

.NET Framework 核心技術導論



課程簡介



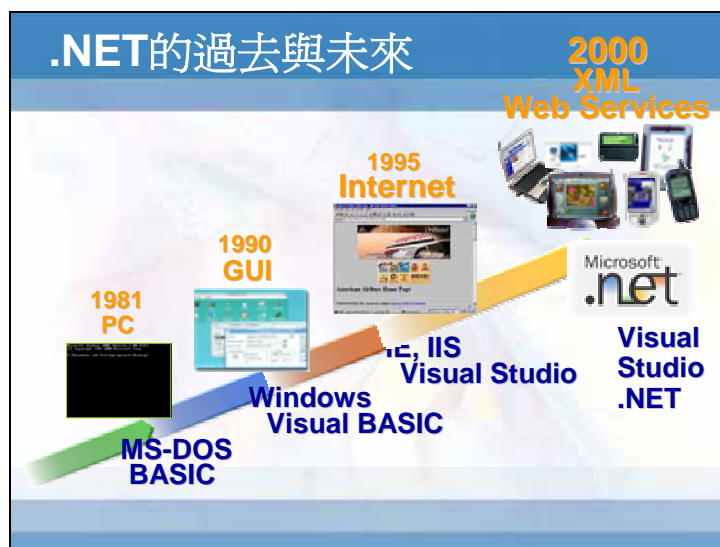
本課程的目的在就.NET Framework 作一簡單的介紹。在進入.NET 主題之前，我們先要了解什麼是.NET，以及.NET 的願景與平台。

在這一章的課程中，將包括：

- Microsoft .NET 的願景與.NET 平台介紹
- .NET Framework 架構
- 共通語言執行期環境(CLR)
- .NET Framework 的應用程式架構 -- 組件(Assembly)
- .NET Framework 類別函式庫
- 共通語言規範與.NET 語言

NOTE:

.NET 平台簡介



在進入.NET 主題之前，我們先要了解什麼是.NET，以及.NET 的願景與平台。在 PC 時代來臨以前，電腦的世界屬於大型主機的天下，而電腦的使用者也只是少數 IT 部門的人員。1980' s PC 的到來，改變了部份企業的工作方式，電腦文書處理取代了傳統打字機，那時後我們用的平台是 MS DOS、BASIC；1990 年，圖型化介面以及 PC 網路的架構的來臨，快速地改變了這個世界與商業運作的模式，我們開始使用 Windows 與 Visual Basic；1995 年 Internet 的到來，推翻了我們溝通的方式，全球吹起了網路風，豐富的資訊、快速的資訊散佈模式，帶給我們一個全新的電子商務環境，我們開始使用 IE、IIS、與 Visual Studio，開發主從架構的 internet 應用程式，及後期以 DCOM 架構分散式應用程式，Windows DNA 的架構。

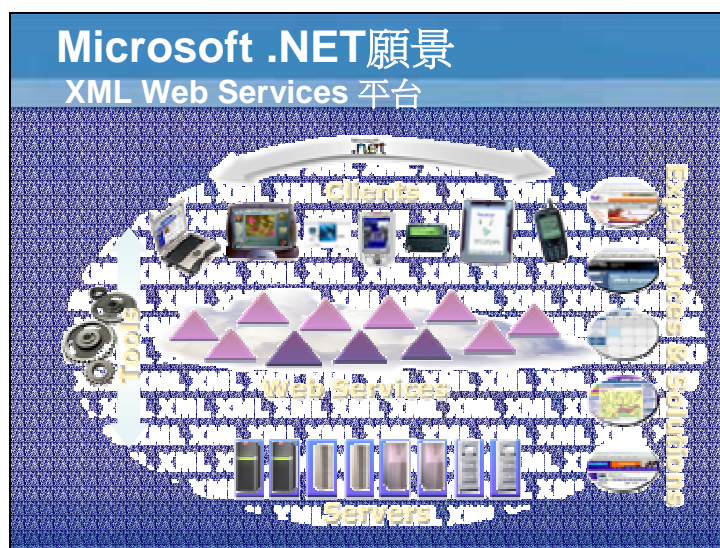
NOTE:

然而到了 2000 年，全球將近有三十億的人口使用 Web，企業與企業間的互動日趨頻繁，使用者對網站的要求不單單只有瀏覽資訊而已，還希望能夠編輯、分析資料、交換資料等，企業開始期待一個嶄新的程式架構，希望能夠讓應用程式、裝置、與企業流程能夠充份的協同合作。

Microsoft 為了解決以上的問題，便提出了一套以 Web Service 為核心的解決方案 Microsoft® .NET。Microsoft .NET 延伸了來自網際網路及作業系統的概念，讓網路本身成為新作業系統的基礎，如此將使開發人員可以輕易的建構出程式，而不再受周邊裝置的侷限，並可透過應用程式充分與網際網路連結。因此 Microsoft .NET 是對現行運算方式的一項重要革新。

NOTE:

.NET 的願景



.NET 之所以對使用者相當重要是因為它讓電腦更易於使用，並具備更強的功能。值得一提的是，它擺脫了以往硬體的限制：讓使用者的資料存在於網路，而非手提電腦中。你可以透過利用各種平臺，包括桌上型電腦、手提電腦、手機或掌上型裝置(PDA)來達到即時傳遞資訊的目的，更可將這些資訊整合於應用程式中。.NET 能讓使用者輕鬆地連結並完成交易，而不需重覆鍵入資料。藉著將多種安全性資料整合入單一使用操作介面的方式，或者甚至是一種程式化的自動決策引擎，.NET 架構將使用者跳脫出目前網頁應用的限制，讓使用者可以隨心所欲地透過任何裝置，在任何時間、任何地點自由地存取資料。

.NET 不僅改變未來應用程式的開發方式，更允許開發人員可以創造出全新的應用程式。這項新發展的核心即是 Web 服務(Web Services)的概念。由於架構在網際網路的開放標準

NOTE:

XML(Extensible Markup Language)之上，透過 SOAP(Simple Object Access Protocol)協定，使得 Web 服務成為網際網路上的一項多功能整合應用服務。

過去，開發人員必須藉助整合本機的系統服務來建構應用程式。這種模式使得開發人員需要使用大量的程式設計資源，並要精準的控制這些應用程式的動作。目前，開發人員正在架構一種複合性的多階(n-Tier)架構系統，這將在網路上整合所有應用程式，然後創造獨到的價值。開發人員所要關注的將只是如何提供與使用這些整合的服務，而非整個系統的重新建構，這樣的結果將節省產品上市的時間、達到更高的開發產能、以及最高品質的軟體。

我們正邁入下一新階段的電腦運算階段，一個由網際網路所觸發的階段，此階段充分地運用的新的網際網路開放標準技術—XML。藉由創新的軟體技術，XML 允許創造出可讓任何人或在任何地方使用的強力應用程式。它增加了應用程式的取得與和軟體的持續連結。在這種方式中，軟體並不單只是從光碟中安裝的東西，而是一種服務，就像是呼叫服務或付費欣賞的電視節目，都需要經由通訊媒介取得資料。

為達到軟體網路化的目標，我們利用兩項特質，一項為更具彈性的多階(n-tier)的運算方式，另一項則是「Web 服務(Web Services)」，其象徵了應用程式發展的下一階段目標。所謂 Web 服務(Web Services)簡單地說，就是一種利用網際網路的標準協定，例如：HTTP、XML，將軟體的功能完全地展現在網際網路或是企業內部網路上運用的軟體服務，你也可以想像它是一種在網頁上的元件編製程序。

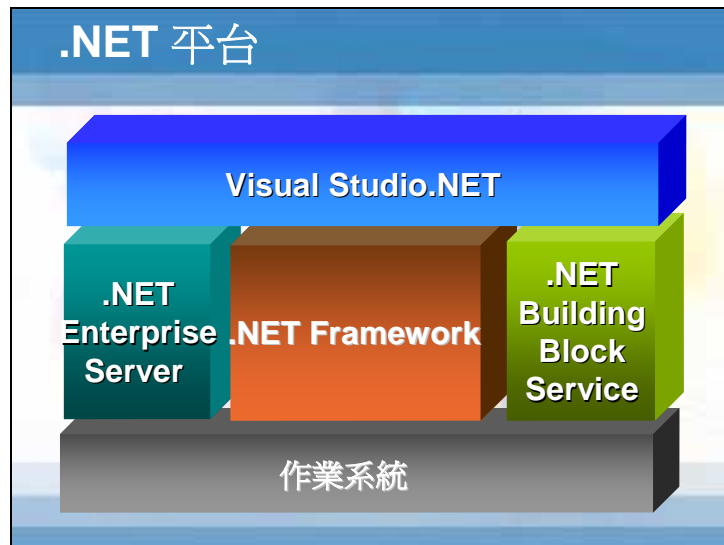
就概念上而言，程式開發者藉由呼叫應用程式介面 (APIs) 將應用程式整合於 Web 服務(Web Service)中，就像呼叫本機服務的

NOTE:

意思一樣。這之間的差別在於 Web 服務呼叫可以透過網路來服務位於遠端的系統。NET 即是架構在這樣的 Web 服務(Web Services)的原則下，微軟目前正透過完整 .NET Framework 平台(包含各項組成部分)，以提供 Web 服務(Web Services)穩固的基礎開發架構。

NOTE:

.NET 平台



.NET 平台大致可分為幾個區塊，最底層為強大穩定的作業系統，尤其是 Windows2000 或是即將上市的 Windows .NET 更是新一代作業系統中最佳的選擇。

在作業系統的基礎上一邊為各式專業分工的伺服器產品，包括 SQL 2000、Exchange 2000、BizTalk、Application Center Server、Content Server 等，另外一邊則為整合許多既有的服務的軟體積木，包括 Passport 身份驗證服務、MSN 即時訊息、行事曆等，提供開發者直接引用。

再來則是一套能夠快速開發解決方案的開發工具，Visual Studio .NET，提供了全新的功能，讓程式開發人員能快速的開發出新一代的應用程式。

中間的部份，則為整個 .NET 應用程式平台的核心，.NET

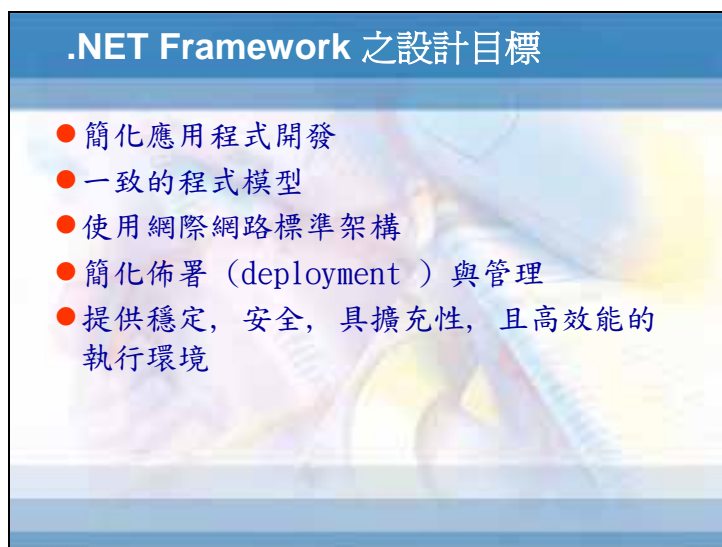
NOTE:

Framework。為什麼 .NET Framework 那麼重要呢？

在開發分散式應用程式時，除了快速開發工具以外，還必須有一套整合與服務這些程式組件的中介軟體，且這一套中介軟體必須能與作業系統充分整合、使用系統所提供的資源與服務。在上一代的應用程式的中介軟體包括 MTS、COM、IIS、MSMQ、以及後來的 COM+，而在 .NET 平台上，如前面所提到的，是以 XML 為基礎的，且要達到 AP-to-AP 與 Programmable Web 的目標，因此需要一個全新的中介軟體，Microsoft 提供給我們一個更強大的中介軟體 .NET Framework。

NOTE:

.NET Framework 之設計目標

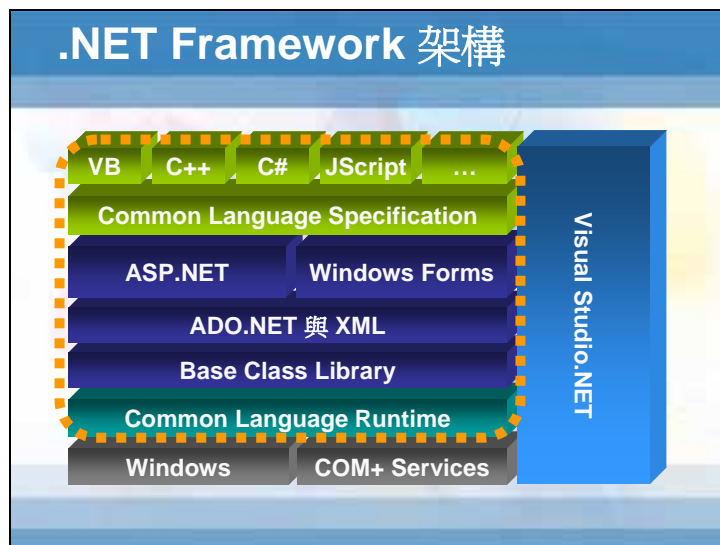


.NET Framework 的設計目標包括：

- 簡化應用程式開發
 - 較少的不必要的雜工、獲取較大的效能
- 一致的程式模型
 - 跨程式語言與應用程式的型別系統
- 使用網際網路標準架構
 - XML、標準通訊協定．．
- 簡化佈署 (deployment) 與管理
 - 元件、版本、可用性
- 提供穩定, 安全, 具擴充性, 且高效能的執行環境

NOTE:

.NET Framework 組成架構



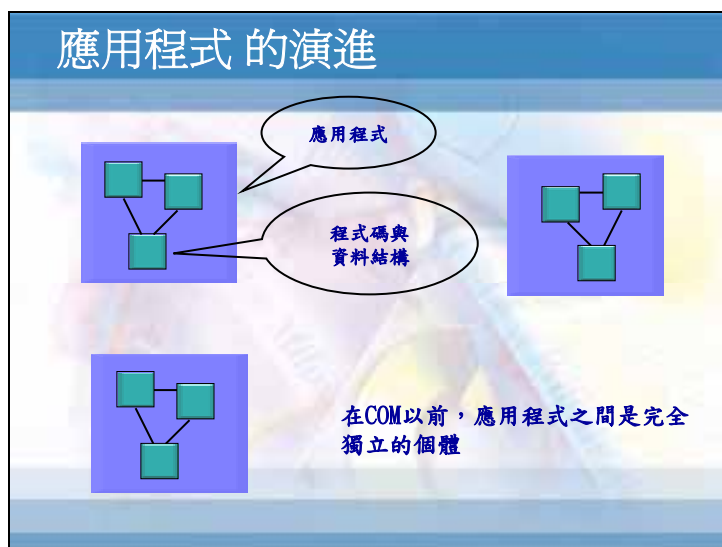
.NET Framework 的架構共分為五大部份，其中 .NET Framework 本身具有的包括共通語言執行期環境 (Common Language Runtime、CLR)、廣泛的類別函式庫，以及定義 .NET 語言的共通語言規範 (CLS)，另外則為承載 .NET Framework 的作業系統平台，及支援 .NET 語言的開發工具 Visual Studio .NET。

最底層的 Common Language Runtime 設計上有點像 Java Virtual Machine，未來開發的 .NET 應用程式，都必須 Run 在 Common Language Runtime 上面。Common Language Runtime 提供了非常多的 Service 讓應用程式在開發上、或執行上更穩定、更快速。

在中間的部份為 Class Library，在以前我們在開發程式的時候，我們可能會用一些 API，COM Library，MFC/ATL，等等，.NET Framework 整合這些 Functionality 提供一致的 Class Library。

NOTE:

Common Language Runtime

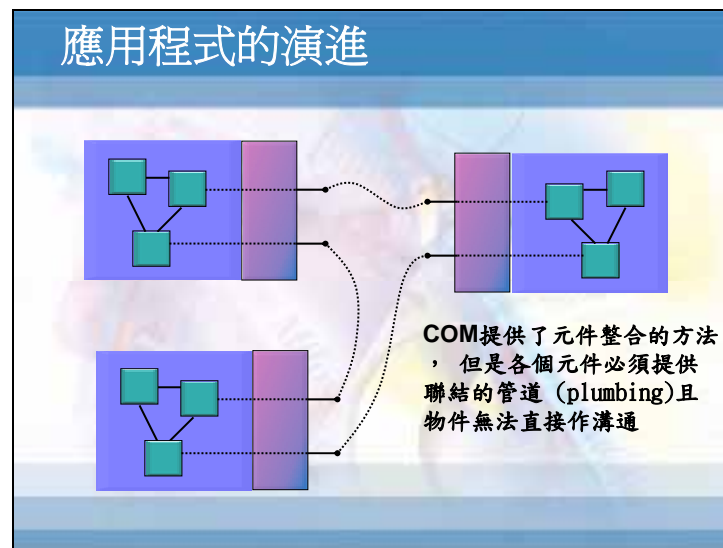


.NET 應用程式必須 Run 在 Common Language Runtime 上。Common Language Runtime 提供了非常多的 Service 讓應用程式在開發上、或執行上更穩定、更快速。

在介紹 CLR 之前，我們先來回顧一下應用程式的演進，在 COM 以前，應用程式之間是完全獨立的個體，應用程式的程式碼與資料結構是個自獨立而封閉的。

NOTE:

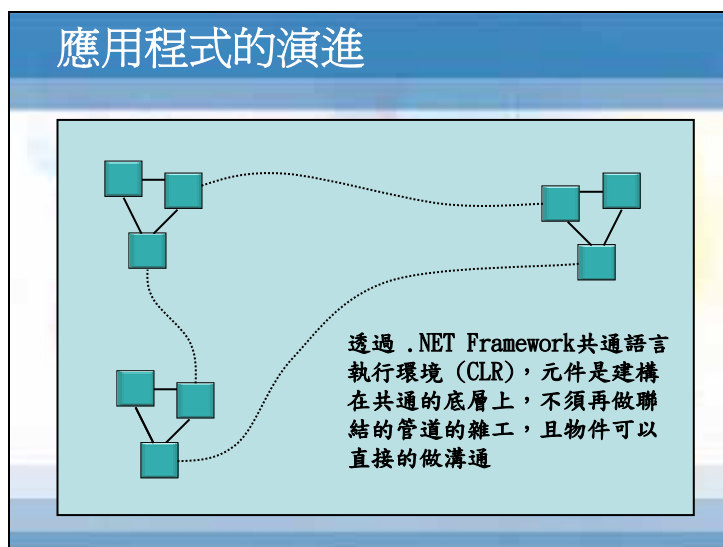
COM 元件的溝通



到了 COM 的時代，COM 提供了元件整合的方法，但是各個原件必須提供聯結的管道，且物件與物件之間的溝通必須透過一些定義的介面，而無法直接地作溝通。

NOTE:

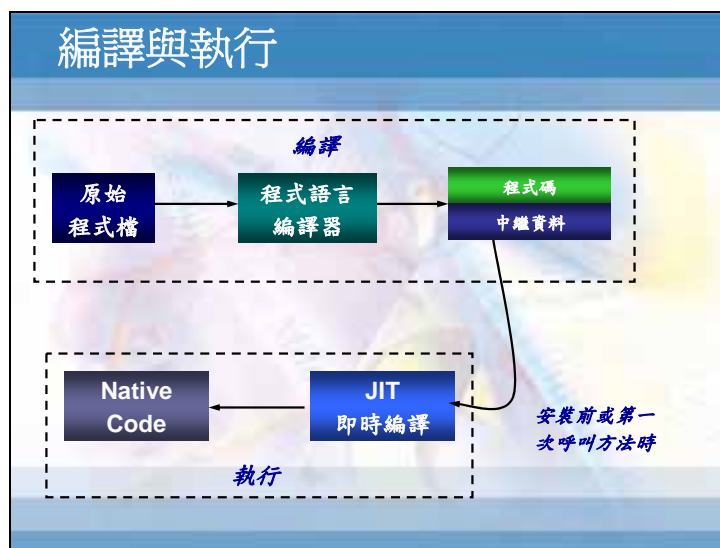
CLR 的執行環境



.NET 時代，透過 .NET Framework 共通語言執行環境 (CLR)，軟體元件是建構在共通的底層上，應用程式與應用程式之間的溝通不須再做聯結管道的雜工，且物件與物件之間可直接的溝通。

NOTE:

.NET 應用程式的編譯與執行



在 Common Language Runtime 上應用程式編譯與執行的方式作了很大的改變，剛剛前面有提到，它有點像 Java 的 Virtual Machine，.NET Framework 的設計觀念，第一階段，開發階段由程式語言的編譯器進行編譯，第二階段的編譯是在執行期，由 CLR 進行編譯。未來不管是用 VB，C#、或是 C++，都可以將程式碼編譯成 EXE 或是 DLL 檔案，但是裡面的程式碼了，卻不是 Native Code，不是 CPU 所能執行的程式碼，而是所謂的 Intermediate Language (IL) 這一階段的編譯是利用各程式語言編譯器將原始程式碼編譯成類似組合語言的中介語言格式 (MSIL)，檔案中包含了中繼語言 (IL) 的部份以及自我描述的中繼資料 (Metadata)。

MSIL 是不相依於 CPU 的一種指令集，也是 .NET Framework 程式的編譯方式。我想各位都有這樣的經驗，當我們高高興興地將編譯過的應用程式安裝到別台電腦上時，卻發現應用程式無法如預

NOTE:

期的執行。由於 MSIL 不會與 CPU 及系統產生相依性，以上因系統或硬體產生的不適性即可消除。

另一方面，由於結合了中繼資料與通用類別系統，因此 MSIL 允許真正的跨語言整合。在執行之前，MSIL 會轉換成機器碼。

第二階段則是在應用程式第一次執行的時候由 CLR 執行即時編譯(JIT)，將中繼語言程式碼編譯成 CPU 可執行的原生碼，儲存在 Cache 中，並交由 CLR 管理。

CLR 也提供開發者另一選擇，即在安裝前先執行第二階段的編譯工作，可節省執行編譯的時間。

NOTE:

CLR 的組成元素



為了提供應用程式執行所需的功能及管理程式碼，CLR 包含了以下幾個重要的成員：

- Class Loader 負責載入執行或引用的元件。
- IL to Native Compilers 負責執行第二階段的編譯工作。
- Code Manager 負責管理程式碼。
- Garbage Collector 負責記憶體管理的部份。
- Security Engine 負責安全檢查機制。
- Debug Engine 提供豐富跨語言的偵錯工具。
- Type Checker 負責執行型別檢查的工作，在 .NET 中不允許

NOTE:

未經宣告的型別。

- Exception Manager 提供結構化且一致的意外處理程序。
- Thread Support 提供多執行緒的支援。
- COM Marshaler 提供與 COM 溝通的管道。
- 最後 Base Class Library Support 則提供對 .NET Framework 基礎型別之支援。

NOTE:

CLR 所提供的服務



透過共通語言執行期環境可以

- 管理執行中的程式碼
 - 確認型別的安全
 - 提供資源回收與錯誤處理
 - 程式碼存取安全
- 提供共通型別系統
 - 值型別
 - 參考型別

NOTE:

- 提供系統資源的存取管道
 - Native API, COM interop, etc.

NOTE:

CLR 的特點

CLR 簡化應用程式開發

- 完全無須元件整合所需的 “ 雜工 ”
- 不再有…

➤Registration	=>自我描述
➤GUIDs	=>階層式的名稱空間
➤.IDL files	=>中繼資料
➤HRESULTs	=>結構化的意外處理
➤IUnknown	=>root object class
➤AddRef/Release	=>garbage collector
➤CoCreateInstance	=>"new" operator

透過 CLR 可以簡化整個應用程式的開發過程，完全無須元件整合所需的雜工。不再有：

Registration	=>自我描述
GUIDs	=>階層式的名稱空間
.IDL files	=>中繼資料
HRESULTs	=>結構化的意外處理
IUnknown	=>root object class
AddRef/Release	=>資源回收 garbage collector
CoCreateInstance	=>"new" operator

NOTE:

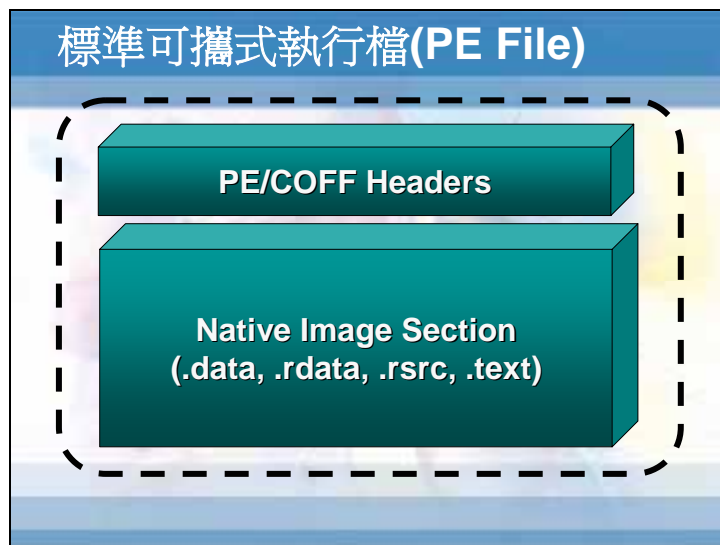
.NET CLR 環境



是不是一旦裝上 .NET Framework 之後，所有的應用程式都是以這種方式執行，並使用 CLR 所提供的資源呢？當然不是，只有 CLR 執行檔，才可以在 CLR 上執行。接下來我們來看看 .NET 的可執行檔與一般程式的可執行檔有何不同。

NOTE:

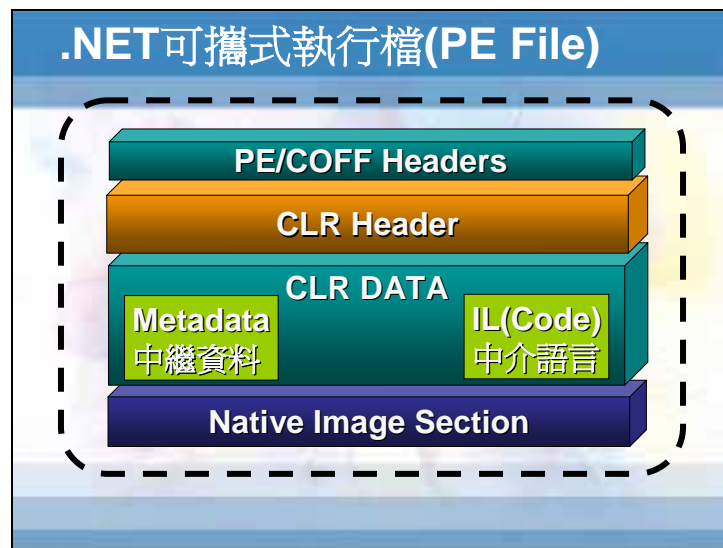
標準可攜式執行檔



在傳統的執行環境中，Windows 的可執行檔(EXE or DLL)必須符合 COFF (Common Object File Format) 所衍生的 PE 檔案格式。任何開發 Windows 程式的編譯器必須依循 PE/COFF 的規範，作業系統才知道如何載入與執行 DLLs 或 EXEs。標準的 PE 檔案分成兩個主要 Section，第一個 Section 為參照到檔案內容的 PE / COFF 檔頭，第二部份為一些原生影像資料。

NOTE:

.NET 可攜式執行檔



為了支援 CLR 的使用，Microsoft 在標準 PE 檔中加入一些新的 Section，其中 CLR Header 記載這個 PE 檔是一個 .NET 的可執行檔；CLR DATA 則包含中繼資料與中介語言程式碼。

NOTE:

實作範例



在這個練習中，我們利用一個小工具取得.NET 執行檔之內容。

1. 開啟 Notepad，輸入下列程式碼

```
Imports System

Public Module MainApp

    Sub Main()

        Console.WriteLine("Hello, World!")

    End Sub

End Module
```

2. 儲存為 Hello.vb

NOTE:

3. 開啟 Visual Studio.NET 命令提示列，並注意路徑。(建議直接建立一個捷徑到程式碼所在資料夾)

4. 編譯程式碼

vbc.exe hello.txt

5. 執行 *hello.exe*

6. 執行 *vcvars32.bat* 設定路徑

7. 執行 *dumpbin.exe hello.exe*，將可看到一部分 PE 檔案的資料。

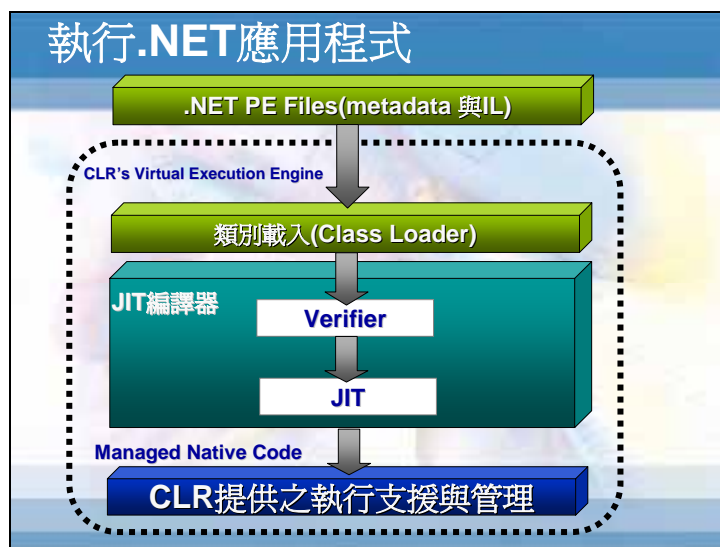
8. 重複上步驟，但加上選項將資料輸出到文字檔以利讀取。

Dumpbin.exe hello.exe /all >pe.txt

9. 開啟 *pe.txt*，尋找 *clr* 的位置。

NOTE:

執行.NET 應用程式



Microsoft 同時也更新了作業系統的 loader，更新過的 loader 會先檢查 PE File 中是否包含 CLR Header，如果存在，作業系統的 loader 將啟動 CLR 並交出 PE File。接下來就由先前所介紹的 Class Loader 負責載入的工作。

Class Loader 也會根據中繼資料所描述，載入應用程式執行所須引用的外部類別，當這些 IL 載入到記憶體後，CLR 會先作型別檢查的動作，再執行 JIT 的編譯。

編譯過的 Native Code 即可以在 CLR 上執行，並使用 CLR 所提供的服務與管理，包括 Security 的檢查，Garbage Collector 等。

NOTE:

.NET Framework 應用程式組件 Assembly



在了解了 CLR 的執行環境之後，接下來則開始探討 .NET Framework 應用程式的—架構—Assembly。

.NET Framework 以組件(Assembly)為佈署、版本控管、與安全控管的基本單位，利用內含的資訊清單(manifest)自我描述。組件的安裝不會相互影響，組件可以是 shared 或 private。數個版本的相同元件可共存甚至是在同一個執行程序下，可以終結 “ DLL Hell” 的問題。

對程式開發者而言，最常發生的版本編製問題，莫過於所謂的 DLL Hell 問題。DLL 難題是指多重應用程式嘗試共用元件，如動態連結程式庫 (DLL) 或 COM 類別時，所產生的一組問題。最典型的問題是，應用程式安裝共用元件的新版本，而該版本無法與電腦上已存在的版本回溯相容。雖然安裝的應用程式可以正常運作，

NOTE:

但必須依賴舊版共用元件的現有應用程式，則無法繼續運作。在某些情況下，造成問題的原因可能更難以想像。例如，假設使用者在造訪某些 Web 網站時，應網站要求而下載了 Microsoft ActiveX® 控制項。下載控制項之後，會取代電腦上現有的控制項版本。倘若電腦上已安裝的應用程式正好必須使用先前的控制項，也可能導致程式停止運作。

通常，使用者都是在一段時間之後，才發現應用程式停止運作。因此很難斷定是在進行哪一個變更之後，致使應用程式受到影響。使用者或許還記得上星期安裝的軟體，但安裝程序本身與目前看到的行為，並沒有明顯的關聯性。而市面上也很少有所謂的偵測工具，可幫助使用者判斷錯誤的根源。

這些問題的導因，是由於系統無法對應用程式的各種元件版本資訊作記錄或實行隔離。因此某應用程式對系統所做的變更，通常會影響電腦上所有應用程式，要以完全隔離的方式建立應用程式，可謂十分困難。

隔離應用程式之所以建構困難的原因之一，是由於目前執行階段環境通常僅允許單一版本的元件或應用程式。這項限制條件意謂著元件作者撰寫程式碼時，必須採用維持回溯相容性的方式，否則在安裝新元件時，就必須冒著破壞現有應用程式的危險。實際上，撰寫永遠回溯相容的程式碼極為困難，但不是不可能。在 .NET 中，「版本共存」觀念是版本編製的核心概念。所謂版本共存是同時在機器上，安裝和執行同一元件的多重版本。有了版本共存元件支援，作者不必受到維護嚴格回溯相容的牽制，因為不同的應用程式可自由使用不同版本的共用元件。

NOTE:

另一個困擾程式開發人員的問題則是部署與安裝的問題，現今安裝應用程式屬於多重步驟程序。一般而言，安裝應用程式涉及複製多個軟體元件到磁碟，以及製作一系列描述元件的登錄項目。

登錄檔和磁碟檔案之間的項目分隔，使得複製應用程式與解除安裝加倍困難。此外，在登錄檔內描述 COM 類別的各種必要項目之間，其關係十分鬆散。這些項目通常包括 coclasses、介面、typelibs 和 DCOM App IDs，更遑論對登錄文件副檔名或元件類別所做的項目。

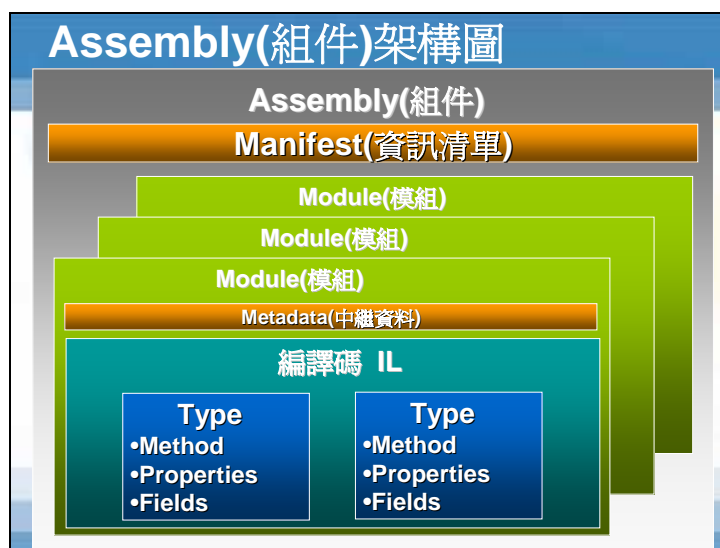
最後，必須依登錄檔的路徑，才能啟動 COM 類別。此舉徹底將部署分散式應用程式的程序複雜化，因為必須處理每一位用戶端電腦，才能做出適當的登錄項目。

另一個目前經常發生的問題，是無法在執行應用程式的同時進行更新。對於 Web 應用程式而言，這是最大的問題，因為 Web 服務必須先停止，然後重新啟動以更新應用程式所使用的 COM 類別。

這些問題主要是由隔離在元件本身以外的元件說明所引起。換句話說，應用程式即非自我說明 (self-describing)，亦非 self-contained。

NOTE:

組件架構



組件是 .NET Framework 用來解決版本編製與部署問題的基本建置單位。組件是型別與資源的部署單位。就許多方面而言，組件等於現在的 DLL。在本質上，組件就是「邏輯 DLL」。

組件透過名為「資訊清單 (Manifest)」的中繼資料自我說明。正如 .NET 使用中繼資料說明型別一般，資訊清單則使用中繼資料來說明包含型別的組件。

.NET 的版本編製是在組件層級內完成，而模組或類別則不予編製。此外，組件可用來共用應用程式之間的程式碼。

程式碼存取安全系統以組件核心。組件的作者在資訊清單中記錄了執行程式碼的整組必要權限，而管理員則根據包含程式碼的組件來許可程式碼權限。

NOTE:

最後，組件同時也是型別系統和執行階段系統的核心，其中可建立出型別的可見界線，並做為型別解析引用的執行階段範圍。

一般而言，組件是由四種元素組成：組件中繼資料（資訊清單）、說明型別的中繼資料、實作型別的中繼語言（IL）程式碼，以及資源組合。並非所有元素皆存在各個組件中，但是要為組件提供有意義的功能，則必須擁有任何一種型別或資源。

這四種元素有多種「包裝」方式。可以是包含整體組件的單一 DLL：資訊清單、型別中繼資料、IL 程式碼以及資源。也可以將某些公用程式碼分割為各個模組，在建立邏輯性的 DLL。這麼做的原因之一，是為了將程式碼下載最佳化。.NET Framework 只有在引用時才下載檔案，因此若經常使用包含程式碼或資源的組件，可以將其分為多個小檔案，以增進下載效率。

NOTE:

資訊清單



資訊清單包含下列組件相關資料：

識別 (identity)：

組件的識別是由三大部份組成：名稱、版本編號，以及文化（選擇性）。

檔案清單：

資訊清單包括所有組成組件的檔案清單。資訊清單在建立期間針對每個檔案，記錄其名稱和內容的加密雜湊。本雜湊於執行階段驗證，以確保部署單位的一致性。

外部引用組件：

NOTE:

組件之間的相依性，必須儲存在呼叫組件的資訊清單中。依存檔案資訊包含版本編號，可適用於執行階段，以確保載入正確的依存檔案版本。

匯出型別與資源：

型別與資料的可見選項，包括「僅於組件內可見」以及「組件外所有呼叫端皆可見」。

權限要求：

組件的權限要求分為下列三組：

- 1) 組件執行的必要權限；
- 2) 要求但即使沒有亦可擁有部份組件功能的權限；
- 3) 絕對禁止的權限。

NOTE:

實作練習



IL Disassembler (Ildasm) SDK 工具對於察看組件中的程式碼和中繼資料十分有用。

1. 開啟 Visual Studio.Net 命令提示字元
2. 輸入 ildasm.exe 開啟檢視視窗
3. 點選[檔案]→[開啟]開啟前面所編輯的 Hello.exe
4. 視窗中將列出組件的相關資訊，如圖 1
5. 展開圖 1 中的 MainApp，以滑鼠點選 Main:void() 兩下

NOTE:

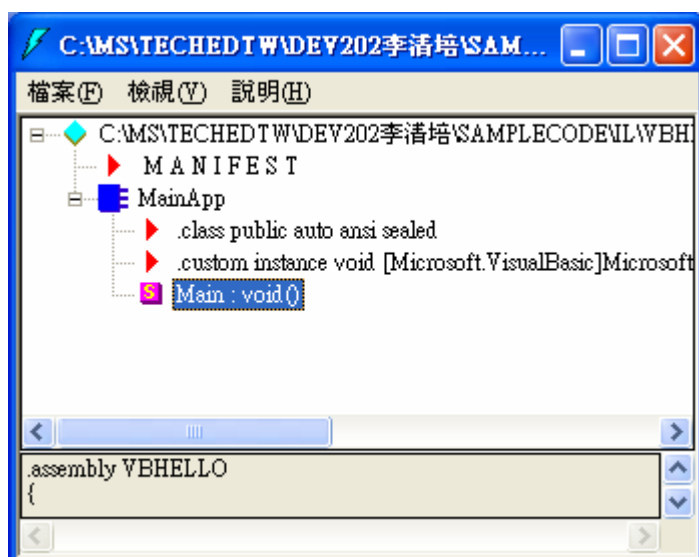


圖 1 IL Disassembler

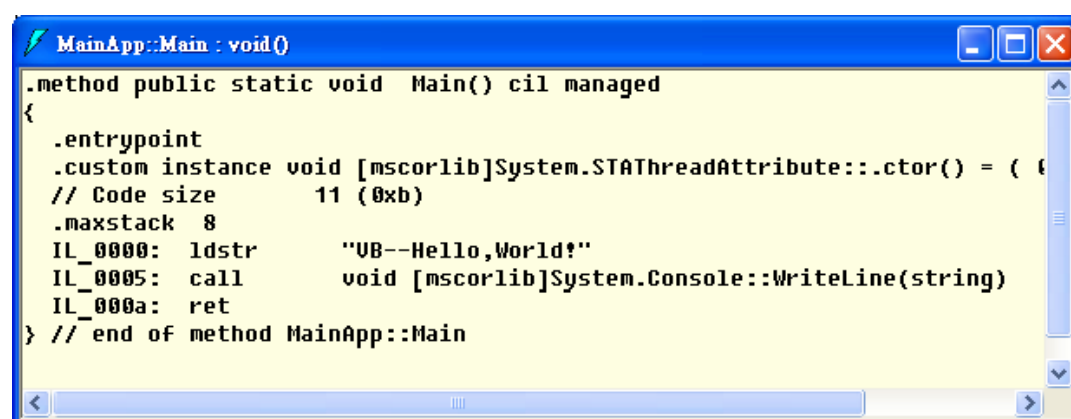
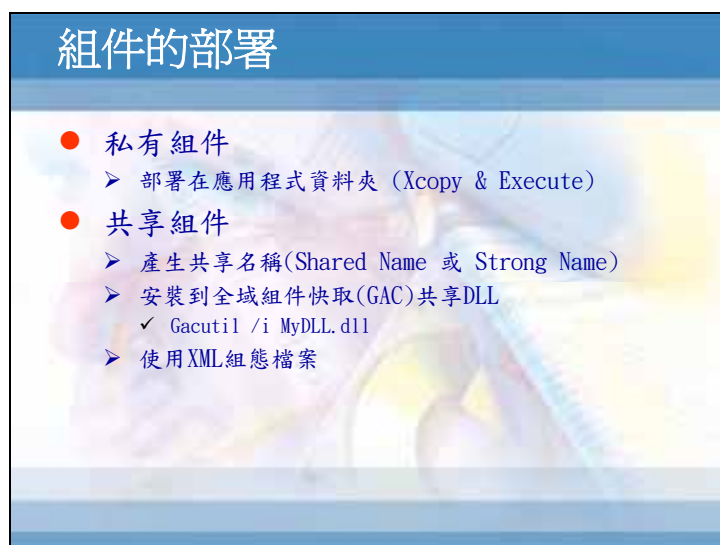


圖 2 由 IL Disassembler 顯示的資訊清單範例

NOTE:

組件的部署



.NET 組件的部署有兩種方式，第一種是私有的，即應用程式專屬組件，是僅限單一應用程式可見的組件，組件直接部署在應用程式的資料夾或其子資料夾中，以 XCOPY & Execute 的方式進行。我們期望 .NET 應用程式中儘量使用這種方式，因為 .NET 架構可促使建立不受其他應用程式變更系統所影響的應用程式。

專屬組件的命名要求很簡單：組件名稱必須是應用程式中唯一的名稱，並不需要是全域唯一的名稱。維持名稱的唯一性並不困難，因為應用程式開發人員可完全控制要將哪些組件隔離至應用程式。

應用程式專屬組件，部署於所在應用程式的目錄結構中。專屬組件可以直接置於應用程式目錄中，或位於其中的子目錄。通用語言執行階段透過探測試(probing)的程序，找出這些組件。Probing 只是利用組件名稱與檔案名稱之間的比對過程。

NOTE:

通用語言執行階段特別以組件引用中記錄的組件名稱，加上「.dll」並於應用程式目錄中尋找該檔案。執行階段會在組件命名的子目錄，或組件文化命名的子目錄中搜尋。例如，開發人員可選擇在包含德文本土化資源的「de」子目錄，以及西班牙文本土化的「es」子目錄中，部署組件。

然而，基於幾項原因，如表達版本原則與安全控管等能力，都可能會促使你選擇建立並使用共享組件。事實上，共享組件擁有加密的加強名稱，即意味著：唯有組件作者才有密鑰可為該組件生產新的版本。

NOTE:

共享組件之要求



共享組件則可被一個以上的應用程式所引用。若欲共享一個組件，便必須給它一個加密的加強名稱（strong name：也稱為共享名稱，shared name），以明確宣示該組件是為此目的而建立。相較之下，私用的組件名稱，只要符合使用它的特定應用程式即可。

對安裝於本機的應用程式而言，共享組件通常明顯地安裝於全域組件快取（global assembly cache：一種由 .NET Framework 負責維護的組件本機快取）。.NET Framework 的版本管理特質就是，安裝於本機的應用程式，在執行上不受下載程式碼的影響。下載程式碼會置於特定的下載快取中；而且，即使有些下載元件在設計上是共享組件，它們在該電腦上也不是隨手可以存取的。

NOTE:

.NET Framework 所推出的這些類別，在設計上皆為共享組件。對於這些共享組件的組態設定，例如版本原則等，都是使用 XML 格式的組態檔進行設定。

NET SDK 包括兩個處理全域組件快取的工作。第一個工具稱為 Gacutil，可將組件新增到 GAC。Gacutil 在程式開發與測試案例中使用方便，不需要建立整個 Windows Installer 封包，即可將組件增加到儲存體內。使用 /install 開關將組件增加到儲存體：

```
Gacutil /i MyDLL.dll
```

第二個工具是 Windows Shell Extension，可利用 Windows Explorer 操控儲存體。圖 3 顯示出全域組件快取。

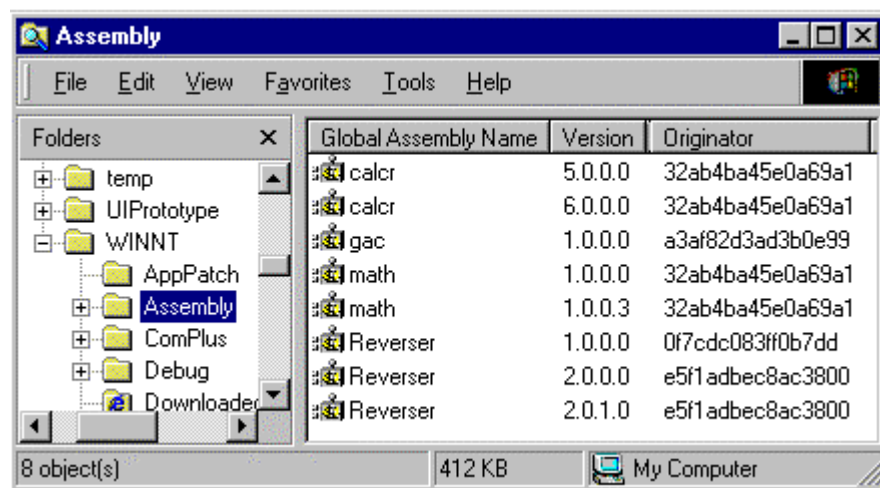


圖 3 全域組件快取

共享組件具有嚴格命名有要求，若欲共享一個組件，便必須給它一個加密的加強名稱（strong name：也稱為共享名稱，shared name），。共享名稱有三個目標：

NOTE:

名稱唯一性：共用組件必須擁有全域唯一的名稱。

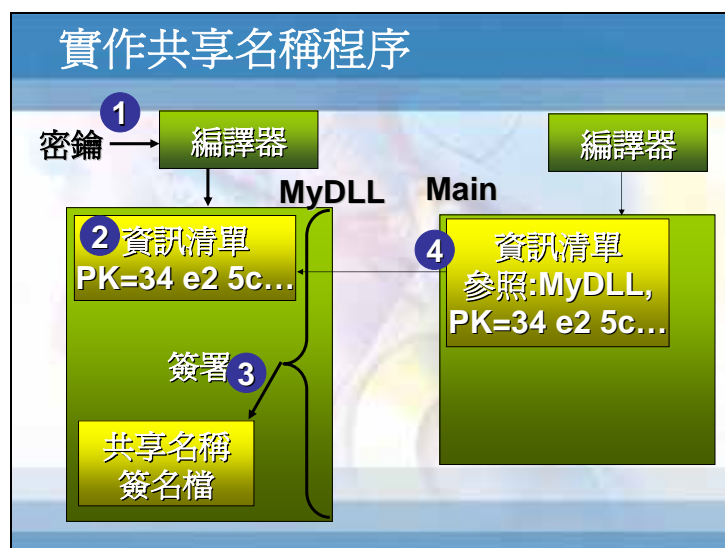
避免盜用名稱：開發人員不希望他人以您的名義發行您個人所有的組件後續版本，不論是意外或蓄意皆同。

提供引用的識別：在解析某組件的引用時，共用名稱可用來保證下載的組件來自預期的發行者。

共用名稱可利用標準公開金鑰加密實行。Public Key 可視為識別的一部份並包含在內，可賦予組件全域唯一的名稱，避免引用錯誤；Private Key 數位簽章則可避免名稱被盜用。

NOTE:

實作共享名稱



共用名稱可利用標準公開金鑰加密實行。一般而言，程序運作方式如下：組件的作者產生一組金鑰（或使用現有金鑰），為含有資訊清單的檔案簽署私密金鑰，並提供公開金鑰給呼叫端使用。為組件製作引用時，呼叫端記錄公開金鑰，以對應用來產生強調名稱的私密金鑰。

產生 Key Pair 的方法也很簡單，直接使用 .NET Framework 所附的工具程式 SN.exe 即可。

加強名稱的簽署，並不像 Authenticode 一樣需要認證。其中，沒有牽涉到任合協力廠，不需支付任何費用，也沒有任何認證鏈結。此外，驗證加強名稱的經常性工作，也比 Authenticode 所需的較少。但是加強名稱並沒有任何敘述，是有關於信任特定出版商的方式。加強名稱可以確定，任一組件的內容未經任何竄改；以及，你在執行時期所載入的組件，是來自原先設定的同一出版商。

NOTE:

但它完全沒有說明，你是否可以信任該出版商的識別身份。當然也可以對組件施以數位簽章，不同的是，Public Key 存在資訊清單中，而數位簽章存在於 PE File 的 section 中。

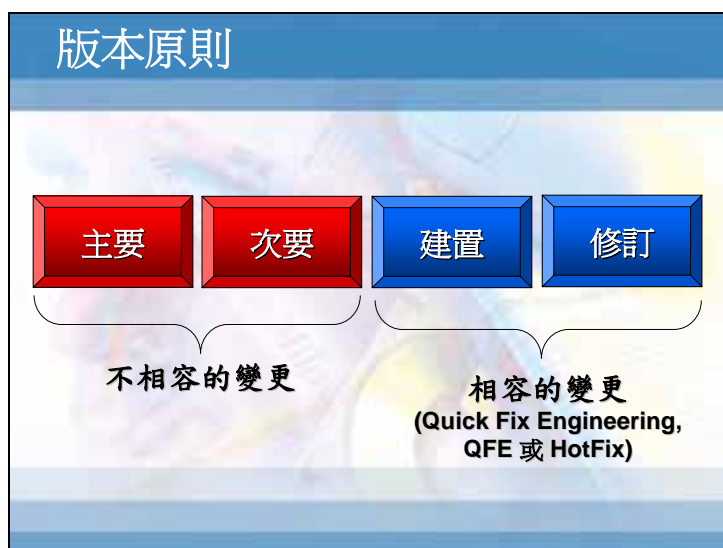
實作過程：

1. 編譯器接受簽署組件的金鑰並將之視為編譯步驟的一部份進行簽署。
2. 在編譯器發出組件時，公開金鑰視為組件識別的一部份來記錄。將公開金鑰視為識別的一部份並包含在內，可賦予組件全域唯一的名稱。
3. 在發出組件之後，包含資訊清單的檔案即以私密金鑰簽署。結果簽名儲存在檔案中。
4. 開發人員叫用以金鑰簽署的組件，由編譯器產生 Main 時，MyDLL 的公開金鑰將視為 MyDLL 引用的一部份，儲存在 Main 的資訊清單中。

在執行階段，.NET 架構需要兩個步驟，才能確保共用名稱可提供開發人員所需的優勢。首先，MyDLL 的共用名稱簽名可於組件安裝至全域組件儲存體時進行驗證（亦可選擇對尚未部署到儲存體內的簽名進行驗證）。驗證簽名可確保 MyDLL 的內容自建立組件後未遭到更改。第二個步驟是針對視為 Main 引用 MyDLL 的一部份而儲存的公開金鑰，是否符合 MyDLL 識別中部份公開金鑰，而進行驗證。若這些金鑰相同，則 Main 的作者可以確定 MyDLL 版本與建立 Main 的 MyDLL 版本，下載自同一個發行者。此金鑰的等值檢查是在執行階段，也就是 Main 引用到 MyDLL 解析完畢後完成。

NOTE:

版本原則



每個資訊清單組件皆記錄有關各個依存檔案建立時的版本資訊。不過在某些案例中，應用程式作者或管理員可能打算在執行階段執行不同的依存檔案版本。例如，管理員無需要求每個應用程式重新編譯，即可部署錯誤修訂程式。同時，若發現安全漏洞或其他嚴重的錯誤，管理員必須有能力指定特定永遠不使用某組件版本。 .NET Framework 可以透過版本原則，在版本繫結中使用這項彈性原則。

版本編號的四個部份分別是主要、次要、建置和修訂。針對主要或次要編號所做的變更，視為不相容性的變更。例如，開發人員已變更部份方法參數類型，或同時移除某些類型。類別載入程式運用本資訊，使得依存檔案組件的不相容版本永遠不預設載入。另一方面，僅針對版本編號中建置和修訂部份的變更，則視為相容。

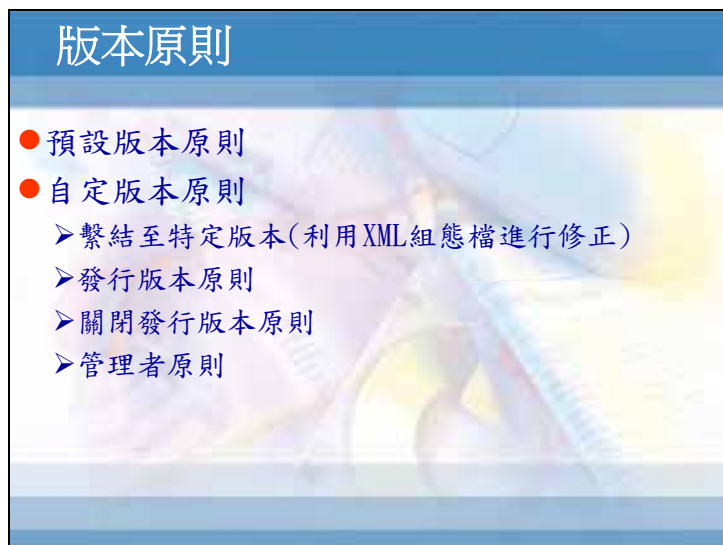
NOTE:

這些變更通常是錯誤修訂程式或安全修補程式，以不中斷呼叫端的方式進行，且不變更型別定義，以達到相容。這些相容變更通常是指 Quick Fix Engineering (QFE) 修訂程式或 hotfix。

開發人員和管理員必須了解版本編號的結構，因為這是通用語言執行階段在組件之間使用版本依存檔案的主要方式。

NOTE:

版本原則



通用語言執行階段可判斷在應用程式引用該組件時，要下載哪些版本的相依組件。在預設情況下，必須和引用的記錄有相同的編號。倘若編號不同，則組譯碼勢必不相容，且無法依預設載入。通常應用程式編譯時，會使用最高的版本編號。

不過在某些案例中，應用程式作者或管理員可能打算在執行階段執行不同的依存檔案版本。.NET 架構可以透過版本原則，在版本繫結中使用這項彈性原則。

由於.NET Framework 允許不同版本的組件並存，因此組件的開發者若希望所有引用該組件的所有應用程式皆使用新的版本，如 Service Pack 等，則依預設版本原則並無法達到這個目標。然而若引用自定版本原則的繫結至特定版本，則須一一的編修每一應用

NOTE:

程式的組態檔，這時候就必須使用發行版本原則。管理員無需要求每個應用程式重新編譯，即可部署錯誤修訂程式。

然而，我想各位都有這樣的經驗，當我們為應用程式安裝修正檔後，卻發現應用程式發生了重大錯誤，甚至無法執行，.NET Framework 提供一個快速簡單的作法，使應用程式恢復原有的樣子，及關閉發行版本原則。

最後.NET Framework 也提供給管理者一個最後的安全防線，若發現安全漏洞或其他嚴重的錯誤，管理員必須有能力指定特定永遠不使用某組件版本。

以上的版本原則皆可利用 XML 組態檔案進行變更。

NOTE:

版本原則組態檔

組態檔案範例

```
<configuration>
<runtime>
  <assemblyBinding xmlns="urn:schemas-microsoft-com:asm.v1">
    <publisherPolicy apply="Yes" />
    <dependentAssembly>
      <assemblyIdentity name="MyDLL"
        publicKeyToken="980d18da84cd69d2" />
      <bindingRedirect
        oldVersion="1.0.0.0" newVersion="1.0.0.1" />
    </dependentAssembly>
  </assemblyBinding>
</runtime>
</configuration>
```

有關版本編製，則有兩個相關檔案：應用程式專屬檔案以及全機器檔案，或者管理員檔案。應用程式專屬檔案位於應用程式的目錄中。本檔案所做的原則陳述僅影響本應用程式。全機器原則檔案目前位於 Windows 目錄中。本檔案由管理員使用，其原則陳述會影響機器中所有應用程式。例如，管理員判斷部份組件的某特定版本，造成安全漏洞，因此要確定不再使用本組件。

自訂原則範例包含：

繫結至特定版本

```
<bindingRedirect oldVersion="2.0.0.0" newVersion="2.0.0.1" />
```

NOTE:

發行者原則

```
<configuration>

  <runtime>

    <assemblyBinding xmlns="urn:schemas-microsoft-com:asm.v1">

      <assemblyIdentity name="policy.1.0.MyDLL" publicKeyToken="980d18da84cd69d2" />

      <dependentAssembly>

        <assemblyIdentity name="MyDLL" publicKeyToken="980d18da84cd69d2" />

        <bindingRedirect oldVersion="1.0.0.0" newVersion="1.0.2.1" />

      </dependentAssembly>

    </assemblyBinding>

  </runtime>

</configuration>
```

拒絕套用發行版本原則，還源自定原則

```
<publisherPolicy apply="no" />
```

NOTE:

實作練習



私有組件部署：

1. 開啟 Notepad，參考下列程式碼，建立 MyDll.vb 程式檔。

```
Imports System
```

```
Public Class SayHello
```

```
Public Function SayHello(ByVal strName as string) As String
```

```
SayHello= strName & " 您好！ "
```

```
End Function
```

```
End Class
```

2. 開啟 Visual Studio.Net 命令提示字元，將 MyDll.vb 編譯為 dll 組件。

NOTE:

Vbc.exe /t:library MyD11.vb

3. 開啟 Notepad，參考下列程式碼，建立 Client.vb 程式檔。

Imports System

Module MyClient

Sub Main()

Dim MyObj As New SayHello()

console.WriteLine(MyObj.SayHello("Your Name"))

End Sub

End Module

4. 開啟 Visual Studio.Net 命令提示字元，將 Client.vb 編譯為 exe 執行檔。

Vbc.exe /r:MyD11.dll Client.vb

5. 執行 Client.exe
6. 修改 MyD11.vb 的輸出字串，重複步驟 2 及步驟 5。(Client 端不需要重新編譯)

共享組件部署：

1. 分別開啟兩個資料夾，[Client]與[Share]
2. 開啟 Visual Studio.Net 命令提示字元，建立金鑰(建議放置於 C:\ 底下，以方便練習)

NOTE:

```
sn.exe -k MyKey.snk
```

3. 開啟 Notepad，在[Share]資料夾中建立 MyDll.vb 的程式碼。

```
Imports System
```

```
Imports System.Reflection
```

```
<assembly:AssemblyVersion("1.0.0.0")>
```

```
<assembly:AssemblyKeyFile("C:\MyKey.snk")>
```

```
public class MyDLL
```

```
    public sub SayHello()
```

```
        Console.WriteLine("Version:1.0.0.0-- Hello!")
```

```
    End Sub
```

```
End Class
```

4. 編譯 MyDll.vb

```
vbc.exe /t:library MyDll.vb
```

5. 將 MyDll.dll 發行到全域組件快取

```
gacutil /i MyDLL.dll
```

6. 開啟 Notepad，在[Cleint]資料夾中建立 Client.vb 的程式碼。

7. 編譯 Client.vb 為 Client1.exe（我們將有兩個 Client 作比較）

NOTE:

```
vbc.exe /r:..\share\MyD11.dll /out:Client1.exe Client.vb
```

8. 執行 Client1.exe

9. 修改 MyD11.vb，變更版本編號，並將輸出之版本編號字串一併修改，以便識別。

```
<assembly:AssemblyVersion("1.0.0.1")>
```

```
Console.WriteLine("Version:1.0.0.1-- Hello!")
```

10. 重複步驟 4, 5 編譯 MyD11.vb，並重新部署。

11. 將 Client.vb 編譯成 Client2.exe

```
vbc.exe /r:..\share\MyD11.dll /out:Client2.exe Client.vb
```

12. 執行 Client2.exe

13. 因為兩個用戶端執行檔編譯時間不同，依預設版本原則將繫節至編譯時的最新版本，同時不同版本元件並存。

<自定版本原則>

14. 開啟命令提示字元，在[Share]資料夾的路徑下，取得公開金鑰字元(publicKeyToken)

```
sn.exe -T MyD11.dll
```

15. 開啟 Notepad，依照下列文字完成組態設定，並儲存為 Client1.exe.config(注意大小寫)

NOTE:

```
<configuration>

  <runtime>

    <assemblyBinding xmlns="urn:schemas-microsoft-com:asm.v1">

      <publisherPolicy apply="yes" />

      <dependentAssembly>

        <assemblyIdentity name="MyDLL" publicKeyToken="參考步驟 14" />

        <bindingRedirect oldVersion="1.0.0.0" newVersion="1.0.0.1" />

      </dependentAssembly>

    </assemblyBinding>

  </runtime>

</configuration>
```

16. 再執行一次 Client1.exe，將看到與 Client2.exe 同樣的執行結果。

< 發行版本原則 >

17. 修改 MyD11.vb，變更版本編號，並將輸出之版本編號字串一併修改，以便識別。

```
<assembly:AssemblyVersion("1.0.2.1")>
```

```
Console.WriteLine("Version:1.0.2.1-- Hello!")
```

18. 重複步驟 4, 5 編譯 MyD11.vb，並重新部署。

NOTE:

19. 開啟 Notepad，依照下列文字完成組態設定，並儲存為 publish.xml(注意大小寫)

```
<configuration>

<runtime>

  <assemblyBinding xmlns="urn:schemas-microsoft-com:asm.v1">

    <assemblyIdentity name="policy.1.0.MyDLL" publicKeyToken="參考步驟 14" />

    <dependentAssembly>

      <assemblyIdentity name="MyDLL" publicKeyToken="參考步驟 14" />

      <bindingRedirect oldVersion="1.0.0.0-1.0.1.99" newVersion="1.0.2.1" />

    </dependentAssembly>

  </assemblyBinding>

</runtime>

</configuration>
```

20. 利用組件工具，將 publish.xml 連結為 dll 組件

```
A1 /link:publish.xml /out:policy.1.0.MyDLL.dll
/Keyfile:C:\MyKey.snk /Version:1.0.0.0
```

(因印刷關係，上述指令請勿中斷)

21. 將 policy.1.0.MyD11.dll 發行到 GAC

22. 執行 Client1.exe

23. 結果將出現 1.0.2.1 的版本說明

NOTE:

<拒絕套用>

24. 修改 Client.exe.config，將套用原則改為 no

```
<publisherPolicy apply="no" />
```

25. 在執行一次 Client1.exe

<管理者原則>

26. 管理者版本原則的組態檔在系統資料夾的
machine.config

27. 開啟 Microsoft .NET Framework Configuration 管理工具

28. 展開[我的電腦]，點選[組件快取]，點選右側窗格中的[在組件快取中檢視組件清單]

29. 找到 MyD11(應該有三個)，確認版本共存

30. 回到左側窗格，點選[已設定的組件]

31. 點選右側窗格的[設定組件]

32. 按下[選擇組件...]，在清單中任選一個 MyD11(同一個 Public Key)，按下完成。

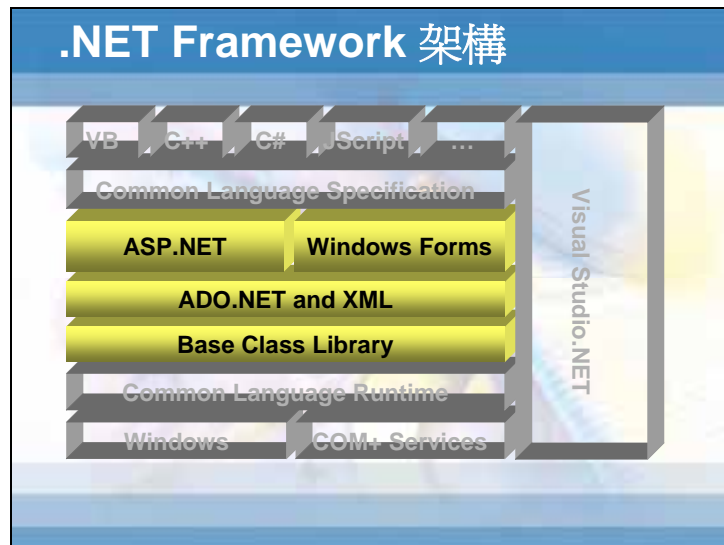
33. 點選繫結原則的頁籤

NOTE:

34. 在底下[要求的版本]欄位輸入 1.0.0.0-1.0.0.99，在[新增版本]的欄位輸入 1.0.2.1
35. 按下確定
36. 重新執行 Client1.exe

NOTE:

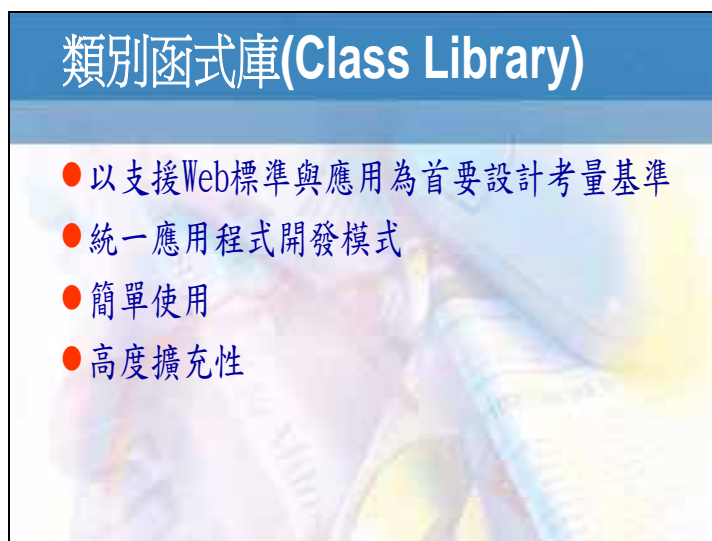
✚ .NET Framework 類別函式庫



以前在開發程式的時候，我們可能會用一些 API，COM Library，MFC/ATL，等等，.NET Framework 整合這些 Functionality 提供一致的 Class Library。

NOTE:

.NET Framework 類別



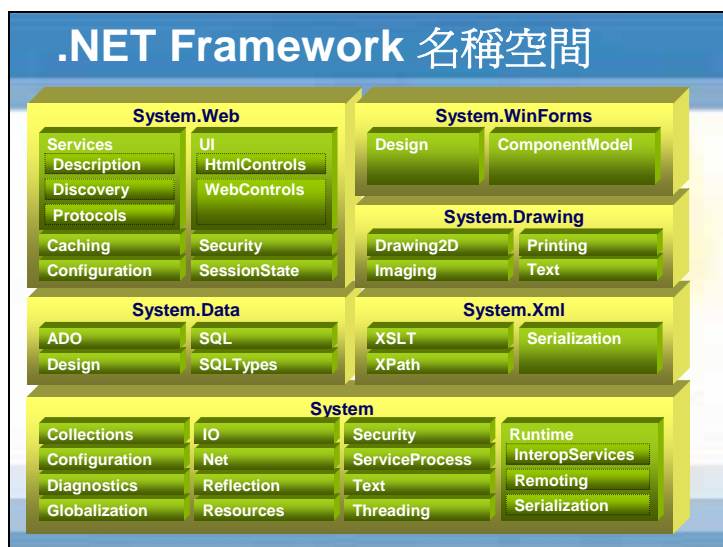
.NET Framework 類別函式庫，以支援 Web 標準與應用為首要設計考量基準，.NET Framework 支援 HTML, XML, SOAP, XSLT, XPath...等標準規範，並運用 loosely connected, stateless Web services 以支援網際網路的分散式應用程式開發。

.NET Framework 統一應用程式開發模式，未來無論您用什麼樣的程式語言，都可以使用一致的 Class Library。 .NET Framework 利用階層式的名稱空間 (namespaces)與一致的类型系統簡化程式的開發。

.NET Framework 提供隨插即用的組件與子系統，並支援跨程式語言的繼承 (inheritance!)，程式設計人員可透過繼承來擴充任何 .NET 類別的功能

NOTE:

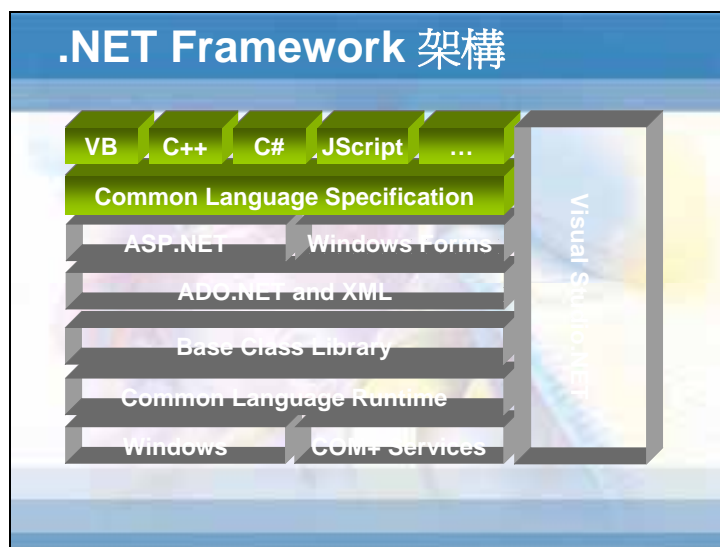
Namespace



要如何引用、延伸這些類別函式庫的功能，以建立自定的類別函式庫呢？面對那麼多的類別函式庫，又面臨了以前使用 API 的窘境：要用那個 API？怎麼用？好在 Microsoft 在這一方面提供了絕佳的解決方案，Microsoft 利用名稱空間(Namespace)的架構，將這些類別分門別類，例如要使用與資料庫相關的，就引用 System.Data 相關名稱空間；若要使用到 Web Service 相關的類別，則引用 System.Web.Services，依此類推。詳細的“族譜”請參閱 .NET Framework SDK 說明文件。

NOTE:

共通語言規範與.NET 語言



由於分散系統使用量日漸增多，互通性成為系統開發人員一個主要的議題。多年以來，互通性問題一直存在。語言互通性議題，所指的不只是像 COM 或 CORBA 等等標準化呼叫模型而已。在 COM 以前，應用程式之間是完全獨立的個體，應用程式的程式碼與資料結構是個自獨立而封閉的。到了 COM 的時代，COM 提供了元件整合的方法，但是各個原件必須提供聯結的管道，且物件與物件之間的溝通必須透過一些定義的介面，而無法直接地作溝通。到了 .NET 時代，透過 .NET Framework 共通語言執行環境(CLR)，軟體元件是建構在共通的底層上，應用程式與應用程式之間的溝通不須再做聯結管道的雜工，且物件與物件之間可直接的溝通。只要是符合共通語言規範 (Common Language Specification) 的程式語言，即可在共通語言執行環境(CLR)上執行，這樣的程式語言，也稱為 .NET 語言。

NOTE:

跨語言的程式開發

跨語言的程式開發

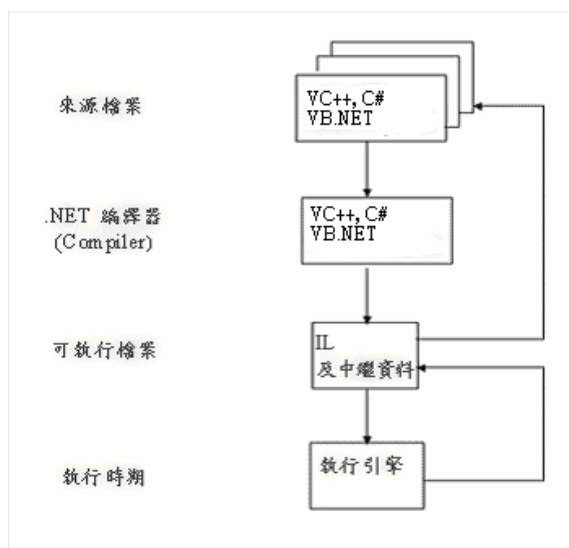
- 內建物件系統，以物件導向為核心
 - 可選擇任何符合共通語言規範的程式語言
- 進階的跨語言功能
 - 跨語言的繼承與偵錯
- 目前已有超過 25 種程式語言支援
 - VB, C++, C#, Java, JScript, Perl, COBOL
- 高效能的工具
 - Debuggers, profilers, code coverage analyzers, 皆可跨語言使用

在 Common Language Runtime 上應用程式編譯與執行方式作了很大的改變，剛剛前面有提到，它有點像 Java 的 Virtual Machine，.NET Framework 的設計觀念，第一階段，開發階段由程式語言的編譯器進行編譯，第二階段的編譯是在執行期，由 CLR 進行編譯。未來不管是用 VB, C#、或是 C++，都可以將程式碼編譯成 EXE 或是 DLL 檔案，但是裡面的程式碼了，卻不是 Native Code，不是 CPU 所能執行的程式碼，而是所謂的 Intermediate Language (IL) 這一階段的編譯是利用各程式語言編譯器將原始程式碼編譯成類似組合語言的中介語言格式 (MSIL)，檔案中包含了中繼語言 (IL) 的部份以及自我描述的中繼資料 (Metadata)。

在下面的示意圖顯示執行時期項目彼此之間的關聯性 (relationship)。在圖表上方的原始程式檔 (Source File)，可以使用像是 VC++ 等等多種語言，讓它包含新型別的定義。這個新型

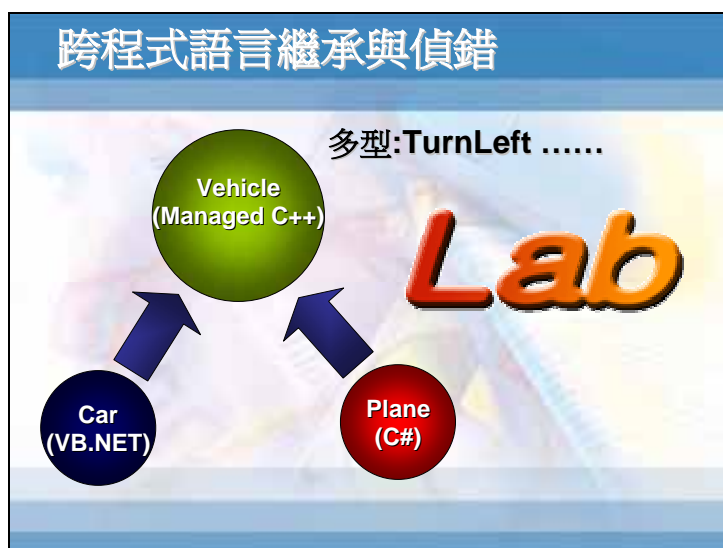
NOTE:

別，將會繼承自 .NET Framework 程式庫的一個型別，像是物件 (Object) 等等。當這個檔案經由 .NET Framework C++ 編譯器編譯之後，由此所產生的 Microsoft Intermediate Language (MSIL)，會保存於一個與新型別的中繼資料在一起的檔案裡。中繼資料所使用的格式，不必依賴定義此型別的程式語言。只要這個新型別的 MSIL 存在，其它原始程式檔（可能用別的語言寫成，像是 C#、或 Visual Basic.NET）就可以匯入此檔案。接下來，此 C++ 型別就可用於某個 Visual Basic.NET 原始程式碼，就好像它真的是 Visual Basic 型別一般。匯入型別的處理序，依語言的不同，可能會重複執行許多次。



NOTE:

範例實作



以下將以一個簡單的例子說明 .NET 語言的跨語言繼承。

建立 Vehicle 類別(C++)

1. 開啟 Microsoft Visual Studio.NET IDE 編輯環境。
2. 點選網頁中[新增專案]按鈕。開啟新增專案視窗。
3. 在左邊專案類型窗格中，點選[Visual C++ 專案]，在右邊範本的窗格中，點選[Managed C++類別庫]。保留其它預設值。
4. 點選方案總管中標頭檔 Vehicle.h
5. 在程式編輯視窗中輸入以下程式碼。

NOTE:


```
#pragma once
using namespace System;
namespace Vehicle
{
    public __gc __interface ISteering
    {
        void TurnLeft();
        void TurnRight();
    };

    public __gc class Vehicle : public ISteering
    {
    public:
        virtual void TurnLeft()
        {
            Console::WriteLine("載具向左轉");
        }

        virtual void TurnRight()
        {
            Console::WriteLine ("載具向右轉");
        }

        virtual void ApplyBrakes()=0;
    };
}
```

NOTE:

建立 Plane 類別(C#)

1. 點選[檔案] → [加入專案]→[新增專案] 按鈕。開啟新增專案視窗。
2. 在左邊專案類型窗格中，點選 Visual C# 專案，在右邊範本的窗格中，點選類別庫。保留其它預設值。
3. 將 Class1.cs 改為 plane.cs，並完成以下程式碼：

```
using System;
namespace Plane
{
    public class Plane : Vehicle.Vehicle
    {
        //以下為可改寫
        //          public override void TurnLeft()
        //          {
        //          Console.WriteLine ("飛機向左轉");
        //          }

        public override void TurnRight()
        {
            Console.WriteLine("飛機向右轉");
        }

        //以下為必須改寫
        public override void ApplyBrakes()
        {
```

NOTE:

```

        Console.WriteLine ("啓動空氣煞車");
    }
}

```

4. 在方案總管中展開[Plane]→[參考]，按滑鼠右鍵，選擇加入參考。選擇 Vehicle 專案。

NOTE:

建立 Car 類別(VB.NET)

1. 點選[檔案] → [加入專案]→[新增專案] 按鈕。開啟新增專案視窗。
2. 在左邊專案類型窗格中，點選 **Visual Basic 專案**，在右邊範本的窗格中，點選**類別庫**。保留其它預設值。
3. 完成以下程式碼，並參考至 Vehicle 專案。

```
Imports System
```

```
Public Class Car
```

```
    Inherits Vehicle.Vehicle
```

```
    '以下為可改寫
```

```
    Public Overrides Sub TurnLeft()
```

```
        Console.WriteLine("汽車向左轉")
```

```
End Sub
```

```
    'Public Overrides Sub TurnRight()
```

```
    '    Console.WriteLine("汽車向右轉")
```

```
    'End Sub
```

```
    '以下為必須改寫
```

```
    Public Overrides Sub ApplyBrakes()
```

```
        Console.WriteLine("汽車嘗試煞車")
```

```
        Throw New Exception("煞車失效")
```

```
    End Sub
```

```
End Class
```

NOTE:

建立測試程式(VB.NET)

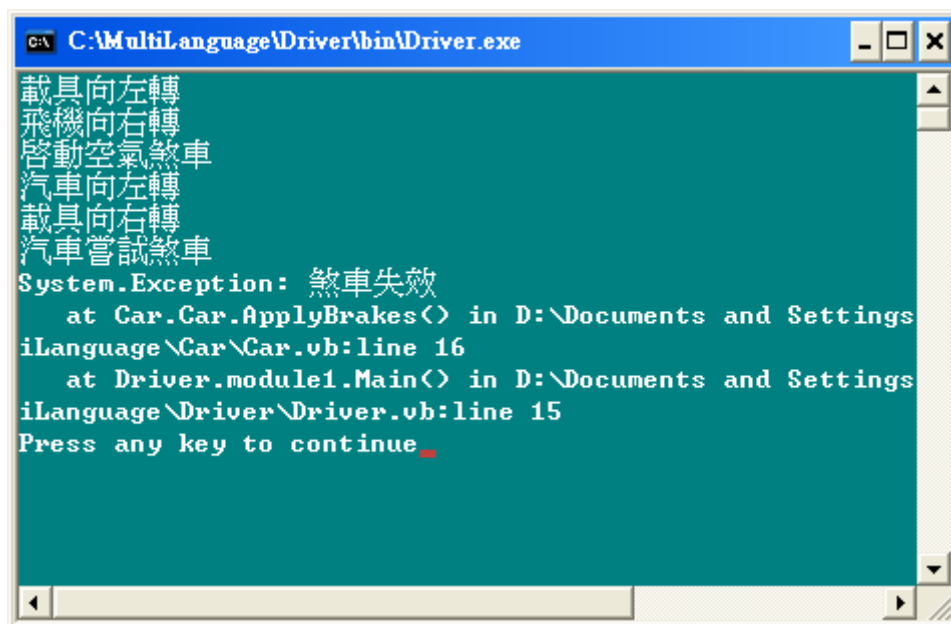
1. 點選[檔案] → [加入專案]→[新增專案] 按鈕。開啟新增專案視窗。
2. 在左邊專案類型窗格中，點選 **Visual Basic 專案**，在右邊範本的窗格中，點選**主控台應用程式**。保留其它預設值。
3. 完成以下程式碼，並參考至 Plane 及 Car 專案。

```
Public Module module1
    Sub Main()
        Try
            Dim v As Vehicle.Vehicle
            v = New Plane.Plane()
            v.TurnLeft()
            v.TurnRight()
            v.ApplyBrakes()
            v = New Car.Car()
            v.TurnLeft()
            v.TurnRight()
            v.ApplyBrakes()
        Catch e As Exception
            Console.WriteLine(e.ToString)
        End Try
    End Sub
End Module
```

NOTE:

直接在 IDE 內執行這個程式

1. 選擇 [偵錯] → [啟動但不偵錯]。



```
C:\MultiLanguage\Driver\bin\Driver.exe
載具向左轉
飛機向右轉
啟動空氣煞車
汽車向左轉
載具向右轉
汽車嘗試煞車
System.Exception: 煞車失效
   at Car.Car.ApplyBrakes() in D:\Documents and Settings
iLanguage\Car\Car.vb:line 16
   at Driver.module1.Main() in D:\Documents and Settings
iLanguage\Driver\Driver.vb:line 15
Press any key to continue_
```

跨語言繼承執行結果

2. 使用 Visual Studio 提供的工具進行偵錯，以逐行執行方式觀察跨語言偵錯環境。

NOTE:

