.NET 分散式交易程式開發 FAQ

作者: 李明儒

.NET 2.0 讓分散式交易的程式開發步入一個簡便與彈性兼備的新紀元，然而在實作過程中，還是免不了會有些顛簸崎嶇。本文以 FAQ 方式彙整分散式交易程式開發在實務上常遇到的問題，希望能提供一些指引。

1. 系統開發中常見的分散式交易(Distributed Transaction)情境爲何?

最常見的情境是多部資料庫伺服器間的資料異動必須包成一個交易。例如: 當兩台 SQL Server 或 SQL Server 與 Oracle Server 間的資料更動作業具有高度相關性時，兩台機器上的更新動作必須要一起成功或一起失敗，不允許 Oracle 資料表新增了匯出記錄，SQL Server 上卻未寫入匯入記錄這類狀況發生。

2. 分散式交易是如何做到的?

Windows 平台上有所謂的 MSDTC(Distributed Transaction Coordinator)，有能力協調多部資料庫伺服器完成兩階段式交易認可(Two-Phase Commit)，其中的實作原理及細節頗爲深奧，但一般.NET 開發者只需知道分散式交易由 DTC 負責執行管理即可。

3. 如何啓用分散式交易?

ASP 時代，我們可以在開頭宣告<%@ Transaction=Required%>，則 ASP 中全部的資料庫動作都會被包成分散式交易。但這樣不分青紅皂白地狂包一通往往不利於效能，如要有效控制啓用交易的範圍，一般標準的做法是建立一顆 Transaction Root COM+元件，Transaction Support Level 設爲"Required"，由它去呼叫要參與 Transaction 的 COM+元件(Transaction Support Level 設爲"Support")，如此，我們可以將 Transaction 發生的範圍，侷限在特定的 COM+元件間。

.NET 1.1 裡要實作分散式交易，需要寫一顆繼承自 System.EnterpriseService.ServicedComponent 的元件，內含更新資料庫的程式邏輯，再設定 TransactionAttribute，然後 Strong-Named/Signed，包上 COM+的皮，註冊放入 COM+ Application 中，原則上還是借用了前述 COM+的運作原理，但手續更加繁瑣。

.NET 2.0 推出 System.Transactions.TransactionScope，讓開發者可以重溫當年 ASP 時代只需宣告，資料庫存取程式碼完全不需修改就能參與交易的方便性，更由於 TransactionScope 可以自由宣告要包成交易的 Code 區段範圍，應用的彈性又更上一層樓。

4. 原本好好的程式碼，啓用分散式交易後就出現連不上資料庫的錯誤?

啓用分散式交易後，程式存取資料庫的方式由原先直接連線資料庫特定 TCP Port(例如: SQL Server 預設爲 1433 Port)，變成要由 MSDTC 透過 RPC 連線與 SQL Server 資料庫溝通或需依賴其他特定服務(例如: 必須要安裝 Oracle Services for MTS 才能讓 Oracle 參與分散式交易)，由於連線方式改變，切換到分散式交易後就無法連線資料庫的狀況還挺常見的。

以 RPC 連線爲例，很容易受到 Client 與 Database Server 間防火牆的限制而無法連通，出現 New transaction cannot enlist in specified transaction coordinator (新增的交易無法編列在指定的交易協調器中) 或 Error 8004d00a.Distributed Transaction error (錯誤 8004d00a。分散式交易錯誤)兩種錯誤訊息。

關於 MSDTC 穿防火牆的問題，微軟有一篇專門的文章提示如何排除。(http://support.microsoft.com/kb/306843/zh-tw) KB 中提到的 DTCPing 是處理 DTC 問題很基本而有效的檢測工具。

5. **系統在 Windows 2003 SP1 上，連線本機 SQL Server 的分散式交易 OK，但連到遠端 SQL 就不行?**

Windows 2003 的預設安裝不允許透過網路進行分散式交易，因此要額外選取安裝"啓用網路 DTC 存取"選項(圖 1)，或是從元件管理中選擇"啓用網路 DTC 存取"(圖 2 Security Configuration 上方選項)亦有同樣效果:

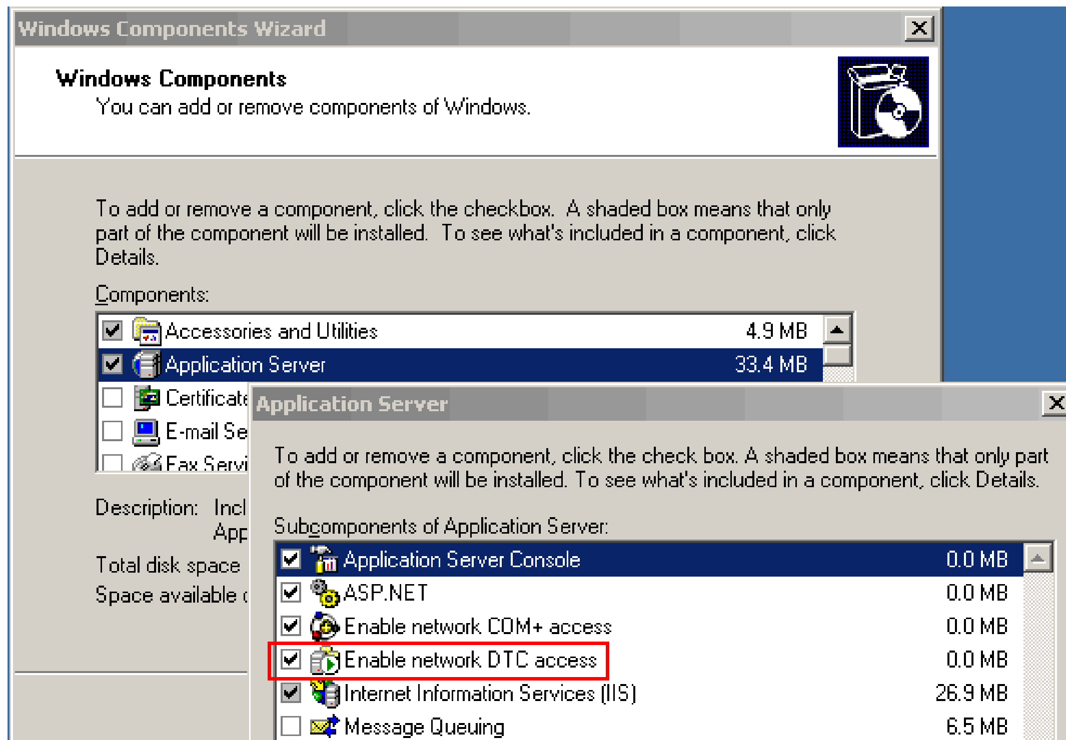

圖 1 於 Windows 2003 上啓用網路 DTC 存取

Windows 2003 SP1 在 MSDTC 的安全性上有些增強，MSDTC 上多了幾個選項，當然預設又是最嚴的選項—Mutual Authentication Required，它是個未來才會生效的選項，現在的效果等同於 Incoming Caller Authentication Required，而且只有在兩台 DTC 都是 Windows XP SP2 或 Windows 2003 才適用。如果 Client 或 SQL 其中一台的 OS 是 Windows 2000 時，需設定爲 No Authentication Required。

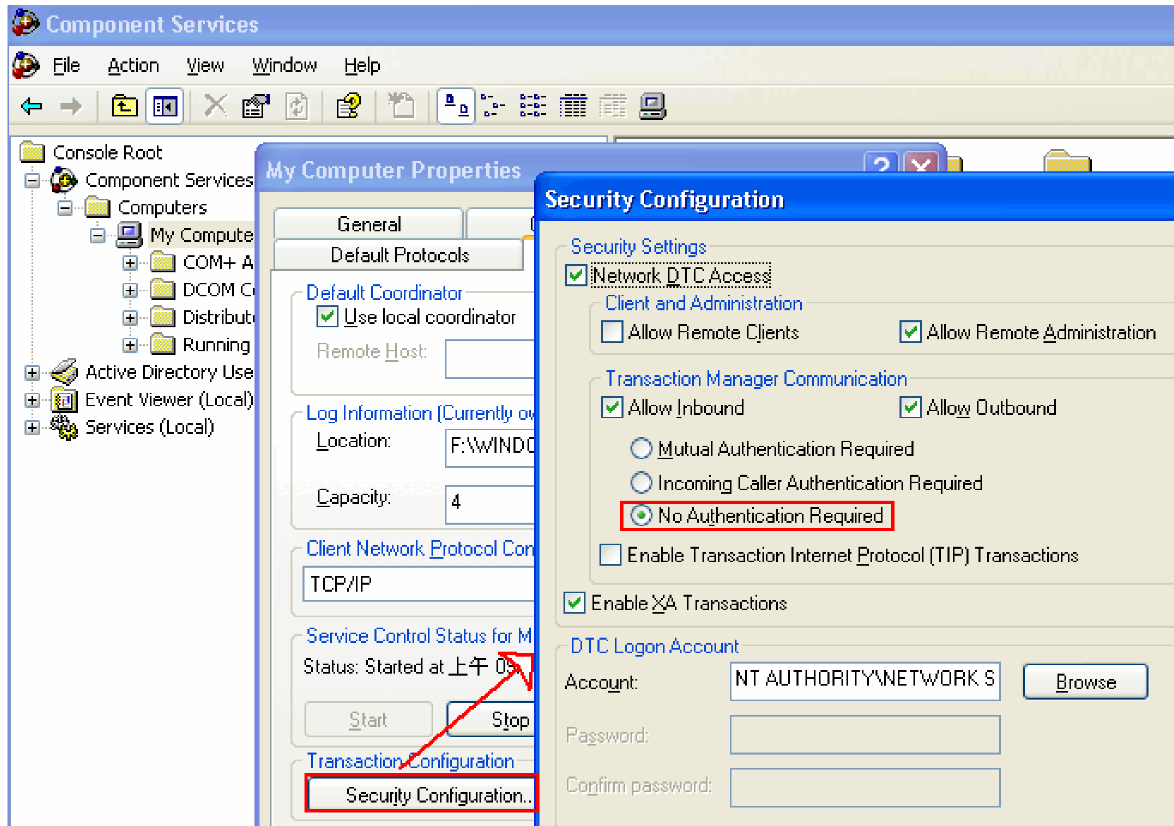

圖 2 MSDTC 的額外安全設定

最後，還有一點要記得，如果你有啓用 XP SP2/Windows 2003 SP1 LAN 網卡上的防火牆，記得要將 MSDTC 加入例外清單，用 UI 設定或執行以下的指令都可以：

```
netsh firewall set allowedprogram %windir%\system32\msdtc.exe MSDTC enable
```

6. **SQL 與 Client 分屬不同 AD Domain 時，分散式交易要如何搞定?**

當 Client 與 SQL Server 分處於不同 AD Domain 時，也很容易導致分散式交易失敗。現象是 Connection.Open()傳回錯誤訊息：New transaction cannot enlist in the specified transaction coordinator.

問題肇因於 MSDTC 靠 RPC 管道溝通，因此 SQL Server 所在的主機也要有能力連回 Client 端，跨 Domain 遇到的狀況多半是因雙方 DNS 不同導致對方機器名稱無法被正確解析，因此請確定 SQL Server 與 Client”彼此相識”！ 最簡單的測試方法是開個 DOS 視窗，分別從 SQL Server 及 Client 用 ping serverMachineName, ping clientMachineName 測試確雙方是否認得對方的機器名稱。不然用 DTCPing 測試也行，還可一併測試 Firewall 等 Issue。

如果發現某一方無法解析機器名稱時，可透過新增 DNS 記錄解決，但另一個簡便的方法是在 windows/system32/drivers/etc/lmhosts(這個檔案預設是不存在的，請將 lmhosts.sam 更名爲 lmhosts，直接拿來修改)裡加上一列如 192.168.1.1 myClient 的宣告，再下個 nbtstat -R，之後 ping 測試如果 OK，問題應該就解決了！

7. **我想讓 Oracle 與 SQL Server 做分散式交易，有何注意事項?**
   經測試，使用 System.Data.OracleClient Namespace 存取資料庫的程式碼，只要被包在有效的 TransactionScope 範圍內，就會參與分散式交易。若程式使用 Oracle 提供的 OPD.NET 元件存取 Oracle，則要視版本而定。依 Oracle 的 ODP.NET FAQ(http://www.Oracle.com/technology/tech/windows/odpnet/faq.html)，ODP.NET 從 10.2.0.2.20 起支援.NET 2.0 的 System.Transactions。另外，記得要安裝 Oracle Services For MTS(http://www.Oracle.com/technology/tech/windows/ora_mts/index.html)。

8. **Oracle 與 SQL Server 做 Transaction，除了寫程式外，有沒有其他更簡單的做法?**
   SQL Server 支援所謂 Linked Server 的概念，而 Oracle 資料庫也可被設定成 Linked Server。如此，我們可以直接在 T-SQL 中引用 UPDATE OracleLinkedServerName..SchemaName.TableName SET ColA='Blah' WHERE ColB='Boo'的寫法，把 ORACLE 當成一般的 SQL 資料庫使喚，而資料更動過程也支援以 BEGIN TRAN/COMMIT TRAN 宣告的交易範圍，算是最簡單的分散式交易實作方式。
   不過 Linked Server 需要預先由管理者新增設定，並指定帳號對應，有些額外的管理部署工作；同時經驗中透過 Linked Server 存取資料庫的執行效率不甚理想，規劃時也應留意。

9. **System.Transactions 配合 SQL 2005 使用時有何特異功能?**
   關鍵在於 LTM(Light Weight Transaction Manager)! 當.NET 2.0 發現整個 TransactionScope 中只涉及一台 SQL Server 2005 時，可以不啓用 OleTx，不驚動 MSDTC，直接使用 LTM 更有效率地完成交易。而最方便的一點是，.NET 2.0 會依參與交易的對象、數目決定使用 LTM 或是 OleTx，對開發人員完全透明。目前只有在連線單一 SQL 2005 時可以支援 LTM，涉及 SQL Server 2000、MSMQ、Oracle 等其他交易成員時，則一律使用 OleTx。

10. **我要如何確認現在是否動用了 MSDTC?**
   最簡單的方法是檢視元件服務管理員的 DTC 交易統計，如圖 3。若啓用了分散式交易，將可以看到成功(Commit)或失敗(Abort)的統計數字增加。
   另一種做法是利用 System.Transactions.Transaction.Current.TransactionInformation，其中有兩個屬性: LocalIdentifier 有值時代表啓用了 LTM，DistributedIdentifier 有值代表啓用了 OleTx。當 System.Transactions.Transaction.Current==null 時則代表未在交易範圍內。

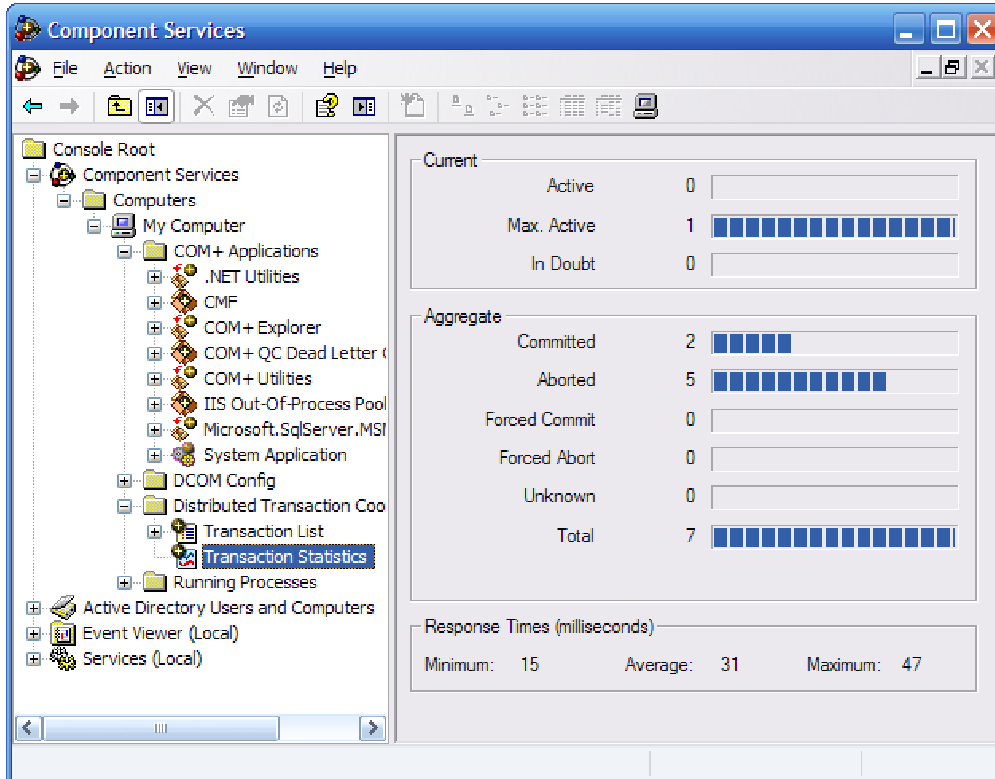

圖 3 MSDTC 的交易統計

11. 我明明只連了一台 SQL 2005，為何還是用了 MSDTC?

如程式範例 1，執行完會發現 DistributedIdentifier 有 GUID 值，而交易統計中也可以看到交易成功數加 1。

程式範例1

```
static string demoCnnStr
    = "Data Source=mySql2005;User Id=blah;Password=blah;Initial Catalog=Lab;";
//查詢資料庫的共用函數
static DataTable getDataTable(string sqlCmd)
{
    SqlConnection cn = new SqlConnection(demoCnnStr);
    SqlDataAdapter da = new SqlDataAdapter(sqlCmd, cn);
    DataTable dt = new DataTable();
    da.Fill(dt);
    return dt;
}
//檢查啓用LTM或OleTx
static void showTrnInfo(string tag)
{
    string msg = "No Transaction!";
    if (Transaction.Current != null)
```

```
    {
        TransactionInformation ti =
        Transaction.Current.TransactionInformation;
        msg = string.Format("LTM:{0} OleTx:{1}", ti.LocalIdentifier,
            ti.DistributedIdentifier);
    }
    Console.WriteLine(string.Format("{0}\n {1}", tag, msg));
}
//Trnasaction中即使SELECT也會導致升級成OleTx
static void TestSelInTran2()
{
    showTrnInfo("P1");
    using (TransactionScope tx = new TransactionScope())
    {
        string sql = "SELECT getdate()";
        showTrnInfo("P2");
        //第一次查詢動作
        DataTable t = getDataTable(sql);
        showTrnInfo("P3");
        //第二次查詢動作
        t = getDataTable(sql);
        showTrnInfo("P4");
        tx.Complete();
    }
}
```

程式範例 1 共檢查交易狀況四次，P1 時爲 No Transaction，P2/P3 則是 LTMId 有 GUID，OleTxId 爲 Guid.Empty，在第二次查詢動作後(P4)，OleTxId 也有了 Guid 值。這印證了.NET 2.0 一開始先啓用 LTM，當發現不符合 LTM 可支援的條件時，再升級成 OleTx。

但都是同樣對同一台 SQL 2005 做查詢，爲何還是無法適用 LTM 模式?

問題出在這裡的二次查詢都重新開啓及關閉連線，雖然在設計實務上，每次資料庫存取都重新開啓及關閉連線可以發揮 Connection Pooling 的最大效益。但在 Transaction Scope 中想讓 SQL 2005 維持使用 LTM，必須所有資料庫查詢都使用單一連線，才不會升級成 OleTx。

程式範例 2 中建立一個 DaraProvider 物件，全程使用單一連線，改寫後的測試，就可保持 LTM 模式。

```
程式範例2:
//可共用連線的資料存取物件
class DataProvider : IDisposable
{
    SqlConnection cn = null;
```

```
    public DataProvider()
    {
        string demoCnnStr
        = "Data Source=labs-web01;User Id=lab;Password=blah;Initial Catalog=Lab;";
        cn = new SqlConnection(demoCnnStr);
        cn.Open();
    }

    public DataTable GetDataTable(string sqlCmd)
    {
        SqlDataAdapter da = new SqlDataAdapter(sqlCmd, cn);
        DataTable dt = new DataTable();
        da.Fill(dt);
        return dt;
    }

    public void ExecCommand(string sqlCmd)
    {
        SqlCommand cmd = new SqlCommand(sqlCmd, cn);
        cmd.ExecuteNonQuery();
    }

    public void Dispose()
    {
        cn.Close();
    }

}
//改用DataProvider物件存取資料庫
static void TestSelInTran3()
{
    using (TransactionScope tx = new TransactionScope())
    {
        using (DataProvider dp = new DataProvider())
        {
            string sql = "SELECT getdate()";
            showTrnInfo("P2");
            //第一次查詢動作
            DataTable t = dp.GetDataTable(sql);
```

```
            showTrnInfo("P3");
            //第二次查詢動作
            t = dp.GetDataTable(sql);
            showTrnInfo("P4");
            tx.Complete();
        }
    }
}
```

12. **我要如何避免不必要的資料庫動作導致Transaction升級成OleTx?**

前面提過如何用共用連線維持LTM模式，但如果在TransactionScope範圍內呼叫了其他廠商的元件，其中雖然查詢了其他資料庫，卻與我們的交易完全無關，想避免因為這些無關交易的連線動作升級成OleTx，可以使用using (TransactionScope ntx = new TransactionScope(TransactionScopeOption.Suppress))將無關交易的部分包起來。如程式範例3。

程式範例3

```
static void TestTran4()
{
    CleanData(false);
    using (TransactionScope tx = new TransactionScope())
    {
        using (DataProvider dp = new DataProvider())
        {
            string sql = "SELECT getdate()";
            DataTable t = dp.GetDataTable(sql);
            showTrnInfo("Before");
            using (TransactionScope ntx = new TransactionScope(
                    TransactionScopeOption.Suppress
                    ))
            {
                //呼叫查詢資料庫的其他元件
                ExtProc ep = new ExtProc();
                ep.DoSomething();
            }
            showTrnInfo("After");
            tx.Complete();
        }
    }
}
```

13. 要保持 LTM 模式還真麻煩，OleTx 與 LTM 的效率差異真值得我們這麼做嗎?

以程式範例 4 實測，TSQLInsert 使用 T-SQL 包成交易，CnTrnInsert 則是用 SqlTransaction，LTMInsert 使用單一連線維持 LTM，OleTxInsert 中 DataHelper 比照程式範例 1，每次執行都重新開啓及關閉資料庫連線，因此會用 OleTx。四個程式都連到同一台 SQL 2005 資料庫，以迴圈各做 500 次，測試結果為:

TSQLInsert : 406ms

CnTrnInsert: 1,078ms

LTMInser: 1,093ms

OleTxInsert: 10,890ms

LTM 與使用 SqlTransaction 的效果相近，而 OleTx 足足慢了近十倍。當程式被呼叫的次數很頻繁時，設法保持在 LTM 模式將產生明顯的效能差異。

程式範例4

```
static void TSQLInsert(int i)
{
    string sql = string.Format(
@"
SET XACT_ABORT ON
BEGIN TRAN
INSERT INTO Emp VALUES ({0},'Emp{0}')
INSERT INTO Cust VALUES ({0},'Cust{0}')
COMMIT TRAN
", i);
    DataHelper.ExecCommand(sql);
}
static void CnTrnInsert(int i)
{
    using (SqlConnection cn = new SqlConnection(DataHelper.DemoCnnStr))
    {
        cn.Open();
        SqlTransaction tn = cn.BeginTransaction();
        string sql = string.Format("INSERT INTO Emp VALUES ({0},'Emp{0}')", i);
        SqlCommand cmd = new SqlCommand(sql, cn, tn);
        cmd.ExecuteNonQuery();
        cmd.CommandText = string.Format("INSERT INTO Cust VALUES ({0},'Cust{0}')", i);
        cmd.ExecuteNonQuery();
        tn.Commit();
        cn.Close();
    }
}
static void LTMInsert(int i)
```

```
{
    using (TransactionScope tx = new TransactionScope())
    {
        using (DataProvider dp = new DataProvider())
        {
            string sql = string.Format("INSERT INTO Emp VALUES ({0},'Emp{0}')", i);
            dp.ExecCommand(sql);
            sql = string.Format("INSERT INTO Cust VALUES ({0},'Cust{0}')", i);
            dp.ExecCommand(sql);
            tx.Complete();
        }
    }
}
static void OleTxInsert(int i)
{
    using (TransactionScope tx = new TransactionScope())
    {
        string sql = string.Format("INSERT INTO Emp VALUES ({0},'Emp{0}')", i);
        DataHelper.ExecCommand(sql);
        sql = string.Format("INSERT INTO Cust VALUES ({0},'Cust{0}')", i);
        DataHelper.ExecCommand(sql);
        tx.Complete();
    }
}
```

14. 如果只連到一台 SQL Server 2005，為何不直接包成一個 SQL 指令送出? 執行效能更好?

在問題 11-13 裡，專注的情境一直都是: 單一 SQL Server 2005，用 TransactionScope 包起來，卻又要極力保有 LTM 的模式。或許有人會想，既然只有單一 DB，為何不將全部的更新動作寫成如下的 T-SQL? (由 13 題的測試可以驗證 T-SQL 交易快了一倍以上)

```
SET XACT_ABORT ON
BEGIN TRAN
… UPDATE COMMAND …
COMMIT TRAN
```

的確，這種做法可以獲得最佳的執行效能。但在實務上，當我們以物件導向開發程式，更新資料庫的程式邏輯常會散落在大大小小的各式物件中，並不像範例程式裡永遠只是單純兩行 INSERT，用膝蓋就可以合併成一個指令。更何況有時元件可能來自別的部門、其他廠商，我們無法掌握所有元件的開發與實作方式，使用 TransactionScope 可以兼顧元件開發自由度與包成交易的要求，簡化了在物件導向系統中實作交易式更新的複雜度。當參與交易的成員不固定時，System.Transactions 允許我們先"獨善其身"，將固定發生的作業侷限在 LTM 模式下，交易管理擴大機制可視情況於必要時才升級為較耗資源的分散式交易，達成系統效能

的最佳化。

15. 開發分散式交易程式時，有什麼效能方面的注意事項?

TransactionScope 的方便性，讓許多開發者輕輕鬆鬆變身爲有能力寫出資料庫交易程式的高手，但欠缺對資料庫交易的正確認識，不當的濫用卻可能導致災難。以下是一些設計應用時的效能準則:

* 永遠記得啓動交易機制是要付出代價的，資料庫必須額外對資料進行鎖定，爲失敗還原(Rollback)做準備，分散式交易要採用複雜的兩階段式交易認可，更傷效能

* 如果交易無可避免，請記住執行效率排序 T-SQL > SqlTransaction = LTM > OleTx (MSDTC)，非不得已，請盡量採用效率較高的做法。

* 在 SQL 中，查詢及更新資料時常會導致資料鎖定，其他人必須等待鎖定解除才能查詢該資料。因此請儘可能縮短交易開始到結束的期間，以提高資料的共用性及多人環境下的整體效能。

* 若交易 A 先鎖定資料 X，要更新資料 Y，同一時間交易 B 先鎖定資料 Y，要更新資料 X，即產生了 Deadlock(死結)。一般解決方式是由資料庫強制讓交易 A 或交易 B 失敗，釋放鎖定以化解僵局。被強制失敗的交易(稱作 Victim)必須 Rollback、稍後再重試，十分不利於效能。要避免 Deadlock 的方法是縮短交易持續期間、減少不必要的鎖定、以及儘可能採用相同的鎖定/更新順序。(例如: 大家一律先更新 X，再更新 Y)