**概述**

* 事务是一个最小的工作单元，不论成功与否都作为一个整体进行工作。
* 由于事务是由几个任务组成的，因此如果一个事务作为一个整体是成功的，则事务中的每个任务都必须成功。如果事务中有一部分失败，则整体事务失败。 当事务失败时，系统返回到事务开始前的状态。这个取消所有变化的过程称为“回滚”( rollback )。
* 例如，如果一个事务成功更新了两个表，在更新第三个表时失败，则系统将两次更新恢复原状，并返回到原始的状态。

**理解事务特性**

* **一个纯粹的事务包含4个特性**
  + **Atomic（原子性） 不可分割性**
  + **Consistent（一致性） 事务的结果和预期应该是一样的**
  + **Isolated（隔离性） 事务是私有的，对其他事务是不可见的。**
  + **Durable（持久性） 一旦事务提交成功，数据的修改时永久的。**

**[TransactionFlow] Attribute**

* **[TransactionFlow]是提定服务操作可以接收事务的模式，该Attribute只有一个特性用来标记服务操作的方法。即：**TransactionFlowOption
  + TransactionFlowOption是一个枚举型，包括三个枚举项
    - NotAllowed：不允许事务，是缺省值；
    - Allowed：允许事务，意味着事务可有可无；
    - Mandatory：强制事务，表示事务是必须的。

[TransactionFlow(TransactionFlowOption.Mandatory)]

int serviceMethod(int value)

{

}

**[ServiceBehavior]事务属性3-1**

TransactionAutoCompleteOnSessionClose

如果想要确保关闭会话后待处理的消息仍然可以完成，应该使用该属性。根据其属性值，事务将会在会话关闭后提交或回滚。

**[ServiceBehavior]事务属性3-2**

TransactionIsolationLevel

用于指示事务隔离级别。

IsolationLevel枚举如下：

ReadUncommitted : 未提交读

ReadCommitted ：已提交读

RepeatableRead ：可重复读

Serializable ：可序列化

ReadUncommitted是最低的隔离级别

Serializable是最高的隔离级别

**[ServiceBehavior]事务属性3-3**

TransactionTimeout

用于指示事务的超时时间，默认为TimeSpan.Zero,表示不会受超时时间的限制

[ServiceBehavior(TransactionAutoCompleteOnSessionClose=true,TransactionIsolationLevel=IsolationLevel.ReadCommitted,TransactionTimeout="00:00:30")]

public Class ServiceClass:IServiceClass

{

}

**理解事务隔离级别2-1**

* 脏读：一个事务会读进还没有被另一个事务提交的数据，所以你会看到一些最后被另一个事务回滚掉的数据。
* 非可重复性读取：一个事务读进一条记录，另一个事务更改了这条记录并提交完毕，这时候第一个事务再次读这条记录时，它已经改变了。
* 幻像读：一个事务用Where子句来检索一个表的数据，另一个事务插入一条新的记录，并且符合Where条件，这样，第一个事务用同一个where条件来检索数据后，就会多出一条记录。

**理解事务隔离级别2-2**

ReadUncommitted：读取未提交数据，该方式在读取数据时保持共享锁定以避免读取已修改的数据，但在事务结束前可以更改这些数据，这导致非可重复读取或幻读。

ReadCommitted：读取提交数据, 发出共享锁定并允许非独占方式的锁定。该方式与读取未提交数据相相似，这种方式看似和读取未提交数据相似，但有一个区别，事务的只读锁在移到下一行的时候，会解锁，而写入锁却只有在事务完成或者被中止后才解锁，事务要等待所有写入锁解锁。

RepeatableRead：可重复性读取，与读取提交数据相似，在查询中使用的所有数据上放置锁，以防止其他用户更新这些数据。防止非可重复读取，但幻读行仍有可能发生。该方式是只读锁也要等到事务结束或者中止才解除

示例：

* Client/Service transaction，最常见的一种事务模型，通常由客户端或服务本身启用一个事务。

设置步骤：

* + (1) 选择一个支持事务的Binding，设置 TransactionFlow = true。
  + (2) 设置 TransactionFlow(TransactionFlowOption.Allowed)。
  + (3) 设置 OperationBehavior(TransactionScopeRequired=true)。

做一个银行转帐的例子：

1. 先创建一个数据表 Account 两个字段 ID balance
2. 创建wcf服务库

接口：

[ServiceContract]

public interface IService1

{

[OperationContract]

[TransactionFlow(TransactionFlowOption.Mandatory)]

void outMoney(int money);

[OperationContract]

[TransactionFlow(TransactionFlowOption.Mandatory)]

void intoMoney(int money);

}

实现：

[ServiceBehavior(InstanceContextMode=InstanceContextMode.Single,ReleaseServiceInstanceOnTransactionComplete=false)]

public class Service1 : IService1

{

[OperationBehavior(TransactionScopeRequired=true)]

public void outMoney(int money) //转出

{

using (SqlConnection conn = new SqlConnection("server=.;uid=sa;pwd=123;database=master"))

{

SqlCommand cmd = new SqlCommand("",conn);

conn.Open();

//得到余额

cmd.CommandText = "select balance from Account where id='A'";

decimal m = (decimal)cmd.ExecuteScalar();

if (m >= money)

{

string sql = "update Account set balance=balance-" + money + " where ID='A'";

cmd.CommandText = sql;

cmd.ExecuteNonQuery();

}

else

{

throw new FaultException("余额不足，转帐失败！");

}

conn.Close();

}

}

[OperationBehavior(TransactionScopeRequired = true)]

public void intoMoney(int money) //转入

{

using (SqlConnection conn = new SqlConnection("server=.;uid=sa;pwd=123;database=master"))

{

string sql = "update Account set balance=balance+" + money + " where ID='B'";

SqlCommand cmd = new SqlCommand(sql, conn);

conn.Open();

cmd.ExecuteNonQuery();

conn.Close();

}

//throw new FaultException("发生人为故障，转帐失败！");

}

}

同时配置文件需要添加绑定，允许事务流

<bindings>

<wsHttpBinding>

<binding name="wshttpbind\_transaction" transactionFlow="true"></binding>

</wsHttpBinding>

</bindings>

Endpoint中binbingConfiguration与上面绑定的名字相对应

<endpoint address ="" binding="wsHttpBinding" bindingConfiguration="wshttpbind\_transaction" contract="WcfServiceLibrary1.IService1">

1. 创建服务端窗体程序，添加相应引用，可启动服务。

//开启服务

ServiceHost host = null;

private void button1\_Click(object sender, EventArgs e)

{

WSDualHttpBinding bind = new WSDualHttpBinding();

bind.TransactionFlow = true;

Uri address = new Uri("http://localhost:3200/bank");

host = new ServiceHost(typeof(WcfServiceLibrary1.Service1), address);

host.AddServiceEndpoint(typeof(WcfServiceLibrary1.IService1), bind, "");

host.Open();

label1.Text = "服务已启动！";

}

1. 创建客户端程序，进行转账操作

//转帐

private void button1\_Click(object sender, EventArgs e)

{

WSDualHttpBinding bind = new WSDualHttpBinding();

bind.TransactionFlow = true;

bind.ClientBaseAddress = new Uri("http://localhost:5100/");

EndpointAddress address = new EndpointAddress("http://localhost:3200/bank");

ChannelFactory<WcfServiceLibrary1.IService1> factory = new ChannelFactory<WcfServiceLibrary1.IService1>(bind, address);

WcfServiceLibrary1.IService1 client = factory.CreateChannel();

int m = int.Parse(textBox1.Text);

using(System.Transactions.TransactionScope tx = new System.Transactions.TransactionScope())

{

try

{

client.outMoney(m);

client.intoMoney(m);

tx.Complete();//提交成功

MessageBox.Show("转帐成功！");

}

catch(FaultException fe)

{

MessageBox.Show(fe.Message);

System.Transactions.Transaction.Current.Rollback();//回滚

}

}

}