最近开始WCF相关知识的学习，虽然实际工作中使用公司自己的一套SOA系统，但微软的一套服务架构还是具有很大的参考意义。除了WCF的一些基础使用，相对比较复杂的内容有分布式的事务和通信的安全等，不过基本都和WS-协议簇相关联。为了引出WCF中的事务处理，今天优先介绍Windows下的事务处理模型，说实话用了很多年的TransactionScope，其实我一直都不知道它到底是如何运作的，回想Java中不管是EJB还是Spring也有和这部分类似的实现。今天通过学习蒋金楠老师的WCF服务框架解析来了解这部分的原理。

首先回忆事务的概念和相关应用机制。事务具有ACID（原子性，一致性，隔离性，持久性）特性，主要的应用场景比如银行转账，大部分的数据库都支持事务操作（SQL Server，Oracle，DB2，MySQL-InnoDB）。事务的实现最常见的有在SQL语句中使用事务BEGIN TRANSACTION, COMMIT TRANSACTION, ROLLBACK TRANSACTION，还可以通过ADO.NET的DbTransaction对象来控制事务，但这些只能支持单个链接的本地事务，而不能支持分布式事务。但实际的项目中，事务的参与者往往不会分布在同一个网络节点，有时是不同类型的事务资源（不仅仅是数据库，也可以是文件系统，注册表等）。分布式事务的典型业务场景有以下三类：将多个资源的访问纳入同一事务；将多个服务纳入同一事务（涉及事务在服务间的流转）；将多个资源和服务纳入同一事务(如下图所示)。



接下来进入核心内容Windows下的事务处理模型，事务处理模型一般包括三种角色：

应用（也包括服务、组件等），资源管理器RM（管理具体事务型资源的软件程序），事务管理器TM（管理整个事务的中间件程序）。事务是为了将一组相关操作作为一个不可分割的整体来执行，保证数据的一致性，实际上就是为应用服务的。应用主要负责开始事务&事务的封送(Marshaling)和传播(Propagation)、提交事务。

事务型资源的存取需要通过资源管理器，按照目标资源是否可以被持久化，相应的资源管理器分为：持久化(Durable)资源管理器，例如数据库管理器和小心队列，当事务回滚时具有可恢复性；易失(Volatile)资源管理器用于管理类似内存的资源，不具可恢复性。其主要职责是帮助应用实现对目标资源的操作、注册到相应的事务管理器便于回滚时接受恢复请求、向事务管理器报告本地事务的结果。事务管理器

事务管理器是整个模型的枢纽，协调所有的事务参与者，提供事务的开始、提交和回滚服务。Windows提供了三种不同的事务管理器，包括轻量级事务管理器LTM、内核事务管理器KTM和分布式事务协调器DTC。不过从这儿也可以看出事务服务并不具有跨平台特性，因而这部分内容主要作为参考学习，重在理解原理。轻量级事务管理器和内核事务管理器分别负责SQLSERVER和Windows文件&注册表，高效但不支持分布式。分布式事务协调器用于管理跨边界的分布式事务，支持Ole-Tx和WS-AT协议，每一台Windows计算机都具有唯一的DTC用于管理本地所有的资源管理器，支持分布式但效率较低。需要注意的是，事务开始默认LTM作为事务管理器，在其进行的过程中，根据实际的情形会进行事务提升（Transaction Promotion）。例如，涉及到注册表的读写就会升级到KTM，涉及到跨域的封送就会提升到DTC。

那么我们不经要问，分布式事务到底是如何实现的呢？其最关键的两个概念分别是事务登记&事务提交树（Transaction Commit Tree）和两阶段提交协议，接下来分别介绍这两个概念。

事务登记（Transaction Enlist）的目的在建立事务参与者之间的关系，促进相互间的协作，整个流程如下图所示：



以上的事务涉及两台机器，事务由ServiceA开启，并将其作为当前执行上下文的环境事务(Ambient Transaction)。当ServiceA调用本机资源管理器（如SQLSERVER）时，会将该RM纳入到本事务中。资源管理器RM向本机的DTC进行事务登记，从而DTC于RM建立起上下级关系。当ServiceA在调用ServiceB时，会将当前事务信息（分布式事务ID和本机DTC信息）封送，Service接到后取出信息重建事务，并将其设置为环境事务具有与原事务相同的ID。同时，ServiceB根据得到的MachineA的DTC消息，让本机DTC对MachineA的DTC进行事务登记，使得两台机器DTC建立上下级关系，之后将RM也纳入该DTC管理。在登记流程结束后就形成了如下事务提交树：



在分布式环境下，事务提交需要保证在操作成功时，所有需要持久化的数据被相应资源管理器RM写入目标资源，而失败时所有RM中数据要恢复到原始状态。为了实现这一目标，分布式事务提交需要采用“两阶段提交”协议，接下来详细介绍这两个阶段：

第一阶段--准备阶段，根节点的DTC向所有事务参与者发起请求，要求他们对本地事务的结果进行投票，如上图所示递归的进行消息传播，相应节点反馈就绪、只读、终止等投票类型。若所有投票结果为就绪和只读就代表提交，如果有任何一个为终止则代表终止提交。同时可以设置超时时限，若超过此时限整个事务进行回滚。

第二阶段—提交或者回滚，根节点DTC根据投票结果对整个事务发起提交或者终止操作，使用和第一阶段相同的消息传播方式完成操作。当遇到事务参与者在完成第一阶段投票后网络断开等异常情况时，该子事务处于“未决态”。再重启此事务后，该DTC会向上级询问最终结果，如果上级不能确认则继续向根节点传播，直到得到答复，若时间太长系统管理者可以强制提交或终止事务。

此外，为了提高提交性能，协议会根据节点所具有的下级节点数决定是否选用单阶段提交协议，就是在节点只有一个唯一下级时直接发起提交。

在完成了原理的剖析后，进入实际的应用，这也是与工作息息相关的部分。WCF基于DTC的Windows事务架构为基础，提供一个完善的分布式事务解决方案。提供了System.Transactions.Transaction的命令式编程模型和System.Transactions.TransactionScope的声明式编程方式。

Transaction类时可以序列化封送的，通过EnlistDurable和EnlistVolatile等方法将资源管理器登记到当前事务，构建事务提交树。Currrent属性表示当前的环境事务（Ambient）,其存储在当前线程的TLS中。TransactionInformation属性表示事务的基本信息，包括创建时间、状态、本地标识和分布式标识。并通过IsolationLevel表示隔离级别，使用Clone和Rollback方法克隆事务和回滚事务，需要注意的是事务的开始和结束需要同一个事务来完成，我们把这种事务称为可提交事务（Commitable Transaction），而其他的相关事务被称为依赖事务（DependentTransaction）。可提交事务通过TransactionOptions结构体设置超时时间和隔离级别，也可以通过如下配置设定。

|  |
| --- |
| <system.transactions>  <defaultSettings timeout="00:01:00"/>  <machineSettings maxTimeout="00:10:00"/>  </system.transactions> |

可提交事务通过Commit方法和异步Begin/EndCommit方法提交事务，例子如下所示：

|  |
| --- |
| private static void Transfer(string accountFrom, string accountTo, double amount) {  Transaction originalTransaction = Transaction.Current;  CommittableTransaction ct = new CommittableTransaction();  try  {  Transaction.Current = ct;  Withdraw(accountFrom, amount);  Deposit(accountTo, amount);  ct.Commit();  }  catch  {  ct.Rollback();  throw;  }  finally {  Transaction.Current = originalTransaction;  ct.Dispose();  }  } |

在Transaction类中存在一个DepedentClone的方法，该方法用于基于当前事务创建其所对应的依赖事务，也就是其所辖的子事务，将当前线程的环境事务传递到新的事务中。接下来的代码演示了通过依赖事务采用异步方式进行银行转账。需要注意的是，由于在调用DependentClone时指定的Options参数为BlockCommitUntilComplete，所以主线程在提交事务时，如果依赖事务未结束会一直等待到超时。还有一个关于事务型方法的实现也很有意思，大家有兴趣可以去看蒋老师的原著。

|  |
| --- |
| private static void Transfer(string accountFrom, string accountTo, double amount)  {  Transaction originalTransaction = Transaction.Current;  CommittableTransaction ct = new CommittableTransaction();  try  {  Transaction.Current = ct;  ThreadPool.QueueUserWorkItem(state => {  Transaction.Current = state as DependentTransaction;  try  {  Withdraw(accountFrom, amount);  Deposit(accountTo, amount);  (state as DependentTransaction).Complete();  }  catch (Exception ex) { Transaction.Current.Rollback(ex); }  finally {  (state as DependentTransaction).Dispose();  Transaction.Current = null;  }  }, Transaction.Current.DependentClone(DependentCloneOption.BlockCommitUntilComplete));  ct.Commit();  }  catch  {  ct.Rollback();  throw;  }  finally  {  Transaction.Current = originalTransaction;  ct.Dispose();  }  } |

最后介绍实际工作中最常见的TransactionScope类，如之前提到的事务型方法就有如下简易实现。

|  |
| --- |
| static void InvokeInTransaction(Action action)  {  using (TransactionScope ts = new TransactionScope())  {  action();  ts.Complete();  }  } |

需要注意的是TransactionScopeOption的设置，有如下三种类型：Required，表示如果存在环境事务则使用，否则进入范围前创建新的事务；RequiresNew，表示总是在该范围创建新事物；Suppress，表示屏蔽环境事务，即所有操作均在无环境事务的情况下执行。这个J2EE Spring中的REQUIRED 、REQUIRES\_NEW   、 NOT\_SUPPORTED很相似，不过对于J2EE与DTC相类似的机制个人还未涉猎，之后有机会再和大家分享了。