微信中可以搜索微软的“小冰”（刚截稿前对面的程序媛告诉我的，挺逗的，能挖掘你的内心哦），然后就可以在编码无聊、寂寞空虚时…你懂得，哈哈！

参考资料：

[1]蒋金楠. WCF全面解析[M]. 上海:电子工业出版社, 2012.