

7

高效率的分散式多層應用系統

現在我知道了如何開發分散式多層應用系統，但是除了讓應用系統能夠正確的執行之外，我還想知道怎麼樣才能夠撰寫出高執行效率的應用系統，讓我開發的分散式多層應用系統表現的比其他人開發的系統來得有效率。

本章重點

- 影響分散式多層應用系統的效率因素
- 使用有效率的遠端呼叫方式
- 調校應用程式伺服器
- TDCOMConnection 和 TSocketConnection 的比較
- Interceptor

7-2 實戰 Delphi 5.0-分散式多層應用系統篇

不論軟體人員在開發任何形式的應用系統時，當整個系統已經步入穩定的階段之後接下來的工作便是如何調整應用系統的執行效率，讓它可以執行的更為迅速。通常對於應用系統執行效率的調整是由資深開發人員來進行的，因為要調校應用系統的執行效率必須對於許多的技術有所瞭解才能夠知道在什麼地方使用最適當的方式來執行應用系統。但是要開發一個高效率的分散式多層應用系統除了系統的調校之外，當程式師在撰寫應用程式時也必須知道如何使用最有效率的方式呼叫遠端的物件。因為不同的呼叫方式在執行效率上會有一倍以上的差距，這對於經常會進行遠端物件呼叫的應用系統來說會有非常大的影響。

本章的內容就在討論如何開發高效率的分散式多層應用系統。由於一個應用系統的執行效率和許多的因素都有關連，例如應用程式伺服器，資料庫，資料庫驅動程式等，所以本章絕無可能討論所有的主題。有一些效率因素的討論是在其他的章節或是書籍之中，例如『資料庫存取的設定，剖析和最佳化調整』一章以及『實戰 Delphi 5.x-高效率資料庫應用系統篇』中有關 ADO 技術的部份。下面的表格簡略的列出了應用系統效率相關的因素，每一個因素對於系統的影響，以及直接相關的人員：

	效率影響	關鍵人
分散式多層應用系統架構	整體應用系統的執行效率， 延展性	Architect，系統 Leader，專案管理者
應用系統領域元件 (domain components)的設計	元件的建立/消滅次數，元 件的訊息流量	元件撰寫者
應用程式伺服器的架構	應用程式伺服器處理用戶端 服務的效率	資深開發人員
資料庫的使用元件，驅動程 式的設定	應用程式伺服器處理資料查 詢/異動的效率	資深開發人員，應用系統程 式師
呼叫遠端物件的方式	應用程式執行效率相差可達 一倍以上	應用系統程式師

在上表中列出的效率因素有一些是屬於分析/設計的主題，所以本章並不會加以討論。在接下來的內容中會討論一些屬於程式技巧以及應用程式伺服器的主題，因為這些技術是每一個軟體人員都可以掌握的，而且對於應用系統的

執行效率也有很大的影響。如果軟體人員能夠掌握本章討論的重點和技術的話，那麼至少可以在整體系統架構影響之外有效的提昇分散式多層應用系統的執行效率。

對於分散式多層應用系統來說如何建製一個高效率的模型是更為重要的，因為多層應用系統不但比傳統主從架構系統多了中介軟體，而且多層應用系統通常是需要服務更多的用戶端使用者，或是執行在一個緩慢的 Internet/Intranet 或是 WAN 環境之中。因此軟體人員必須對於什麼會影響應用系統的執行效率有清楚的瞭解，才能夠避免撰寫出差勁的程式碼。當軟體人員在開發分散式多層應用系統時必須把下列的事情牢記在心：

減少網路的 roundtrip

減少需要在網路中流動的資料

瞭解各種不同資料型態的傳遞負荷成本

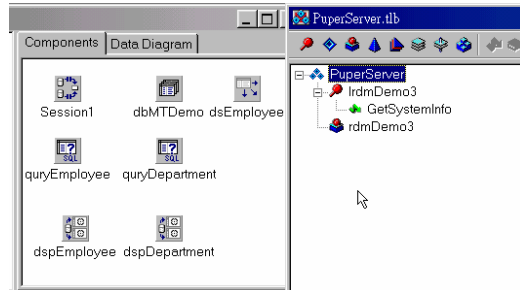
使用最有效率的方式進行遠端呼叫

這些原則即使是 Inprise 的 Delphi R&D 工程師也是在遵守的，例如 Delphi5.x 之中的 MIDAS 3.0 比起 MIDAS 2.0 有更好的執行效率，這便是因為 MIDAS 3.0 大幅的減少了網路的 roundtrip 以及流動的資料。在下面的小節中將會一一的討論這幾個因素的意義，以及它們對於多層應用系統執行效率的影響。

7-1 使用有效率的遠端呼叫方式

這個因素可以說是軟體人員最容易掌握的而且對於多層應用系統的執行效率也有非常大的影響的，但卻也是被許多人忽略的。在開發分散式多層應用系統時不管應用程式伺服器是 MIDAS 伺服器，MTS 伺服器，或是 CORBA 伺服器，通常我們都會在應用程式伺服器中定義一些服務讓用戶端應用程式呼叫使用。例如應用程式伺服器中可能有一些會計物件提供過帳的服務，或是有產品物件提供產品資訊查詢的服務。圖 7-1 便是一個應用程式伺服器，在這個應用程式伺服器的遠端資料模組中定義了一個方法 GetSystemInfo 可以讓用戶端應用程式呼叫以取得應用程式伺服器一些系統的資訊。

7-4 實戰 Delphi 5.0-分散式多層應用系統篇



7-1 應用程式伺服器中定義的方法可以被用戶端呼叫使用

當用戶端應用程式需要取得應用程式伺服器的系統資訊時，軟體人員應該如何呼叫這個方法呢？使用的呼叫方式是不是有效率呢？這是接下來要討論的主題。一般來說對於應用程式伺服器提供的服務，用戶端應用程式有數種不同的呼叫方式，分別是：

Late Binding 呼叫方式
Early Binding 呼叫方式
Dispatch Table 呼叫方式

經常軟體人員最常使用的方式是 Late Binding，但這也是最沒有效率的方
式，只是它在使用上非常的方便，所以許多人並不知道這種呼叫方式對於應用
系統執行效率的影響。在下面的小節中將會一一的說明並且比較這些不同的呼
叫方式對於執行效率的影響。

7-1-1 最常使用的遠端呼叫方式-Late Binding

在圖 7-1 中的 GetSystemInfo 範例中，一般軟體人員要呼叫這個方法時通
常會使用如下的程式碼：

```
//使用 Late Binding  
...  
DCOMConnection1.AppServer.GetSystemInfo(vTime, vRDM, vThread);  
...
```

軟體人員藉由 TDCOMConnection(或是 TSocketConnection, TCORBAConnection)的 AppServer 特性取得遠端應用程式伺服器，再呼叫其中的 GetSystemInfo 方法。這種方式可以正確的呼叫到遠端應用程式伺服器之中

提供的服務。但是你必須知道的是 AppServer 特性回傳的是一個 Variant 型態的變數，當程式透過 Variant 呼叫遠端方法時，Delphi 會以 IDispatch 介面動態的和遠端應用程式伺服器溝通，組合呼叫方法和參數，最後再進行呼叫。所以這種呼叫方式需要執行非常多額外的工作，因此並不是很有效率。只是因為 TDCOMConnection 元件直接提供了 AppServer 這個特性可以讓軟體人員很方便的呼叫遠端方法，所以幾乎所有的人都是使用這種方式(當然也包括懶惰的我在內)。那麼這種呼叫方式的效率如何呢？稍後你就會知道。

7-1-2 最有效率的遠端呼叫方式-Early Binding

由於前面使用 Late Binding 的呼叫方式是在程式執行時才動態進行遠端方法呼叫，所以效率並不會很好。因此一些聰明的程式師會使用所謂的 Early Binding 的方式來呼叫遠端方法。使用 Early Binding 的好處是 Delphi 在編譯應用程式時便可以檢查程式師使用的語法是不是正確，傳遞的參數形態是不是正確，以及編譯器可以事先便產生所有必要的執行碼，因此不需要應用程式執行時再使用動態方式和遠端應用程式伺服器溝通，**所以在執行效率上比起使用 Late Binding 更有效率，也更安全**。那麼既然 Early Binding 有這麼多的好處，為什麼很少人使用呢？這是因為它在使用上比較麻煩一點，而且需要一些程式技巧。下面的程式碼便是使用 Early Binding 的方式呼叫相同的遠端方法：

```
//使用 Early Binding
...
begin
  (DCOMConnection1.GetServer as IrdmDemo3).GetSystemInfo(vTime,
    vRDM, vThread);
  ...
end;
...
```

在上面的程式碼中，當我們呼叫 GetSystemTime 時，是先把 AppServer 特性回傳的 Variant 直接轉換為 IrdmDemo3 介面，再呼叫 GetSystemTime。但是當你使用上面的程式碼在用戶端機器中呼叫遠端的方法時，Early Binding 的呼叫方式卻會在 COM 執行時期函式庫中顯示一個錯誤，那就是有名的『Interface not supported』。這是因為由於用戶端應用程式直接使用了遠端應用程式伺服器的介面來進行遠端方法呼叫，但是這個介面只存在遠端應用程式

7-6 實戰 Delphi 5.0-分散式多層應用系統篇

伺服器之中，用戶端機器並未有註冊這個應用程式伺服器介面的資訊，所以 COM 無法取得應用程式伺服器介面的支援。因此要解決這個問題我們必須在用戶端的機器中註冊應用程式伺服器的介面，你可以使用兩種方法來註冊應用程式伺服器的介面資訊，它們是：

直接在用戶端機器之中先執行應用程式伺服器一次，讓應用程式伺服器在用戶端機器中註冊應用程式伺服器的介面資訊，在用戶端機器中註冊應用程式伺服器的 Type Library 資訊

使用第一種方法時，你可以先在用戶端應用程式中執行所有使用 Early Binding 存取的應用程式伺服器，然後再把這些應用程式伺服器從用戶端機器中刪除。如果你使用第二種方法的話，那麼你需要在用戶端應用程式啟動時，載入應用程式伺服器的 Type Library 並且註冊它，之後就可以使用 Early Binding 的方式呼叫遠端方法了。例如在這個範例中我於表格的 OnCreate 事件處理函式中加入了如下的程式碼：

```
procedure TForm2.FormCreate(Sender: TObject);
begin
  OleCheck(LoadTypeLib('puperserver.tlb', servertlb));
end;
```

LoadTypeLib 可以載入指定的 Type Library 並且註冊它。因此你可以把所有需要以 Early Binding 存取的應用程式伺服器的 Type Library 檔案(.tlb 的檔案)拷貝到用戶端機器的特定目錄之下，例如 C:\AppServerTLB，然後使用

```
OleCheck(LoadTypeLib('c:\AppServerTlb\puperserver.tlb',
servertlb));
```

的程式碼載入指定的 Type Library。

使用 Early Binding 雖然可以大幅增加遠端呼叫的執行效率，但是天下沒有白吃的午餐，當你使用視覺化 Type Library 編輯器異動應用程式伺服器介面時，例如增加其中的方法，或是修改其中的參數，那麼你必須重新拷貝這個 Type Library 到用戶端機器的 C:\AppServerTLB 目錄之下，否則用戶端的 Type Library 和應用程式伺服器的服務就不一致了。當然你也可以把所有的 Type Library 檔案拷貝到網路的全域目錄之下，然後讓用戶端都從這個目錄中載入 Type Library，如此就可以保證用戶端應用程式能夠取得最新的 Type Library 的資訊了。另外 Early Binding 有一個限制，那就是它只能使用在 DCOM 的通訊協定之中。

7-1-3 DCOM/Socket 都可以使用的呼叫方式-Dispatch Table

Early Binding 的方式雖然很有效率，但可惜的是不是所有的多層應用系統都可以使用它。例如如果你是使用 Socket 通訊協定的話，就無法使用 Early Binding。那麼這是不是代表在 socket 通訊協定中就只能使用 Late Binding 呢？當然不，在 Delphi 中有另外一種方法可以讓程式師在 DCOM 或是 Socket 通訊

協定中都能夠使用比 Late Binding 更有效率的方式，那就是 Dispatch Table 呼叫方式。

由於 Delphi 支援 COM/DCOM 的 dual interface，所以程式師可以選擇使用比 IDispatch 介面更有效率的 Dispatch Table 方式呼叫遠端方法，更好的是即使是在 Socket 通訊協定中仍然可以使用 Dispatch Table。那麼要如何在程式碼中使用 Dispatch Table 呢？很簡單，當你在 Delphi 中使用視覺化 Type Library 編輯器定義應用程式伺服器的服務後，Delphi 便會在 Type library 的 wrapper 類別中自動產生 Dispatch Table 的介面。你可以在 Delphi 中開啟應用程式伺服器的 wrapper 類別原始程式(通常是 XXX_TLB.PAS 這個檔案，XXX 代表你的應用程式伺服器的名稱)，在 wrapper 類別中會有一個以 disp 結尾的介面，這個介面就是應用程式伺服器的 Dispatch Table，例如圖 7-1 的範例應用程式伺服器在它的 wrapper 類別中有如下的 Dispatch Table 介面：

```
IrdmDemo3Disp = dispinterface
  ['{AFC25B93-37E0-11D3-AA94-0080C8518D04}']
...
```

有了這個介面之後，我們便可以使用它以 Dispatch Table 的方式進行遠端呼叫。例如在下面的程式碼中首先宣告一個 IrdmDemo3Disp 的變數 aDisp，然後把 TDCOMConnection 的 AppServer 特性先轉換為 IDispatch，再轉換為 IrdmDemo3Disp 介面，再指定給 aDisp 變數，最後就可以使用這個變數呼叫遠端方法了。

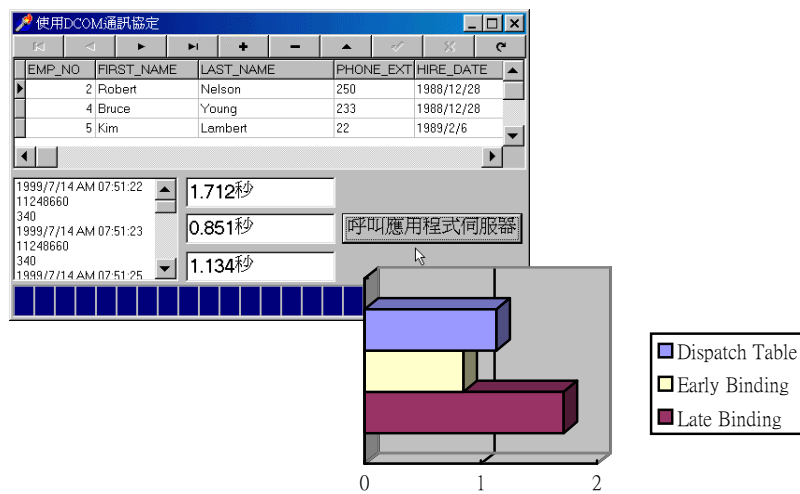
```
//使用 Dispatch Interface
var
  aDisp : IrdmDemo3Disp;
begin
  ...
  begin
    aDisp := IrdmDemo3Disp(IDispatch(DCOMConnection1.AppServer));
    aDisp.GetSystemInfo(vTime, vRDM, vThread);
    ...
  end;
  ...
end;
```

由於 Dispatch Table 介面可以由視覺化 Type Library 編輯器自動產生，而且又能夠使用同時使用在 DCOM 和 Socket 通訊協定之中，因此如果你不想像上一小節討論的一樣在用戶端分發應用程式伺服器的 Type Library，那麼你就可以考慮使用 Dispatch Table 的呼叫方式，因為它幾乎和 Late Binding 一樣很方便，而且在執行效率上是比 Late Binding 來得良好。

7-1-4 不同遠端呼叫方式的效率差距

在你瞭解了呼叫遠端方法有這麼多不同的方式之後，那麼再瞭解這些方式在執行效率上的差異之後，相信你就會知道如何在多層應用系統中使用最有效率的遠端呼叫方法了。

下面的畫面和圖形便是我撰寫一個測試程式以不同的方式呼叫圖 7-1 應用程式伺服器中 GetSystemInfo 方法的結果。



7-10 實戰 Delphi 5.0-分散式多層應用系統篇

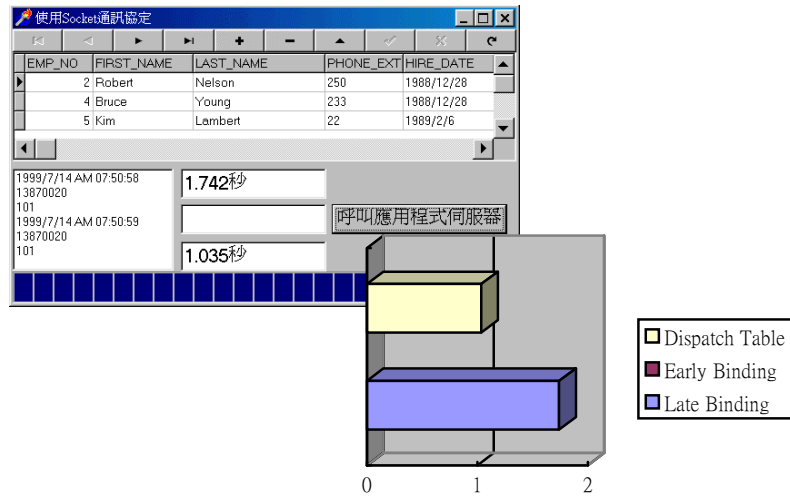


圖 7-2 不同的遠端呼叫方式的執行效率差異

下面的表格詳細的列出了每一種遠端呼叫方法使用的時間以及執行效率的差異：

使用DCOM	Late Binding	Early Binding	Dispatch Table
時間	1.712	0.851	1.134
效率	-	快 100%	快 70%以上
使用Socket	Late Binding	Early Binding	Dispatch Table
時間	1.742	N/A	1.035
效率	-	無法使用	快 70%以上

從上表中可以知道，Early Binding 比起 Late Binding 快了 100% 以上，即使你嫌麻煩，不想在用戶端中安裝 Type Library，那麼多寫一、二行程式碼，使用 Dispatch Table 的方式仍然能夠讓你在呼叫遠端方法時快上 70%，你又何樂而不為呢？而且在一般的分散式多層應用系統中，經常會有大量的遠端物件，遠端方法的呼叫。尤其是在真正使用物件導向，或是 CBD(Component Based Design)開發模型的應用系統之中更是如此。加快遠端呼叫的效率可以明顯的增加整個應用系統的執行效率。

7-2 不同資料型態的傳遞負荷成本

在前面有提到需要減少在網路中流動的資料，這除了需要程式師小心的規劃遠端方法傳遞的資料以及參數之外，另外一個需要注意的地方便是當程式碼在傳遞各種不同型態的資料時，程式師必須知道每一種資料型態的成本。換句話說如果程式師能夠減少資料傳遞成本，當然就可以增加應用程式的執行效率，因為這可以有效的降低網路的負荷。

在本書前面的章節中有許多的範例都是使用 Variant 或是 OleVariant 在傳遞資料或是參數。但是這些範例都是為了在使用上方便的原因，因為 Variant 能夠代表任何型態的資料。但是在程式師真正開發分散式應用系統時，一定要注意不要任意的使用 Variant，因為 Variant 雖然好用，但是相對上它使用的成本也非常的高。例如如果你使用 Variant 來代表整數的話，那麼整數本身只佔 4 個位元組，但是使用 Variant 來代表它則需要 20 個位元組。當這樣的 Variant 在網路上傳遞時是比直接傳遞整數來得緩慢。

因此瞭解每一種資料型態的成本並且注意在應用程式中適當的使用它們也可以有效的增加應用系統的執行效率。下面的表格列出了一些常用的資料型態的成本：

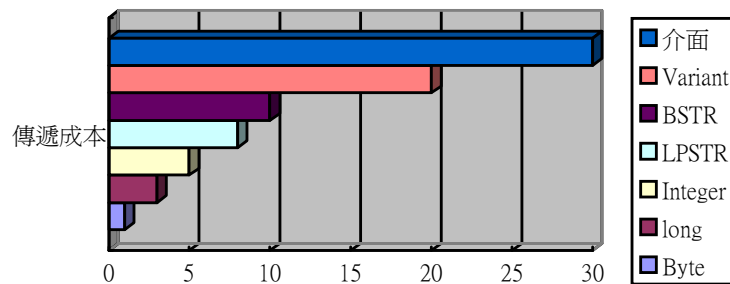


圖 X-3 各種資料型態的傳遞成本

從上圖中可以知道，使用 Variant 的成本會比你真正需要的資料型態來得昂貴的許多。例如以 Variant 來代替 Integer 將會付出數倍的成本，而不只是一

點點的額外成本。此外請注意使用 long 比使用 Integer 便宜，在傳遞字串時也必須注意不同的字串型態在成本上也有所不同。

7-3 減少網路的 roundtrip

程式師必須知道在分散式應用系統中，遠端呼叫是非常昂貴的事情。因此如果應用系統能夠儘量避免不必要的遠端呼叫的話，就可以有效的增加應用系統的執行效率，因為這可以減少網路的 roundtrip。Delphi 5.0 的 MIDAS 3.0 之所以比 MIDAS 2.0 有將近 1 倍以上的效率就是因為 3.0 有效的降低了應用系統需要的遠端呼叫。例如在 Delphi 4.0 的 MIDAS 2.0 中，當用戶端需要取得遠端資料時，必須先取得應用程式伺服器之中的 IProvider 介面，再藉由這個介面呼叫遠端方法。下面即是 MIDAS 2.0 取得遠端資料的典型程式碼：

```
function GetDataTheOldWay(Connection: TDCOMConnection;
TDataSetProviderName:
string; Params: OleVariant; Metadata: Boolean);
var
    TDataSetProvider: ITDataSetProvider;
    RecsOut: Integer;
begin
    TDataSetProvider :=
        Connection.GetTDataSetProvider(TDataSetProviderName); //2 個
        roundtrips, 一個 ITDataSetProvider 介面
    TDataSetProvider.Reset(Metadata);
        //1 個 roundtrip
    TDataSetProvider.SetParams(Params);
        //1 個 roundtrip
    Result := TDataSetProvider.GetRecords(-1, RecsOut); //2 個
    roundtrip
    { 6 網路 roundtrips, 1 個額外的介面 }
end;
```

你可以看到在 MIDAS 2.0 中一次取得資料的動作需要花費 6 個網路 roundtrips 而且存取一個額外的 IProvider 介面，這是很高的成本。而下面則是 MIDAS 3.0 進行相同工作時使用的程式碼：

```
function GetDataTheNewWay(Connection: TDCOMConnection;
TDataSetProviderName:
string; Params: OleVariant; Metadata: Boolean);
var
Options, RecsOut: Integer;
begin
if Metadata then Options := INCLUDE_METADATA else Options := 0;
Result := Connection.GetServer.AS_GetRecords(TDataSetProviderName,
-1,
RecsOut, Options, Params, NULL);
{ 1 個網路，不需要額外取得介面}
end;
```

MIDAS 3.0 只需要 1 個網路 roundtrip 即可，也不需要取得額外的遠端介面，所以 MIDAS 3.0 比起 MIDAS 2.0 減少了許多網路 roundtrip 的成本，因此也遠比 MIDAS 2.0 有效率。從這個存取資料的典型程式碼中可以知道，MIDAS 3.0 至少快了 5 倍以上。因此如何降低網路 roundtrip 應該是程式師特別注意的事情。

如何能夠降低網路的 roundtrip 牽涉到許多的技術，其中最重要的便是程式師如何設計在中介應用程式伺服器之中的企業物件。一般來說中介企業物件如果依照它的工作精細度來區分的話，那麼大致上可以分為兩種：Finer grained 和 Larger grained 企業物件。下面是它們的說明。

7-3-1 Larger Grained 企業物件

一般來說需要重覆使用的企業物件應該儘量避免成為 Larger grained 企業物件，那麼什麼是 Larger grained 企業物件呢？應該在什麼時候使用它呢？

一般來說 Larger grained 企業物件是指其運作邏輯比較複雜，而比較無法重覆使用的。例如一個接受客戶訂單的企業物件，它必須同時使用數個其他的物件(即下一小節的 Finer grained 企業物件)來完成工作。通常在 Larger grained

企業物件中會有判斷的程式碼，例如一個客戶的信用額度是否足夠？客戶的訂購量如果很大的話，是否有折扣等。所以通常 Larger grained 企業物件是比較困難重覆使用在其他的應用程式之中。Larger grained 企業物件即是前面介紹分散式物件中的控制物件或是協調物件。

7-3-2 Finer Grained 企業物件

Finer grained 企業物件是指比較適合重覆使用的企業物件。例如具備會計邏輯的企業物件，或是能夠處理員工邏輯的企業物件。這種企業物件能夠使用在許多不同的應用程式中，所以一般來說程式師應該盡量把這些需要或是能夠使用在各種不同場合的程式碼封裝成這種元件。由於這些具備特定邏輯的企業物件只和它本身的領域知識有關，比較沒有整合其他元件的程式碼或是執行判斷邏輯的程式碼，所以稱為 Finer grained 企業物件。

一般來說這種企業物件由於比較會因為企業邏輯或是資料庫的改變而需要修改，因此程式師應該盡量不要把這種企業物件輸出給用戶端應用程式使用。而應該盡量讓 Larger grained 企業物件使用這些企業物件來完成用戶端應用程式的要求。Finer grained 企業物件事實上也是前面章節在介紹分散式物件種類時之中的功能物件，實體物件，以及資料物件等。

7-3-3 有效率的使用 Finer 和 Larger Grained 企業物件

程式師設計的 Larger grained 企業物件以及 Finer grained 企業物件會影響分散式應用系統的執行效率。如果一個系統設計了許多的 Finer grained 企業物件能夠執行各種不同的工作，那麼程式師絕對不應該直接輸出所有的 Finer grained 企業物件給用戶端使用。而應該再額外的開發一些 Larger grained 企業物件，讓這些 Larger grained 企業物件呼叫 Finer grained 企業物件執行工作，再輸出這些 Larger grained 企業物件給用戶端應用程式使用。

例如在下面的圖形中如果用戶端應用程式直接呼叫 Finer grained 企業物件，那麼會進行許多的遠端方法呼叫，所以在執行效率上當然不會好。如果我們稍為修改這個架構，讓用戶端應用程式呼叫 Larger grained 企業物件，再讓 Larger grained 企業物件呼叫 Finer grained 企業物件，那麼遠端方法呼叫會減少

許多，執行效率當然比較好。而且 Larger grained 企業物件再呼叫 Finer grained 企業物件的服務時這些呼叫通常都是 in-process 呼叫，所以執行效率也會再增加。

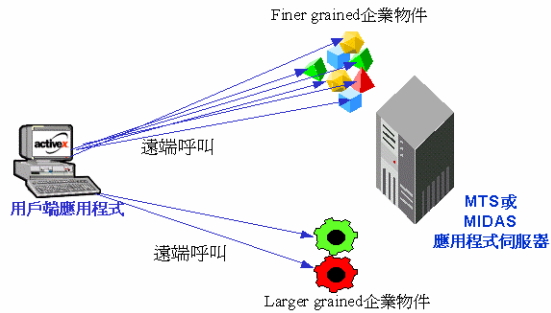


圖 7-3 用戶端應用程式直接呼叫 Finer grained 企業物件

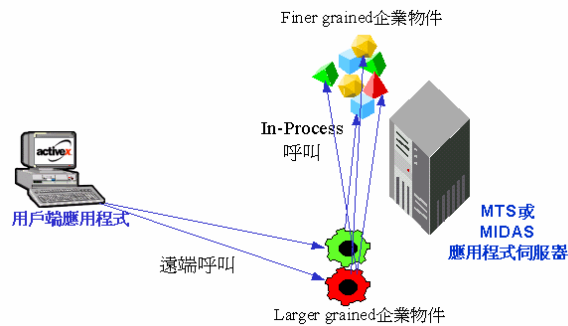


圖 7-4 用戶端應用程式呼叫 Larger grained 企業物件以執行工作

目前在 Enterprise Java Bean 的伺服器之中，也是鼓勵程式師應該遵照這種架構開發。這樣不但可以增加企業物件的重覆使用率，也可以增加分散式應用系統的執行效率。

7-4 瞭解你撰寫的程式碼的意義

在前面的討論中你應該已經瞭解了在多層應用系統中影響應用程式執行效率的原因，也知道了使用那一種呼叫方式是最有效率的。但是在軟體開發人員平日撰寫程式碼時，可能不知不覺的犯了一些錯誤，或是不瞭解撰寫的程式碼

意義，因此造成了應用程式執行效率的不理想。在本小節中讓我們使用數個實際的範例讓各位瞭解如何辨明這些陷阱，讓你的程式碼能夠正確而且有效率的執行。

7-4-1 程式碼範例 1

在許多的應用程式中經常會需要更新大量的資料。因此許多程式師便會在一個大迴圈中使用如下的程式碼來更新資料：

```
For iCount := 1 to UpdateRecordCount do
Begin
...
ClientDataSet1.ApplyUpdates(0);
End;
```

上面的程式碼可以正確的執行，但是卻非常的沒有效率。因為在每一次迴圈之中，用戶端應用程式都會從應用程式伺服器之中取得 IAppServer 介面，這是一個非常昂貴的動作，因為每一個迴圈都需要花費兩次網路的 roundtrip 取得 IAppServer 介面並且呼叫 IAppServer 之中的 AS_ApplyUpdates 方法。而此外這個程式碼傳遞了最昂貴的資料-『介面』。對於這種緩慢的程式碼，程式師應該想辦法改善。

因此如果你有大量的資料需要更新，那麼你至少應該使用如下的程式碼：

```
var
  UpdateIF : IAppServer;
...
UpdateIF := DCOMConnection1.AppServer;
For iCount := 1 to UpdateRecordCount do
Begin
  ...
  IAppServer.AS_ApplyUpdates(Delta...);
End;
```

這個新的程式碼在進行大量資料更新之前，先從應用程式伺服器之中取得 IAppServer 介面並且儲存在用戶端應用程式的變數之中。在稍後需要更新資料時只需要直接呼叫 IAppServer 介面的方法即可。這樣在每一次的迴圈之中只需要一次遠端呼叫即可，比第一個程式碼至少快了一倍以上。

如果上面的 UpdateRecordCount 是 1000 筆資料的話，那麼下面的表格便是這兩種方法花費的成本：

	遠端呼叫次數	網路roundtrip
第一個程式碼	2000	2000
第二個程式碼	1001	1001

你可以看到第一個程式碼無論在遠端呼叫次數，網路 roundtrip 都比第二個程式碼多了許多，而且第一個程式碼的動作又比第二個程式碼昂貴了許多(因為第一個程式碼傳遞了大量的介面)。從這個範例我們可以知道，程式師應該儘量避免無謂的介面傳遞和遠端呼叫。

當然第二個程式碼仍然可以執行的更快，只要你使用前面章節的 Early Binding 或是 Dispatch Table 都可以再增加許多的執行效率。

7-4-2 程式碼範例 2

在前面的章節中說明了 Finer Grained 物件和 Larger Grained 物件。事實上 Larger Grained 也可以視為前面章節說明的協調物件，而 Finer Grained 則是各種控制物件，功能物件，實體物件或是資料物件。在應用系統中適當的使用 Finer Grained 物件和 Larger Grained 物件對於應用程式的執行效率也有很大的

影響。讓我們以一個範例來說明適當的使用各種分散式物件對於應用系統執行效率的影響。

假設現在在應用程式伺服器中有一個分散式物件稱為 `sysProcessObject`，它有一個方法 `GetAllProcesses` 可以回傳一台特定機器之中所有目前正在執行的程序(Process)的名稱，就像 Windows 的 Task Manager 一樣。這個物件可以接受一台機器的名稱並且回傳這台機器中所有正在執行的程序名稱。另外有一個分散式物件稱為 `sysMachineName`，這個物件，它有一個方法 `GetServerName` 方法可以回傳執行這個物件的機器的名稱。

現在有一台應用程式伺服器 `LWServer`，它正在執行 `sysMachineName`。現在如果我希望知道在 `LWServer` 之中目前所有正在執行的程序名稱，那麼我可以使用的程式碼來取得這些資訊：

建立遠端 `sysMachineName` 物件

```
sMachineName := DCOMConnection1.AppServer.GetServerName;
```

建立遠端 `sysProcessObject` 物件

```
aProcessList :=
```

```
DCOMConnection2.AppServer.GetAllProcesses(sMachineName);
```

顯示所有的程序名稱

上面的程式碼雖然可以正確的取得 `LWServer` 之中所有執行的程序名稱，但是我們也可以使用另外一種方法來取得程序名稱的資訊。這個新的方法便是在應用程式伺服器之中建立一個新的物件稱為 `sysServerProcesses`，它提供一個方法 `GetAllProcesses` 可以回傳伺服器之中所有執行的程序名稱。而 `sysServerProcesses` 的 `GetAllProcesses` 方法則是如同上面的程式碼一樣，是分別建立 `sysMachineName` 物件取得機器名稱，再建立 `sysProcessObject` 物件取得所有執行的程序名稱。因此現在只需要使用如下的程式碼即可以完成和上面的程式碼相同的工作：

建立遠端 `sysServerProcesses` 物件

```
aProcessList := DCOMConnection1.AppServer.GetAllProcesses;
```

顯示所有的程序名稱

雖然這兩個程式碼可以完成相同的工作，但是它們卻有非常大的差異。首先第二個程式碼雖然建立了一個新的物件 `sysServerProcesses`，但是它是一個協

調物件，因為它分別建立了 `sysMachineName` 物件和 `sysProcessObject` 物件來完成它的工作。同時也為用戶端應用程式隱藏了 `sysMachineName` 物件和 `sysProcessObject` 物件，讓程式師面對的物件比較少。

另外一個差別就是執行效率的差異。下面的表格列出了這兩個程式碼執行時消耗的資源成本：

	建立的遠端物件	建立的本 地物件	進行的遠端呼 叫次數	網路的 roundtrip
第一個程式碼	2	0	2	3
第二個程式碼	1	2	1	1

如果我們把上表中每一個數字加起來，那麼從上表中可以看到第一個程式碼不但總執行次數較多，並且第一個方法的每一個行動都非常的昂貴。因此第二個程式碼執行的效率比起第一個程式碼來得有效率的多。

從上面的分析可以知道，第二個程式碼由於適當的使用了協調物件，所以它不但在設計比較良好，在執行效率上也比較好。因此當軟體開發人員開發應用程式伺服器時，絕不是在應用程式伺服器之中定義一大堆的物件讓用戶端應用程式呼叫。這樣不但會增加用戶端應用程式設計師瞭解系統架構的困難度，事實上也會降低應用系統執行的效率，實在不可不慎。

從上面的兩個範例可以看到程式師使用的遠端呼叫方式以及撰寫程式碼的方式和技巧都會深深的影響多層應用系統執行效率，因此程式師實在不可不慎。此外當程式師使用物件導向分析/物件導向設計的方式開發以分散式物件為架構的應用系統時，如何安排和呼叫分散式物件更會對於應用系統的執行效率有巨大的影響，由於本書不是討論如何以分散式物件架構開發應用系統的書籍，因此也就不再這裡多做討論了，希望有機會能夠在其他的書籍中再和各位討論這個重要的主題和技術。

7-5 調校應用程式伺服器的執行效率

在前面的小節中討論了影響多層應用系統重要的因素，也說明了如何使用最有效率的方式進行遠端呼叫。但是多層應用系統除了用戶端應用程式需要最佳化外，應用程式伺服器對於整個系統的執行效率也有重大的影響。因此軟

體開發人員對於應用程式伺服器的執行效率也必須加以注意，在瞭解了如何使用最有效率以及最適合你的應用系統使用的呼叫方式之後，你應該能夠好好的調校你的用戶端應用程式的執行效率。現在讓我們移轉焦點到多層應用系統的中間部份，那就是應用程式伺服器的執行效率。從本小節開始即將對於如何調整應用程式伺服器做一個詳細的討論。

7-5-1 TDataSetProvider 元件的設定

在開發多層應用系統時，遠端資料模組之中的 TDataSetProvider 元件是一個非常重要的元件，因為許多重要的事件都是由它觸發的，此外用戶端在異動資料回資料庫時，也是由它真正負責更新資料回資料庫。既然 TDataSetProvider 元件是從後端資料庫中取得資料和異動資料回資料庫的關鍵元件，當然它的執行效率就影響了整個多層應用系統的表現。所以增加 TDataSetProvider 的效率就等於增加了多層應用系統的效率。

在試著增加 TDataSetProvider 元件的效率之前，你應該先閱讀前面討論多層應用系統如何異動資料的章節。現在就讓我們看看 TDataSetProvider 元件特性值的設定會對應用程式伺服器執行行為有什麼影響。

TDataSetProvider 元件的 UpdateMode 特性值會影響 TDataSetProvider 元件產生什麼樣的 SQL 敘述來更新資料。在內定上 TDataSetProvider 元件的 UpdateMode 是設定為 upWhereAll，這是最嚴格的設定。因為當 TDataSetProvider 元件要更新一筆資料之前，它會先使用這筆資料尚未被異動之前的舊欄位資料先試著在資料庫中找到這筆要異動的資料。當 TDataSetProvider 元件在找尋資料時，UpdateMode 如果設定為 upWhereAll，那麼 TDataSetProvider 元件只會在它從資料庫中找到和所有舊欄位數值完全一樣的記錄時，才允許應用程式伺服器更新它。如果 TDataSetProvider 元件無法找到和所有舊欄位數值完全一樣的記錄時，它就會產生一個許多人熟悉的錯誤『Record has been changed by another user』。

到底 UpdateMode 設定為 upWhereAll 是讓 TDataSetProvider 元件執行什麼樣的程式碼呢？下圖便是一個 TDataSetProvider 元件的 UpdateMode 設定為 upWhereAll 的情形：

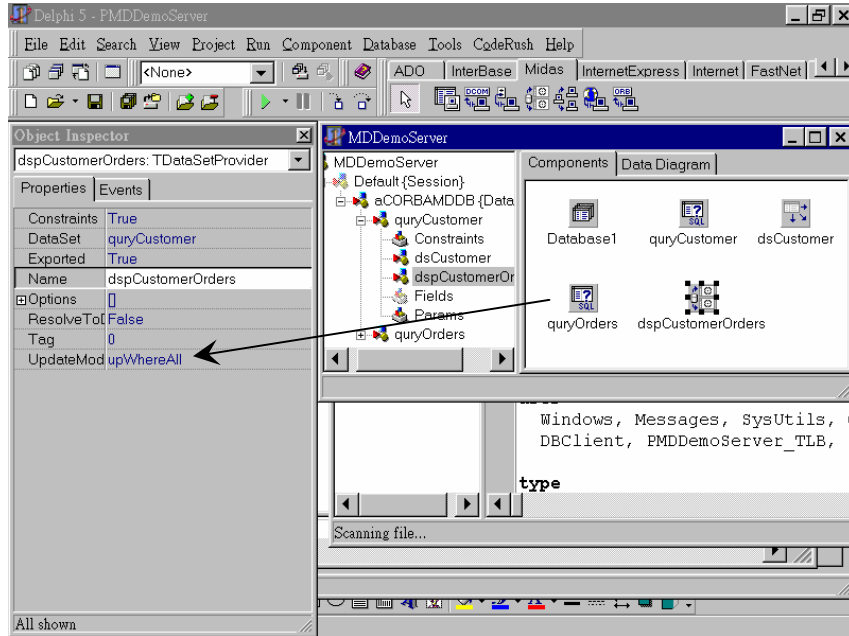


圖 7-5 TDataSetProvider 元件設定 UpdateMode 為 upWhereAll

當應用程式伺服器的 TDataSetProvider 如此設定時，那麼當圖 7-6 的用戶端應用程式執行並且異動資料時：

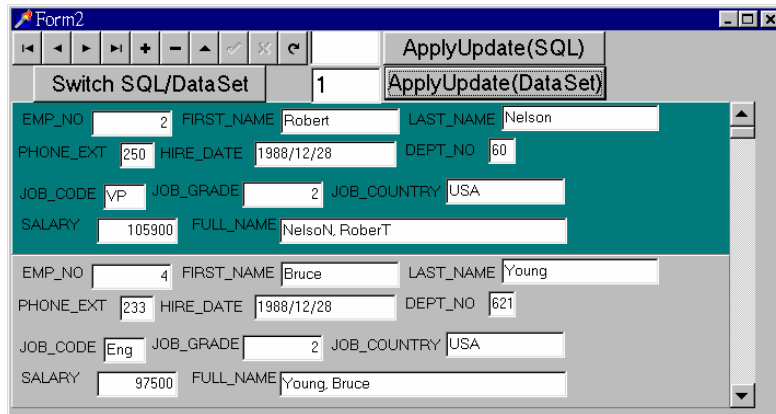


圖 7-6 異動資料的用戶端應用程式

從 SQL Monitor 中便可以看到如下的結果：

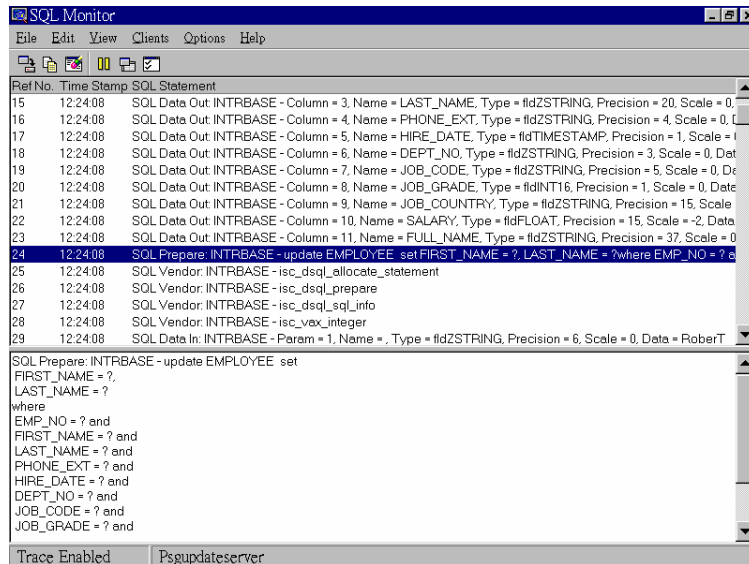


圖 7-7 UpdateMode 設定為 upWhereAll 時 BDE/IDAPI 的 SQL 敘述

從上圖中可以看到，當 UpdateMode 設定為 upWhereAll 時，TDataSetProvider 元件在產生 SQL 敘述更新資料時，在 where 句子中它會使用所有的舊欄位資料來尋找要被更新的資料。如果在這個 SQL 敘述執行之前，已經有其他的使用者在其他的機器中異動了這筆資料，那麼 TDataSetProvider 元件就會產生『Record has been changed by another user』的錯誤。

從上面的觀察知道，TDataSetProvider 元件的 UpdateMode 特性會影響產生的 SQL 敘述。如果使用 upWhereAll 的話，不但是最嚴格的方式，由於在尋找資料時也必須比對所有的欄位數值，所以是比較緩慢的更新方式。除了 upWhereAll 之外，UpdateMode 也可以設定為 upWhereKeyOnly 和 upWhereChanged。一般來說，如果應用程式不允許使用者改變記錄的鍵值，那麼把 UpdateMode 設定為 upWhereKeyOnly 那麼應用程式伺服器將會執行的快一點。UpdateMode 設定為 upWhereKeyOnly 是表示當 TDataSetProvider 元件在資料庫中尋找要被更新的資料時，是直接使用資料的索引值來搜尋資料的。下面的圖形是前面的範例中把 TDataSetProvider 元件的 UpdateMode 設定為 upWhereKeyOnly 時由 SQL Monitor 觀察到的程式碼。從這個程式碼中可以看到使用了 upWhereKeyOnly 之後，TDataSetProvider 元件產生的 SQL 敘述是

比較簡潔也比較有效率，因為它直接使用 Employee 的索引欄位值 EMP_NO 來搜尋資料。

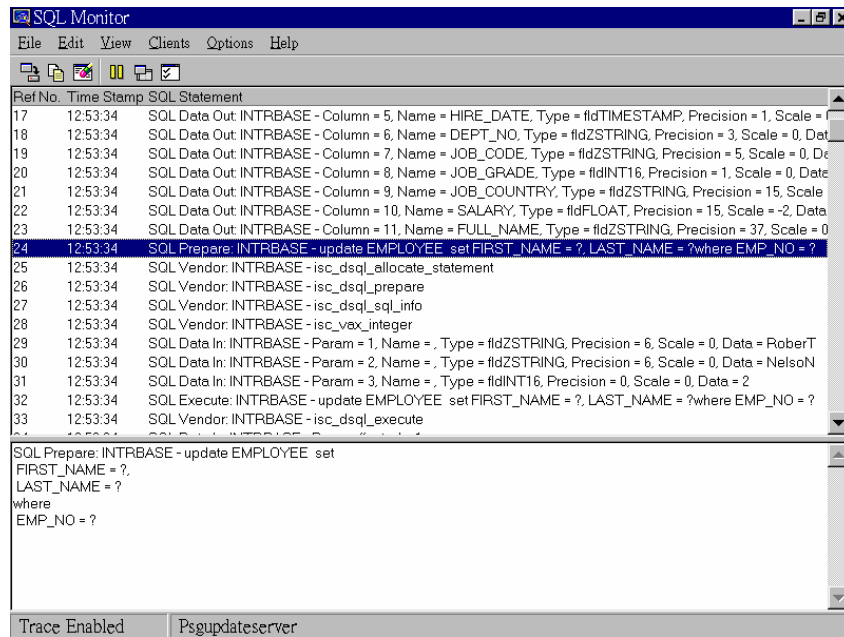


圖 7-8 TDataSetProvider 元件設定 UpdateMode 為 upWhereKeyOnly 時 BDE/IDAPI 的 SQL 敘述

設定 UpdateMode 為 upWhereKeyOnly 不但需要比對的欄位較少，而且使用鍵值通常都可以藉重使用資料庫的索引，因此在尋找資料和異動資料時都會比較快速。所以當程式師在開發應用程式伺服器時，如果不需要最嚴格的資料整合性，或是在鍵值不會被異動的情形下，那麼設定 TDataSetProvider 元件的 UpdateMode 為 upWhereKeyOnly 會有最好的執行效率。

綜合上面觀察的結果，我把 TDataSetProvider 中 UpdateMode 特性值的設定對於 TDataSetProvider 如何產生 SQL 敘述的影響整理成如下的參考供你在選擇使用那一個 UpdateMode 特性值的參考：

UpdateMode 特性值	產生的 SQL 敘述
UpWhereAll	在 SQL 敘述的 Where 中使用所有的資料表欄位，即所有欄位的 ProviderFlags 值只要包含 pflnWhere 值，那麼這個欄位

	就會使用在 Where 子句中做為搜尋的條件
UpWhereKeyOnly	在 SQL 敘述的 Where 中只使用所資料表的鍵值欄位做為搜尋，比對的欄位，即只有包含 pflnKey 值的欄位才會出現在 Where 子句中做為搜尋的條件
UpWhereChanged	在 SQL 敘述的 Where 中使用所資料表的鍵值欄位，以及任何被異動過的欄位做為搜尋，比對的欄位。即包含 pflnKey 值以及被修改過的欄位才會出現在 Where 子句中做為搜尋的條件

7-6 應用程式伺服器中 TQuery 元件的設定

TDataSetProvider 元件除了像上一小節一樣可以自動產生 SQL 敘述更新資料之外，也可以使用它連結的資料集來更新資料。如果 TDataSetProvider 元件是使用資料集更新資料的話，那麼到底是由 TDataSetProvider 元件控制 SQL 敘述或是由資料集控制 SQL 敘述呢？下圖是 TDataSetProvider 元件使用它連結的 TQuery 元件更新資料的結果。由於這個範例中的 TDataSetProvider 元件和 TQuery 元件其 UpdateMode 都是設定為 upWhereAll，所以看起來是和上一節中 TDataSetProvider 元件的 UpdateMode 設定為 upWhereAll 是一樣的。

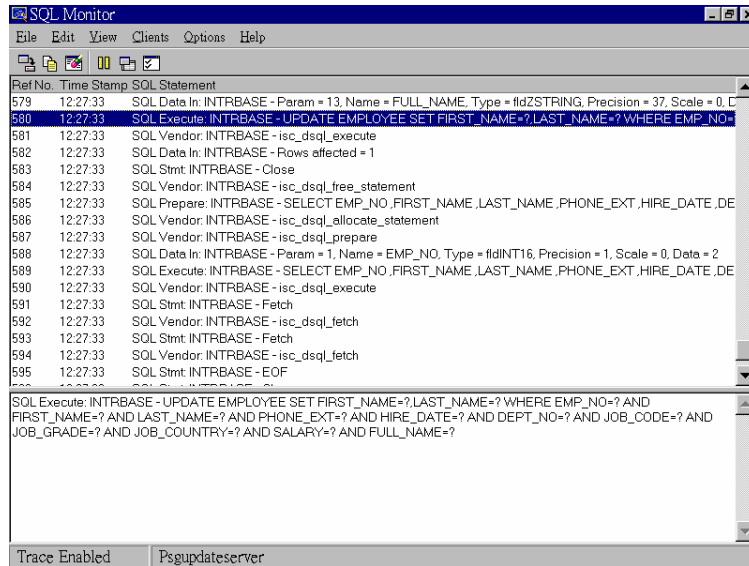


圖 7-9 TQuery 元件設定 UpdateMode 為 upWhereAll 時的 SQL 敘述

現在讓我們把遠端資料模組開啟，並且如下圖般把 TDataSetProvider 元件的 UpdateMode 設定為 upWhereAll，把 TQuery 元件的 UpdateMode 設定為 upWhereKeyOnly，然後再次使用用戶端應用程式異動資料來觀察到底是由誰控制產生的 SQL 敘述，圖 7-11 是執行的結果。

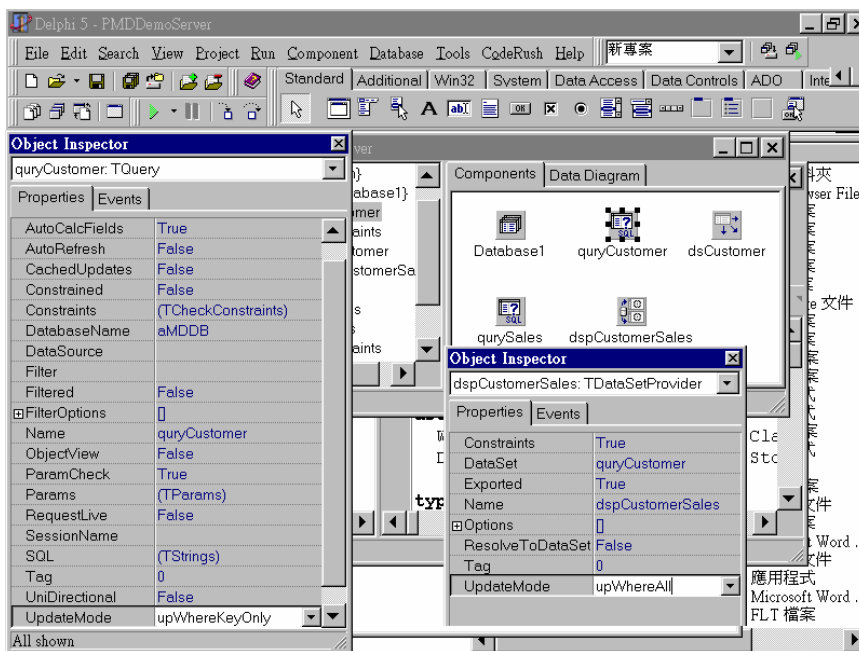


圖 7-10 TQuery 元件設定 UpdateMode 為 upWhereKeyOnly 而 TDataSetProvider 元件的 UpdateMode 設定為 upWhereAll

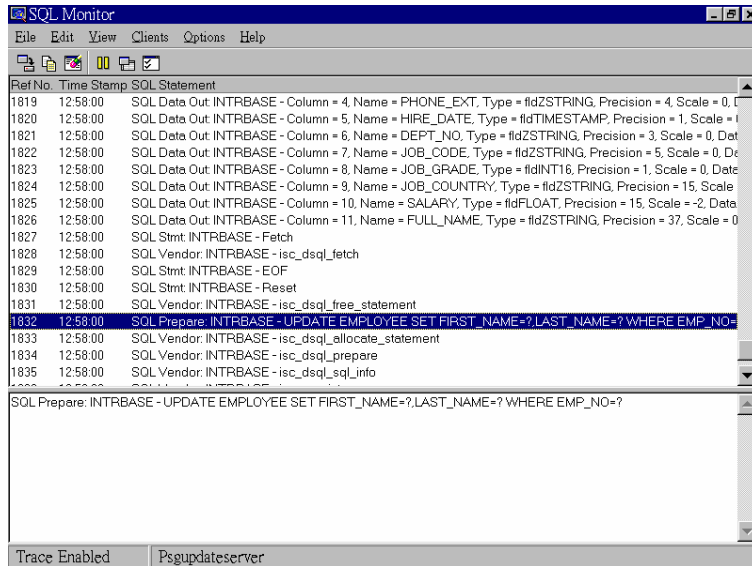


圖 7-11 TQuery 元件設定 UpdateMode 為 upWhereKeyOnly 而 TDataSetProvider 元件的 UpdateMode 設定為 upWhereAll 時的 SQL 敘述

從上圖中可以證明，當 TDataSetProvider 元件是使用它連結的資料集來更新資料時，自動產生的 SQL 敘述是由資料集的設定值來控制的，TDataSetProvider 元件的設定值不再有影響力，所以程式師在使用這個方式之下，應該花精神調整資料集的特性值。

7-7 應用程式伺服器異動資料的行為

當用戶端應用程式傳遞異動資料給應用程式伺服器時，TDataSetProvider 元件會根據它的 ResolveToDataSet 特性來決定如何異動資料回資料庫。當 ResolveToDataSet 特性是 False 時，TDataSetProvider 元件是自己產生更新資料的 SQL 敘述把異動的資料更新回資料庫，而當 ResolveToDataSet 特性是 True 時，它則呼叫它連結的資料集元件把異動的資料更新回資料庫。這兩種不同的方式對於應用程式伺服器的執行效率有什麼不同呢？使用那一種比較有效率呢？要知道答案，我們必須分析這兩種方法在異動資料時執行的程式碼。

下面是使用 SQL Monitor 觀察 TDataSetProvider 元件自己產生 SQL 敘述時，異動兩筆資料回 Employee 資料表的結果。讓我們分析看看這些由 SQL Monitor 產生的程式碼。

```

24      13:09:43  SQL Prepare: INTRBASE - update EMPLOYEE set
FIRST_NAME = ?,
LAST_NAME = ?
where
EMP_NO = ?
...
32      13:09:43  SQL Execute: INTRBASE - update EMPLOYEE set
FIRST_NAME = ?,
LAST_NAME = ?
where
EMP_NO = ?
...
40      13:09:43  SQL Execute: INTRBASE - update EMPLOYEE set
FIRST_NAME = ?,
LAST_NAME = ?
where
EMP_NO = ?
...
52      13:09:43  SQL Prepare: INTRBASE - SELECT EMP_NO, FIRST_NAME,
LAST_NAME, PHONE_EXT, HIRE_DATE, DEPT_NO, JOB_CODE, JOB_GRADE,
JOB_COUNTRY, SALARY, FULL_NAME
FROM EMPLOYEE Employee

```

從上面的程式碼中可以看到當用戶端應用程式異動二筆資料並且讓 TDataSetProvider 元件產生 SQL 敘述更新時，TDataSetProvider 元件首先會在 24 行要求資料庫 prepare 由 TDataSetProvider 元件自動產生的 SQL 敘述，然後分別在 32 行和 40 行執行這個 SQL 敘述。當資料更新回資料庫之後，TDataSetProvider 會再 prepare 一個 Select 敘述從資料庫中取得所有最新的資料。

從這個分析中可以知道，TDataSetProvider 元件自動產生的 SQL 敘述對於一次異動的所有資料只會要求資料庫 prepare 一次 SQL 敘述。每次更新資料時，只是代入不同的動態參數，這種執行方式非常的快速。現在再觀察如果 TDataSetProvider 元件使用資料集更新資料的話，會有什麼結果。

下面是 TDataSetProvider 元件使用資料集更新資料的結果，同樣讓我們分析一下這些程式碼：

```

562      13:07:20  SQL Prepare: INTRBASE - UPDATE EMPLOYEE SET
FIRST_NAME=?,LAST_NAME=? WHERE EMP_NO=?

```

要求資料庫 prepare SQL 敘述

```

...
...
570 13:07:20 SQL Execute: INTRBASE - UPDATE EMPLOYEE SET
FIRST_NAME=?,LAST_NAME=? WHERE EMP_NO=?

```

執行 SQL 敘述

```

571 13:07:20 SQL Vendor: INTRBASE - isc_dsql_execute
572 13:07:20 SQL Data In: INTRBASE - Rows affected = 1
573 13:07:20 SQL Stmt: INTRBASE - Close
574 13:07:20 SQL Vendor: INTRBASE - isc_dsql_free_statement
575 13:07:20 SQL Prepare: INTRBASE - SELECT
EMP_NO ,FIRST_NAME ,LAST_NAME ,PHONE_EXT ,HIRE_DATE ,DEPT_NO ,JOB_CO
DE ,JOB_GRADE ,JOB_COUNTRY ,SALARY ,FULL_NAME FROM EMPLOYEE WHERE
EMP_NO=?

```

從資料庫中重新取得剛才異動的資料

```

...
...
579 13:07:20 SQL Execute: INTRBASE - SELECT
EMP_NO ,FIRST_NAME ,LAST_NAME ,PHONE_EXT ,HIRE_DATE ,DEPT_NO ,JOB_CO
DE ,JOB_GRADE ,JOB_COUNTRY ,SALARY ,FULL_NAME FROM EMPLOYEE WHERE
EMP_NO=?
...
...
588 13:07:20 SQL Prepare: INTRBASE - UPDATE EMPLOYEE SET
FIRST_NAME=?,LAST_NAME=? WHERE EMP_NO=?

```

再次要求資料庫 prepare SQL 敘述

```

...
596 13:07:20 SQL Execute: INTRBASE - UPDATE EMPLOYEE SET
FIRST_NAME=?,LAST_NAME=? WHERE EMP_NO=?
...
601 13:07:20 SQL Prepare: INTRBASE - SELECT
EMP_NO ,FIRST_NAME ,LAST_NAME ,PHONE_EXT ,HIRE_DATE ,DEPT_NO ,JOB_CO
DE ,JOB_GRADE ,JOB_COUNTRY ,SALARY ,FULL_NAME FROM EMPLOYEE WHERE
EMP_NO=?
...
605 13:07:20 SQL Execute: INTRBASE - SELECT
EMP_NO ,FIRST_NAME ,LAST_NAME ,PHONE_EXT ,HIRE_DATE ,DEPT_NO ,JOB_CO
DE ,JOB_GRADE ,JOB_COUNTRY ,SALARY ,FULL_NAME FROM EMPLOYEE WHERE
EMP_NO=?
606 13:07:20 SQL Vendor: INTRBASE - isc_dsql_execute
607 13:07:20 SQL Stmt: INTRBASE - Fetch

```

從上面的程式碼中可以知道，當 TDataSetProvider 元件使用資料集更新資料時。對於每一筆更新的資料都會要求資料庫 prepare 一次 SQL 敘述。這比起前面讓 TDataSetProvider 元件自動產生 SQL 敘述的方式來得非常沒有效率。因為更新多筆資料時，每一筆資料的資料表綱要應該是一樣的，並不需要為相同的 SQL 敘述準備 prepare 這麼多次。但是如果你仔細觀察使用資料集更新資

料時，它在更新之後，只會取回異動的資料，而不是像前面是取得所有的資料。所以使用資料集更新資料的方式雖然在執行更新資料的 SQL 敘述比較沒有效率，但是在從資料庫取得異動之後的資料卻是有效率的多了，

下面是本節觀察的結論重點：

- 角 當 TDataSetProvider 元件是自動產生異動資料的 SQL 敘述時，由於程式無法控制產生的 SQL 敘述，所以在異動資料時最好是一次讓 TDataSetProvider 元件異動多筆資料。因為如果用戶端一次異動一筆資料，那麼 TDataSetProvider 元件會為一筆資料產生一次 SQL 敘述，要求資料庫 prepare 一次 SQL 敘述，再執行這個 SQL 敘述。
- 角 當 TDataSetProvider 元件使用連結的資料集異動資料時，會為每一筆異動的資料 prepare 一次 SQL 敘述，不管用戶端是一次異動一筆資料，或是一次異動多筆資料。所以如果程式師需要更好的執行效率時，可以考慮使用另外的 TQuery 元件執行事先寫好的 SQL 敘述，並且在應用程式伺服器啟動時先 prepare 這些 TQuery 元件的 SQL 敘述，並且在應用程式伺服器結束時 UnPrepare 這些 TQuery 元件的 SQL 敘述。這樣的作法就像前面章節討論異動多個資料表時使用的技巧一樣。如此一來不但可以避免每一筆異動資料都需要 prepare 一次 SQL 敘述的情形，也可以避免 TDataSetProvider 元件需要在異動資料時花時間產生 SQL 敘述，執行效率可以有效的增加。
- 角 如果你不在意使用一個特定的資料庫時，你可以考慮在應用程式伺服器中使用 TStoredProc 元件，在應用程式伺服器異動資料時直接呼叫資料庫之中的儲存程序 (Stored Procedure) 來更新資料。
- 角 讓 TDataSetProvider 元件使用資料集元件更新資料，再配合執行已經 Prepare 好的 SQL 敘述會是最有效率的方式。



請注意，上述觀察的結果可能因為你使用的 BDE/IDAPI 版本或是使用的資料庫而有所不同的行為，你應該學習上面討論分析的精神來觀察你的應

用程式的執行行為，進而根據你使用的 BDE/IDAPI，資料庫和應用程式的架構來調整應用程式伺服器的執行效率。

7-8 DCOM 應用程式伺服器 vs. Socket (TCP/IP) 伺服器

在程式師開發應用程式伺服器時，通常都是開發使用 DCOM 通訊協定的應用程式伺服器或是使用 Socket 通訊協定的應用程式伺服器。雖然使用 DCOM 或是使用 Socket 通訊協定各有優缺點，但是許多程式師更關心的問題應該是使用那一種應用程式伺服器比較有效率？為什麼那一種通訊協定比另外一種通訊協定有效率？本小節的目的就是討論這些問題，以便讓程式師有比較好的選擇。

為了先測試到底是 DCOM 應用程式伺服器比較有效率，還是 Socket 應用程式伺服器有效率。我撰寫了四支應用程式，分測試 DCOM 應用程式伺服器和 Socket 應用程式伺服器的效率。

圖 7-12 是 DCOM 應用程式伺服器和 Socket 應用程式伺服器執行畫面在隨機新增 10000 筆資料時執行的畫面之一。從圖中可以看出 DCOM 應用程式伺服器是比 Socket 應用程式伺服器來得有效率。

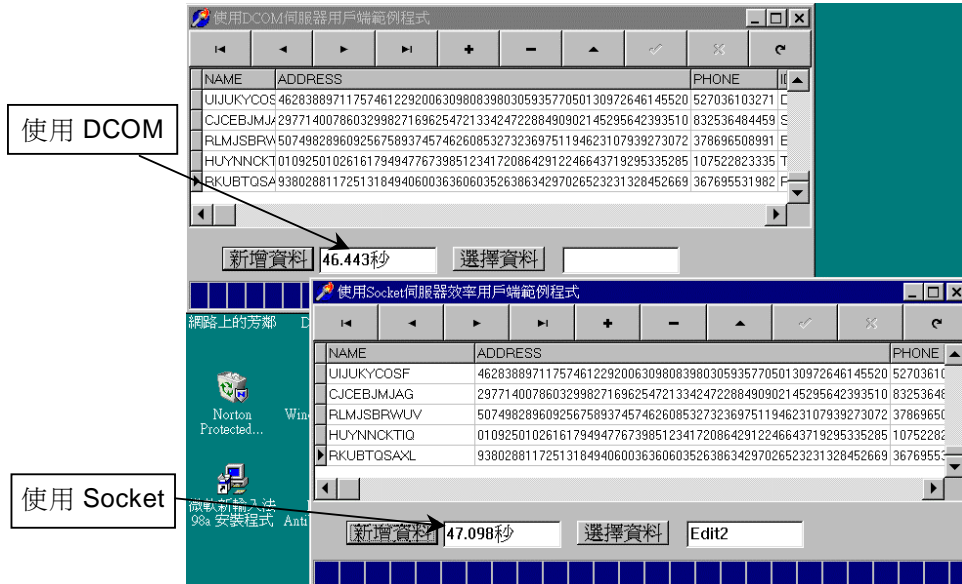


圖 7-12 DCOM 應用程式伺服器 和 Socket 應用程式伺服器執行畫面 (用戶端應用程式和應用程式伺服器都在同一台機器)

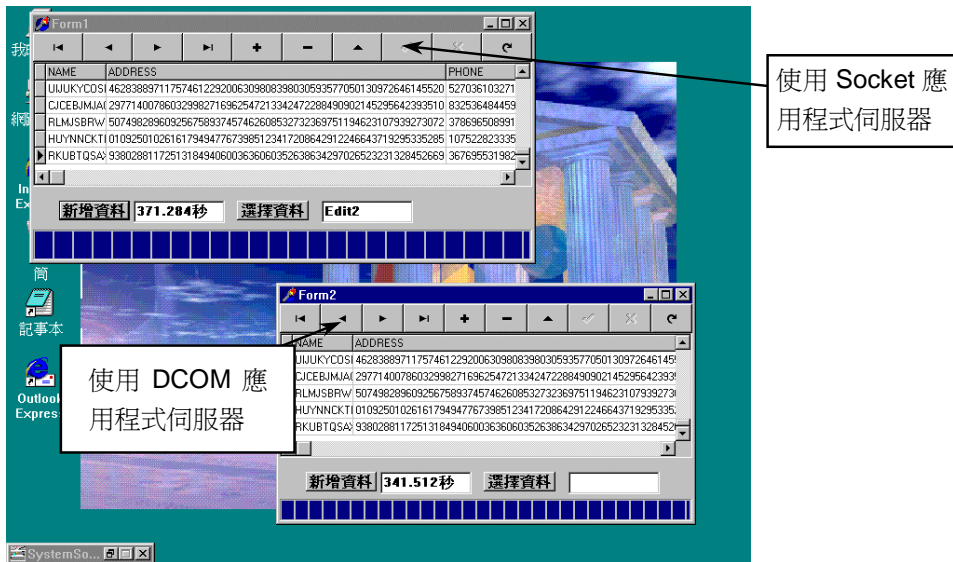


圖 7-13 DCOM 應用程式伺服器 和 Socket 應用程式伺服器執行畫面 (用戶端應用程式和應用程式伺服器在不同的機器)

下面的表格是 DCOM 應用程式伺服器 和 Socket 應用程式伺服器執行的時間比較表，圖 7-14 則是使用圖形來顯示這兩種應用程式伺服器的執行效率比較。

用戶端應用程式和應用程式伺服器在同一台機器	花費時間
使用 DCOM 通訊協定	46.443
使用 Socket 通訊協定	47.098
用戶端應用程式和應用程式伺服器在不同台機器	花費時間
使用 DCOM 通訊協定	341.512
使用 Socket 通訊協定	371.284

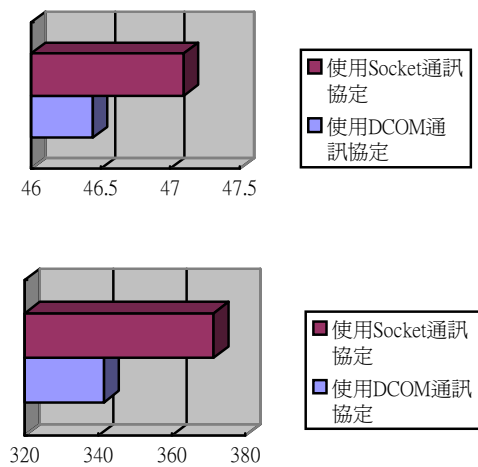


圖 7-14 DCOM 應用程式伺服器和 Socket 應用程式伺服器執行效率圖

在我的各種測試中，一般來說 DCOM 應用程式伺服器會比 Socket 應用程式伺服器有 5 % 到 10 % 的差異。當然，實際的差距會根據應用程式的型態，機器的組態而受到影響。雖然 DCOM 應用程式伺服器在執行速度上享有優勢，但是 DCOM 初次連結的速度卻令人不敢恭維。我在許多朋友或是其他地方看到用戶端應用程式初次連結 DCOM 應用程式伺服器時，都需要花費許多的時間。在我自己已經調整過的機器中進行範例 DCOM 應用程式伺服器和 Socket 應用程式伺服器初次連結的時間測試，其結果如下表：

	花費時間
--	------

使用 DCOM 初次連結	2 到 5 秒
使用 Socket 初次連結	小於 1 秒

從這個表格中可以看到 Socket 應用程式伺服器在連結速度上是明顯的快於 DCOM 應用程式伺服器。我想對使用 DCOM 做為連結通訊協定的應用程式伺服器而言，許多程式師可能經常會發現第一支連結應用程式伺服器的用戶端應用程式需要一段不短的時間才能夠完成首次的連結，但是對於使用 Socket 通訊協定的應用程式伺服器而言，用戶端的應用程式卻能夠很快的連結上。為什麼使用 DCOM 的應用程式伺服器需要如此久的時間？能夠改善 DCOM 的初次連結時間嗎？

DCOM 初次連結需要花費許久的時間是一個普遍的現象，雖然我無法確定真正的原因，但是我的觀察是有兩個主要的原因會造成 DCOM 延遲的連結速度。

第一個影響效率的因素便是用戶端機器中所使用的網路連結通訊協定。當程式師分發多層應用系統時，請注意用戶端的機器是否有同時安裝多種不同的網路連結通訊協定。如果用戶端機器同時使用了多種網路通訊協定將會影響多層應用系統的執行效率。例如下圖是筆者觀察的一個用戶端機器，由於這台機器同時使用了數種不同的通訊協定，所以在這台機器中用戶端應用程式在連結和執行的速度上並不很快速。



圖 7-15 用戶端機器使用太多不同的通訊協定會影響執行效率

第二個影響的因素是機器中 DNS 的設定，這個設定會大幅的影響用戶端機器在連結應用程式伺服器時的速度。如果用戶端機器設定了許多不同的

DNS，那麼用戶端應用程式在連結 DCOM 應用程式伺服器時需要花費很長的時間（有時實在令人受不了）。所以在分發多層應用系統時，程式師或是安裝人員如果發現用戶端應用程式在初次執行時需要很久的時間，那麼請檢查一下機器中 DNS 的設定。圖 7-16 是筆者一台機器的設定，由於這台機器設定了兩個 DNS，所以在初次連結時，DCOM 似乎會去這兩個 DNS 中搜尋應用程式伺服器的機器，以致造成連結的緩慢。這是系統開發人員在安裝或是微調應用系統時必須注意的事情。



圖 7-16 電腦中的 DNS 設定會影響 DCOM 初次的連結速度

現在回到為什麼 DCOM 的應用程式伺服器在執行時會比 Socket 應用程式伺服器有效率，這是因為 DCOM 通訊協定在傳送資料時會對傳遞的資料進行壓縮的動作。此外 Microsoft 對於 DCOM 也做了一些效率上的調整，例如當 DCOM 在取得遠端伺服器端 COM/DCOM 物件的方法，或是呼叫介面之中的方法時會把數個介面或是呼叫的方法封裝在一個網路的封包之中，這樣可以減少網路的負荷進而增加應用系統的執行效率，這些都是 DCOM 應用程式伺服器能夠比 Socket 應用程式伺服器比較有效率的原因。

在瞭解了前面的分析之後，要使用那一種的應用程式伺服器就需要程式師自己考慮了。是需要連結速度較快的應用程式伺服器或是比較的有效率的應用程式伺服器。一般來說如果你希望應用系統有嚴格的安全控制的話，那麼使用 DCOM 是比較好的選擇，但是如果你的多層應用系統需要使用在 Internet 或是需要跨越數個 router 或是網域的話，那麼使用 Socket 是比較簡單也比較可行的。

7-9 欄位物件的 TDataSetProviderFlags 特性

在前面討論如何開發多層應用系統時，已經說明了 TField 元件的 ProviderFlags 特性的意義以及它的功能。本小節只是提醒程式師，ProviderFlags 特性可以控制 TDataSetProvider 元件如何異動資料的行為，此外當 TDataSetProvider 元件在異動資料之前，一定需要先在資料庫中找到要被異動的資料，而欄位物件 ProviderFlags 的 PfInWhere 值就決定了 TDataSetProvider 會使用那些欄位來搜尋資料。

當 TDataSetProvider 元件搜尋資料時，它會檢查是否有欄位是 pfInKey 的值，如果有的話，就會使用這個索引欄位來輔助搜尋資料。由於 pfInKey 欄位通常可以利用資料庫的索引，所以可以快速的找到資料。這是程式師可以增加 TDataSetProvider 元件執行效率的地方之一。此外就如同前面章節所說的，善用 ProviderFlags 可以讓程式師隨心所欲的控制用戶端能夠異動的資料。

最後要提醒各位的是，雖然 ProviderFlags 賦予了程式師絕大的控制能力，但是由於 TDataSetProvider 元件在異動資料時，對於每一筆異動的資料都會檢查欄位物件的 ProviderFlags 值，所以如果程式師在 BeforeUpdateRecord 等事件處理函式中執行了太多控制 ProviderFlags 值的程式碼，那麼應用程式伺服器整體的執行效率就會緩慢下來。所以程式師要記得在控制能力和執行效率之間必須做一個良好的規劃。

7-10 COM 執行緒模型的限制

不管程式師如何的調整 Delphi 的 MIDAS 元件，只要是使用 Delphi 的 TDCOMConnection 元件製作使用 COM/DCOM 的應用程式伺服器，那麼程式師就可以利用所有 COM/DCOM 提供的功能。當然這也表示應用程式伺服器也必須受到目前 COM/DCOM 功能的限制。

當程式師在設計遠端資料模組時，Delphi 會詢問程式師要使用什麼樣的 COM/DCOM 執行緒模型，以及什麼樣的 COM 物件樣例。如果程式師能夠適

當而且正確的使用執行緒模型和物件樣例，那麼就可以明顯的增加應用系統的執行效率。在稍後的『COM/DCOM 執行緒模型和多層應用系統』一章中會詳細的說明如何開發更有效率的應用程式，但是在真正討論如何在多層應用系統中充分利用 COM/DCOM 的執行緒模型之前，我們必須先知道一些 COM/DCOM 在執行緒模型方面的限制。

多層應用系統使用 COM/DCOM 執行緒模型能夠增加應用程式執行效率的原因，基本上是因為應用程式伺服器能夠為每一個它所服務的用戶端產生一個獨立的執行緒，所以每一個用戶端應用程式基本上不會受到其他用戶端應用程式的影響。但是一個應用程式伺服器能夠在合理的執行效率下同時服務多少個用戶端應用程式是受到作業系統對於執行緒管理的因素所影響的。在 COM/DCOM 中，Microsoft 的文件說明當一個伺服器產生多於 16 個執行緒時伺服器執行的效率就會開始下降。但是我的實驗結果顯示，應用程式伺服器產生的執行緒如果超過 30 個時，執行效率才有開始下降的趨勢。請注意執行效率開始下降並不代表執行速度不好，而是指執行速度開始受影響。上面話如果使用 Delphi 的術語的話，可以說成如果每一個用戶端應用程式在連結應用程式伺服器時都使用獨立執行緒服務的話，那麼就代表當有 30 個用戶端應用程式連結相同的應用程式伺服器時，應用程式伺服器的執行效率就可能開始下降。

當然這只是一般的情形，如果你的應用程式伺服器設計的良好而且伺服端機器的硬體設備也好的話，那麼可能可以增加一個應用程式伺服器能夠服務的用戶端機器。但是一台單一的伺服端機器所能服務的用戶端也受到 BDE/IDAPI 的影響，請參考下一節的說明。

如果你閱讀過上一章『COM/DCOM 執行緒模型和多層應用系統』的話，那麼你應該知道 Delphi 的多層應用系統的執行行為和執行效率是和 COM/DCOM 的執行緒模型有很大的關連的(當然，這是指你使用 DCOM 或是 Socket 連結連結通訊協定)。就目前而言，這也說 Windows 2000 還沒有推出之前，使用 Apartment 執行緒模型是最適合的 COM/DCOM 執行緒模型。因為在上一章我們已經瞭解在 COM/DCOM 中切換不同的執行緒模型時是非常耗時的工作，而 Delphi 5 的核心 DLL-MIDAS.DLL 便是使用 Apartment 的執行緒模型。因此各位撰寫的應用程式伺服器最好便是使用 Apartment 執行緒模型，不

必使用 Free 或是 Both 執行緒模型，這樣可以保證你的應用程式伺服器擁有最好的執行效率。此外 MTS 也是使用 Apartment 執行緒模型，所以即使你的中介軟體是使用 MTS，使用 Apartment 也有最好的執行效率。

但是到了 Windows 2000 情形就不一樣了，Microsoft 在 Windows 2000 中終於改變為了相容 Windows 3.1 而無法提供最適合的 COM/DCOM 執行緒模型的歷史包袱。Windows 2000 的執行緒模型不但在延展性上更為良好，也提供了一個最適合使用在應用程式伺服器和分散式物件之中的執行緒模型-Neutral 執行緒模型。我相信在未來分散式多層應用系統在 Windows 2000 中將有更好，更有執行效率的表現。

7-11 資料存取引擎的限制-BDE/IDAPI 和 ADO

許多我認識的朋友經常詢問我，在開發分散式多層應用系統時，一台執行應用程式伺服器的機器到底能夠服務多少台用戶端機器？對於這個問題通常很難回答，因為這個問題所牽涉到的東西包括了執行應用程式伺服器的機器硬體設備的能力，應用程式伺服器是如何撰寫的，前一小節所討論的 COM/DCOM 執行緒模型的限制，以及本節所要討論的 BDE/IDAPI 的限制。所以要回答一台執行應用程式伺服器的機器到底能夠服務多少台用戶端機器這個問題，必須至少在瞭解了這些東西之後，才能夠有一個基本的答案。為什麼說知道了這些東西之後只能有基本的答案呢？因為在實際建製多層應用系統的執行環境時，除了這些考慮之外，程式師還必須根據實際用戶端應用程式執行的效率來微調用戶端的數目。此外在開發應用程式伺服器時也可以使用一些進階的技巧來增加應用程式伺服器可以服務的用戶端的數目。所以程式師必須有通盤的瞭解才能夠建製出一個良好的分散式執行環境，這也是經驗的程式師的價值所在。

例如在 Delphi 3.0 實戰篇中提到了應用程式伺服器執行時使用 Windows NT 的 Task Manager 觀察它情形。在圖 7-17 中是一個應用程式伺服器機器在執行應用程式伺服器之前記憶體和 CPU 使用的情形，而圖 7-18 則是應用程式伺服器在啟動之後資源使用的情形。

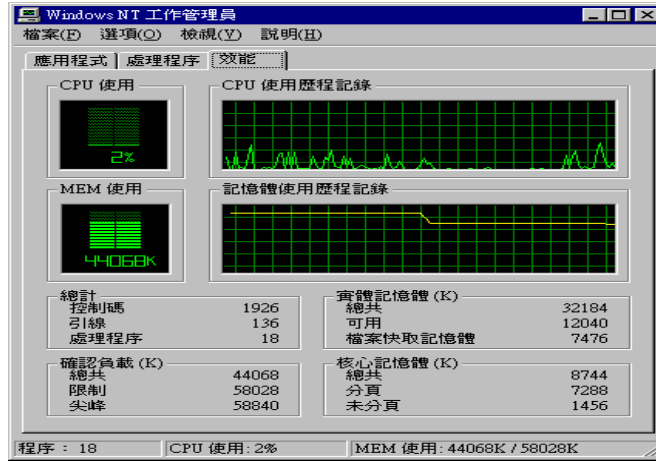


圖 7-17 應用程式伺服器執行前的資源使用情形

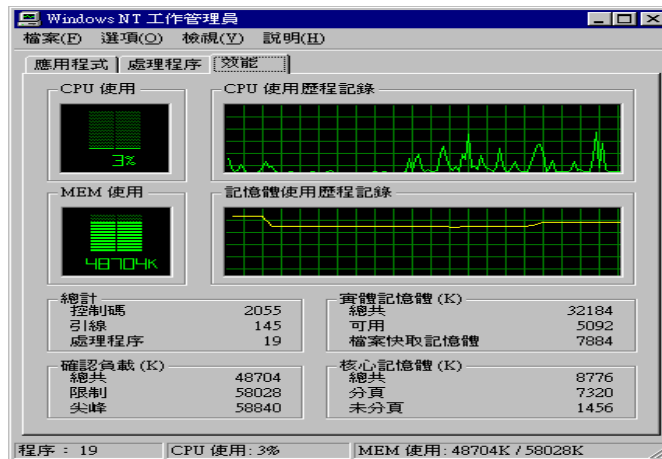


圖 7-18 應用程式伺服器執行後的資源使用情形

從這二個圖形中可以看到一個應用程式伺服器（這個應用程式伺服器是 Multiple Instance 型態）需要大概 4M 到 6M 的記憶體。而圖 7-19 則是用戶端應用程式在新增 1000 筆資料當時資源使用的情形，你可以看到在應用程式伺服器真正的更新資料到後端資料庫的時候會需要額外的記憶體和 CPU 使用的時間：

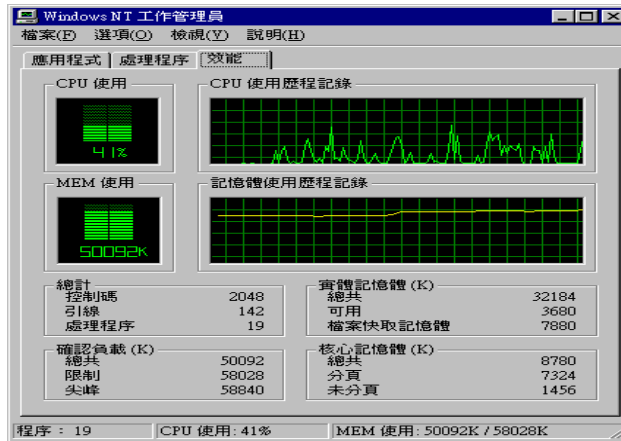


圖 7-19 應用程式伺服器執行後的資源使用情形

一般來說，應用程式伺服器需要的 4M 到 6M 的記憶體，但是當用戶端連結的機器增加時，它需要的記憶體就會增加。此外當應用程式伺服器處理的資料筆數比較多的時候，它也會需要額外的記憶體。不過一台應用程式伺服器在理論上能夠服務的用戶端機器必須受限於 BDE/IDAPI 的限制。因為應用程式伺服器是使用 BDE/IDAPI 存取資料庫的（當然，如果你使用 ODBC 的話，那麼就受到 ODBC 的限制）。

BDE/IDAPI 目前限制一台執行 BDE/IDAPI 的機器最多只能同時有 48 個不同的連結資料庫的 Session，而每一個 Session 可以擁有最多 256 個用戶端使用者。這也就是說，在一個分散式多層應用系統中，如果每一個用戶端使用一個獨立的 Session 連結資料庫的話，那麼一台應用程式伺服器最多只能服務 48 台用戶端機器（在每一台用戶端機器執行一支應用程式的情形下，如果用戶端同時執行多個使用不同的 Session 的應用程式，那麼應用程式伺服器可以服務的用戶端機器會再減少）。而如果每一個用戶端應用程式在使用應用程式伺服器時，應用程式伺服器遠端資料模組中的 TDatabase 元件的 Handle shared 設定為 True，那麼就代表大家都是使用相同的 Session 連結資料庫，那麼應用程式伺服器最多可以服務 256 台用戶端機器。

由於 BDE/IDAPI（或是 ODBC）的限制，所以當多層應用系統有許多用戶端機器的狀況下，通常都需要有二台以上的應用程式伺服器。這樣不但可以服務數目較多的用戶端，也可以提供平均負荷和容錯能力的功能。當然，在有些情形下，可能一台應用程式伺服器需要服務多於 48 個用戶端，例如股票查詢系統，或是商品查詢系統。如果你需要開發這種應用系統，那麼你應該使用無

狀態物件以及使用 session pooling 的技術巧。所謂 session pooling 是指應用程式伺服器建立一定數目的 session 例如 40 個，然後不管有多少連結的用戶端都只使用這 40 個 session 來連結資料庫，如此一來就可以突破 BDE/IDAPI 或是 ODBC 的限制了。在 Delphi 的 Demos\MIDAS\Pooler 子目錄之下就有一個 session pooling 的範例。

下面的內容是前面幾個小節的整理，你可以參考一下是不是瞭解了這幾節的說明：

如果程式師在設計應用程式伺服器時，是設定 TDatabase 元件的 HandleShared 為 True，那麼所有的連結到這個應用程式伺服器的用戶端應用程式都使用相同的 Session 連結到資料庫。即使每一個用戶端都使用獨立的執行緒服務，每一個用戶端應用程式彼此仍然會影響對方。而應用程式伺服器在產生超過 30 個用戶端應用程式時，應用程式伺服器的執行效率就可能開始下降。一台應用程式伺服器最多可以服務 256 個用戶端應用程式。

如果程式師在設計應用程式伺服器時，是設定 TDatabase 元件的 HandleShared 為 False 而且在遠端資料模組中使用 TSession 元件並且設定它的 AutoSessionName 特性值為 True，那麼所有的連結到這個應用程式伺服器的用戶端應用程式都使用獨立的 Session 連結到資料庫。那麼即使每一個用戶端都使用獨立的執行緒服務，一台應用程式伺服器最多可以服務 48 個用戶端應用程式。

Delphi 5 在資料存取方面有了非常大的突破，那就是 Delphi 5 支援了原生的 ADO 元件，讓 Delphi 的程式師可以使用 ADO 存取各種資料庫，例如 MS SQL Server 7.0，Access 2000，Oracle 等。那麼 Delphi 的程式師要如何選擇使用那一個資料存取引擎來存取資料呢？事實上就筆者的觀點來看，使用 ADO 似乎是不可避免的趨勢，因為 ADO 不但可以存取各種關連資料庫，也可以存取其他來源的資料，所以是一個通用的資料存取介面。ADO 允許平台使用一個統一的介面存取各種不同形式的資料。因此不但 BDE/IDAPI，連 ODBC，DAO/Jet Engine 等資料存取引擎將來都會慢慢的被取代掉。ADO 在存取關連資料庫方面的表現愈來愈好，延展性/穩定性也在每一個版本中不斷的提昇。但是我們是不是現在就要捨棄 BDE/IDAPI/ODBC 使用 ADO 呢？這就和你使用的關連資料庫有關了，就目前來說如果你使用的資料庫是屬於 Microsoft 系列的話，那麼使用 ADO 絕對是正確的，例如 Microsoft SQL Server 7.0 只有使

用 ADO 才能夠充分的發揮它的功能，而 BDE/IDAPI/ODBC 都無法如此。但是如果你是使用其他的資料庫，例如 Oracle，Sybase 或是 Informix 的話，那麼我想目前還不是使用 ADO 的時機，因為這些關連資料庫目前還沒有真正的 ADO 驅動程式，因此目前還是使用 BDE/IDAPI 存取這些資料庫是最穩定也是最有效率的方式。在未來必須等這些關連資料庫廠商真正的為他們的資料庫推出 ADO 驅動程式之後才適合使用 ADO 存取這些關連資料庫。

雖然目前你可以使用 ODBC Provider For ADO 來存取 Oracle，Sybase，或是 Informix 等目前沒有真正 ADO 驅動程式的資料庫時，但是這種存取方式有許多的問題，許多的臭蟲以及效率不彰等的問題。所以你如果使用這種方式存取這些資料庫的話，我勸你還是三思而後行。有關 BDE/IDAPI，ADO 等深入的討論請參考『實戰 Delphi 5.x-高效率資料庫應用系統』一書的說明。

7-12 Interceptor

自從 Delphi 4 開始在多層應用系統加入了一個非常有趣而且重要的技術，那就是所謂的 Interceptor 的技術。什麼是 Interceptor 呢？從字義上說，Interceptor 是『攔截者』的意思，而這種技術也正是這個意思，這種技術可以攔截多層應用系統中用戶端和伺服器進出的資料。

在主從架構或是分散式應用系統中，用戶端和伺服端的互動可以使用下圖來代表：

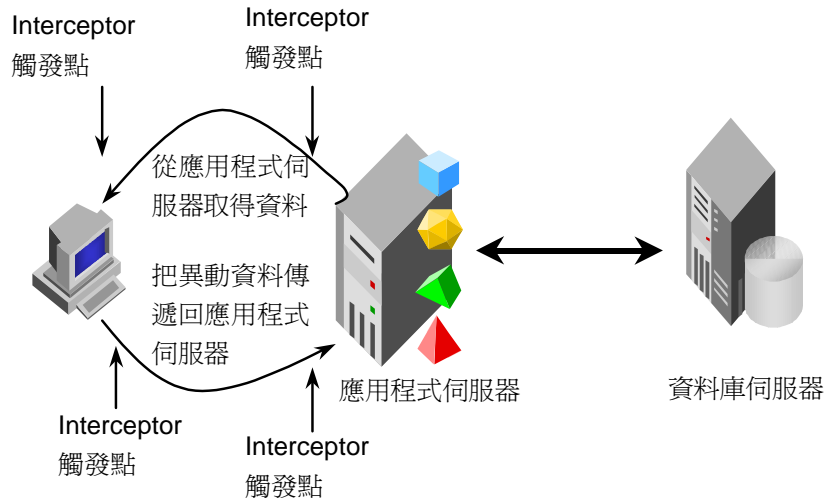


圖 7-20 主從架構或是多層應用系統中的 Interceptor

從上圖中可以看到在主從架構或是多層應用系統中，Delphi 4 加入了四個攔截：

- 角 當資料從應用程式伺服器輸出資料到用戶端時。這是應用程式伺服器的 Data Out 攔截點。
- 角 當資料從應用程式伺服器進入用戶端時。這是用戶端應用程式的 Data In 攔截點。
- 角 當資料從用戶端傳遞到應用程式伺服器時。這是用戶端應用程式的 Data Out 攔截點。
- 角 當資料從用戶端進入應用程式伺服器時。這是應用程式伺服器的 Data In 攔截點。

Delphi 5 允許程式師在這四個攔截點加入特別的程式碼以處理這些在網路中傳遞的資料。那麼使用攔截點有什麼好處呢？嗯，好處很多。如果你的應用系統需要傳遞敏感的資料時，像薪資資料，金融資料，那麼程式師就可以利用這四個攔截點進行資料加密的功能。例如在這四個攔截點呼叫 SSL (Secure Socket Layer) 對資料加密。又例如你的應用系統需要傳遞大量的圖形或是語音資料，由於這些資料通常都很龐大，所以需要花費許多網路傳輸的時間，並且造成網路嚴重的負荷。在這種環境下即使你的機器很快也沒有什麼作用，因

為大部份的時間都花費在網路之上。此時程式師就可以利用這四個攔截點對這些資料做壓縮的動作。由於壓縮可以大幅減少網路需要傳遞的資料量，所以可以大幅降低這個應用程式執行的時間。

Delphi 5 的 Demos\MIDAS\Intrcpt 子目錄之下有一個實際使用攔截點的範例。這個範例就是在分散式多層應用系統中壓縮用戶端和應用程式伺服器中傳遞的資料。由於這個範例實作了攔截點的技術，所以在這裡我就解釋一下這個範例是如何工作的。

要使用攔截點的技術，Delphi 要求程式師必須實作 Delphi 定義的 IDataIntercept 介面。IDataIntercept 介面只定義了兩個虛擬方法 DataIn 和 DataOut。DataIn 和 DataOut 的意義就是前面圖 7-20 的中的四個 Data In 和 Data Out 的意義。所以在這個範例中下面的 TDataCompressor 類別就宣告實作了 IDataIntercept 的 DataIn 和 DataOut 這二個虛擬方法：

```
{
  The interception object needs to implement IDataIntercept
  defined in
  SConnect.pas. This interface has 2 procedures DataIn and DataOut
  described
  below.
}
TDataCompressor = class (TComObject, IDataIntercept)
protected
  procedure DataIn (const Data: IDataBlock) ; stdcall;
  procedure DataOut (const Data: IDataBlock) ; stdcall;
end;
```

由於 DataOut 是當資料離開應用程式伺服器或是用戶端時觸發的，所以在這個方法之中必須壓縮傳遞出去的資料。下面的程式碼就是 DataOut 方法：

```
{
  DataOut is called whenever data is leaving the client or server.
  Use this
  procedure to compress or encrypt data.
}
procedure TDataCompressor.DataOut (const Data: IDataBlock) ;
var
```

```

InStream, OutStream: TMemoryStream;
ZStream: TCompressionStream;
Size: Integer;
begin
  InStream := TMemoryStream.Create;
  try
    { Skip BytesReserved bytes of data }
    InStream.Write (Pointer (Integer (Data.Memory) +
Data.BytesReserved) ^, Data.Size) ;
    Size := InStream.Size;
    OutStream := TMemoryStream.Create;
    try
      ZStream := TCompressionStream.Create (clFastest, OutStream) ;
      try
        ZStream.CopyFrom (InStream, 0) ;
      finally
        ZStream.Free;
      end;
      { Clear the datablock, then write the compressed data back into
the
  datablock }
      Data.Clear;
      Data.Write (Size, SizeOf (Integer) ) ;
      Data.Write (OutStream.Memory^, OutStream.Size) ;
    finally
      OutStream.Free;
    end;
  finally
    InStream.Free;
  end;
end;

```

把傳遞出去的資料寫入 InStream 之中準備壓縮

壓縮傳遞出去的資料到 ZStream 之中

把壓縮資料的大小拷貝到傳遞出去的 Data Packet 之中

把壓縮資料本身拷貝到傳遞出去的 Data Packet 之中

DataOut 方法的觀念非常的簡單，它首先從傳入的 Data Packet 之中取出要傳遞的資料，壓縮這些資料，最後再把經過處理的資料拷貝到 Data Packet 之中。當 DataOut 方法執行完之後，這個經過處理的 Data Packet 就會傳遞到用戶端或是應用程式伺服器。此時在網路上傳輸的資料就是經過處理過的資料。

下面的 DataIn 方法就是執行和 DataOut 相反的程式碼。它首先從傳入的 Data Packet 參數中取得已經經過壓縮的資料，然後解壓縮到 OutStream 之中，最後再把解壓縮的資料拷貝回 Data Packet 之中。在 DataIn 方法執行完畢之後，還完的資料就回傳遞到用戶端或是應用程式伺服器之中。

```

{
  DataIn is called whenever data is coming into the client or server.
  Use this
  procedure to uncompress or decrypt data.
}
procedure TDataCompressor.DataIn (const Data: IDataBlock) ;
var
  Size: Integer;
  InStream, OutStream: TMemoryStream;
  ZStream: TDecompressionStream;
  p: Pointer;
begin
  InStream := TMemoryStream.Create;
  try
    { Skip BytesReserved bytes of data }
    p := Pointer (Integer (Data.Memory) + Data.BytesReserved) ;
    Size := PInteger (p) ^;
    p := Pointer (Integer (p) + SizeOf (Size) ) ;
    InStream.Write (p^, Data.Size - SizeOf (Size) ) ;
    OutStream := TMemoryStream.Create;
    try
      InStream.Position := 0;
      ZStream := TDecompressionStream.Create (InStream) ;
      try
        OutStream.CopyFrom (ZStream, Size) ;
      finally
        ZStream.Free;
      end;
    end;
  { Clear the datablock, then write the uncompressed data back into
  the
    datablock }
  Data.Clear;
  Data.Write (OutStream.Memory^, OutStream.Size) ;

```

把壓縮資料的大小拷貝到傳遞去出的 Data Packet 之中

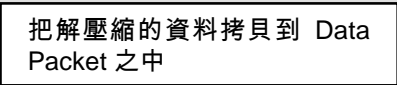
取得壓縮資料本身的大小

取得資料本身的起始處

拷貝壓縮的資料到 InStream 之中

解壓縮傳遞進來的資料到 OutStream 之中

```
finally
  OutStream.Free;
end;
finally
  InStream.Free;
end;
end;
```



經過上面的說明，你應該可以瞭解使用 Interceptor 技術並不困難。只需要實作 DataIn 和 DataOut 方法即可。例如現在你想加入加密的功能，那麼你可以修改這個範例，並且在已經經過壓縮的資料之後，再使用一串字串對這些壓縮的資料進行 XOR 的程式碼。到了 DataIn 方法之中，先使用相同的字串反 XOR 回來，再解壓縮就可以了。這樣不但具有壓縮的功能，也有了初步加密的能力。當然你也可以使用更為複雜的加密功能，例如 SSL 或是 RSA。

事實上 Interceptor 技術早已經在 CORBA 之中使用了，而未來 Microsoft 的 COM+ 中也將會加入 Interceptor 的觀念。看來 Delphi 是第一個引入這個觀念和技術的 RAD 工具。在 Delphi 的 Inrcpt 範例中雖然說明是使用於 Socket 應用程式伺服器，但是使用 DCOM 通訊協定的應用程式伺服器也可以使用這個技術，程式師只需要在適當的事件處理函式中進行這些工作即可。

7-13 結論

本章討論的內容對於任何想要開發分散式多層應用系統的程式師來說都是非常重要的，因為執行效率對於任何應用系統來說都是非常關鍵的因素。當程式師在開發多層應用系統時，應該對於會影響應用系統執行效率的關鍵因素都瞭若指掌，如此才能夠有效的調校應用系統進而提昇系統的執行效率。

除了資料庫伺服器之外，程式師應該儘量減少不必要的遠端呼叫以及在網路上流動的資料量，因為網路的 roundtrip 和資料流量是影響分散式應用系統都關鍵的因素。從這些觀念，我們衍生出了如何使用最有效率的遠端方法呼叫，以避免額外的網路 roundtrip。也瞭解了不同資料型態在傳遞時的成本，因此在傳遞資料時應該使用成本最小的資料型態來傳遞資料，而不應該漫無目的浪費網路的使用率。

