Chương 5

Marshaling và Remoting

Những ngày mà các chương trình được tích hợp chạy trên cùng một process duy nhất và trên một máy duy nhất đã qua rồi, quá lỗi thời. Ngày nay, các chương trình thường bao gồm những thành phần (được gọi là "cấu kiện", component) phức tạp chạy trên nhiều process (giống như những "đường đua") xuyên qua hệ thống mạng. Web làm cho việc chạy các ứng dụng mang tính phát tán (distributed application) ngày càng dễ dàng hơn theo vô số cách khó tưởng tượng nổi chỉ một vài năm về trước, và chiều hướng là ngày càng đi đến việc phân tán trách nhiệm.

Chiều hướng thứ hai là tập trung khía cạnh business lô gic lên các server đồ sộ. Ngoài mặt xem ra nghịch lý, nhưng trong thực tế các đối tượng mang tính business (sản xuất kinh doanh) thì theo hướng tập trung, trong khi giao diện người sử dụng thì lại phân tán.

Tác dụng rõ ràng nhất là các đối tượng có khả năng nói chuyện với nhau mà ở cách xa nhau. Các đối tượng chạy trên server lo thụ lý giao diện web của người sử dụng cần có khả năng tương tác với các đối tượng business nằm ở những server được tập trung tại trụ sở chính của công ty.

Tiến trình di chuyển một đối tượng xuyên qua một ranh giới được gọi là *remoting* (hành trình đi xa). Ranh giới hiện hữu ở nhiều cấp độ trừu tượng khác nhau trong chương trình của bạn. Ranh giới rõ ràng nhất là giữa các đối tượng chạy trên những máy tính khác nhau.

Tiến trình chuẩn bị một đối tượng được gởi đi xa được gọi là *marshaling*[1]. Trên một máy đơn độc các đối tượng có thể được marshal xuyên phạm trù, xuyên app domain hoặc xuyên ranh giới process.

Một *process* thực chất là một ứng dụng đang chạy. Nếu một đối tượng trên trình soạn thảo văn bản muốn tương tác với một đối tượng trên bảng tính, đối tượng này phải liên lạc xuyên ranh giới.

Các process được chia thành *application domain* (thường được gọi là "app domain"), và app domain lại được chia thành *context* (phạm trù). App domain hành động như là

[1] Marshaling là "cho vào nề nếp quân ngũ". Bạn có thể hình dung marshaling như là việc các nhà sản xuất đóng hàng vào container chuyển đi theo đường hàng hải, hoặc đóng gói gởi theo đường bưu điện.

những process nhẹ cân, còn context sẽ tạo ra những ranh giới theo đấy các đối tượng cùng chia sẻ các qui tắc tương tự có thể nằm trong lòng ranh giới. Có lúc, các đối tượng sẽ được marshal xuyên qua cả ranh giới context lẫn ranh giới app domain, cũng như xuyên qua ranh giới process và máy tính.

Khi một đối tượng nằm ở xa, nó có vẻ được gởi đi thông qua đường dây nối liền máy tính này qua máy tính kia. Nếu bạn ở phía kia đường dây bạn có thể nghĩ rằng bạn nhìn thấy đối tượng và đang nói chuyện với đối tượng. Nhưng thật ra bạn không nói chuyện với đối tượng mà là đang nói chuyện với một *proxy*, hoặc một cái mô phỏng. Nhiệm vụ của proxy là tiếp nhận thông điệp của bạn rồi chuyển cho đối tác. Ngoài ra, giữa bạn và đối tác còn có vô số "sink".

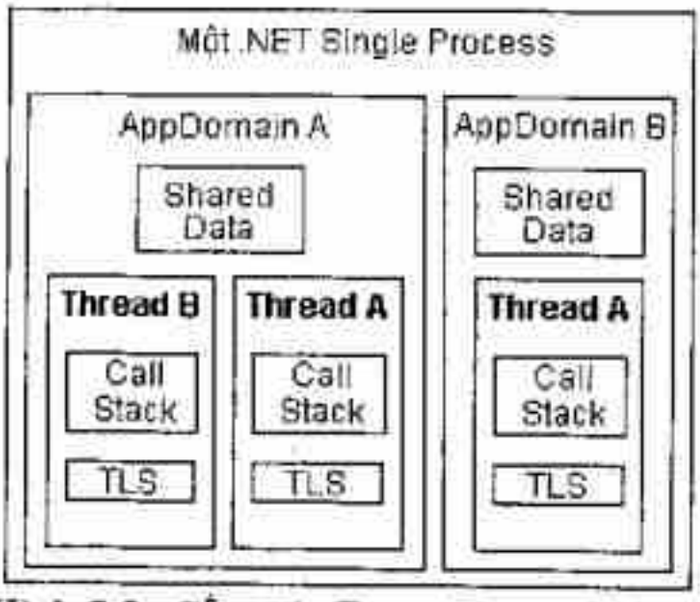
Sink là một đối tượng có nhiệm vụ tăng cường cơ chế liên lạc. Khi phía đối tác chuyển một cái gì không hợp lệ theo cơ chế, thì sink sẽ cho ngưng ngay việc liên lạc. Khi phía đối tác trả lời, thì nó chuyển câu trả lời cho những sink khác nhau cho đến khi tới tay proxy, và proxy sẽ nói chuyện với bạn.

Việc liên lạc chuyển thông điệp của bạn sẽ được thực hiện thông qua một *channel* (kênh liên lạc). Công việc của kênh là biết thế nào chuyển thông điệp từ nơi này qua nơi kia. Channel làm việc với một bộ định dạng (*formatter*). Formatter bảo đảm là thông điệp theo đúng dạng thức. Formatter giữ vai trò lặng lẽ của một thông dịch viên nói ngôn ngữ của bạn và ngôn ngữ của phía đối tác, cả hai có thể là không cùng một ngôn ngữ.

Chương này sẽ minh hoạ làm thế nào các đối tượng có thể được marshal xuyên qua các ranh giới khác nhau, và làm thế nào proxy và stub (là đối tác của proxy) có thể tạo một ảo tưởng là đối tượng của bạn chạy xuyên qua đường dây mạng đến tận máy tính của bạn hoặc đi vòng quanh thế giới. Ngoài ra, chương này sẽ giải thích vai trò của formatter, channel và sink, và cách dùng các khái niệm này trong lập trình.

5.1 Application Domains

Một process thực chất là một ứng dụng đang chạy. Mỗi ứng dụng .NET đều chạy trong process riêng của mình (giống như đường đua thể thao). Nếu bạn cho mở cùng lúc Word, Excel và Visual Studio .NET, thì bạn có 3 process đang chạy. Mỗi process lại phân nhỏ thành một hoặc nhiều application domain, mỗi application domain (gọi tắt là AppDomain) sẽ hoàn toàn được cách ly khỏi các AppDomain khác trong lòng process. Một app domain hành động như là một process nhưng sử dụng ít nguồn lực hơn. Các ứng dụng chạy trên AppDomain khác nhau sẽ không có khả năng chia sẻ bất cứ thông tin nào (biến toàn cục hoặc vùng mục tin static) trừ phi chúng tuân thủ .NET remoting protocol. Hình 5-01 cho thấy toàn cảnh Process, AppDomain và Thread.



Hình 5-1: Cấu trúc Process,AppDomain

App domain có thể được khởi động hoặc cho ngưng một cách hoàn toàn độc lập. Chúng được xem như là an toàn, "nhẹ cân" (nghĩa là không sử dụng nhiều nguồn lực) và linh hoạt cũng như mang tính "chấp nhận sai lầm" (fault tolerance). Nếu bạn khởi động một đối tượng trong một app domain thứ hai và nó "sụm bà chè" (crash), thì nó sẽ cho đi đong app domain nhưng toàn bộ chương trình thì không hề hấn gì. Bạn có thể tưởng tượng là Web Server phải sử dụng app domain để cho chạy đoạn mã của người sử dụng, và nếu đoạn mã có vấn đề, thì Web Server có thể duy trì công tác.

Một app domain sẽ được gói ghém bởi một thể hiện của lớp **AppDomain**. Lớp này gồm một số hàm hành sự và thuộc tính. Bảng 5-01 liệt kê một số quan trọng:

Bảng 5-01: Các thành viên cốt lõi của lớp System.AppDomain

Các thành viên	Mô tả
CurrentDomain	Thuộc tính public static này trả về app domain hiện hành đối với mạch trình hiện hành.
CreateDomain()	Hàm overloaded public static này tạo một app domain mới trong process hiện hành.
GetCurrentThreadID()	Hàm public static trả về mã nhận diện mạch trình hiện hành.
Unload()	Hàm public static lo gỡ bỏ AppDomain được khai báo.
FriendlyName	Thuộc tính public trả về tên thân thiện đối với AppDomain này.
DefineDynamicAssembly()	Hàm overloaded public cho phép định nghĩa một dynamic assembly trong AppDomain hiện hành.
ExecuteAssembly()	Hàm public lo thi hành assembly được chỉ định.
GetData()	Hàm public cho phép đi lấy trị được trữ trong app domain hiện hành.
Load()	Hàm public lo nạp một assembly vào app domain hiện hành.
SetAppDomainPolicy()	Hàm public lo đặt để cơ chế an toàn (security policy) đối với AppDomain hiện hành.
SetData()	Hàm public cho phép đưa dữ liệu vào thuộc tính app domain được chỉ định.

Ngoài ra, AppDomain cũng hỗ trợ vô số tình huống khác nhau, bao gồm **AssemblyLoad, AssemblyResolve, ProcessExit**, và **ResourceResolve**, được phát pháo khi assembly được tìm thấy, nạp, chạy và gỡ bỏ.

Mỗi process đều có một app domain khởi sự, và có thể có những app domain bổ sung khi bạn tạo ra chúng. Mỗi app domain hiện hữu đúng trong một process. Mãi tới nay, tất cả các chương trình trong tập sách này chỉ có một app domain đơn độc; app domain mặc nhiên. Mỗi process đều có app domain mặc nhiên riêng. Trong phần lớn các chương trình bạn viết ra, bạn chỉ cần app domain mặc nhiên là đủ.

Tuy nhiên, đôi lúc một domain đơn độc là không đủ. Có thể bạn muốn tạo một app domain thứ hai (sử dụng hàm **CreateDomain()** nếu bạn muốn cho chạy một thư viện viết bởi một lập trình viên khác). Có thể, bạn không tin tưởng cho lắm thư viện này và bạn muốn cách ly nó khỏi domain riêng của bạn để khi một hàm hành sự nào đó trên thư viện sụm thì chỉ domanin bị cách ly mới bị ảnh hưởng mà thôi.

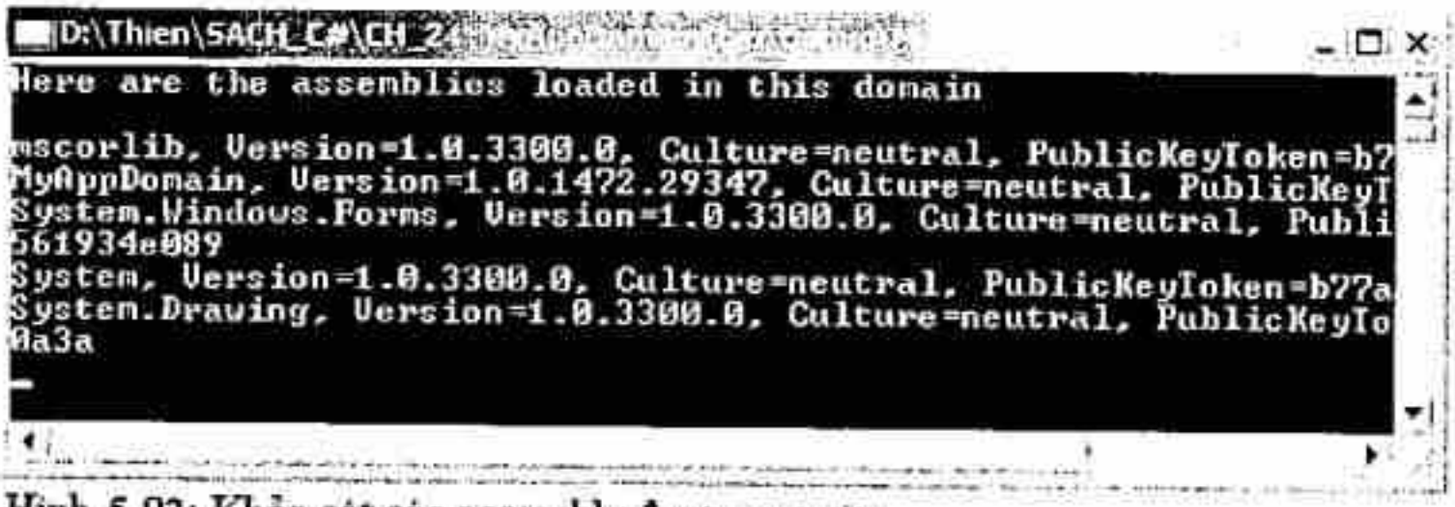
Cũng có thể là thư viện kia đòi hỏi một môi trường an ninh khác đi; tạo một app domain thứ hai cho phép hai môi trường an ninh chung sống hoà bình. Mỗi app domain có riêng cho mình một môi trường an ninh, và như vậy app domain được dùng như là ranh giới an ninh.

App domain không phải là mạch trình [2](thread) và phải phân biệt khỏi mạch trình. Một mạch trình chỉ hiện hữu trong một app domain vào một lúc nào đó, mà một mạch trình có thể truy cập (và báo cáo) vào app domain nó đang thi hành. App domain được dùng để cách ly các ứng dụng; trong lòng một app domain có thể có nhiều mạch trình cùng hoạt động vào một lúc nào đó. Xem lại hình 5-01.

5.1.1 Chơi một chút với AppDoamin

Để xem app domain hoạt động thế nào, ta thử xem thí dụ 5-01 sau đây, với kết xuất là hình 5-02:

[2] Có người dịch là "tiểu trình".



Hình 5-02: Khảo sát các assembly được nạp vào

Thí dụ 5-01: Khảo sát các assembly được nạp vào một AppDomain

```
using System;
using System.Windows.Forms;
using System.Reflection;   // cần namespace này để làm việc với
                           // kiểu dữ liệu Assembly
namespace MyAppDomain
{
    public class MyAppDomain
    {
        public static void PrintAllAssemblies()
        {
            // Yêu cầu AppDomain liệt kê tất cả các assembly
            // được nạp vào
            AppDomain ad = AppDomain.CurrentDomain;
            Assembly[] loadedAss = ad.GetAssemblies();
            Console.WriteLine("Here are the assemblies loaded in this
                               domain\n");

            // In ra tên trọn vẹn của mỗi assembly
            foreach(Assembly a in loadedAss)
            {
                Console.WriteLine(a.FullName);
            }
        }

        public static int Main(string[] args)
        {
            // Ép nạp assembly Windows Forms
            MessageBox.Show("Loaded System.Windows.Forms.dll");
            PrintAllAssemblies();
            Console.ReadLine(); // dùng ngăn cửa sổ console biến đi
            return 0;
        }
    }
}
```

Bạn để ý, trong chương trình này bạn sử dụng đến namespace **System.Reflection,** lo định nghĩa kiểu dữ liệu **Assembly** mà hàm **PrintAllAssemblies()** sẽ dùng đến. **Assembly** đã được đề cập đến ở chương 4, "Tìm hiểu về Attribute và Reflection", đi trước.

Hàm static **PrintAllAssemblies** nhận lấy qui chiếu về AppDomain hiện hành và liệt kê ra những assembly hiện đang "đóng quân" trên app domain. Bạn để ý hàm hành sự **Main()** cho hiển thị một message box yêu cầu assembly resolver nạp assembly System.Windows.Forms.dll (đến phiên DLL này nạp các assembly được qui chiếu khác). Hình 5-02 cho thấy kết xuất.

5.1.2 Một thí dụ sử dụng AppDomain

Muốn xem app domain hoạt động thế nào, ta thử dàn dựng một thí dụ. Giả sử ta muốn chương trình hiển lộ một lớp **Shape**, nhưng ở trên một app domain thứ hai.

Bạn để ý: Không có lý do gì chính đáng phải đưa lớp **Shape** vào một app domain thứ hai, ngoại trừ việc muốn minh hoạ các kỹ thuật này hoạt động thế nào. Tuy nhiên, đối với những đối tượng phức tạp hơn, có thể ta cần một app domain thứ hai cung cấp một môi trường an ninh khác. Ngoài ra, khi tạo ra những lớp mới, có thể bạn sẽ đưa vào những hành xử nhiều rủi ro, và để cho an toàn có thể bạn bắt đầu các lớp mới này trên một app domain thứ hai.

Thông thường, bạn nạp lớp **Shape** từ một assembly riêng biệt, nhưng để cho đơn giản thí dụ bạn chỉ cần đưa định nghĩa của lớp **Shape** vào cùng mã nguồn như với các đoạn mã khác. Ngoài ra, trên một môi trường sản xuất, có thể bạn cho chạy các hàm hành sự của lớp **Shape** trên một mạch trình riêng biệt. Nhưng để cho đơn giản, tạm thời bạn quên đi mạch trình. Threading sẽ được đề cập đến ở chương 6, "Mạch trình và Đồng bộ hoá", đi sau. Như vậy, thí dụ sẽ đơn giản hơn và ta chỉ tập trung vào những chi tiết tạo và sử dụng app domain cũng như marshal các đối tượng xuyên ranh giới phân chia các app domain.

5.1.2.1 Tạo và Sử dụng App Domain

Muốn tạo một app domain mới bạn cho triệu gọi hàm static **CreateDomain()** thuộc lớp **AppDomain**:

```
AppDomain ad2 = AppDomain.CreateDomain("Shape Domain", null, null);
```

Lệnh trên tạo mới một app domain ad2 mang tên thân thiện Shape Domain. Tên thân thiện chỉ dành cho lập trình không cần biết đến cách biểu diễn nội tại của app domain.

Bạn có thể kiểm tra tên thân thiện của app domain bạn đang làm việc dựa trên thuộc tính **System.AppDomain.CurrentDomain.FriendlyName.**

Một khi bạn đã hiển lộ một đối tượng **AppDomain**, bạn có thể tạo những thể hiện các lớp, giao diện, v.v.. sử dụng hàm hành sự **CreateInstance()**. Sau đây là dấu ấn của hàm hành sự:

```
public ObjectHandle CreateInstance(
    string assemblyName,
    string typeName,
    bool ignoreCase,
    BindingFlags bindingAttr,
    Binder binder,
    object[] args,
    CultureInfo culture,
    object[] activationAttributes,
    Evidence securityAttributes);
```

Và đây là cách dùng hàm hành sự này:

```
ObjectHandle oh = ad2.CreateInstance(
"ProgCSharp",              // tên assembly
"ProgCSharp.Shape",        // tên kiểu dữ liệu với namespace
false,                     // ignore case
System.Reflection.BindingFlags.CreateInstance,     // flag
null,                      // binder
new object[] {3, 5},       // args
null,                      // culture
null,                      // activation attribute
null );                    // security attribute
```

Thông số đầu tiên (**ProgCSharp**) là tên của assembly, còn thông số thứ hai (**ProgCSharp.Shape**) là tên lớp. Tên phải được "fully qualified" kèm theo namespace.

Một đối-tượng *binder* cho phép việc kết nối động (dynamic binding) của một assembly vào lúc chạy. Công việc của binder là cho phép bạn chuyển giao thông tin liên quan đến đối tượng bạn muốn tạo, tạo đối tượng này cho bạn và gắn kết qui chiếu của bạn vào đối tượng này. Trong phần lớn trường hợp, kể cả thí dụ này, bạn sẽ sử dụng binder mặc nhiên bằng cách trao qua null.

Lẽ dĩ nhiên là bạn có thể viết một binder riêng cho mình, chẳng hạn bằng cách kiểm tra mã nhận diện ID so với quyền hạn truy cập đối với một căn cứ dữ liệu rồi chuyển hướng việc gắn kết qua một đối tượng khác, dựa trên mã nhận diện hoặc quyền hạn truy cập của bạn.

Bạn để ý: Từ "binding", gắn kết, thường ám chỉ việc gắn liền một tên đối tượng vào một đối tượng. "Dynamic binding" ám chỉ khả năng việc gắn kết được thực hiện khi chúng ta

đang chạy, so với khi đang thiết kế. Trong thí dụ này, đối tượng **Shape** được gắn kết vào biến thể hiện (instance variable) vào lúc chạy, thông qua hàm **CreateInstance()**.

Binding flags giúp binder hành xử một cách tinh tế hơn vào lúc gắn kết. Trong thí dụ này, bạn sử dụng trị **CreateInstance** của enum **BindingFlags.** Thông thường binder mặc nhiên đi tìm những lớp public để gắn kết, nhưng bạn có thể thêm flag cho phép đi tìm những lớp private nếu quyền hạn truy cập cho phép.

Khi bạn gắn kết một assembly vào lúc chạy, bạn không khai báo assembly phải nạp vào lúc biên dịch mà xác định assembly nào bạn muốn theo lập trình và gắn kết biến của bạn vào assembly này khi chương trình chạy.

Hàm constructor mà chúng tôi triệu gọi nhận hai số nguyên, phải được đưa vào một bản dãy đối tượng (**new object[] {3,5}**). Bạn có thể trao null đối với thông tin culture vì chúng tôi chọn trị culture mặc nhiên (**en**) và cũng sẽ không khác thông tin liên quan đến activation attribute và security attribute

Đối tượng mà bạn nhận về sẽ là một *object handle* (mục quản đối tượng). Đây là một **type** được dùng để trao một đối tượng (trong tình trạng được bao bọc - wrapped state) giữa nhiều app domain không nạp metadata đối với các đối tượng được bao bọc trong mỗi đối tượng theo đấy **ObjectHandle** di chuyển. Bạn có thể đi lấy bản thân đối tượng hiện thời bằng cách triệu gọi hàm **UnWrap()** đối với mục quản đối tượng, và cho ép kiểu đối tượng kết xuất lên kiểu dữ liệu hiện thời - trong trường hợp này là **Shape**.

Hàm **CreateInstance()** cho phép tạo đối tượng trong một app domain mới. Nếu bạn muốn tạo mới đối tượng thông qua **new**, nó sẽ được tạo trong app domain hiện hành.

5.1.2.2 Marshalling xuyên ranh giới App Domain

Bạn đã tạo một đối tượng **Shape** trong Shape domain, nhưng bạn truy cập nó thông qua một đối tượng **Shape** trong domain nguyên thủy. Muốn truy xuất đối tượng shape trong một domain khác, bạn phải marshal đối tượng xuyên qua ranh giới của domain.

Marshalling là tiến trình chuẩn biến một đối tượng trước khi chuyển nó đi xuyên qua một ranh giới. Giống như khi bạn gởi một món đồ cho ai đó ở tận đâu đó qua bưu điện, bạn phải cho đóng gói theo đúng tiêu chuẩn và ghi đầy đủ chi tiết trên gói hàng. Hành động này được gọi là marshalling. Marshalling có thể được thực hiện theo trị (by value) hoặc theo qui chiếu (by reference). Khi một đối tượng được marshal theo trị, thì một bản sao sẽ được thực hiện và được chuyển đi. Với bản sao này trên tay, bạn có thể làm gì tùy thích (nhật tu, thêm bớt v.v..) nhưng bản gốc nguyên thủy sẽ không hề hấn gì.

Marshalling theo qui chiếu, nghĩa là bản gốc nằm tại chỗ, bạn không gởi đi bản sao mà lại một proxy. Khi bạn làm gì trên proxy, thì tác dụng sẽ ảnh hưởng lên bản gốc giống như là bạn thấy ngay bản gốc trước mặt mình.

Tìm hiểu marshalling với proxy

Khi bạn marshal theo qui chiếu, thì CLR sẽ cung cấp cho đối tượng phía triệu gọi một *transparent proxy* (TP). Công việc của TP là nhận về mọi thứ liên quan đến triệu gọi hàm của bạn (trị trả về, các thông số v.v..) khỏi stack và nhét nó vào một đối tượng có thi công giao diện **IMessage**. **IMessage** này sẽ được trao qua cho một đối tượng **RealProxy**.

RealProxy là một lớp abstract mà từ đấy các proxy sẽ được dẫn xuất. Bạn có thể thi công proxy thật sự riêng của mình, hoặc bất cứ các đối tượng khác trong process ngoại trừ TP. Proxy mặc nhiên thực thụ sẽ thụ lý **IMessage** đối với một loạt những đối tượng sink.

Bất cứ số lượng sink nào có thể được sử dụng tùy thuộc vào số cơ chế mà bạn muốn tăng cường, nhưng đúng sink chót trong chuỗi sink sẽ đưa **IMessage** vào một **Channel**. Kênh sẽ được chia thành kênh phía khách và kênh phía server và công việc của kênh là "đưa đò" xuyên ranh giới. Kênh chịu trách nhiệm hiểu thấu nghi thức giao thông vận chuyển. Dạng thức hiện thời đối với một thông điệp khi xuyên biên giới sẽ được quản lý bởi một *formatter* (bộ định dạng). .NET Framework cung cấp hai loại formatter: loại thứ nhất mang tên **SOAP** (Simple Object Access Protocol), được xem là mặc nhiên đối với kênh HTTP, và loại thứ hai mang tên **Binary Formatter** được xem là mặc nhiên đối với kênh TCP/IP. Không ai cấm bạn tạo ra cho mình một formatter đối với kênh riêng của bạn nếu bạn khoái làm việc này.

Một khi thông điệp đã băng qua ranh giới, nó sẽ được tiếp đón bởi kênh phía server, và một formatter, lo tạo lại **IMessage** và chuyển cho một hoặc nhiều sink phía server. Sink cuối cùng trên chuỗi sink này là **StackBuilder**, lo nhận **IMessage**, trả về lại cho stack frame theo đấy nó xuất hiện như là một triệu gọi hàm đối với server.

Khai báo phương pháp marshalling

Trong thí dụ kế tiếp, muốn minh hoạ sự phân biệt giữa marshal by value và marshal by reference , bạn cho biết đối tượng **Shape** sẽ được marshal theo qui chiếu, nhưng lại trao cho nó một biến thành viên kiểu dữ liệu **Point** mà bạn sẽ khai báo như là được marshal theo trị.

Bạn để ý mỗi lần bạn muốn tạo một đối tượng, mà đối tượng này lại có ý vượt biên thì bạn phải chọn cách nó sẽ được marshal thế nào. Thông thường các đối tượng không thể được marshal chi cả; bạn phải thân chinh cho biết đối tượng có biến marshal hay không, hoặc bằng trị hoặc bằng qui chiếu.

Cách dễ nhất khai báo cho biết một đối tượng được marshal theo trị là sử dụng attribute **[Serializable]**:

```
[Serializable]
public class Point
```

Khi một đối tượng được serialize, thì trạng thái nội tại của đối tượng sẽ được biến thành một stream dùng để marshal hoặc dùng cất trữ. Chi tiết về serialization đã được đề cập đến ở chương 1, "Xuất nhập dữ liệu & Sản sinh hàng loạt đối tượng".

Cách dễ nhất khai báo cho biết một đối tượng được marshal theo qui chiếu là cho dẫn xuất lớp của nó từ **MarshalByRefObject**:

```
public class Shape: MarshalByRefObject
```

Lớp **Shape** sẽ chỉ có một biến thành viên, **upperLeft**. Biến này mang kiểu dữ liệu **Point**, lo cầm giữ tọa độ của góc bên trái phía trên của đối tượng shape. Hàm contructor của **Shape** sẽ được khởi gán bởi thành viên **Point**:

```
public Shape(int upperLeftX, int upperLeftY)
{
   Console.WriteLine("[{0}] {1}",
                     System.AppDomain.CurrentDomain.FriendlyName,
                     "Shape constructor");
   upperLeft = new Point(upperLeftX, upperLeftY);
}
```

Lớp **Shape** có một hàm hành sự, **ShowUpperLeft**, sự lo hiển thị vị trí của **upperLeft**:

```
public void ShowUpperLeft()
{
   Console.WriteLine("[{0}] Upper left: {1}, {2}",
                     System.AppDomain.CurrentDomain.FriendlyName,
                     upperLeft.X, upperLeft.Y);
}
```

Ngoài ra, lớp **Shape** còn cung cấp một hàm hành sự trả về biến thành viên **upperLeft**.

```
public Point GetUpperLetf()
{
   return upperLeft;
}
```

Lớp **Point** rất đơn giản. Nó có một hàm constructor lo khởi gán hai biến thành viên toạ độ và các hàm accessor đi lấy trị của chúng. Một khi bạn đã tạo ra đối tượng **Shape**, bạn có thể yêu cầu xem toạ độ:

```
s1.ShowUpperLeft(); // yêu cầu hiển thị tọa độ
```

Sau đó yêu cầu đối tượng trả về toạ độ upperLeft như là đối tượng **Point** mà bạn sẽ thay đổi toạ độ:

```
Point localPoint = s1.GetUpperLeft();
localPoint.X = 500;
localPoint.Y = 600;
```

Yêu cầu **Point** in ra toạ độ, và sau đó yêu cầu **Shape** in ra toạ độ của nó. Như vậy, việc thay đổi trên đối tượng **Point** có được phản ánh lên **Shape** hay không? Điều này tùy thuộc vào việc **Point** được marshal thế nào. Nếu đối tượng **Point** được marshal theo trị, thì đối tượng **localPoint** sẽ là một bản sao, và đối tượng **Shape** sẽ không bị ảnh hưởng bởi việc thay đổi trị của các biến của **localPoint**. Nếu ngược lại, bạn thay đổi đối tượng được marshal theo reference, thì bạn sẽ có một proxy đối với biến **upperLeft** hiện thời, và việc thay đổi này sẽ ảnh hưởng lên **Shape**. Thí dụ 5-02 minh hoạ các giải thích trên, và hình 5-03 là kết xuất:

Hình 5-03: Kết xuất thí dụ 5-02.

Thí dụ 5-02: Marshalling xuyên biên giới app domain

```
using System;
using System.Runtime.Remoting;
using System.Reflection;

namespace MarshalAppDomain
```

```
{
    // đối với marshal by reference bạn cho comment out attribute
    // [Serializable] và uncomment base class
    [Serializable]
    public class Point  //: MarshalByRefObject
    {
        public Point(int x, int y)
        {   Console.WriteLine("[{0}] {1}",
                System.AppDomain.CurrentDomain.FriendlyName,
                "Point constructor");
            this.x = x;
            this.y = y;
        }
        public int X
        {
            get
            {   Console.WriteLine("[{0}] {1}",
                    System.AppDomain.CurrentDomain.FriendlyName,
                    "Point x.get");
                return this.x;
            }
            set
            {   Console.WriteLine("[{0}] {1}",
                    System.AppDomain.CurrentDomain.FriendlyName,
                    "Point x.set");
                this.x = value;
    }
        }

        public int Y
        {
            get
            {   Console.WriteLine("[{0}] {1}",
                    System.AppDomain.CurrentDomain.FriendlyName,
                    "Point y.get");
                return this.y;
            }
            set
            {   Console.WriteLine("[{0}] {1}",
                    System.AppDomain.CurrentDomain.FriendlyName,
                    "Point y.set");
                this.y = value;
            }
        }
        private int x;
        private int y;
    }

    // Lớp Shape được marshal theo qui chiếu
    public class Shape: MarshalByRefObject
    {
        public Shape(int upperLeftX, int upperLeftY)
        {   Console.WriteLine("[{0}] {1}",
```

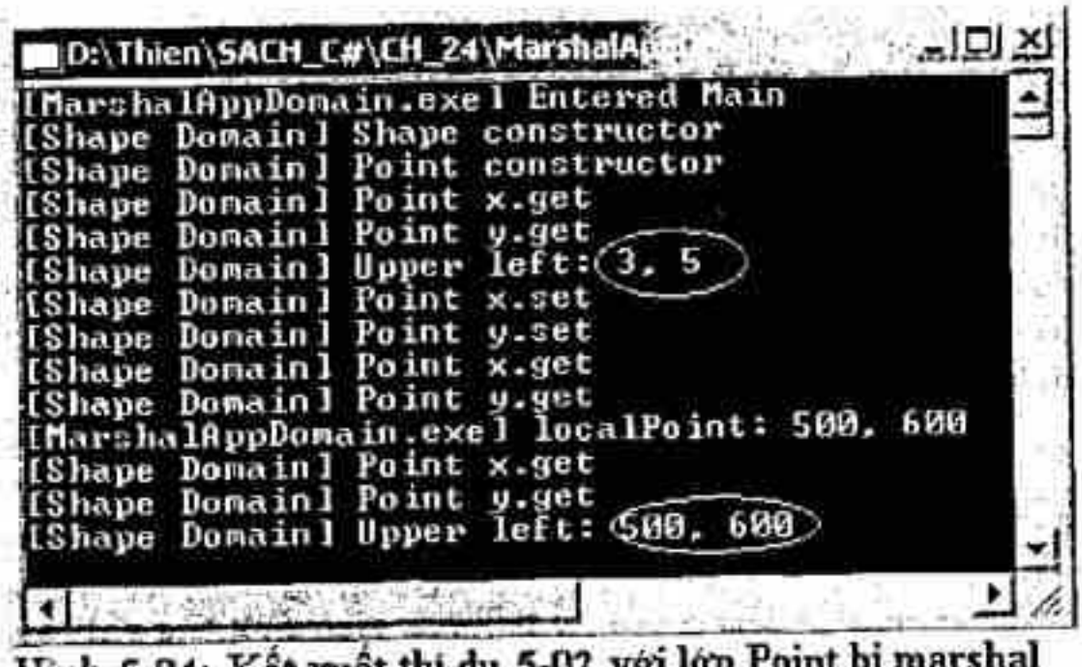
```
            System.AppDomain.CurrentDomain.FriendlyName,
            "Shape constructor");
        upperLeft = new Point(upperLeftX, upperLeftY);
    }

    public void ShowUpperLeft()
    {   Console.WriteLine("[{0}] Upper left: {1}, {2}",
            System.AppDomain.CurrentDomain.FriendlyName,
            upperLeft.X, upperLeft.Y);
    }

    public Point GetUpperLetf()
    {   return upperLeft;
    }
    private Point upperLeft;
}

public class Testter
{
    public static void Main()
    {   Console.WriteLine("[{0}] {1}",
            System.AppDomain.CurrentDomain.FriendlyName,
            "Entered Main");

        // tạo một app domain mới
        AppDomain ad2 = System.AppDomain.CreateDomain(
                                                "Shape Domain");

        // Assembly a = Assembly.LoadFrom("MarshalAppDomain.exe");
        // Object theShape = a.CreateInstance("Shape");
        // hiển lộ một đối tượng Shape
        ObjectHandle oh = ad2. CreateInstance(
            "MarshalAppDomain",          // tên assembly
            "MarshalAppDomain.Shape", // tên kiểu với namespace
            false,                       // ignore case
            System.Reflection.BindingFlags.CreateInstance, // flag
            null,                        // binder
            new object[] {3, 5},         // args
            null,                        // culture
            null,                        // activation attribute
            null );                      // security attribute

        Shape s1 = (Shape) oh.Unwrap();

        s1.ShowUpperLeft();     // yêu cầu đối tượng hiển thị

        // lấy một bản sao local? proxy? và gán trị mới
        Point localPoint = s1.GetUpperLetf();
        localPoint.X = 500;
        localPoint.Y = 600;

        // cho hiển thị trị của đối tượng Point local
        Console.WriteLine("[{0}] localPoint: {1}, {2}",
```

```
                System.AppDomain.CurrentDomain.FriendlyName,
                localPoint.X, localPoint.Y);
                s1.ShowUpperLeft(); // cho hiển thị trị một lần nữa
                Console.ReadLine();
            }
        }
    }
```

Hình 5-03 (trang 11) cho thấy kết xuất của chương trình thí dụ 5-02 kể trên. Cũng thí dụ trên, bây giờ bạn comment out attribute [Serializable] và uncomment base class như sau:

```
// [Serializable]
public class Point: MarshalByRefObject
```

rồi cho chạy lại chương trình, và hình 5-04 cho thấy kết xuất:



Hình 5-04: Kết xuất thí dụ 5-02 với lớp Point bị marshal theo reference.

Lần này bạn nhận một proxy đối với đối tượng Point và các thuộc tính được đặt để thông qua proxy trên biến thành viên nguyên thủy của Point. Do đó những thay đổi được phản ảnh liền lên bản thân Shape. Bạn so sánh hai hình 5-03 và 5-04.

5.2 Phạm trù (context)

App domain lại được chia nhỏ thành *context* (phạm trù). Context được xem như là ranh giới trong lòng những đối tượng cùng chia sẻ sử dụng các qui tắc (rule). Việc sử dụng các qui tắc này bao gồm việc đồng bộ hoá các phiên giao dịch. (Xem chương 6, "Mạch trình và Đồng bộ hoá").

5.2.1 Các đối tượng contex-bound và context-agile

Các đối tượng có thể hoặc là *context-bound* hoặc *context-agile*. Nếu các đối tượng thuộc loại context-bound (được gắn kết với phạm trù) thì chúng hiện hữu ngay trong phạm trù, và khi muốn tương tác với chúng thông điệp phải được marshal. Còn khi chúng thuộc loại context-agile (nhanh nhạy theo phạm trù) thì chúng hành động ngay trong lòng phạm trù của đối tượng phía triệu gọi; nghĩa là các hàm hành sự của chúng sẽ thi hành ngay trong phạm trù của đối tượng phía triệu gọi và không nhất thiết phải được marshal.

Giả sử bạn có một đối tượng A tương tác với căn cứ dữ liệu và như thế đối tượng A được đánh dấu là chịu hỗ trợ các phiên giao dịch. Điều này tạo ra một phạm trù. Các hàm hành sự triệu gọi A sẽ xảy ra trong lòng phạm trù bảo vệ mà các phiên giao dịch tự trang bị. Đối tượng A có thể quyết định cho trở lộn lui lại (rollback) phiên giao dịch, và những hành động đã được thực hiện từ lần trình duyệt[3] (commit) chót sẽ bị hoá giải (undone).

Giả sử, bạn có một đối tượng B khác thuộc loại context-agile. Bây giờ ta giả định là đối tượng A chuyển một qui chiếu căn cứ dữ liệu cho đối tượng B, rồi sau đó triệu gọi các hàm trên B. Có thể A và B nằm trong một mối liên hệ call-back, theo đấy B sẽ làm vài việc chi đó và sau đó gọi lại A để trao lại kết quả xử lý. Vì B là context-agile, hàm hành sự của B được thi hành trong phạm trù của phía đối tượng triệu gọi là A; như vậy nó sẽ hưởng sự bảo vệ giao dịch của đối tượng A. Những thay đổi mà B thực hiện đối với căn cứ dữ liệu sẽ bị hoá giải nếu đối tượng A rollback giao dịch, vì các hàm hành sự của B được thi hành trong phạm trù của phía triệu gọi.

Như vậy B phải là context-agile hay là context-bound? Trong trường hợp ta vừa quan sát, B hoạt động tốt khi đang là context-agile. Bây giờ ta giả định một lớp hiện hữu nữa: lớp C. Lớp C không có các phiên giao dịch, và nó sẽ triệu gọi hàm hành sự của B lo thay đổi gì đó trên căn cứ dữ liệu. Bây giờ A cố rollback, nhưng rất tiếc phần việc mà B làm cho C lại nằm trong phạm trù C và như vậy không nhận sự hỗ trợ giao dịch của A, và như vậy công việc mà B đã làm giùm cho C không thể bị hoá giải.

Nếu B được đánh dấu context-bound, và khi A tạo ra B, B sẽ kế thừa phạm trù của A. Trong trường hợp này, khi C cho triệu gọi một hàm hành sự trên B nó phải được marshal xuyên ranh giới phạm trù, nhưng khi B cho thi hành hàm hành sự nó phải ở trong phạm trù của phiên giao diện của A. Như vậy là tốt hơn.

Điều này sẽ chạy nếu B là context-bound nhưng không có attribute. Lẽ dĩ nhiên là B có thể có riêng context attribute riêng cho mình, và những attribute này có thể ép B vào trong một phạm trù khác với A. Thí dụ, B có thể có một transaction attribute được đánh

[3] Ở đây, cụm từ "trình duyệt" đừng nhầm với browser (mà người ta dịch là trình duyệt) có nghĩa là "trình lên để duyệt xét" (commit).

dấu là **RequireNew**. Trong trường hợp này, khi B được tạo nó sẽ nhận một phạm trù mới, như vậy sẽ không ở trong phạm trù của A. Như vậy, khi A cho rollback, công việc B đã thực hiện sẽ không được rollback. Có thể bạn cho đánh dấu **RequireNew** đối với B vì B là một hàm kiểm toán (audit function). Khi A thực hiện một hành động trên căn cứ dữ liệu nó thông báo B lo nhật tu một audit trail (theo dõi kiểm toán). Bạn không muốn công việc của B bị hoá giải khi A rollback phiên giao dịch. Bạn muốn B nằm trong phạm trù giao dịch riêng của một và chỉ rollback khi B phạm sai lầm, chứ không phải của A.

Như vậy, một đối tượng sẽ có 3 lựa chọn: lựa chọn thứ nhất là context-agile, hoạt động trong phạm trù của đối tượng phía triệu gọi. Lựa chọn thứ hai là context-bound (được thực hiện bằng cách dẫn xuất từ **ContextBoundObject**) nhưng lại không có attribute và như thế sẽ hoạt động trong phạm trù của phía tạo dựng. Lựa chọn thứ ba là context-bound với context attribute, và như thế chỉ sẽ hoạt động trong phạm trù nào khớp với context attribute.

Việc bạn quyết định lựa chọn nào tùy thuộc việc đối tượng của bạn sẽ được sử dụng thế nào. Nếu đối tượng của bạn là một đối tượng đơn giản, như một máy tính bỏ túi (calculator) chẳng hạn, không cần đến đồng bộ hoá hoặc phiên giao dịch hoặc bất cứ hỗ trợ phạm trù nào, thì nên chọn context-agile là hiệu quả nhất. Nếu đối tượng của bạn phải sử dụng phạm trù của đối tượng tạo ra nó, thì đối tượng này phải là context-bound không mang attribute. Cuối cùng, nếu đối tượng của bạn có những yêu cầu phạm trù riêng, bạn phải cho đối tượng này là context-bound với attribute thích ứng.

5.2.2 Cho marshal xuyên biên giới phạm trù

Ta sẽ không cần đến proxy khi muốn truy xuất các đối tượng context-agile trong lòng một app domain đơn độc. Khi một đối tượng nằm trong một phạm trù muốn truy xuất một đối tượng context-bound nằm trong phạm trù thứ hai, nó phải truy xuất thông qua một proxy, và vào lúc đó, hai cơ chế phạm trù phải được tuân thủ. Nghĩa là theo chiều hướng này một phạm trù sẽ tạo ra ranh giới; cơ chế sẽ được tuân thủ ở ngay ranh giới phân chia hai phạm trù.

Thí dụ, khi bạn đánh dấu một đối tượng là context-bound thông qua attribute **System.Runtime.Remoting.Synchronization**, bạn cho biết là bạn muốn hệ thống quản lý việc đồng bộ hoá đối với đối tượng này. Tất cả các đối tượng nằm ngoài phạm trù này phải vượt qua phạm trù ranh giới để đến với một trong những đối tượng này, và vào lúc đó cơ chế đồng bộ hoá sẽ được áp dụng.

Các đối tượng sẽ được marshal khác nhau xuyên ranh giới phạm trù tùy vào việc các đối tượng này được tạo ra thế nào:

- Các đối tượng điển hình thường không được marshal chi cả; trong lòng app domain các đối tượng này thường thuộc context-agile.

- Các đối tượng được đánh dấu bởi attribute **Serializable** thường được marshal theo trị xuyên app domain và thuộc loại context-agile.

- Các đối tượng được dẫn xuất từ **MarshelByRefObject** sẽ được marshal theo qui chiếu xuyên app domain và thuộc loại context-agile.

- Các đối tượng được dẫn xuất từ **ContextBoundObject** sẽ được marshal theo qui chiếu xuyên app domain cũng như theo qui chiếu xuyên ranh giới phạm trù.

5.3 Remoting[4]

Ngoài việc được marshal xuyên ranh giới app domain và ranh giới phạm trù, các đối tượng có thể được marshal xuyên ranh giới các process, kể cả xuyên ranh giới các máy tính. Khi một đối tượng được marshal, hoặc theo trị hoặc theo qui chiếu (thông qua một proxy), xuyên process hoặc xuyên máy tính, thì nó được gọi là *remoting*.

5.3.1 Tìm hiểu kiểu dữ liệu đối tượng Server

Có hai kiểu dữ liệu của các đối tượng server được hỗ trợ bởi .NET đối với remoting: *well-known* và *client-activated*. Việc thông thương liên lạc đối với các đối tượng quá quen thuộc (well-known) sẽ được thiết lập mỗi lần một thông điệp được gởi đi bởi khách hàng. Sẽ không có việc kết nối lâu dài cố định với các đối tượng well-known, như với các đối tượng được khởi động bởi khách hàng (client-activated).

Các đối tượng well-known lại được chia thành hai nhóm: *singleton* và *single-call*. Với một đối tượng well-known singleton, tất cả các thông điệp gởi cho đối tượng đến từ mọi khách hàng sẽ được chuyển cho (dispatched) một đối tượng đơn độc chạy trên server. Đối tượng sẽ được tạo ra khi server được khởi động và nằm đó chờ cung cấp dịch vụ cho bất cứ khách hàng nào tiếp xúc được đối tượng. Các đối tượng well-known phải có một hàm constructor *không thông số*.

Với một đối tượng well-known single-call, thì mỗi thông điệp mới từ khách hàng sẽ được thụ lý bởi một đối tượng mới. Việc này rất tiện lợi đối với những server farm (trang trại server) theo đấy một loạt thông điệp từ một khách hàng sẽ được thụ lý đến phiên bởi những máy tính khác nhau tùy thuộc vào sự cân bằng tải (load balancing).

[4] Remoting có nghĩa là việc gì dính dáng đến hoạt động từ xa. Tạm thời chúng tôi không dịch.

Còn các đối tượng client-activated điển hình được sử dụng bởi lập trình viên nào tạo những server tận tụy (dedicated server) được tạo ra để cung cấp dịch vụ đối với một khách hàng mà họ viết cho. Theo kịch bản này, client và server tạo một kết nối và duy trì kết nối này cho tới khi nhu cầu của khách hàng được thoả mãn.

5.3.2 Khai báo một Server với một Interface

Cách hay nhất để hiểu remoting là đi thẳng vào một thí dụ. Bạn thử tạo một máy tính bỏ túi mang 4 chức năng (cộng trừ nhân chia) trên web service (xem chương 9, "Lập trình Web Service") thi công một giao diện như theo thí dụ 5-03.

Thí dụ 5-03: Giao diện Calculator

```
using System;

namespace CalcServer
{
    public interface ICalc
    {
        double Add(double x, double y);
        double Sub(double x, double y);
        double Mult(double x, double y);
        double Div(double x, double y);
    }
}
```

Nếu bạn chạy trên Visual Studio .NET, bạn cho tạo một dự án mới kiểu C# Class Library, cho mang tên **ICalc.cs**, rồi cho Build thành **ICalc.dll**. Nếu bạn sử dụng **csc.exe** trên command line thì bạn cho ghi đoạn mã trên lên Notepad, cho cất trữ dưới tên **ICalc.cs** rồi cho biên dịch tập tin trên ở command line bằng cách khỏ vào

```
csc ICalc.cs /t:library
```

Có rất nhiều lợi điểm khi cho thi công một server thông qua một giao diện. Nếu bạn thi công calculator như là một lớp, khách hàng phải kết nối với lớp này để có thể khai báo những thể hiện trên client. Điều này giảm đi rất nhiều những lợi điểm của remoting vì những thay đổi trên server đòi hỏi việc định nghĩa lớp phải được nhật tu trên client. Nói cách khác, client và server phải được gắn kết (lệ thuộc) với nhau một cách quá chặt chẽ, như vậy không tốt. Chính giao diện giúp giảm bớt sự gắn kết quá chặt này. Thật ra, bạn có thể nhật tu việc thi công này trên server, miễn là server làm tròn khế ước mà giao diện đã đặt ra, còn phía client thì khỏi phải thay đổi gì cả.

5.3.3 Xây dựng một Server

Muốn xây dựng một server được dùng trong thí dụ này, bạn cho tạo **CalcServer.cs** trên một dự án mới kiểu **C# Console Application** rồi cho **Build**. Hoặc bạn cũng có thể khỏ đoạn mã vào Notepad, cho cất trữ dưới dạng tập tin **CalcServer.cs**, rồi khỏ vào lệnh sau đây trên command line:

```
csc CalcServer.cs /t:exe
```

Lớp **Calculator** sẽ thi công giao diện **ICalc**. Nó được dẫn xuất từ **MarshalByRef Object** do đó nó sẽ thông qua một proxy của calculator đối với ứng dụng khách hàng:

```
public class Calculator: MarshalByRefObject, Icalc
```

Việc thi công đòi hỏi nhiều hơn là một hàm constructor và những hàm hành sự đơn giản thi công 4 chức năng. Trong thí dụ này, bạn đưa phần lô gic đối với server vào hàm **Main()** của **CalcServer.cs**.

Công việc đầu tiên của bạn là tạo một channel. Sử dụng HTTP như là phương tiện chuyển tải vì nó đơn giản và không đòi hỏi một kết nối TCP/IP. Bạn có thể dùng kiểu dữ liệu **HTTPChannel** do .NET cung cấp:

```
HttpChannel chan = new HttpChannel(65100);
```

Bạn để ý là bạn đăng ký kênh lên cổng TCP/IP 65100. Chương 1, "Xuất nhập dữ liệu & Sản sinh hằng loạt đối tượng" đã đề cập đến số cổng. Tiếp theo, bạn cho đăng ký kênh với lớp **ChannelServices** sử dụng hàm static **RegisterChannel**:

```
ChannelServices.RegisterChannel(chan);
```

Bước này báo cho .NET biết bạn sẽ cung cấp HTTP services đối với cổng 65100, giống như IIS đã làm đối với cổng 80. Vì bạn đã đăng ký một HTTP channel và không cung cấp một formatter riêng của mình, nên các triệu gọi hàm sẽ dùng SOAP như là formatter mặc nhiên.

Bây giờ, bạn sẵn sàng yêu cầu lớp **RemotingConfigurationp** cho đăng ký đối tượng well-known của mình. Bạn phải trao kiểu dữ liệu cho đối tượng bạn muốn đăng ký kèm theo *endpoint*. *Endpoint* là một tên mà **RemotingConfiguration** sẽ gắn liền với kiểu dữ liệu của bạn. Nó hoàn tất địa chỉ. Nếu địa chỉ IP nhận diện máy tính và cổng nhận diện kênh thì endpoint nhận diện ứng dụng hiện hành sẽ cung cấp dịch vụ. Muốn lấy kiểu dữ liệu của đối tượng bạn có thể triệu gọi hàm static **GetType()** của lớp **Type**, trả về một đối tượng **Type**. Bạn trao qua tên đầy đủ của đối tượng mà bạn muốn có:

```
Type calcType = Type.GetType("CalcServer.Calculator");
```

Ngoài ra, bạn trao qua kiểu dữ liệu enum **WellKnownObjectMode** cho biết bạn đăng ký như là **Singleton** hoặc **SingleCall**:

```
RemotingConfiguration.RegisterWellKnownServiceType(
        calcType,"theEndPoint",WellKnownObjectMode.Singleton);
```

Việc triệu gọi hàm **RegisterWellKnownServiceType** không đưa một byte nào lên mạng mà chỉ là dùng reflection để xây dựng một proxy đối với đối tượng của bạn.

Thí dụ 5-04 cho thấy toàn bộ đoạn mã nguồn:

Thí dụ 5-04: Calculator Server

```
using System;
using System.Runtime.Remoting;
using System.Runtime.Remoting.Channels;
using System.Runtime.Remoting.Channels.Http;
namespace CalcServer
{
    // thi công lớp Calculator

    public class Calculator: MarshalByRefObject, ICalc
    {
        public Calculator()
        {
            Console.WriteLine("Calculator constructor");
        }
        public double Add(double x, double y)
        {
            Console.WriteLine("Add {0} + {1}", x,y);
            return x+y;
        }
        public double Sub(double x, double y)
        {
            Console.WriteLine("Sub {0} - {1}", x,y);
            return x-y;
        }
        public double Div(double x, double y)
        {
            Console.WriteLine("Div {0} / {1}", x,y);
            return x/y;
        }
        public double Mult(double x, double y)
        {
            Console.WriteLine("Mult {0} * {1}", x,y);
            return x*y;
        }
    }
```