**概述**

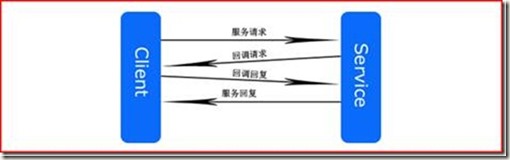
双工（Duplex）模式的消息交换方式体现在消息交换过程中，参与的双方均可以向对方发送消息。基于双工MEP消息交换可以看成是多个基本模式下（比如请求-回复模式和单项模式）消息交换的组合。双工MEP又具有一些变体，比如典型的订阅-发布模式就可以看成是双工模式的一种表现形式。双工消息交换模式使服务端回调（Callback）客户端操作成为可能。

在Wcf中不是所有的绑定协议都支持回调操作，BasicHttpBinding，WSHttpBinding绑定协议不支持回调操作；NetTcpBinding和NetNamedPipeBinding绑定支持回调操作；WSDualHttpBinding绑定是通过设置两个HTTP信道来支持双向通信，所以它也支持回调操作。

**两种典型的双工MEP**

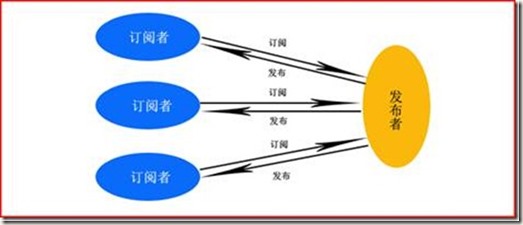
**1.请求过程中的回调**

这是一种比较典型的双工消息交换模式的表现形式，客户端在进行服务调用的时候，附加上一个回调对象；服务在对处理该处理中，通过客户端附加的回调对象（实际上是调用回调服务的代理对象）回调客户端的操作（该操作在客户端执行）。整个消息交换的过程实际上由两个基本的消息交换构成，其一是客户端正常的服务请求，其二则是服务端对客户端的回调。两者可以采用请求-回复模式，也可以采用单向（One-way）的MEP进行消息交换。下**图**描述了这样的过程，服务调用和回调都采用请求-回复MEP。

[](http://images.cnitblog.com/blog/243452/201312/05183159-fffa5c8022f84d8a818e12c7d4aa396d.jpg)

**2.订阅-发布**

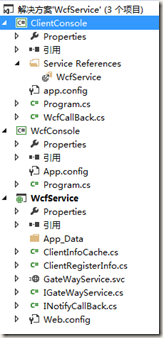
订阅-发布模式是双工模式的一个典型的变体。在这个模式下，消息交换的双方变成了订阅者和发布者，若干订阅者就某个主题向发布者申请订阅，发布者将所有的订阅者保存在一个订阅者列表中，在某个时刻将主题发送给该主题的所有订阅者。实际上基于订阅-发布模式的消息交换也可以看成是两个基本模式下消息交换的组合，申请订阅是一个单向模式的消息交换（如果订阅者行为得到订阅的回馈，该消息交换也可以采用请求-回复模式）；而主题发布也是一个基于单向模式的消息交换过程。订阅-发布消息交换模式如下**图**所示。

[](http://images.cnitblog.com/blog/243452/201312/05183204-570c3a4e2b144eda9a8821b1e815a12c.jpg)

**示例**

接下来我们将会创建一个简单的Wcf通信服务，包括使使用NetTcpBinding实现双工通信，和监控双工通信过程中的客户端和服务端一方断开后的捕捉事件。

项目如图所示

[](http://images.cnitblog.com/blog/243452/201312/05183423-c0d385e3c30947518392cad8bef846d8.png)

**第一步：**

先创建IGateWayService和INotifyCallBack接口

[ServiceContract(CallbackContract = typeof(INotifyCallBack))]

public interface IGateWayService

{

[OperationContract]

void RegisterClient(string clientName);

[OperationContract]

string GetData(int value);

[OperationContract]

CompositeType GetDataUsingDataContract(CompositeType composite);

}

// 使用下面示例中说明的数据约定将复合类型添加到服务操作。

[DataContract]

public class CompositeType

{

bool boolValue = true;

string stringValue = "Hello ";

[DataMember]

public bool BoolValue

{

get { return boolValue; }

set { boolValue = value; }

}

[DataMember]

public string StringValue

{

get { return stringValue; }

set { stringValue = value; }

}

}

INotifyCallBack.cs如下：

public interface INotifyCallBack

{

[OperationContract(IsOneWay = true)]

void NotifyFunction(string sender);

}

记住在IGateWayService接口上方设置Attribute [ServiceContract(CallbackContract = typeof(INotifyCallBack))] 这样设置表示这个接口是支持回调的。

接下来定义一个ClientRegisterInfo.cs来定义客户端的名字和客户端的INotifyCallBack属性，再定义一个Timer 来调用INotifyCallBack给客户端发送消息。再通过

wcf 的ICommunicationObject来定义通信出错和关闭的事件。

public class ClientRegisterInfo

{

public ClientRegisterInfo()

{

\_senderTimer.Elapsed += OnSenderMessage;

\_senderTimer.Start();

}

private void OnSenderMessage(object sender, ElapsedEventArgs e)

{

if (\_notifyCallBack != null)

{

var communication = \_notifyCallBack as ICommunicationObject;

if(communication.State==CommunicationState.Opened)

\_notifyCallBack.NotifyFunction(DateTime.Now.ToString());

}

}

public Timer \_senderTimer=new Timer(10\*1000);

private INotifyCallBack \_notifyCallBack;

public INotifyCallBack NotifyCallBack

{

get { return \_notifyCallBack; }

set

{

lock (\_syncNotifyObj)

{

\_notifyCallBack = value;

if (\_notifyCallBack != null)

{

var communication = \_notifyCallBack as ICommunicationObject;

if (communication != null)

{

communication.Closed += OnChannelClose;

communication.Faulted += OnChannelFault;

}

}

}

}

}

private readonly object \_syncNotifyObj = new object();

private void OnChannelFault(object sender, EventArgs e)

{

ClientInfoCache.Instance.Remove(this);

}

private void OnChannelClose(object sender, EventArgs e)

{

ClientInfoCache.Instance.Remove(this);

}

public string ClientName { get; set; }

}

再定义一个单例来保存客户端的信息。

public class ClientInfoCache

{

private static readonly object SyncObj = new object();

private static ClientInfoCache \_instance;

public static ClientInfoCache Instance

{

get

{

lock (SyncObj)

{

if (\_instance == null)

\_instance = new ClientInfoCache();

}

return \_instance;

}

}

private ClientInfoCache()

{

\_clientList = new List<ClientRegisterInfo>();

}

private List<ClientRegisterInfo> \_clientList;

private static object SyncOperator = new object();

/// <summary>

/// Add client entity

/// </summary>

/// <param name="entity">client entity</param>

public void Add(ClientRegisterInfo entity)

{

if (entity == null) return;

lock (SyncOperator)

{

var findClient =

\_clientList.FirstOrDefault(

t => t.ClientName.Equals(entity.ClientName, StringComparison.OrdinalIgnoreCase));

if (findClient == null)

\_clientList.Add(entity);

else

{

findClient.NotifyCallBack = entity.NotifyCallBack;

}

}

}

/// <summary>

/// Remove client

/// </summary>

/// <param name="entity">Client entity</param>

public void Remove(ClientRegisterInfo entity)

{

lock (SyncOperator)

{

\_clientList.Remove(entity);

}

}

}

再新建个控制台运应程序来启动Wcf,代码如下：

public class Program

{

static void Main(string[] args)

{

StartListener();

}

private static void StartListener()

{

try

{

using (var host = new ServiceHost(typeof(GateWayService)))

{

host.Opened += delegate

{

Console.WriteLine("[Server] Begins to listen request on " + host.BaseAddresses[0]);

};

host.Open();

Console.Read();

}

}

catch (Exception ex)

{

}

}

}

在App.config设置配置如下：

<?xml version="1.0" encoding="utf-8" ?>

<configuration>

<system.serviceModel>

<bindings>

<netTcpBinding>

<binding name="longTimeoutBinding" closeTimeout="01:10:00" openTimeout="01:10:00"

receiveTimeout="10:10:00" sendTimeout="10:10:00" maxBufferPoolSize="655350000"

maxBufferSize="655350000" maxReceivedMessageSize="655350000">

<readerQuotas maxDepth="32" maxStringContentLength="655350000"

maxArrayLength="655350000" maxBytesPerRead="655350000" maxNameTableCharCount="655350000" />

<reliableSession inactivityTimeout="23:59:59" />

<security mode="None" />

</binding>

</netTcpBinding>

</bindings>

<behaviors>

<serviceBehaviors>

<behavior name="NewBehavior">

<serviceMetadata httpGetEnabled="True" httpGetUrl="Http://localhost:7789/" httpsGetEnabled="True"/>

<serviceDebug includeExceptionDetailInFaults="False" />

<serviceThrottling maxConcurrentCalls="1000" maxConcurrentSessions="1000" maxConcurrentInstances="1000" />

</behavior>

</serviceBehaviors>

</behaviors>

<services>

<service name="WcfService.GateWayService" behaviorConfiguration="NewBehavior" >

<endpoint address="net.tcp://localhost:7788/GatewayService.svc" binding="netTcpBinding" contract="WcfService.IGateWayService" name="WcfService\_GateWayService" bindingConfiguration="longTimeoutBinding" >

</endpoint>

<endpoint address="mex" binding="mexTcpBinding" contract="IMetadataExchange" ></endpoint>

<host >

<baseAddresses >

<add baseAddress="net.tcp://localhost:7788/GatewayService.svc" />

<add baseAddress="Http://localhost:7789/" />

</baseAddresses>

</host >

</service>

</services>

</system.serviceModel>

</configuration>

longTimeoutBinding是设置传输的属性，如最大传输大小，TimeOut的时间等。

在客户端新建个WcfCallBack.cs 继承IGateWayServiceCallback接口,代码如下。

[CallbackBehavior(ConcurrencyMode = ConcurrencyMode.Multiple)]

public class WcfCallBack : IGateWayServiceCallback

{

public void NotifyFunction(string sender)

{

Console.WriteLine("Get a message,message info is {0}", sender);

}

}

设置属性[CallbackBehavior(ConcurrencyMode = ConcurrencyMode.Multiple)]表示服务器是通过并发的给客户端来发送消息的。

控制台代码如下

class Program

{

private static GateWayServiceClient \_client;

static void Main(string[] args)

{

var cb = new WcfCallBack();

var context = new InstanceContext(cb);

\_client = new GateWayServiceClient(context);

\_client.RegisterClient("Test1");

((ICommunicationObject)\_client).Closed += OnChannelClose;

((ICommunicationObject)\_client).Faulted += OnChannelFaulted;

Console.WriteLine("Input Q to exit.");

while (string.Compare(Console.ReadLine(), ConsoleKey.Q.ToString(), StringComparison.OrdinalIgnoreCase) != 0)

{

}

}

private static void OnChannelFaulted(object sender, EventArgs e)

{

if (FaultedEvent != null)

FaultedEvent(sender, e);

}

private static void OnChannelClose(object sender, EventArgs e)

{

if (CloseEvent != null)

CloseEvent(sender, e);

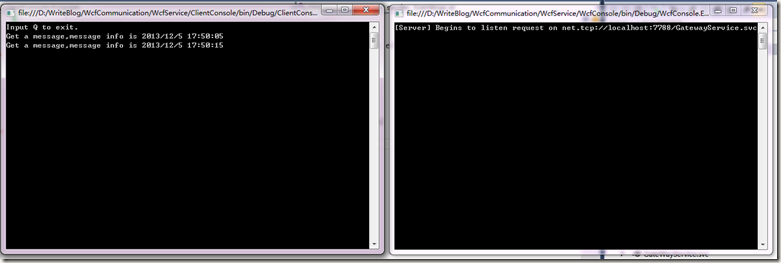
}

public static EventHandler CloseEvent;

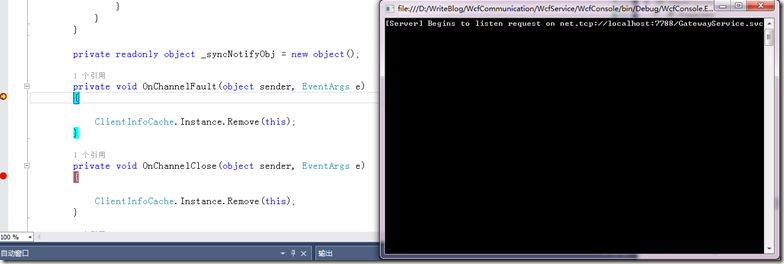
public static EventHandler FaultedEvent;

}

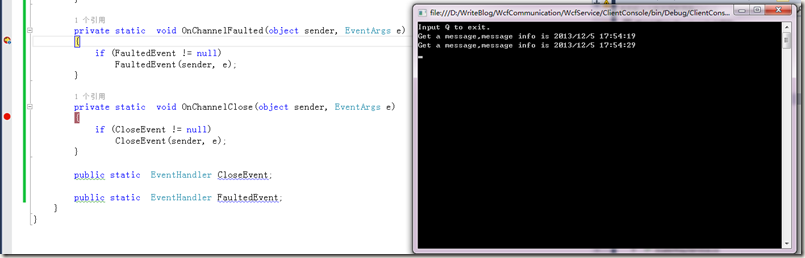
运行的结果如下图：

[](http://images.cnitblog.com/blog/243452/201312/05183659-4913dd4573ef4cb3bea752462d0e85bd.png)

当我关闭客户端时，能捕捉到Closed和Faulted事件

[](http://images.cnitblog.com/blog/243452/201312/05183712-2264116203d645bea51f924fb47e328e.png)

当我关闭服务端时，在客户端能捕捉到Faulted事件

[](http://images.cnitblog.com/blog/243452/201312/05183814-9614053f9f804e949f08d4cb07806ca9.png)

http://www.cnblogs.com/springyangwc/p/3460178.html