如前所述，**Type** 对象是轻量级的对象引用。要更多地了解类型本身，必须获取一个 **TypeInfo** 对象，后者才代表类型定义。可调用 **System.Reflection.IntrospectionExtensions** 的 **GetTypeInfo** 扩展方法将 **Type** 对象转换成 **TypeInfo** 对象。

```
Type typeReference = ...; // 例如: o.GetType()或者 typeof(Object)
TypeInfo typeDefinition = typeReference.GetTypeInfo();
```

另外，虽然作用不大，但还可调用 **TypeInfo** 的 **AsType** 方法将 **TypeInfo** 对象转换为 **Type** 对象。

```
TypeInfo typeDefinition = ...;
Type typeReference = typeDefinition.AsType();
```

获取 **TypeInfo** 对象会强迫 CLR 确保已加载类型的定义程序集，从而对类型进行解析。这个操作可能代价高昂。如果只需要类型引用(**Type** 对象)，就应避免这个操作。但一旦获得了 **TypeInfo** 对象，就可查询类型的许多属性进一步了解它。大多数属性，比如 **IsPublic**，**IsSealed**，**IsAbstract**，**IsClass** 和 **IsValueType** 等，都指明了与类型关联的标志。另一些属性，比如 **Assembly**，**AssemblyQualifiedName**，**FullName** 和 **Module** 等，则返回定义该类型的程序集或模块的名称以及类型的全名。还可查询 **BaseType** 属性来获取对类型的基类型的引用。除此之外，还有许多方法能提供关于类型的更多信息。文档描述了 **TypeInfo** 公开的所有方法和属性。

23.3.3 构建 Exception 派生类型的层次结构

以下代码使用本章讨论的许多概念将一组程序集加载到 AppDomain 中，并显示最终从 **System.Exception** 派生的所有类。顺便说一句，20.4 节“FCL 定义的异常类”展示的 **Exception** 层次结构就是用这个程序显示的。

```
public static void Go() {
   // 显式加载想要反射的程序集
   LoadAssemblies();

   // 对所有类型进行筛选和排序
   var allTypes =
      (from a in AppDomain.CurrentDomain.GetAssemblies()
       from t in a.ExportedTypes
       where typeof(Exception).GetTypeInfo().IsAssignableFrom(t.GetTypeInfo())
       orderby t.Name
       select t).ToArray();

   // 生成并显示继承层次结构
   Console.WriteLine(WalkInheritanceHierarchy(new StringBuilder(), 0,
                     typeof(Exception), allTypes));

   private static StringBuilder WalkInheritanceHierarchy(
      StringBuilder sb, Int32 indent, Type baseType, IEnumerable<Type> allTypes) {
      String spaces = new String(' ', indent * 3);
      sb.AppendLine(spaces + baseType.FullName);
      foreach (var t in allTypes) {
         if (t.GetTypeInfo().BaseType != baseType) continue;
         WalkInheritanceHierarchy(sb, indent + 1, t, allTypes);
      }
      return sb;
   }
```

```
   private static void LoadAssemblies() {
    String[] assemblies = {
        "System,                          PublicKeyToken={0}",
        "System.Core,                     PublicKeyToken={0}",
        "System.Data,                     PublicKeyToken={0}",
        "System.Design,                   PublicKeyToken={1}",
        "System.DirectoryServices,        PublicKeyToken={1}",
        "System.Drawing,                  PublicKeyToken={1}",
        "System.Drawing.Design,           PublicKeyToken={1}",
        "System.Management,               PublicKeyToken={1}",
        "System.Messaging,                PublicKeyToken={1}",
        "System.Runtime.Remoting,         PublicKeyToken={0}",
        "System.Security,                 PublicKeyToken={1}",
        "System.ServiceProcess,           PublicKeyToken={1}",
        "System.Web,                      PublicKeyToken={1}",
        "System.Web.RegularExpressions,   PublicKeyToken={1}",
        "System.Web.Services,             PublicKeyToken={1}",
        "System.Xml,                      PublicKeyToken={0}",
    };

    String EcmaPublicKeyToken = "b77a5c561934e089";
    String MSPublicKeyToken = "b03f5f7f11d50a3a";

    // 获取包含 System.Object 的程序集的版本,
    // 假定其他所有程序集都是相同的版本
    Version version = typeof(System.Object).Assembly.GetName().Version;

    // 显式加载想要反射的程序集
    foreach (String a in assemblies) {
       String AssemblyIdentity =
          String.Format(a, EcmaPublicKeyToken, MSPublicKeyToken) +
             ", Culture=neutral, Version=" + version;
       Assembly.Load(AssemblyIdentity);
    }
}
```

23.3.4　构造类型的实例

获得对 **Type** 派生对象的引用之后，就可以构造该类型的实例了。FCL 提供了以下几个机制。

- System.Activator 的 CreateInstance 方法

  Activator 类提供了静态 **CreateInstance** 方法的几个重载版本。调用方法时既可传递一个 **Type** 对象引用，也可传递标识了类型的 **String**。直接获取类型对象的几个版本较为简单。你要为类型的构造器传递一组实参，方法返回对新对象的引用。

  用字符串来指定类型的几个版本则稍微复杂一些。首先必须指定另一个字符串来标识定义了类型的程序集。其次，如果正确配置了远程访问(remoting)选项，这些方法还允许构造远程对象。第三，这些版本返回的不是对新对象的引用，而是一个 **System.Runtime.Remoting.ObjectHandle** 对象(从 **System.MarshalByRefObject** 派生)。**ObjectHandle** 类型允许将一个 AppDomain 中创建的对象传至其他 AppDomain，期间不强迫对象具体化(materialize)。准备好具体化这个对象时，请调用 **ObjectHandle** 的 **Unwrap** 方法。在一个 AppDomain 中调用该方法时，它将定义了要具体化的类型的程序集加载到这个 AppDomain 中。如果对象按引用封送，会创建代理类型和对象。如果对象按值封送，对象的副本会被反序列化。

- System.Activator 的 CreateInstanceFrom 方法
  Activator 类还提供了一组静态 **CreateInstanceFrom** 方法。它们与 **CreateInstance** 的行为相似，只是必须通过字符串参数来指定类型及其程序集。程序集用 **Assembly** 的 **LoadFrom**(而非 Load)方法加载到调用 **AppDomain** 中。由于都不接受 Type 参数，所以返回的都是一个 **ObjectHandle** 对象引用，必须调用 **ObjectHandle** 的 **Unwrap** 方法进行具体化。

- System.AppDomain 的方法
  AppDomain 类型提供了 4 个用于构造类型实例的实例方法(每个都有几个重载版本)，包括 **CreateInstance**，**CreateInstanceAndUnwrap**，**CreateInstanceFrom** 和 **CreateInstanceFromAndUnwrap**。这些方法的行为和 **Activator** 类的方法相似，区别在于它们都是实例方法，允许指定在哪个 AppDomain 中构造对象。另外，带 **Unwrap** 后缀的方法还能简化操作，不必执行额外的方法调用。

- System.Reflection.ConstructorInfo 的 Invoke 实例方法
  使用一个 **Type** 对象引用，可以绑定到一个特定的构造器，并获取对构造器的 **ConstructorInfo** 对象的引用。然后，可利用 **ConstructorInfo** 对象引用来调用它的 **Invoke** 方法。类型总是在调用 AppDomain 中创建，返回的是对新对象的引用。本章稍后会详细讨论该方法。



> 注意 CLR 不要求值类型定义任何构造器。但这会造成一个问题，因为上述列表中的所有机制都要求调用构造器来构造对象。然而，**Activator** 的 **CreateInstance** 方法允许在不调用构造器的情况下创建值类型的实例。要在不调用构造器的情况下创建值类型的实例，必须调用 **CreateInstance** 方法获取单个 **Type** 参数的版本或者获取 **Type** 和 **Boolean** 参数的版本。

利用前面列出的机制，可为除数组(**System.Array** 派生类型)和委托(**System.MulticastDelegate** 派生类型)之外的所有类型创建对象。创建数组需要调用 **Array** 的静态 **CreateInstance** 方法(有几个重载的版本)。所有版本的 **CreateInstance** 方法获取的第一个参数都是对数组元素 **Type** 的引用。**CreateInstance** 的其他参数允许指定数组维数和上下限的各种组合。创建委托则要调用 **MethodInfo** 的静态 **CreateDelegate** 方法。所有版本的 **CreateDelegate** 方法获取的第一个参数都是对委托 **Type** 的引用。**CreateDelegate** 方法的其他参数允许指定在调用实例方法时应将哪个对象作为 **this** 参数传递。

构造泛型类型的实例首先要获取对开放类型的引用，然后调用 **Type** 的 **MakeGenericType** 方法并向其传递一个数组(其中包含要作为类型实参使用的类型)[①]。然后，获取返回的 **Type** 对象并把它传给上面列出的某个方法。下面是一个例子：

```
using System;
```

① 要想进一步了解开放类型、封闭类型、类型参数和类型实参等术语，请参见第 12 章“泛型”。——译注

```
using System.Reflection;

internal sealed class Dictionary<TKey, TValue> { }

public static class Program {
   public static void Main() {
      // 获取对泛型类型的类型对象的引用
      Type openType = typeof(Dictionary<,>);

      // 使用 TKey=String、TValue=Int32 封闭泛型类型①
      Type closedType = openType.MakeGenericType(typeof(String), typeof(Int32));

      // 构造封闭类型的实例
      Object o = Activator.CreateInstance(closedType);

      // 证实能正常工作
      Console.WriteLine(o.GetType());
   }
}
```

编译并运行上述代码得到以下输出：

```
Dictionary`2[System.String,System.Int32]
```

23.4　设计支持加载项的应用程序

构建可扩展应用程序时，接口是中心。可用基类代替接口，但接口通常是首选的，因为它允许加载项开发人员选择他们自己的基类。例如，假定要写一个应用程序来无缝地加载和使用别人创建的类型。下面描述了如何设计这种应用程序。

- 创建“宿主 SDK”(Host SDK)程序集，它定义了一个接口，接口的方法作为宿主应用程序与加载项之间的通信机制使用。为接口方法定义参数和返回类型时，请尝试使用 MSCorLib.dll 中定义的其他接口或类型。要传递并返回自己的数据类型，也在“宿主 SDK”程序集中定义。一旦搞定接口定义，就可为这个程序集赋予强名称(参见第 3 章)，然后把它打包并部署到合作伙伴和用户那里。发布后要避免对该程序集中的类型做出任何重大的改变。例如，不要以任何方式更改接口。但如果定义了任何数据类型，在类型中添加新成员是完全允许的。对程序集进行任何修改之后，可能需要使用一个发布者策略文件来部署它(也参见第 3 章的讨论)。



> **注意**　之所以能使用 MSCorLib.dll 中定义的类型，是因为 CLR 总是加载与 CLR 本身的版本匹配的那个版本的 MSCorLib.dll。此外，一个 CLR 实例只会加载一个版本的 MSCorLib.dll。换言之，永远不会出现多个不同版本的 MSCorLib.dll 都加载的情况(详见第 3 章)。最后结果就是，绝不会出现类型版本不匹配的情况。这还有助于减少应用程序对内存的需求。

- 当然，加载项开发人员会在加载项程序集中定义自己的类型。这些程序集将引用你的

① 开放类型变成了封闭类型。——译注

"宿主"程序集中的类型。加载项开发人员可按自己的步调推出程序集的新版本，而宿主应用程序能正常使用加载项中的类型，不会出任何纰漏。

- 创建单独的"宿主应用程序"程序集，在其中包含你的应用程序的类型。这个程序集显然要引用"宿主 SDK"程序集，并使用其中定义的类型。可自由修改"宿主应用程序"程序集的代码。由于加载项开发人员不会引用这个"宿主应用程序"程序集，所以随时都能推出"宿主应用程序"程序集的新版本，这不会对加载项开发人员产生任何影响。

本节包含了一些非常重要的信息。跨程序集使用类型时，需要关注程序集的版本控制问题。要花一些时间精心建构，将跨程序集通信的类型隔离到它们自己的程序集中。要避免以后更改这些类型的定义。但是，如果真的要修改类型定义，一定要修改程序集的版本号，并为新版本的程序集创建发布者策略文件。

下面来看一个非常简单的例子，它综合运用了所有这些知识。首先是 HostSDK.dll 程序集的代码：

```
using System;

namespace Wintellect.HostSDK {
   public interface IAddIn {
      String DoSomething(Int32 x);
   }
}
```

其次是 AddInTypes.dll 程序集的代码，其中定义了两个公共类型，它们实现了 HostSDK.dll 的接口。要生成该程序集，必须引用 HostSDK.dll 程序集：

```
using System;
using Wintellect.HostSDK;

public sealed class AddIn_A : IAddIn {
   public AddIn_A() {
   }
   public String DoSomething(Int32 x) {
      return "AddIn_A: " + x.ToString();
   }
}

public sealed class AddIn_B : IAddIn {
   public AddIn_B() {
   }
   public String DoSomething(Int32 x) {
      return "AddIn_B: " + (x * 2).ToString();
   }
}
```

然后是一个简单的 Host.exe 程序集(控制台应用程序)的代码。生成该程序集必须引用 HostSDK.dll 程序集。为了发现有哪些可用的加载项类型，以下宿主代码假定类型是在一个以.dll 文件扩展名结尾的程序集中定义的，而且这些程序集已部署到和宿主的 EXE 文件相同的目录中。Microsoft 的"托管可扩展性框架"(Managed Extensibility Framework，MEF)是在我刚才描述的各种机制的顶部构建的，它还提供了加载项注册和发现机制。构建动态可扩展应用程序时，强烈建议研究一下 MEF，它能简化本章描述的一些操作。

```
using System;
using System.IO;
using System.Reflection;
using System.Collections.Generic;
using Wintellect.HostSDK;

public static class Program {
   public static void Main() {
      // 查找宿主 EXE 文件所在的目录
      String AddInDir = Path.GetDirectoryName(Assembly.GetEntryAssembly().Location);

      // 假定加载项程序集和宿主 EXE 文件在同一个目录
      var AddInAssemblies = Directory.EnumerateFiles(AddInDir, "*.dll");

      // 创建可由宿主使用的所有加载 Type 的一个集合
      var AddInTypes =
         from file in AddInAssemblies
         let assembly = Assembly.Load(file)
         from t in assembly.ExportedTypes // 公开导出的类型
         // 如果类型实现了 IAddIn 接口，该类型就可由宿主使用
         where t.IsClass && typeof(IAddIn).GetTypeInfo().IsAssignableFrom(t.GetTypeInfo())
         select t;
      // 初始化完成：宿主已发现了所有可用的加载项

      // 下面示范宿主如何构造加载项对象并使用它们
      foreach (Type t in AddInTypes) {
         IAddIn ai = (IAddIn) Activator.CreateInstance(t);
         Console.WriteLine(ai.DoSomething(5));
      }
   }
}
```

这个简单的宿主/加载项例子没有用到 AppDomain。但在实际应用中，每个加载项都可能要在自己的 AppDomain 中创建，每个 AppDomain 都有自己的安全性和配置设置。当然，如果希望将加载项从内存中移除，可以卸载相应的 AppDomain。为了跨 AppDomain 边界通信，可告诉加载项开发人员从 **MashalByRefObject** 派生出他们自己的加载类型。但另一个更常见的办法是让宿主应用程序定义自己的、从 **MashalByRefObject** 派生的内部类型。每个 AppDomain 创建好后，宿主要在新 AppDomain 中创建它自己的 **MashalByRefObject** 派生类型实例。宿主的代码(位于默认 AppDomain 中)将与它自己的类型(位于其他 AppDomain 中)通信，让后者载入加载项程序集，并创建和使用加载的类型的实例。

23.5　使用反射发现类型的成员

到目前为止，本章的重点一直都是构建动态可扩展应用程序所需的反射机制，包括程序集加载、类型发现以及对象构造。要获得好的性能和编译时的类型安全性，应尽量避免使用反射。如果是动态可扩展应用程序，构造好对象后，宿主代码一般要将对象转型为编译时已知的接口类型或者基类。这样访问对象的成员就可以获得较好的性能，而且可以确保编译时的类型安全性。

本章剩余部分将从其他角度探讨反射，目的是发现并调用类型的成员。一般利用这个功能创建开发工具和实用程序，查找特定编程模式或者对特定成员的使用，从而对程序集进行分析。例子包括 ILDasm，FxCopCmd.exe 以及 Visual Studio 的 Windows 窗体/WPF/Web 窗

体设计器。另外，一些类库也利用这个功能发现和调用类型的成员，为开发人员提供便利和丰富的功能。例子包括执行序列化/反序列化以及简单数据绑定的类库。

23.5.1 发现类型的成员

字段、构造器、方法、属性、事件和嵌套类型都可以定义成类型的成员。FCL 包含抽象基类 **System.Reflection.MemberInfo**，封装了所有类型成员都通用的一组属性。**MemberInfo** 有许多派生类，每个都封装了与特定类型成员相关的更多属性。图 23-1 是这些类型的层次结构。

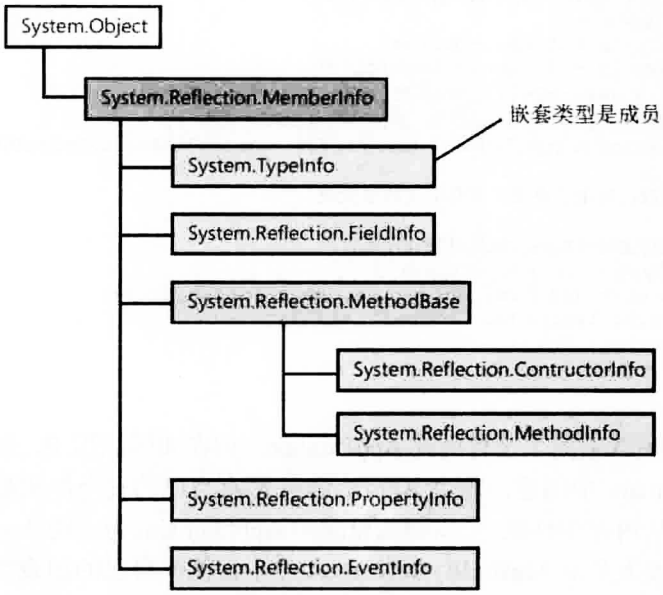


图 23-1 封装了类型成员信息的反射类型层次结构

以下程序演示了如何查询类型的成员并显示成员的信息。代码处理的是由调用 AppDomain 加载的所有程序集定义的所有公共类型。对每个类型都调用 **DeclaredMembers** 属性以返回由 **MemberInfo** 派生对象构成的集合；每个对象都引用类型中定义的一个成员。然后，显示每个成员的种类(字段、构造器、方法和属性等)及其字符串值(调用 **ToString** 来获取)。

```
using System;
using System.Reflection;

public static class Program {
   public static void Main() {
      // 遍历这个 AppDomain 中加载的所有程序集
      Assembly[] assemblies = AppDomain.CurrentDomain.GetAssemblies();
      foreach (Assembly a in assemblies) {
         Show(0, "Assembly: {0}", a);

         // 查找程序集中的类型
         foreach (Type t in a.ExportedTypes) {
            Show(1, "Type: {0}", t);

            // 发现类型的成员
            foreach (MemberInfo mi in t.GetTypeInfo().DeclaredMembers) {
```

```
                String typeName = String.Empty;
                if (mi is Type) typeName = "(Nested) Type";
                if (mi is FieldInfo) typeName = "FieldInfo";
                if (mi is MethodInfo) typeName = "MethodInfo";
                if (mi is ConstructorInfo) typeName = "ConstructoInfo";
                if (mi is PropertyInfo) typeName = "PropertyInfo";
                if (mi is EventInfo) typeName = "EventInfo";
                Show(2, "{0}: {1}", typeName, mi);
            }
        }
    }
}

private static void Show(Int32 indent, String format, params Object[] args) {
    Console.WriteLine(new String(' ', 3 * indent) + format, args);
}
}
```

编译并运行上述代码会产生大量输出。下面摘录了其中一小部分：

```
Assembly: mscorlib, Version=4.0.0.0, Culture=neutral, PublicKeyToken=b77a5c561934e089
   Type: System.Object
      MethodInfo: System.String ToString()
      MethodInfo: Boolean Equals(System.Object)
      MethodInfo: Boolean Equals(System.Object, System.Object)
      MethodInfo: Boolean ReferenceEquals(System.Object, System.Object)
      MethodInfo: Int32 GetHashCode()
      MethodInfo: System.Type GetType()
      MethodInfo: Void Finalize()
      MethodInfo: System.Object MemberwiseClone()
      MethodInfo: Void FieldSetter(System.String, System.String, System.Object)
      MethodInfo: Void FieldGetter(System.String, System.String, System.Object ByRef)
      MethodInfo: System.Reflection.FieldInfo GetFieldInfo(System.String, System.String)
      ConstructoInfo: Void .ctor()
   Type: System.Collections.Generic.IComparer`1[T]
      MethodInfo: Int32 Compare(T, T)
   Type: System.Collections.IEnumerator
      MethodInfo: Boolean MoveNext()
      MethodInfo: System.Object get_Current()
      MethodInfo: Void Reset()
      PropertyInfo: System.Object Current
   Type: System.IDisposable
      MethodInfo: Void Dispose()
   Type: System.Collections.Generic.IEnumerator`1[T]
      MethodInfo: T get_Current()
      PropertyInfo: T Current
   Type: System.ArraySegment`1[T]
      MethodInfo: T[] get_Array()
      MethodInfo: Int32 get_Offset()
      MethodInfo: Int32 get_Count()
      MethodInfo: Int32 GetHashCode()
      MethodInfo: Boolean Equals(System.Object)
      MethodInfo: Boolean Equals(System.ArraySegment`1[T])
      MethodInfo: Boolean op_Equality(System.ArraySegment`1[T], System.ArraySegment`1[T])
      MethodInfo: Boolean op_Inequality(System.ArraySegment`1[T], System.ArraySegment`1[T])
      ConstructoInfo: Void .ctor(T[])
      ConstructoInfo: Void .ctor(T[], Int32, Int32)
      PropertyInfo: T[] Array
      PropertyInfo: Int32 Offset
      PropertyInfo: Int32 Count
      FieldInfo: T[] _array
      FieldInfo: Int32 _offset
```

由于 **MemberInfo** 类是成员层次结构的根，所以有必要更深入地研究一下它。表 23-1 展示了 **MemberInfo** 类提供的几个只读属性和方法。这些属性和方法是一个类型的所有成员都

通用的。不要忘了 **System.TypeInfo** 从 **MemberInfo** 派生。**TypeInfo** 也提供了表 23-1 列出的所有属性。

表 23-1 MemberInfo 的所有派生类型都通用的属性和方法

成员名称	成员类型	说明
Name	一个 **String** 属性	返回成员名称
DeclaringType	一个 **Type** 属性	返回声明成员的 **Type**
Module	一个 **Module** 属性	返回声明成员的 **Module**
CustomAttributes	该属性返回一个 **IEnumerable<CustomAttributeData>**	返回一个集合，其中每个元素都标识了应用于该成员的一个定制特性的实例。定制特性可应用于任何成员。虽然 **Assembly** 不从 **MemberInfo** 派生，但它提供了可用于程序集的相同属性

在查询 **DeclaredMembers** 属性所返回的集合中，每个元素都是对层次结构中的一个具体类型的引用。虽然 **TypeInfo** 的 **DeclaredMembers** 属性能返回类型的所有成员，但还可利用 **TypeInfo** 提供的一些方法返回具有指定字符串名称的成员类型。例如，利用 **TypeInfo** 的 **GetDeclaredNestedType**，**GetDeclaredField**，**GetDeclaredMethod**，**GetDeclaredProperty** 和 **GetDeclaredEvent** 方法，可分别返回一个 **TypeInfo**、**FieldInfo**、**MethodInfo**、**PropertyInfo** 和 **EventInfo** 对象引用。而利用 **GetDeclaredMethods** 方法能返回由 **MethodInfo** 对象构成的集合，这些对象描述了和指定字符串名称匹配的一个(多个)方法。

图 23-2 总结了用于遍历反射对象模型的各种类型。基于 AppDomain，可发现其中加载的所有程序集。基于程序集，可发现构成它的所有模块。基于程序集或模块，可发现它定义的所有类型。基于类型，可发现它的嵌套类型、字段、构造器、方法、属性和事件。命名空间不是这个层次结构的一部分，因为它们只是从语法角度将相关类型聚集到一起。CLR 不知道什么是命名空间。要列出程序集中定义的所有命名空间，需枚举程序集中的所有类型，并查看其 **Namespace** 属性。

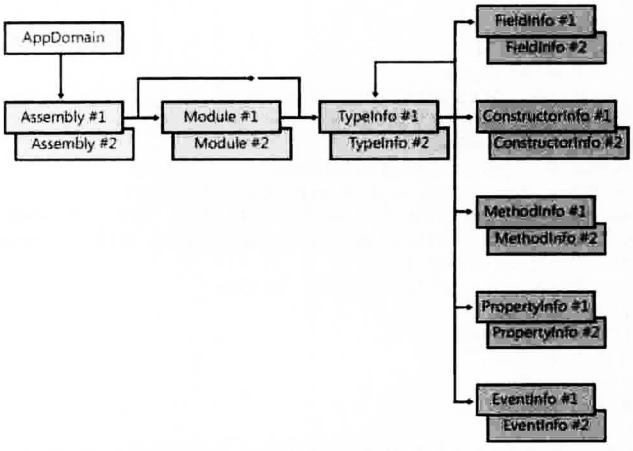


图 23-2 应用程序用于遍历反射对象模型的各种类型

基于一个类型，还可发现它实现的接口。基于构造器、方法、属性访问器方法或者事件的添加/删除方法，可调用 **GetParameters** 方法来获取由 **ParameterInfo** 对象构成的数组，从而了解成员的参数的类型。还可查询只读属性 **ReturnParameter** 获得一个 **ParameterInfo** 对象，它详细描述了成员的返回类型。对于泛型类型或方法，可调用 **GetGenericArguments** 方法来获得类型参数的集合。最后，针对上述任何一项，都可查询 **CustomAttributes** 属性来获得应用于它们的自定义定制特性的集合。

23.5.2　调用类型的成员

发现类型定义的成员后可调用它们。“调用”(invoke)的确切含义取决于要调用的成员的种类。表 23-2 展示了为了调用(invoke)一种成员而需调用(call)的方法。

表 23-2　如何调用成员

成员类型	调用(invoke)成员而需调用(call)的方法
FieldInfo	调用 **GetValue** 获取字段的值
	调用 **SetValue** 设置字段的值
ConstructorInfo	调用 **Invoke** 构造类型的实例并调用构造器
MethodInfo	调用 **Invoke** 来调用类型的方法
PropertyInfo	调用 **GetValue** 来调用的属性的 get 访问器方法
	调用 **SetValue** 来调用属性的 set 访问器方法
EventInfo	调用 **AddEventHandler** 来调用事件的 add 访问器方法
	调用 **RemoveEventHandler** 来调用事件的 remove 访问器方法

PropertyInfo 类型代表与属性有关的元数据信息(参见第 10 章“属性”)；也就是说，**PropertyInfo** 提供了 **CanRead**、**CanWrite** 和 **PropertyType** 只读属性，它们指出属性是否可读和可写，以及属性的数据类型是什么。**PropertyInfo** 还提供了只读 **GetMethod** 和 **SetMethod** 属性，它们返回代表属性 get 和 set 访问器方法的 **MethodInfo** 对象。**PropertyInfo** 的 **GetValue** 和 **SetValue** 方法只是为了提供方便；在内部，它们会自己调用合适的 **MethodInfo** 对象。为了支持有参属性(C#的索引器)，**GetValue** 和 **SetValue** 方法提供了一个 **Object[]**类型的 **index** 参数。

EventInfo 类型代表与事件有关的元数据信息(参见第 11 章“事件”)。**EventInfo** 类型提供了只读 **EventHandlerType** 属性，返回事件的基础委托的 **Type**。**EventInfo** 类型还提供了只读 **AddMethod** 和 **RemoveMethod** 属性，返回为事件增删委托的方法的 **MethodInfo** 对象。增删委托可调用这些 **MethodInfo** 对象，也可调用 **EventInfo** 类型提供的更好用的 **AddEventHandler** 和 **RemoveEventHandler** 方法。

以下示例应用程序演示了用反射来访问类型成员的各种方式。**SomeType** 类包含多种成员：一个私有字段(**m_someField**)；一个公共构造器(**SomeType**)，它获取一个传引用的 **Int32** 实参；一个公共方法(**ToString**)；一个公共属性(**SomeProp**)；以及一个公共事件(**SomeEvent**)。定义好 **SomeType** 类型后，我提供了三个不同的方法，它们利用反射来访问 **SomeType** 的

成员。三个方法用不同的方式做相同的事情。

- **BindToMemberThenInvokeTheMember** 方法演示了如何绑定到成员并调用它。
- **BindToMemberCreateDelegateToMemberThenInvokeTheMember** 方法演示了如何绑定到一个对象或成员，然后创建一个委托来引用该对象或成员。通过委托来调用的速度很快。如果需要在相同的对象上多次调用相同的成员，这个技术的性能比上一个好。
- **UseDynamicToBindAndInvokeTheMember** 方法演示了如何利用 **C#**的 **dynamic** 基元类型(参见第 5 章“基元类型、引用类型和值类型”)简化成员访问语法。此外，在相同类型的不同对象上调用相同成员时，这个技术还能提供不错的性能，因为针对每个类型，绑定都只会发生一次。而且可以缓存起来，以后多次调用的速度会非常快。用这个技术也可以调用不同类型的对象的成员。

```
using System;
using System.Reflection;
using Microsoft.CSharp.RuntimeBinder;
using System.Linq;

// 该类用于演示反射机制，
// 其中定义了一个字段、构造器、方法、属性和一个事件
internal sealed class SomeType {
   private Int32 m_someField;
   public SomeType(ref Int32 x) { x *= 2; }
   public override String ToString() { return m_someField.ToString(); }
   public Int32 SomeProp
   {
      get { return m_someField; }
      set {
        if (value < 1)
           throw new ArgumentOutOfRangeException("value");
        m_someField = value;
      }
   }
   public event EventHandler SomeEvent;
   private void NoCompilerWarnings() { SomeEvent.ToString(); }
}

public static class Program {
   public static void Main()
   {
      Type t = typeof(SomeType);
      BindToMemberThenInvokeTheMember(t);
      Console.WriteLine();

      BindToMemberCreateDelegateToMemberThenInvokeTheMember(t);
      Console.WriteLine();

      UseDynamicToBindAndInvokeTheMember(t);
      Console.WriteLine();
   }

   private static void BindToMemberThenInvokeTheMember(Type t) {
      Console.WriteLine("BindToMemberThenInvokeTheMember");

      // 构造实例
      Type ctorArgument = Type.GetType("System.Int32&");   // 或者typeof(Int32).MakeByRefType();
      ConstructorInfo ctor = t.GetTypeInfo().DeclaredConstructors.First(
         c => c.GetParameters()[0].ParameterType == ctorArgument);
      Object[] args = new Object[] { 12 }; // 构造器的实参
```

```
    Console.WriteLine("x before constructor called: " + args[0]);
    Object obj = ctor.Invoke(args);
    Console.WriteLine("Type: " + obj.GetType());
    Console.WriteLine("x after constructor returns: " + args[0]);

    // 读写字段
    FieldInfo fi = obj.GetType().GetTypeInfo().GetDeclaredField("m_someField");
    fi.SetValue(obj, 33);
    Console.WriteLine("someField: " + fi.GetValue(obj));

    // 调用方法
    MethodInfo mi = obj.GetType().GetTypeInfo().GetDeclaredMethod("ToString");
    String s = (String)mi.Invoke(obj, null);
    Console.WriteLine("ToString: " + s);

    // 读写属性
    PropertyInfo pi = obj.GetType().GetTypeInfo().GetDeclaredProperty("SomeProp");
    try
    {
       pi.SetValue(obj, 0, null);
    }
    catch (TargetInvocationException e) {
       if (e.InnerException.GetType() != typeof(ArgumentOutOfRangeException)) throw;
       Console.WriteLine("Property set catch.");
    }
    pi.SetValue(obj, 2, null);
    Console.WriteLine("SomeProp: " + pi.GetValue(obj, null));

    // 为事件添加和删除委托
    EventInfo ei = obj.GetType().GetTypeInfo().GetDeclaredEvent("SomeEvent");
    EventHandler eh = new EventHandler(EventCallback); // See ei.EventHandlerType
    ei.AddEventHandler(obj, eh);
    ei.RemoveEventHandler(obj, eh);
}

// 添加到事件的回调方法
private static void EventCallback(Object sender, EventArgs e) { }

private static void BindToMemberCreateDelegateToMemberThenInvokeTheMember(Type t) {
    Console.WriteLine("BindToMemberCreateDelegateToMemberThenInvokeTheMember");

    // 构造实例(不能创建对构造器的委托)
    Object[] args = new Object[] { 12 }; // 构造器实参
    Console.WriteLine("x before constructor called: " + args[0]);
    Object obj = Activator.CreateInstance(t, args);
    Console.WriteLine("Type: " + obj.GetType().ToString());
    Console.WriteLine("x after constructor returns: " + args[0]);

    // 注意：不能创建对字段的委托

    // 调用方法
    MethodInfo mi = obj.GetType().GetTypeInfo().GetDeclaredMethod("ToString");
    var toString = mi.CreateDelegate<Func<String>>(obj);
    String s = toString();
    Console.WriteLine("ToString: " + s);

    // 读写属性
    PropertyInfo pi = obj.GetType().GetTypeInfo().GetDeclaredProperty("SomeProp");
    var setSomeProp = pi.SetMethod.CreateDelegate<Action<Int32>>(obj);
    try {
       setSomeProp(0);
    }
    catch (ArgumentOutOfRangeException) {
       Console.WriteLine("Property set catch.");
    }
    setSomeProp(2);
```

```
            var getSomeProp = pi.GetMethod.CreateDelegate<Func<Int32>>(obj);
            Console.WriteLine("SomeProp: " + getSomeProp());

            // 向事件增删委托
            EventInfo ei = obj.GetType().GetTypeInfo().GetDeclaredEvent("SomeEvent");
            var addSomeEvent = ei.AddMethod.CreateDelegate<Action<EventHandler>>(obj);
            addSomeEvent(EventCallback);
            var removeSomeEvent = ei.RemoveMethod.CreateDelegate<Action<EventHandler>>(obj);
            removeSomeEvent(EventCallback);
        }

        private static void UseDynamicToBindAndInvokeTheMember(Type t) {
            Console.WriteLine("UseDynamicToBindAndInvokeTheMember");

            // 构造实例(不能创建对构造器的委托)
            Object[] args = new Object[] { 12 }; // 构造器的实参
            Console.WriteLine("x before constructor called: " + args[0]);
            dynamic obj = Activator.CreateInstance(t, args);
            Console.WriteLine("Type: " + obj.GetType().ToString());
            Console.WriteLine("x after constructor returns: " + args[0]);

            // 读写字段
            try {
                obj.m_someField = 5;
                Int32 v = (Int32)obj.m_someField;
                Console.WriteLine("someField: " + v);
            }
            catch (RuntimeBinderException e) {
                // 之所以会执行到这里，是因为字段是私有的
                Console.WriteLine("Failed to access field: " + e.Message);
            }

            // 调用方法
            String s = (String)obj.ToString();
            Console.WriteLine("ToString: " + s);

            // 读写属性
            try {
                obj.SomeProp = 0;
            }
            catch (ArgumentOutOfRangeException) {
                Console.WriteLine("Property set catch.");
            }
            obj.SomeProp = 2;
            Int32 val = (Int32)obj.SomeProp;
            Console.WriteLine("SomeProp: " + val);

            // 从事件增删委托
            obj.SomeEvent += new EventHandler(EventCallback);
            obj.SomeEvent -= new EventHandler(EventCallback);
        }
    }

    internal static class ReflectionExtensions {
      // 这个辅助扩展方法简化了创建委托的语法
      public static TDelegate CreateDelegate<TDelegate>(this MethodInfo mi, Object target =
    null){
          return (TDelegate)(Object)mi.CreateDelegate(typeof(TDelegate), target);
      }
    }
```

生成并运行上述代码得到以下输出：

```
BindToMemberThenInvokeTheMember
x before constructor called: 12
Type: SomeType
```

```
x after constructor returns: 24
someField: 33
ToString: 33
Property set catch.
SomeProp: 2

BindToMemberCreateDelegateToMemberThenInvokeTheMember
x before constructor called: 12
Type: SomeType
x after constructor returns: 24
ToString: 0
Property set catch.
SomeProp: 2

UseDynamicToBindAndInvokeTheMember
x before constructor called: 12
Type: SomeType
x after constructor returns: 24
Failed to access field: “SomeType.m_someField”不可访问，因为它具有一定的保护级
别
ToString: 0
Property set catch.
SomeProp: 2
```

注意，**SomeType** 构造器唯一的参数就是传引用的 **Int32**。上述代码演示了如何调用这个构造器，如何在构造器返回后检查修改过的 **Int32** 值。在 **BindToMemberThenInvokeTheMember** 方法靠近顶部的地方，我调用 **Type** 的 **GetType** 方法并传递字符串**"System.Int32&"**。其中的“&”表明参数是传引用的。这个符号是类型名称的巴克斯-诺尔范式(BNF)语法的一部分，详情请参考文档。在注释中，我还解释了如何用 **Type** 的 **MakeByRefType** 方法获得相同效果。

23.5.3　使用绑定句柄减少进程的内存消耗

许多应用程序都绑定了一组类型(**Type** 对象)或类型成员(**MemberInfo** 派生对象)，并将这些对象保存在某种形式的集合中。以后，应用程序搜索这个集合，查找特定对象，然后调用(invoke)这个对象。这个机制很好，只是有个小问题：**Type** 和 **MemberInfo** 派生对象需要大量内存。所以，如果应用程序容纳了太多这样的对象，但只是偶尔调用，应用程序消耗的内存就会急剧增加，对应用程序的性能产生负面影响。

CLR 内部用更精简的方式表示这种信息。CLR 之所以为应用程序创建这些对象，只是为了方便开发人员。CLR 不需要这些大对象就能运行。如果需要保存/缓存大量 **Type** 和 **MemberInfo** 派生对象，开发人员可以使用运行时句柄(runtime handle)代替对象以减小工作集(占用的内存)。FCL 定义了三个运行时句柄类型(全部都在 **System** 命名空间中)，包括 **RuntimeTypeHandle**，**RuntimeFieldHandle** 和 **RuntimeMethodHandle**。三个类型都是值类型，都只包含一个字段，也就是一个 **IntPtr**；这使类型的实例显得相当精简(相当省内存)。**IntPtr** 字段是一个句柄，引用了 AppDomain 的 Loader 堆中的一个类型、字段或方法。因此，现在需要以一种简单、高效的方式将重量级的 **Type** 或 **MemberInfo** 对象转换为轻量级的运行时句柄实例，反之亦然。幸好，使用以下转换方法和属性可轻松达到目的。

- 要将 Type 对象转换为一个 **RuntimeTypeHandle**，调用 **Type** 的静态 **GetTypeHandle**

方法并传递那个 **Type** 对象引用。

- 要将一个 **RuntimeTypeHandle** 转换为 **Type** 对象，调用 **Type** 的静态方法 **GetTypeFromHandle**，并传递那个 **RuntimeTypeHandle**。
- 要将 **FieldInfo** 对象转换为一个 **RuntimeFieldHandle**，查询 **FieldInfo** 的实例只读属性 **FieldHandle**。
- 要将一个 **RuntimeFieldHandle** 转换为 **FieldInfo** 对象，调用 **FieldInfo** 的静态方法 **GetFieldFromHandle**。
- 要将 **MethodInfo** 对象转换为一个 **RuntimeMethodHandle**，查询 **MethodInfo** 的实例只读属性 **MethodHandle**。
- 要将一个 **RuntimeMethodHandle** 转换为一个 **MethodInfo** 对象，调用 MethodInfo 的静态方法 **GetMethodFromHandle**。

以下示例程序获取许多 **MethodInfo** 对象，把它们转换为 **RuntimeMethodHandle** 实例，并演示了转换前后的工作集的差异：

```
using System;
using System.Reflection;
using System.Collections.Generic;

public sealed class Program {
    private const BindingFlags c_bf = BindingFlags.FlattenHierarchy |
       BindingFlags.Instance | BindingFlags.Static |
       BindingFlags.Public | BindingFlags.NonPublic;

    public static void Main() {
       // 显示在任何反射操作之前堆的大小
       Show("Before doing anything");

       // 为 MSCorlib.dll 中的所有方法构建 MethodInfo 对象缓存
       List<MethodBase> methodInfos = new List<MethodBase>();
       foreach (Type t in typeof(Object).Assembly.GetExportedTypes()) {
          // 跳过任何泛型类型
          if (t.IsGenericTypeDefinition) continue;

          MethodBase[] mb = t.GetMethods(c_bf);
          methodInfos.AddRange(mb);
       }

       // 显示当绑定所有方法之后，方法的个数和堆的大小
       Console.WriteLine("# of methods={0:N0}", methodInfos.Count);
       Show("After building cache of MethodInfo objects");

       // 为所有 MethodInfo 对象构建 RuntimeMethodHandle 缓存
       List<RuntimeMethodHandle> methodHandles =
          methodInfos.ConvertAll<RuntimeMethodHandle>(mb => mb.MethodHandle);

       Show("Holding MethodInfo and RuntimeMethodHandle cache");
       GC.KeepAlive(methodInfos);       // 阻止缓存被过早垃圾回收

       methodInfos = null;              // 现在允许缓存垃圾回收
       Show("After freeing MethodInfo objects");

       methodInfos = methodHandles.ConvertAll<MethodBase>(
```

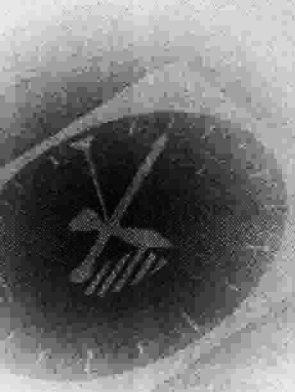
```
            rmh => MethodBase.GetMethodFromHandle(rmh));
        Show("Size of heap after re-creating MethodInfo objects");
        GC.KeepAlive(methodHandles);     // 阻止缓存被过早垃圾回收
        GC.KeepAlive(methodInfos);       // 阻止缓存被过早垃圾回收

        methodHandles = null;            // 现在允许缓存垃圾回收
        methodInfos = null;              // 现在允许缓存垃圾回收
        Show("After freeing MethodInfos and RuntimeMethodHandles");
    }

    private static void Show(String s) {
        Console.WriteLine("Heap size={0,12:N0} - {1}",
            GC.GetTotalMemory(true), s);
    }
}
```

编译并运行以上程序，在我的机器上得到以下输出：

```
Heap size=     22,060 - Before doing anything
# of methods=52,737
Heap size=  3,996,640 - After building cache of MethodInfo objects
Heap size=  4,207,656 - Holding MethodInfo and RuntimeMethodHandle cache
Heap size=  3,847,352 - After freeing MethodInfo objects
Heap size=  4,058,336 - Size of heap after re-creating MethodInfo objects
Heap size=     81,224 - After freeing MethodInfos and RuntimeMethodHandles
```



第 24 章　运行时序列化

本章内容：

- 序列化/反序列化快速入门
- 使类型可序列化
- 控制序列化和反序列化
- 格式化器如何序列化类型实例
- 控制序列化/反序列化的数据
- 流上下文
- 将类型序列化为不同的类型以及将对象反序列化为不同的对象
- 序列化代理
- 反序列化对象时重写程序集和/或类型

序列化是将对象或对象图[①]转换成字节流的过程。**反序列化**是将字节流转换回对象图的过程。在对象和字节流之间转换是很有用的机制。下面是一些例子。

- 应用程序的状态(对象图)可轻松保存到磁盘文件或数据库中，并在应用程序下次运行时恢复。ASP.NET 就是利用序列化和反序列化来保存和还原会话状态的。
- 一组对象可轻松复制到系统的剪贴板，再粘贴回同一个或另一个应用程序。事实上，Windows 窗体和 Windows Presentation Foundation(WPF)就利用了这个功能。
- 一组对象可克隆并放到一边作为“备份”；与此同时，用户操纵一组“主”对象。
- 一组对象可轻松地通过网络发送给另一台机器上运行的进程。Microsoft .NET Framework 的 Remoting(远程处理)架构会对按值封送(marshaled by value)的对象进行序列化和反序列化。这个技术还可跨 AppDomain 边界发送对象，具体如第 22 章“CLR

① 本书将 object graph 翻译成“对象图”。对象图是一个抽象的概念，代表的是对象系统在特定时间点的一个视图。另一个常用的术语 object diagram 则是指总体 object graph 的一个子集。普通的对象模型(比如 UML 类图)描述的是对象之间的关系，而对象图侧重于它们的实例在特定时间点的状态。在面向对象应用程序中，相互关联的对象构成了一个复杂的网络。一个对象可能拥有或包含另一个对象，或者容纳了对另一个对象的引用。这样一来，不同的对象便相互链接起来了。这个对象网络便是对象图。它是一种比较抽象的结构，可在讨论应用程序的状态时使用它。注意，在.NET Framework SDK 中文文档中，由对象相互连接而构成的对象图被称为“连接对象图形”。——译注

寄宿和 AppDomain”所述。

除了上述应用，一旦将对象序列化成内存中的字节流，就可方便地以一些更有用的方式处理数据，比如进行加密和压缩。

由于序列化如此有用，所以许多程序员耗费了大量时间写代码执行这些操作。历史上，这种代码很难编写，相当繁琐，还容易出错。开发人员需要克服的难题包括通信协议、客户端/服务器数据类型不匹配(比如低位优先/高位优先[①]问题)、错误处理、一个对象引用了其他对象、**in** 和 **out** 参数以及由结构构成的数组等。

让人高兴的是，.NET Framework 内建了出色的序列化和反序列化支持。上述所有难题都迎刃而解，而且.NET Framework 是在后台悄悄帮你解决的。开发者现在只需负责序列化之前和反序列化之后的对象处理，中间过程由.NET Framework 负责。

本章解释了.NET Framework 如何公开它的序列化和序列化服务。对于几乎所有数据类型，这些服务的默认行为已经足够。也就是说，几乎不需要做任何工作就可以使自己的类型“可序列化”。但对于少量类型，序列化服务的默认行为是不够的。幸好，序列化服务的扩展性极佳，本章将解释如何利用这些扩展性机制，在序列化或反序列化对象时采取一些相当强大的操作。例如，本章演示了如何将对象的“版本 1”序列化到磁盘文件，一年后把它反序列化成“版本 2”的对象。



注意 本章重点在于 CLR 的运行时序列化技术。这种技术对 CLR 数据类型有很深刻的理解，能将对象的所有公共、受保护、内部甚至私有字段序列化到压缩的二进制流中，从而获得很好的性能。要把 CLR 数据类型序列化成 XML 流，请参见 **System.Runtime.Serialization.NetDataContractSerializer** 类。.NET Framework 还提供了其他序列化技术，它们主要是为 CLR 数据类型和非 CLR 数据类型之间的互操作而设计的。这些序列化技术用的是 **System.Xml.Serialization.XmlSerializer** 类和 **System.Runtime.Serialization.DataContractSerializer** 类。

24.1 序列化/反序列化快速入门

下面先来看一些代码：

```
using System;
using System.Collections.Generic;
using System.IO;
using System.Runtime.Serialization.Formatters.Binary;

internal static class QuickStart {
   public static void Main() {
```

① little-endian/big-endian，也译成小端和大端。——译注

```
        // 创建对象图以便把它们序列化到流中
        var objectGraph = new List<String> { "Jeff", "Kristin", "Aidan", "Grant" };
        Stream stream = SerializeToMemory(objectGraph);

        // 为了演示，将一切都重置
        stream.Position = 0;
        objectGraph = null;

        // 反序列化对象，证明它能工作
        objectGraph = (List<String>) DeserializeFromMemory(stream);
        foreach (var s in objectGraph) Console.WriteLine(s);
    }

    private static MemoryStream SerializeToMemory(Object objectGraph) {
        // 构造流来容纳序列化的对象
        MemoryStream stream = new MemoryStream();

        // 构造序列化格式化器来执行所有真正的工作
        BinaryFormatter formatter = new BinaryFormatter();

        // 告诉格式化器将对象序列化到流中
        formatter.Serialize(stream, objectGraph);

        // 将序列化好的对象流返回给调用者
        return stream;
    }

    private static Object DeserializeFromMemory(Stream stream) {
        // 构造序列化格式化器来做所有真正的工作
        BinaryFormatter formatter = new BinaryFormatter();

        // 告诉格式化器从流中反序列化对象
        return formatter.Deserialize(stream);
    }
}
```

一切似乎都很简单！**SerializeToMemory** 方法构造一个 **System.IO.MemoryStream** 对象。这个对象标明要将序列化好的字节块放到哪里。然后，方法构造一个 **BinaryFormatter** 对象(在 **System.Runtime.Serialization.Formatters.Binary** 命名空间中定义)。格式化器是实现了 **System.Runtime.Serialization.IFormatter** 接口的类型，它知道如何序列化和反序列化对象图。FCL 提供了两个格式化器：**BinaryFormatter**(本例用的就是它)和 **SoapFormatter**(在 **System.Runtime.Serialization.Formatters.Soap** 命名空间中定义，在 **System.Runtime.Serialization.Formatters.Soap.dll** 程序集中实现)。



> **注意**　从.NET Framework 3.5 开始便废了 **SoapFormatter** 类，不要在生产代码中使用它。但在调试序列化代码时，它仍有一定用处，因为它能生成便于阅读的 XML 文本。要在生产代码中使用 XML 序列化和反序列化，请参见 **XmlSerializer** 和 **DataContractSerializer** 类。

序列化对象图只需调用格式化器的 **Serialize** 方法，并向它传递两样东西：对流对象的引用，以及对想要序列化的对象图的引用。流对象标识了序列化好的字节应放到哪里，它可以是从 **System.IO.Stream** 抽象基类派生的任何类型的对象。也就是说，对象图可序列化成一个 **MemoryStream**，**FileStream** 或者 **NetworkStream** 等。

Serialize 的第二个参数是一个对象引用。这个对象可以是任何东西，可以是一个 **Int32**，**String**，**DateTime**，**Exception**，**List<String>** 或者 **Dictionary<Int32, DateTime>** 等。**objectGraph** 参数引用的对象可引用其他对象。例如，**objectGraph** 可引用一个集合，而这个集合引用了一组对象。这些对象还可继续引用其他对象。调用格式化器的 **Serialize** 方法时，对象图中的所有对象都被序列化到流中。

格式化器参考对每个对象的类型进行描述的元数据，从而了解如何序列化完整的对象图。序列化时，**Serialize** 方法利用反射来查看每个对象的类型中都有哪些实例字段。在这些字段中，任何一个引用了其他对象，格式化器的 **Serialize** 方法就知道那些对象也要进行序列化。

格式化器的算法非常智能。它们知道如何确保对象图中的每个对象都只序列化一次。换言之，如果对象图中的两个对象相互引用，格式化器会检测到这一点，每个对象都只序列化一次，避免发生死循环。

在上述代码的 **SerializeToMemory** 方法中，当格式化器的 **Serialize** 方法返回后，**MemoryStream** 直接返回给调用者。应用程序可以按照自己希望的任何方式利用这个字节数组的内容。例如，可以把它保存到文件中、复制到剪贴板或者通过网络发送等。

DeserializeFromStream 方法将流反序列化为对象图。该方法比用于序列化对象图的方法还要简单。在代码中，我构造了一个 **BinaryFormatter**，然后调用它的 **Deserialize** 方法。这个方法获取流作为参数，返回对反序列化好的对象图中的根对象的一个引用。

在内部，格式化器的 **Deserialize** 方法检查流的内容，构造流中所有对象的实例，并初始化所有这些对象中的字段，使它们具有与当初序列化时相同的值。通常要将 **Deserialize** 方法返回的对象引用转型为应用程序期待的类型。



注意 下面是一个有趣而实用的方法，它利用序列化创建对象的深拷贝(或者说克隆体)：

```
private static Object DeepClone(Object original) {
   // 构造临时内存流
   using (MemoryStream stream = new MemoryStream()) {

      // 构造序列化格式化器来执行所有实际的工作
      BinaryFormatter formatter = new BinaryFormatter();

      // 这一行在本章 24.6 节“流上下文”解释
      formatter.Context = new StreamingContext(StreamingContextStates.Clone);

      // 将对象图序列化到内存流中
      formatter.Serialize(stream, original);

      // 反序列化前，定位到内存流的起始位置
      stream.Position = 0;

      // 将对象图反序列化成一组新对象，
      // 向调用者返回对象图(深拷贝)的根
      return formatter.Deserialize(stream);
   }
}
```

有几点需要注意。首先，是由你来保证代码为序列化和反序列化使用相同的格式化器。例如，不要写代码用 **SoapFormatter** 序列化一个对象图，再用 **BinaryFormatter** 反序列化。**Deserialize** 如果解译不了流的内容会抛出 **System.Runtime.Serialization.SerializationException** 异常。

其次，可将多个对象图序列化到一个流中，这是很有用的一个操作。例如，假定有以下两个类定义：

```
[Serializable] internal sealed class Customer   { /* ... */ }
[Serializable] internal sealed class Order      { /* ... */ }
```

然后，在应用程序的主要类中定义了以下静态字段：

```
private static List<Customer> s_customers       = new List<Customer>();
private static List<Order>    s_pendingOrders   = new List<Order>();
private static List<Order>    s_processedOrders = new List<Order>();
```

现在，可利用如下所示的方法将应用程序的状态序列化到单个流中：

```
private static void SaveApplicationState(Stream stream) {
   // 构造序列化格式化器来执行所有实际的工作
   BinaryFormatter formatter = new BinaryFormatter();

   // 序列化我们的应用程序的完整状态
   formatter.Serialize(stream, s_customers);
   formatter.Serialize(stream, s_pendingOrders);
   formatter.Serialize(stream, s_processedOrders);
}
```

要重新构建应用程序的状态，可以使用如下所示的一个方法反序列化状态：

```
private static void RestoreApplicationState(Stream stream) {
   // 构造序列化格式化器来执行所有实际的工作
   BinaryFormatter formatter = new BinaryFormatter();

   // 反序列化应用程序的完整状态(和序列化时的顺序一样)
   s_customers = (List<Customer>)      formatter.Deserialize(stream);
   s_pendingOrders = (List<Order>)     formatter.Deserialize(stream);
   s_processedOrders = (List<Order>)   formatter.Deserialize(stream);
}
```

最后一个注意事项与程序集有关。序列化对象时，类型的全名和类型定义程序集的全名会被写入流。**BinaryFormatter** 默认输出程序集的完整标识，其中包括程序集的文件名(无扩展名)、版本号、语言文化以及公钥信息。反序列化对象时，格式化器首先获取程序集标识信息，并通过调用 **System.Reflection.Assembly** 的 **Load** 方法(参见 23.1 节“程序集加载”)，确保程序集已加载到正在执行的 AppDomain 中。

程序集加载好之后，格式化器在程序集中查找与要反序列化的对象匹配的类型。找不到匹配类型就抛出异常，不再对更多的对象进行反序列化。找到匹配的类型，就创建类型的实例，并用流中包含的值对其字段进行初始化。如果类型中的字段与流中读取的字段名不完全匹配，就抛出 **SerializationException** 异常，不再对更多的对象进行反序列化。本章以后会讨论一些高级机制，它们允许你覆盖某些行为。

本节讲述了序列化和反序列化对象图的基础知识。之后的小节将讨论如何定义自己的可序列化类型。还讨论了如何利用一些机制对序列化和反序列化进行更好的控制。



重要提示 有的可扩展应用程序使用 **Assembly.LoadFrom** 加载程序集，然后根据加载的程序集中定义的类型来构造对象。这些对象序列化到流中是没有问题的。但在反序列化时，格式化器会调用 **Assembly** 的 **Load** 方法(而非 **LoadFrom** 方法)来加载程序集。大多数情况下，CLR 都将无法定位程序集文件，从而造成 **SerializationException** 异常。许多开发人员对这个结果深感不解。序列化都能正确进行，他们当然预期反序列化也是正确的。

如果应用程序使用 **Assembly.LoadFrom** 加载程序集，再对程序集中定义的类型进行序列化，那么在调用格式化器的 **Deserialize** 方法之前，我建议你实现一个方法，它的签名要匹配 **System.ResolveEventHandler** 委托，并向 **System.AppDomain** 的 **AssemblyResolve** 事件注册这个方法。(**Deserialize** 方法返回后，马上向事件注销这个方法。)现在，每次格式化器加载一个程序集失败，CLR 都会自动调用你的 **ResolveEventHandler** 方法。加载失败的程序集的标识(identity)会传给这个方法。方法可以从程序集的标识中提取程序集文件名，并用这个名称来构造路径，使应用程序知道去哪里寻找文件。然后，方法可调用 **Assembly.LoadFrom** 加载程序集，最后返回对结果程序集的引用。

24.2 使类型可序列化

设计类型时，设计人员必须郑重地决定是否允许类型的实例序列化。类型默认是不可序列化的。例如，以下代码可能不会像你希望的那样工作：

```
internal struct Point { public Int32 x, y; }

private static void OptInSerialization() {
   Point pt = new Point { x = 1, y = 2 };
   using (var stream = new MemoryStream()) {
      new BinaryFormatter().Serialize(stream, pt); // 抛出SerializationException
   }
}
```

格式化器的 **Serialize** 方法会抛出 **System.Runtime.Serialization.SerializationException** 异常。问题在于，**Point** 类型的开发人员没有显式地指出 **Point** 对象可以序列化。为了解决这个问题，开发者必须像下面这样向类型应用定制特性 **System.SerializableAttribute**(注意该特性在 **System** 而不是 **System.Runtime.Serialization** 命名空间中定义)。

```
[Serializable]
internal struct Point { public Int32 x, y; }
```

重新生成并运行，就会像预期的那样工作，**Point** 对象会顺利序列化到流中。序列化对象图时，格式化器会确认每个对象的类型都是可序列化的。任何对象不可序列化，格式化器的 **Serialize** 方法都会抛出 **SerializationException** 异常。



注意　序列化对象图时，也许有的对象的类型能序列化，有的不能。考虑到性能，在序列化之前，格式化器不会验证对象图中的所有对象都能序列化。所以，序列化对象图时，在抛出 **SerializationException** 异常之前，完全有可能已经有一部分对象序列化到流中。如果发生这种情况，流中就会包含已损坏的数据。序列化对象图时，如果你认为也许有一些对象不可序列化，那么写的代码就应该能得体地从这种情况中恢复。一个方案是先将对象序列化到一个 **MemoryStream** 中。然后，如果所有对象都成功序列化，就可以将 **MemoryStream** 中的字节复制到你真正希望的目标流中(比如文件和网络)。

SerializableAttribute 这个定制特性只能应用于引用类型(**class**)、值类型(**struct**)、枚举类型(**enum**)和委托类型(**delegate**)。注意，枚举和委托类型总是可序列化的，所以不必显式应用 **SerializableAttribute** 特性。除此之外，**SerializableAttribute** 特性不会被派生类型继承。所以，给定以下两个类型定义，那么 **Person** 对象可序列化，**Employee** 对象则不可：

```
[Serializable]
internal class Person { ... }

internal class Employee : Person { ... }
```

解决方案是也向 **Employee** 类型应用 **SerializableAttribute** 特性：

```
[Serializable]
internal class Person { ... }

[Serializable]
internal class Employee : Person { ... }
```

上述问题很容易修正，反之则不然。如果基类型没有应用 **SerializableAttribute** 特性，那么很难想象如何从它派生出可序列化的类型。但这样设计是有原因的：如果基类型不允许它的实例序列化，它的字段就不能序列化，因为基对象实际是派生对象的一部分。这正是为什么 **System.Object** 已经很体贴地应用了 **SerializableAttribute** 特性的原因。



注意　一般建议将你定义的大多数类型都设置成可序列化。毕竟，这样能为类型的用户提供很大的灵活性。但必须注意的是，序列化会读取对象的所有字段，不管这些字段声明为 **public**，**protected**，**internal** 还是 **private**。如果类型的实例要包含敏感或安全数据(比如密码)，或者数据在转移之后便没有含义或者没有值，就不应使类型变得可序列化。

如果使用的类型不是为序列化而设计的，而且手上没有类型的源代码，无法从源头添加序列化支持，也不必气馁。在本章最后的 24.9 节“反序列化对象时重写程序集和/或类型”中，我会解释如何使任何不可序列化的类型变得可序列化。

24.3 控制序列化和反序列化

将 **SerializableAttribute** 定制特性应用于类型，所有实例字段(**public**，**private** 和 **protected** 等)都会被序列化[①]。但类型可能定义了一些不应序列化的实例字段。一般有两个原因造成我们不想序列化部分实例字段。

- 字段含有反序列化后变得无效的信息。例如，假定对象包含 Windows 内核对象(如文件、进程、线程、互斥体、事件、信号量等)的句柄，那么在反序列化到另一个进程或另一台机器之后，就会失去意义。因为 Windows 内核对象是跟进程相关的值。
- 字段含有很容易计算的信息。这时要选出那些无须序列化的字段，减少需要传输的数据，增强应用程序的性能。

以下代码使用 **System.NonSerializedAttribute** 定制特性指出类型中不应序列化的字段。注意，该特性也在 **System**(而非 **System.Runtime.Serialization**)命名空间中定义。

```
[Serializable]
internal class Circle {
   private Double m_radius; // 半径

   [NonSerialized]
   private Double m_area; // 面积

   public Circle(Double radius) {
      m_radius = radius;
      m_area = Math.PI * m_radius * m_radius;
   }

   ...
}
```

在上述代码中，**Circle** 的对象可以序列化。但格式化器只会序列化对象的 **m_radius** 字段的值。**m_area** 字段的值不会被序列化，因为该字段已应用了 **NonSerializedAttribute** 特性。注意，该特性只能应用于类型中的字段，而且会被派生类型继承。当然，可向一个类型中的多个字段应用 **NonSerializedAttribute** 特性。

假定代码像下面这样构造了一个 **Circle** 对象：

```
Circle c = new Circle(10);
```

在内部，**m_area** 字段会设置成一个约为 314.159 的值。这个对象序列化时，只有 **m_radius** 字段的值(10)才会写入流。这正是我们希望的，但当流反序列化成 **Circle** 对象时，就会遇到一个问题。反序列化时，**Circle** 对象的 **m_radius** 字段会被设为 10，但它的 **m_area** 字段会被初始化成 0——而不是 314.159！

---

① 在标记了**[Serializable]**特性的类型中，不要用 C#的“自动实现的属性”功能来定义属性。这是由于字段名是由编译器自动生成的，而生成的名称每次重新编译代码时都不同。这会阻止类型被反序列化。详情参见 10.1.1 节“自动实现的属性”。

以下代码演示了如何修改 **Circle** 类型来修正这个问题：

```
[Serializable]
internal class Circle {
   private Double m_radius;  // 半径

   [NonSerialized]
   private Double m_area;  // 面积

   public Circle(Double radius) {
      m_radius = radius;
      m_area = Math.PI * m_radius * m_radius;
   }

   [OnDeserialized]
   private void OnDeserialized(StreamingContext context) {
      m_area = Math.PI * m_radius * m_radius;
   }
}
```

修改过的 **Circle** 类包含一个标记了 **System.Runtime.Serialization.OnDeserializedAttribute** 定制特性的方法[①]。每次反序列化类型的实例，格式化器都会检查类型中是否定义了应用了该特性的方法。如果是，就调用该方法。调用这个方法时，所有可序列化的字段都会被正确设置。在该方法中，可能需要访问这些字段来执行一些额外的工作，从而确保对象的完全反序列化。

在上述 **Circle** 修改版本中，我调用 **OnDeserialized** 方法，使用 **m_radius** 字段来计算圆的面积，并将结果放到 **m_area** 字段中。这样 **m_area** 就有了我们希望的值(314.159)。

除了 **OnDeserializedAttribute** 这个定制特性，**System.Runtime.Serialization** 命名空间还定义了包括 **OnSerializingAttribute**，**OnSerializedAttribute** 和 **OnDeserializingAttribute** 在内的其他定制特性。可将它们应用于类型中定义的方法，对序列化和反序列化过程进行更多的控制。在下面这个类中，这些特性被应用于不同的方法：

```
[Serializable]
public class MyType {
   Int32 x, y; [NonSerialized] Int32 sum;

   public MyType(Int32 x, Int32 y) {
      this.x = x; this.y = y; sum = x + y;
   }

   [OnDeserializing]
   private void OnDeserializing(StreamingContext context) {
      // 举例：在这个类型的新版本中，为字段设置默认值
   }

   [OnDeserialized]
   private void OnDeserialized(StreamingContext context) {
      // 举例：根据字段值初始化瞬时状态(比如 sum 的值)
```

① 要在对象反序列化时调用一个方法，**System.Runtime.Serialization.OnDeserialized** 定制特性是首选方案，而不是让类型实现 **System.Runtime.Serialization.IDeserializationCallback** 接口的 **OnDeserialization** 方法。

```
        sum = x + y;
    }

    [OnSerializing]
    private void OnSerializing(StreamingContext context) {
        // 举例：在序列化前，修改任何需要修改的状态
    }

    [OnSerialized]
    private void OnSerialized(StreamingContext context) {
        // 举例：在序列化后，恢复任何需要恢复的状态
    }
}
```

使用这 4 个属性中的任何一个时，你定义的方法必须获取一个 **StreamingContext** 参数(在本章后面的 24.6 节“流上下文”中讨论)并返回 **void**。方法名可以是你希望的任何名称。另外，应将方法声明为 **private**，以免它被普通的代码调用；格式化器运行时有充足的安全权限，所以能调用私有方法。



注意 序列化一组对象时，格式化器首先调用对象的标记了 **OnSerializing** 特性的所有方法。接着，它序列化对象的所有字段。最后，调用对象的标记了 **OnSerialized** 特性的所有方法。类似地，反序列化一组对象时，格式化器首先调用对象的标记了 **OnDeserializing** 特性的所有方法。然后，它反序列化对象的所有字段。最后，它调用对象的标记了 **OnDeserialized** 特性的所有方法。

还要注意，在反序列化期间，当格式化器看到类型提供的一个方法标记了 **OnDeserialized** 特性时，格式化器会将这个对象的引用添加到一个内部列表中。所有对象都反序列化之后，格式化器反向遍历列表，调用每个对象的 **OnDeserialized** 方法。调用这个方法后，所有可序列化的字段都会被正确设置，可访问这些字段来执行任何必要的、进一步的工作，从而将对象完整地反序列化。之所以要以相反的顺序调用这些方法，因为这样才能使内层对象先于外层对象完成反序列化。

例如，假定一个集合对象(比如 **Hashtable** 或 **Dictionary**)内部用一个哈希表维护它的数据项列表。集合对象类型可实现一个标记了 **OnDeserialized** 特性的方法。即使集合对象先反序列化(先于它包含的数据项)，它的 **OnDeserialized** 方法也会最后调用(在调用完它的数据项的所有 **OnDeserialized** 方法之后)。这样一来，所有数据项在反序列化后，它们的所有字段都能得到正确的初始化，以便计算出一个好的哈希码值。然后，集合对象创建它的内部哈希桶，并利用数据项的哈希码将数据项放到桶中。本章稍后的 24.5 节“控制序列化/反序列化的数据”会提供一个例子，它展示了 **Dictionary** 类如何利用这个技术。

如果序列化类型的实例，在类型中添加新字段，然后试图反序列化不包含新字段的对象，格式化器会抛出 **SerializationException** 异常，并显示一条消息告诉你流中要反序列化的数

据包含错误的成员数目。这非常不利于版本控制，因为我们经常都要在类型的新版本中添加新字段。幸好，这时可以利用 **System.Runtime.Serialization.OptionalFieldAttribute** 特性。

类型中新增的每个字段都要应用 **OptionalFieldAttribute** 特性。然后，当格式化器看到该特性应用于一个字段时，就不会因为流中的数据不包含这个字段而抛出 **SerializationException**。

24.4　格式化器如何序列化类型实例

本节将深入探讨格式化器如何序列化对象的字段。掌握这些知识后，可以更容易地理解本章后面要解释的一些更高级的序列化和反序列化技术。

为了简化格式化器的操作，FCL 在 **System.Runtime.Serialization** 命名空间提供了一个 **FormatterServices** 类型。该类型只包含静态方法，而且该类型不能实例化。以下步骤描述了格式化器如何自动序列化类型应用了 **SerializableAttribute** 特性的对象。

1. 格式化器调用 **FormatterServices** 的 **GetSerializableMembers** 方法：

   ```
   public static MemberInfo[] GetSerializableMembers(Type type, StreamingContext context);
   ```

   这个方法利用反射获取类型的 **public** 和 **private** 实例字段(标记了 **NonSerializedAttribute** 特性的字段除外)。方法返回由 **MemberInfo** 对象构成的数组，其中每个元素都对应一个可序列化的实例字段。

2. 对象被序列化，**System.Reflection.MemberInfo** 对象数组传给 **FormatterServices** 的静态方法 **GetObjectData**：

   ```
   public static Object[] GetObjectData(Object obj, MemberInfo[] members);
   ```

   这个方法返回一个 **Object** 数组，其中每个元素都标识了被序列化的那个对象中的一个字段的值。这个 **Object** 数组和 **MemberInfo** 数组是并行(parallel)的；换言之，**Object** 数组中的元素 0 是 **MemberInfo** 数组中的元素 0 所标识的那个成员的值。

3. 格式化器将程序集标识和类型的完整名称写入流中。

4. 格式化器然后遍历两个数组中的元素，将每个成员的名称和值写入流中。

以下步骤描述了格式化器如何自动反序列化类型应用了 **SerializableAttribute** 特性的对象。

1. 格式化器从流中读取程序集标识和完整类型名称。如果程序集当前没有加载到 AppDomain 中，就加载它(这一点前面已经讲过了)。如果程序集不能加载，就抛出一个 **SerializationException** 异常，对象不能反序列化。如果程序集已加载，格式化器将程序集标识信息和类型全名传给 **FormatterServices** 的静态方法 **GetTypeFromAssembly**：

   ```
   public static Type GetTypeFromAssembly(Assembly assem, String name);
   ```

   这个方法返回一个 **System.Type** 对象，它代表要反序列化的那个对象的类型。

2. 格式化器调用 **FormatterServices** 的静态方法 **GetUninitializedObject**：

```
public static Object GetUninitializedObject(Type type);
```

这个方法为一个新对象分配内存，但不为对象调用构造器。然而，对象的所有字节都被初始化为成 **null** 或 **0**。

3. 格式化器现在构造并初始化一个 **MemberInfo** 数组，具体做法和前面一样，都是调用 **FormatterServices** 的 **GetSerializableMembers** 方法。这个方法返回序列化好、现在需要反序列化的一组字段。

4. 格式化器根据流中包含的数据创建并初始化一个 **Object** 数组。

5. 将新分配对象、**MemberInfo** 数组以及并行 **Object** 数组(其中包含字段值)的引用传给 **FormatterServices** 的静态方法 **PopulateObjectMembers**：

```
public static Object PopulateObjectMembers(
   Object obj, MemberInfo[] members, Object[] data);
```

这个方法遍历数组，将每个字段初始化成对应的值。到此为止，对象就算是被彻底反序列化了。

24.5 控制序列化/反序列化的数据

本章前面讨论过，控制序列化和反序列化过程的最佳方式就是使用 **OnSerializing**，**OnSerialized**，**OnDeserializing**，**OnDeserialized**，**NonSerialized** 和 **OptionalField** 等特性。然而，在一些极少见的情况下，这些特性不能提供你想要的全部控制。此外，格式化器内部使用的是反射，而反射的速度是比较慢的，这会增大序列化和反序列化对象所花的时间。为了对序列化/反序列化的数据进行完全的控制，并避免使用反射，你的类型可实现 **System.Runtime.Serialization.ISerializable** 接口，它的定义如下：

```
public interface ISerializable {
    void GetObjectData(SerializationInfo info, StreamingContext context);
}
```

这个接口只有一个方法，即 **GetObjectData**。但实现这个接口的大多数类型还实现了一个特殊的构造器，我稍后会详细描述它。



重要提示 **ISerializable** 接口最大的问题在于，一旦类型实现了它，所有派生类型也必须实现它，而且派生类型必须保证调用基类的 **GetObjectData** 方法和特殊构造器。此外，一旦类型实现了该接口，便永远不能删除它，否则会失去与派生类型的兼容性。所以，密封类实现 **ISerializable** 接口是最让人放心的。使用本章前面描述的各种定制特性，**ISerializable** 接口的所有问题都可以避免。



重要提示　**ISerializable** 接口和特殊构造器旨在由格式化器使用。但其他代码可能调用 **GetObjectData** 来返回敏感数据。另外，其他代码可能构造对象，并传入损坏的数据。因此，建议向 **GetObjectData** 方法和特殊构造器应用以下特性：

```
[SecurityPermissionAttribute(SecurityAction.Demand,
SerializationFormatter = true)]
```

格式化器序列化对象图时会检查每个对象。如果发现一个对象的类型实现了 **ISerializable** 接口，就会忽略所有定制特性，改为构造新的 **System.Runtime.Serialization.SerializationInfo** 对象。该对象包含了要为对象序列化的值的集合。

构造 **SerializationInfo** 对象时，格式化器要传递两个参数：**Type** 和 **System.Runtime.Serialization.IFormatterConverter**。**Type** 参数标识要序列化的对象。唯一性地标识一个类型需要两个部分的信息：类型的字符串名称及其程序集标识(包括程序集名、版本、语言文化和公钥)。构造好的 **SerializationInfo** 对象包含类型的全名(通过在内部查询 **Type** 的 **FullName** 属性)，这个字符串会存储到一个私有字段中。如果你想获取类型的全名，可查询 **SerializationInfo** 的 **FullTypeName** 属性。类似地，构造器获取类型的定义程序集(通过在内部查询 **Type** 的 **Module** 属性，再查询 **Module** 的 **Assembly** 属性，再查询 **Assembly** 的 **FullName** 属性)，这个字符串会存储在一个私有字段中。如果你想获取程序集的标识，可查询 **SerializationInfo** 的 **AssemblyName** 属性。



注意　虽然可以设置一个 **SerializationInfo** 的 **FullTypeName** 和 **AssemblyName** 属性，但不建议这样做。如果想更改被序列化的类型，建议调用 **SerializationInfo** 的 **SetType** 方法，传递对目标 **Type** 对象的引用。调用 **SetType** 可确保类型的全名和定义程序集被正确设置。本章后面的 24.7 节“类型序列化为不同类型以及对象反序列化为不同对象”将展示调用 **SetType** 的一个例子。

构造好并初始化好 **SerializationInfo** 对象后，格式化器调用类型的 **GetObjectData** 方法，向它传递对 **SerializationInfo** 对象的引用。**GetObjectData** 方法决定需要哪些信息来序列化对象，并将这些信息添加到 **SerializationInfo** 对象中。**GetObjectData** 调用 **SerializationInfo** 类型提供的 **AddValue** 方法的众多重载版本之一来指定要序列化的信息。针对要添加的每个数据，都要调用一次 **AddValue**。

以下代码展示了 **Dictionary<TKey, TValue>**类型如何实现 **ISerializable** 和 **IDeserializationCallback** 接口来控制其对象的序列化和反序列化。

```
[Serializable]
public class Dictionary<TKey, TValue>: ISerializable, IDeserializationCallback {
    // 私有字段放在这里(未列出)
```

```
private SerializationInfo m_siInfo; // 只用于反序列化

// 用于控制反序列化的特殊构造器(这是 ISerializable 需要的)
[SecurityPermissionAttribute(SecurityAction.Demand, SerializationFormatter = true)]
protected Dictionary(SerializationInfo info, StreamingContext context) {
     // 反序列化期间，为 OnDeserialization 保存 SerializationInfo
     m_siInfo = info;
}

// 用于控制序列化的方法
 [SecurityCritical]
public virtual void GetObjectData(SerializationInfo info, StreamingContext context) {

     info.AddValue("Version", m_version);
     info.AddValue("Comparer", m_comparer, typeof(IEqualityComparer<TKey>));
     info.AddValue("HashSize", (m_ buckets == null) ? 0 : m_buckets.Length);
     if (m_buckets != null) {
          KeyValuePair<TKey, TValue>[] array = new KeyValuePair<TKey, TValue>[Count];
          CopyTo(array, 0);
          info.AddValue("KeyValuePairs", array, typeof(KeyValuePair<TKey, TValue>[]));
     }
}

// 所有 key/value 对象都反序列化好之后调用的方法
public virtual void IDeserializationCallback.OnDeserialization(Object sender) {
     if (m_siInfo == null) return; // 从不设置，直接返回

     Int32 num = m_siInfo.GetInt32("Version");
     Int32 num2 = m_siInfo.GetInt32("HashSize");
     m_comparer = (IEqualityComparer<TKey>)
     m_siInfo.GetValue("Comparer", typeof(IEqualityComparer<TKey>));
if (num2 != 0) {
     m_buckets = new Int32[num2];
     for (Int32 i = 0; i < m_buckets.Length; i++) m_buckets[i] = -1;
     m_entries = new Entry<TKey, TValue>[num2];
     m_freeList = -1;
     KeyValuePair<TKey, TValue>[] pairArray = (KeyValuePair<TKey, TValue>[])
     m_siInfo.GetValue("KeyValuePairs", typeof(KeyValuePair<TKey, TValue>[]));
     if (pairArray == null)
          ThrowHelper.ThrowSerializationException(
               ExceptionResource.Serialization_MissingKeys);

     for (Int32 j = 0; j < pairArray.Length; j++) {
          if (pairArray[j].Key == null)
               ThrowHelper.ThrowSerializationException(
                    ExceptionResource.Serialization_NullKey);

               Insert(pairArray[j].Key, pairArray[j].Value, true);
          }
     } else { m_buckets = null; }
     m_version = num;
     m_siInfo = null;
}
```

每个 **AddValue** 方法都获取一个 **String** 名称和一些数据。数据一般是简单的值类型，比如 **Boolean**，**Char**，**Byte**，**SByte**，**Int16**，**UInt16**，**Int32**，**UInt32**，**Int64**，**UInt64**，**Single**，**Double**，**Decimal** 或者 **DateTime**。然而，还可以在调用 **AddValue** 时向它传递对一个 **Object**(比如一个 **String**)的引用。**GetObjectData** 添加好所有必要的序列化信息之后，会返回至格式化器。



> **注意**　务必调用 **AddValue** 方法的某个重载版本为自己的类型添加序列化信息。如果一个字段的类型实现了 **ISerializable** 接口，就不要在字段上调用 **GetObjectData**。相反，调用 **AddValue** 来添加字段；格式化器会注意到字段的类型实现了 **ISerializable**，会帮你调用 **GetObjectData**。如果自己在字段对象上调用 **GetObjectData**，格式化器便不知道在对流进行反序列化时创建新对象。

现在，格式化器获取已添加到 **SerializationInfo** 对象的所有值，并把它们都序列化到流中。注意，我们还向 **GetObjectData** 方法传递了另一个参数，也就是对一个 **System.Runtime.Serialization.StreamingContext** 对象的引用。大多数类型的 **GetObjectData** 方法都会完全忽略这个参数，所以我现在不准备讨论它。相反，我准备把它放到本章后面的 24.6 节“流上下文”讨论。

知道了如何设置序列化所需的全部信息之后，再来看反序列化。格式化器从流中提取一个对象时，会为新对象分配内存(通过调用 **System.Runtime.Serialize.FormatterServices** 类型的静态 **GetUninitializedObject** 方法)。最初，这个对象的所有字段都设为 **0** 或 **null**。然后，格式化器检查类型是否实现了 **ISerializable** 接口。如果存在这个接口，格式化器就尝试调用一个特殊构造器，它的参数和 **GetObjectData** 方法的完全一致。

如果你的类是密封类，强烈建议将这个特殊构造器声明为 **private**。这样可防止任何代码不慎调用它，从而提升安全性。如果不是密封类，应该将这个特殊构造器声明为 **protected**，确保只有派生类才能调用。注意，无论这个特殊构造器是如何声明的，格式化器都能调用它。

构造器获取一个 **SerializationInfo** 对象引用。在这个 **SerializationInfo** 对象中，包含了对象序列化时添加的所有值。特殊构造器可调用 **GetBoolean**，**GetChar**，**GetByte**，**GetSByte**，**GetInt16**，**GetUInt16**，**GetInt32**，**GetUInt32**，**GetInt64**，**GetUInt64**，**GetSingle**，**GetDouble**，**GetDecimal**，**GetDateTime**，**GetString** 和 **GetValue** 等任何一个方法，向它传递与序列化一个值所用的名称对应的字符串。上述每个方法返回的值再用于初始化新对象的各个字段。

反序列化对象的字段时，应调用和对象序列化时传给 **AddValue** 方法的值的类型匹配的 **Get** 方法。换言之，如果 **GetObjectData** 方法调用 **AddValue** 时传递的是一个 **Int32** 值，那么在反序列化对象时，应该为同一个值调用 **GetInt32** 方法。如果值在流中的类型和你试图获取(Get)的类型不符，格式化器会尝试用一个 **IFormatterConverter** 对象将流中的值转型成你指定的类型。

前面说过，构造 **SerializationInfo** 对象时，要向它传递类型实现了 **IFormatterConverter** 接口的一个对象。由于是格式化器负责构造 **SerializationInfo** 对象，所以要由它选择它想要的 **IFormatterConverter** 类型。Microsoft 的 **BinaryFormatter** 和 **SoapFormatter** 类型总是构造 **System.Runtime.Serialization.FormatterConverter** 类型的实例。Microsoft 的格式化器

没有提供任何方式让你选择不同的 **IFormatterConverter** 类型。

FormatterConverter 类型调用 **System.Convert** 类的各种静态方法在不同的核心类型之间对值进行转换，比如将一个 **Int32** 转换成一个 **Int64**。然而，为了在其他任意类型之间转换一个值，**FormatterConverter** 要调用 **Convert** 的 **ChangeType** 方法将序列化好的(或者原始的)类型转型为一个 **IConvertible** 接口，再调用恰当的接口方法。所以，要允许一个可序列化类型的对象反序列化成一个不同的类型，可考虑让自己的类型实现 **IConvertible** 接口。注意，只有在反序列化对象时调用一个 **Get** 方法，但发现它的类型和流中的值的类型不符时，才会使用 **FormatterConverter** 对象。

特殊构造器也可以不调用上面列出的各个 **Get** 方法，而是调用 **GetEnumerator**。该方法返回一个 **System.Runtime.Serialization.SerializationInfoEnumerator** 对象，可用该对象遍历 **SerializationInfo** 对象中包含的所有值。枚举的每个值都是一个 **System.Runtime.Serialization.SerializationEntry** 对象。

当然，完全可以定义自己的类型，让它从实现了 **ISerializable** 的 **GetObjectData** 方法和特殊构造器类型派生。如果你的类型也实现了 **ISerializable**，那么在你实现的 **GetObjectData** 方法和特殊构造器中，必须调用基类中的同名方法，确保对象能正确序列化和反序列化。这一点务必牢记，否则对象是不能正确序列化和反序列化的。下一节将解释如何正确地定义基类型未实现 **ISerializable** 接口一个 **ISerializable** 类型。

如果你的派生类型中没有任何额外的字段，因而没有特殊的序列化/反序列化需求，就完全不必实现 **ISerializable**。和所有接口成员相似，**GetObjectData** 是 **virtual** 的，调用它可以正确地序列化对象。此外，格式化器将特殊构造器视为“已虚拟化”(virtualized)。换言之，反序列化期间，格式化器会检查要实例化的类型。如果那个类型没有提供特殊构造器，格式化器会扫描基类，直到它找到实现了特殊构造器的一个类。



重要提示 特殊构造器中的代码一般从传给它的 **SerializationInfo** 对象中提取字段。提取字段后，不保证对象已完全反序列化，所以特殊构造器中的代码不应尝试操纵它提取的对象。

如果你的类型必须访问提取的对象中的成员(比如调用方法)，建议你的类型提供一个应用了 **OnDeserialized** 特性的方法，或者让类型实现 **IDeserializationCallback** 接口的 **OnDeserialization** 方法(就像前面的 **Dictionary** 示例中那样)。调用该方法时，所有对象的字段都已设置好。然而，对于多个对象来说，它们的 **OnDeserialized** 或 **OnDeserialization** 方法的调用顺序是没有保障的。所以，虽然字段可能已初始化，但你仍然不知道被引用的对象是否已完全反序列化好(如果那个被引用的对象也提供了一个 **OnDeserialized** 方法或者实现了 **IDeserializationCallback**)。

要实现 ISerializable 但基类型没有实现怎么办？

前面讲过，**ISerializable** 接口的功能非常强大，允许类型完全控制如何对类型的实例进行序列化和反序列化。但这个能力是有代价的：现在，该类型还要负责它的基类型的所有字段的序列化。如果基类型也实现了 **ISerializable** 接口，那么对基类型的字段进行序列化是很容易的。调用基类型的 **GetObjectData** 方法即可。

总有一天需要定义类型来控制它的序列化，但发现它的基类没有实现 **ISerializable** 接口。在这种情况下，派生类必须手动序列化基类的字段，具体的做法是获取它们的值，并把这些值添加到 **SerializationInfo** 集合中。然后，在你的特殊构造器中，还必须从集合中取出值，并以某种方式设置基类的字段。如果基类的字段是 **public** 或 **protected** 的，那么一切都很容易实现。如果是 **private** 字段，就很难或者根本不可能实现。

以下代码演示了如何正确实现 **ISerializable** 的 **GetObjectData** 方法和它的隐含的构造器，使基类的字段能被序列化：

```
[Serializable]
internal class Base {
    protected String m_name = "Jeff";
    public Base() { /* Make the type instantiable */ }
}

[Serializable]
internal class Derived : Base, ISerializable {
    private DateTime m_date = DateTime.Now;
    public Derived() { /* Make the type instantiable*/ }

    // 如果这个构造器不存在，便会引发一个 SerializationException 异常、
    // 如果这个类不是密封类，这个构造器就应该是 protected 的
     [SecurityPermissionAttribute(SecurityAction.Demand, SerializationFormatter = true)]
    private Derived(SerializationInfo info, StreamingContext context) {
        // 为我们的类和基类获取可序列化的成员集合
        Type baseType = this.GetType().BaseType;
        MemberInfo[] mi = FormatterServices.GetSerializableMembers(baseType, context);

        // 从 info 对象反序列化基类的字段
        for (Int32 i = 0; i < mi.Length; i++) {
            // 获取字段，并把它设为反序列化好的值
            FieldInfo fi = (FieldInfo)mi[i];
            fi.SetValue(this, info.GetValue(baseType.FullName + "+" + fi.Name, fi.FieldType));
        }

        // 反序列化为这个类序列化的值
        m_date = info.GetDateTime("Date");
    }

     [SecurityPermissionAttribute(SecurityAction.Demand, SerializationFormatter = true)]
    public virtual void GetObjectData(SerializationInfo info, StreamingContext context) {
        // 为这个类序列化希望的值
        info.AddValue("Date", m_date);

        // 获取我们的类和基类的可序列化的成员
        Type baseType = this.GetType().BaseType;
        MemberInfo[] mi = FormatterServices.GetSerializableMembers(baseType, context);
```

```
            // 将基类的字段序列化到 info 对象中
            for (Int32 i = 0; i < mi.Length; i++) {
                // 为字段名附加基类型全名作为前缀
                info.AddValue(baseType.FullName + "+" + mi[i].Name,
                    ((FieldInfo)mi[i]).GetValue(this));
            }
        }

        public override String ToString() {
            return String.Format("Name={0}, Date={1}", m_name, m_date);
        }
    }
```

上述代码有一个名为 **Base** 的基类，它只用 **SerializableAttribute** 定制特性进行了标识。从 **Base** 派生的是 **Derived** 类，它除了也用 **SerializableAttribute** 特性进行标识，还实现了 **ISerializable** 接口。为了使局面变得更有趣，两个类都定义了名为 **m_name** 的一个 **String** 字段。调用 **SerializationInfo** 的 **AddValue** 方法时不能添加多个同名的值。在上述代码中，解决这个问题的方案是在字段名前附加类名作为前缀，从而对每个字段进行标识。例如，当 **GetObjectData** 方法调用 **AddValue** 来序列化 **Base** 的 **m_name** 字段时，写入的值的名称是“**Base+m_name**”。

24.6 流上下文

前面讲过，一组序列化好的对象可以有许多目的地：同一个进程、同一台机器上的不同进程、不同机器上的不同进程等。在一些比较少见的情况下，一个对象可能想知道它要在什么地方反序列化，从而以不同的方式生成它的状态。例如，如果对象中包装了 Windows 信号量(semaphore)对象，如果它知道要反序列化到同一个进程中，就可决定对它的内核句柄(kernel handle)进行序列化，这是因为内核句柄在一个进程中有效。但如果要反序列化到同一台计算机的不同进程中，就可决定对信号量的字符串名称进行序列化。最后，如果要反序列化到不同计算机上的进程，就可决定抛出异常，因为信号量只在一台机器内有效。

本章提到的大量方法都接受一个 **StreamingContext**(流上下文)。**StreamingContext** 结构是一个非常简单的值类型，它只提供了两个公共只读属性，如表 24-1 所示。

表 24-1 StreamingContext 的公共只读属性

成员名称	成员类型	说明
State	**StreamingContextStates**	一组位标志(bit flag)，指定要序列化/反序列化的对象的来源或目的地
Context	**Object**	一个对象引用，对象中包含用户希望的任何上下文信息

接受一个 **StreamingContext** 结构的方法能检查 **State** 属性的位标志，判断要序列化/反序列化的对象的来源或目的地。表 24-2 展示了可能的位标志值。

表 24-2　StreamingContextStates 的标志

标志名称	标志值	说明
CrossProcess	0x0001	来源或目的地是同一台机器的不同进程
CrossMachines	0x0002	来源或目的地在不同机器上
File	0x0004	来源或目的地是文件。不保证反序列化数据的是同一个进程
Persistence	0x0008	来源或目的地是存储(store)，比如数据库或文件。不保证反序列化数据的是同一个进程
Remoting	0x0010	来源或目的地是远程的未知位置。这个位置可能在(也可能不在)同一台机器上
Other	0x0020	来源或目的地未知
Clone	0x0040	对象图被克隆。序列化代码可认为是由同一进程对数据进行反序列化，所以可安全地访问句柄或其他非托管资源
CrossAppDomain	0x0080	来源或目的地是不同的 AppDomain
All	0x00FF	来源或目的地可能是上述任何一个上下文。这是默认设定

知道如何获取这些信息后，接着讨论如何设置。**IFormatter** 接口(同时由 **BinaryFormatter** 和 **SoapFormatter** 类型实现)定义了 **StreamingContext** 类型的可读/可写属性 **Context**。构造格式化器时，格式化器会初始化它的 **Context** 属性，将 **StreamingContextStates** 设为 **All**，将对额外状态对象的引用设为 **null**。

格式化器构造好之后，就可以使用任何 **StreamingContextStates** 位标志来构造一个 **StreamingContext** 结构，并可选择传递一个对象引用(对象中包含你需要的任何额外的上下文信息)。现在，在调用格式化器的 **Serialize** 或 **Deserialize** 方法之前，你只需要将格式化器的 **Context** 属性设为这个新的 **StreamingContext** 对象。在本章前面的 24.1 节“序列化/反序列化快速入门”中，已通过 **DeepClone** 方法演示了如何告诉格式化器，对一个对象图进行序列化/反序列化的唯一目的就是克隆对象图中的所有对象。

24.7　类型序列化为不同类型以及对象反序列化为不同对象

.NET Framework 的序列化架构是相当全面的，本节要讨论如何设计类型将自己序列化或反序列化成不同的类型或对象。下面列举了一些有趣的例子。

- 有的类型(比如 **System.DBNull** 和 **System.Reflection.Missing**)设计成每个 AppDomain 一个实例。经常将这些类型称为**单实例**(singleton)类型。给定一个 **DBNull** 对象引用，序列化和反序列化它不应造成在 AppDomain 中新建一个 **DBNull** 对象。反序列化后，返回的引用应指向 AppDomain 中现有的 **DBNull** 对象。

- 对于某些类型(例如 **System.Type** 和 **System.Reflection.Assembly**，以及其他反射类型，例如 **MemberInfo**)，每个类型、程序集或者成员等都只能有一个实例。例如，假定一个数组中的每个元素都引用一个 **MemberInfo** 对象，其中 5 个元素引用的都是一个

MemberInfo 对象。序列化和反序列化这个数组后，那 5 个元素引用的应该还是一个 **MemberInfo** 对象(而不是分别引用 5 个不同的对象)。除此之外，这些元素引用的 **MemberInfo** 对象还必须实际对应于 AppDomain 中的一个特定成员。轮询数据库连接对象或者其他任何类型的对象时，这个功能也是很好用的。

- 对于远程控制的对象，CLR 序列化与服务器对象有关的信息。在客户端上反序列化时，会造成 CLR 创建一个代理对象。这个代理对象的类型有别于服务器对象的类型，但这对于客户端代码来说是透明的(客户端不需要关心这个问题)。客户端直接在代理对象上调用实例方法。然后，代理代码内部会调用远程发送给服务器，由后者实际执行请求的操作。

下面来看看一些示例代码，它们展示了如何正确地序列化和反序列化单实例类型：

```
// 每个 AppDomain 应该只有这个类型的一个实例
[Serializable]
public sealed class Singleton : ISerializable {
    // 这是该类型的一个实例
    private static readonly Singleton s_theOneObject = new Singleton();

    // 这些是实例字段
    public String Name = "Jeff";
    public DateTime Date = DateTime.Now;

    // 私有构造器，允许这个类型构造单实例
    private Singleton() { }

    // 该方法返回对单实例的引用
    public static Singleton GetSingleton() { return s_theOneObject; }

    // 序列化一个 Singleton 时调用的方法
    // 我建议在这里使用一个显式接口方法实现(EIMI)
     [SecurityPermissionAttribute(SecurityAction.Demand, SerializationFormatter = true)]
    void ISerializable.GetObjectData(SerializationInfo info, StreamingContext context) {
        info.SetType(typeof(SingletonSerializationHelper));
        // 不需要添加其他值
    }

     [Serializable]
    private sealed class SingletonSerializationHelper : IObjectReference {
        // 这个方法在对象(它没有字段)反序列化之后调用
        public Object GetRealObject(StreamingContext context) {
            return Singleton.GetSingleton();
        }
    }

    // 注意：特殊构造器是不必要的，因为它永远不会调用
}
```

Singleton 类所代表的类型规定每个 AppDomain 只能存在它的一个实例。以下代码测试 **Singleton** 的序列化和反序列化代码，保证 AppDomain 中只有 **Singleton** 类型的一个实例：

```
private static void SingletonSerializationTest() {
   // 创建数组，其中多个元素引用一个 Singleton 对象
   Singleton[] a1 = { Singleton.GetSingleton(), Singleton.GetSingleton() };
   Console.WriteLine("Do both elements refer to the same object? "
     + (a1[0] == a1[1])); // "True"

   using (var stream = new MemoryStream()) {
```

```
        BinaryFormatter formatter = new BinaryFormatter();

        // 先序列化再反序列化数组元素
        formatter.Serialize(stream, a1);
        stream.Position = 0;
        Singleton[] a2 = (Singleton[])formatter.Deserialize(stream);

        // 证明它的工作和预期的一样：
        Console.WriteLine("Do both elements refer to the same object? "
           + (a2[0] == a2[1])); // "True"
        Console.WriteLine("Do all elements refer to the same object? "
           + (a1[0] == a2[0])); // "True"
    }
}
```

现在，我们通过分析代码来理解所发生的事情。**Singleton** 类型加载到 AppDomain 中时，CLR 调用它的静态构造器来构造一个 **Singleton** 对象，并将对它的引用保存到静态字段 **s_theOneObject** 中。**Singleton** 类没有提供任何公共构造器，这防止了其他任何代码构造该类的其他实例。

在 **SingletonSerializationTest** 中，我们创建包含两个元素的一个数组；每个元素都引用 **Singleton** 对象。为了初始化两个元素，我们调用 **Singleton** 的静态 **GetSingleton** 方法。这个方法返回对一个 **Singleton** 对象的引用。对 **Console** 的 **WriteLine** 方法的第一个调用显示“True”，证明两个数组元素引用同一个对象。

现在，**SingletonSerializationTest** 调用格式化器的 **Serialize** 方法序列化数组及其元素。序列化第一个 **Singleton** 时，格式化器检测到 **Singleton** 类型实现了 **ISerializable** 接口，并调用 **GetObjectData** 方法。这个方法调用 **SetType**，向它传递 **SingletonSerializationHelper** 类型，告诉格式化器将 **Singleton** 对象序列化成一个 **SingletonSerializationHelper** 对象。由于 **AddValue** 没有调用，所以没有额外的字段信息写入流。由于格式化器自动检测出两个数组元素都引用一个对象，所以格式化器只序列化一个对象。

序列化数组之后，**SingletonSerializationTest** 调用格式化器的 **Deserialize** 方法。对流进行反序列化时，格式化器尝试反序列化一个 **SingletonSerializationHelper** 对象，这是格式化器之前被“欺骗”所序列化的东西。(事实上，这正是为什么 **Singleton** 类不提供特殊构造器的原因；实现 **ISerializable** 接口时通常都要求提供这个特殊构造器。)构造好 **SingletonSerializationHelper** 对象后，格式化器发现这个类型实现了 **System.Runtime.Serialization.IObjectReference** 接口。这个接口在 FCL 中是像下面这样定义的：

```
public interface IObjectReference {
    Object GetRealObject(StreamingContext context);
}
```

如果类型实现了这个接口，格式化器会调用 **GetRealObject** 方法。这个方法返回在对象反序列化好之后你真正想引用的对象。在我的例子中，**SingletonSerializationHelper** 类型让 **GetRealObject** 返回对 AppDomain 中已经存在的 **Singleton** 对象的一个引用。所以，当格式化器的 **Deserialize** 方法返回时，**a2** 数组包含两个元素，两者都引用 AppDomain 的 **Singleton** 对象。用于帮助进行反序列化的 **SingletonSerializationHelper** 对象立即变得“不可达”了[①]，

① 没有谁引用它了。——译注

将来会被垃圾回收。

对 **WriteLine** 的第二个调用显示“True”，证明 **a2** 数组的两个元素都引用同一个对象。第三个(也是最后一个)**WriteLine** 调用也显示“True”，证明两个数组中的元素引用的是同一个对象。

24.8 序列化代理

前面讨论了如何修改一个类型的实现，控制该类型如何对它本身的实例进行序列化和反序列化。然而，格式化器还允许不是“类型实现的一部分”的代码重写该类型“序列化和反序列化其对象”的方式。应用程序代码之所以要重写(覆盖)类型的行为，主要是出于两方面的考虑。

- 允许开发人员序列化最初没有设计成要序列化的类型。
- 允许开发人员提供一种方式将类型的一个版本映射到类型的一个不同的版本

简单地说，为了使这个机制工作起来，首先要定义一个“代理类型”(surrogate type)，它接管对现有类型进行序列化和反序列化的行动。然后，向格式化器登记该代理类型的实例，告诉格式化器代理类型要作用于现有的哪个类型。一旦格式化器要对现有类型的实例进行序列化或反序列化，就调用由你的代理对象定义的方法。下面用一个例子演示这一切是如何工作的。

序列化代理类型必须实现 **System.Runtime.Serialization.ISerializationSurrogate** 接口，它在 FCL 中像下面这样定义：

```
public interface ISerializationSurrogate {
    void GetObjectData(Object obj, SerializationInfo info, StreamingContext context);

    Object SetObjectData(Object obj, SerializationInfo info, StreamingContext context,
        ISurrogateSelector selector);
}
```

让我们分析使用了该接口的一个例子。假定程序含有一些 **DateTime** 对象，其中包含用户计算机的本地值。如果想把 **DateTime** 对象序列化到流中，同时希望值用国际标准时间(世界时)序列化，那么应该如何操作呢？这样一来，就可以将数据通过网络流发送给世界上其他地方的另一台机器，使 **DateTime** 值保持正确。虽然不能修改 FCL 自带的 **DateTime** 类型，但可以定义自己的序列化代理类，它能控制 **DateTime** 对象的序列化和反序列化方式。下面展示了如何定义代理类：

```
internal sealed class UniversalToLocalTimeSerializationSurrogate : ISerializationSurrogate {
    public void GetObjectData(Object obj, SerializationInfo info, StreamingContext context) {
        // 将DateTime从本地时间转换成UTC
        info.AddValue("Date", ((DateTime)obj).ToUniversalTime().ToString("u"));
    }

    public Object SetObjectData(Object obj, SerializationInfo info, StreamingContext context,
        ISurrogateSelector selector) {
        // 将DateTime从UTC转换成本地时间
```

```
            return DateTime.ParseExact(info.GetString("Date"), "u", null).ToLocalTime();
        }
    }
```

GetObjectData 方法在这里的工作方式与 **ISerializable** 接口的 **GetObjectData** 方法差不多。唯一的区别在于，**ISerializationSurrogate** 的 **GetObjectData** 方法要获取一个额外的参数——对要序列化的“真实”对象的引用。在上述 **GetObjectData** 方法中，这个对象转型为 **DateTime**，值从本地时间转换为世界时，并将一个字符串(使用通用完整日期/时间模式来格式化)添加到 **SerializationInfo** 集合。

SetObjectData 方法用于反序列化一个 **DateTime** 对象。调用这个方法时要向它传递一个 **SerializationInfo** 对象引用。**SetObjectData** 从这个集合中获取字符串形式的日期，把它解析成通用完整日期/时间模式的字符串，然后将结果 **DateTime** 对象从世界时转换成计算机的本地时间。

传给 **SetObjectData** 第一个参数的 **Object** 有点儿奇怪。在调用 **SetObjectData** 之前，格式化器分配(通过 **FormatterServices** 的静态方法 **GetUninitializedObject**)要代理的那个类型的实例。实例的字段全是 **0/null**，而且没有在对象上调用构造器。**SetObjectData** 内部的代码为了初始化这个实例的字段，可以使用传入的 **SerializationInfo** 中的值，并让 **SetObjectData** 返回 **null**。另外，**SetObjectData** 可以创建一个完全不同的对象，甚至创建不同类型的对象，并返回对新对象的引用。在这种情况下，格式化器会忽略对传给 **SetObjectData** 的对象的任何更改。

在我的例子中，**UniversalToLocalTimeSerializationSurrogate** 类扮演了 **DateTime** 类型的代理的角色。**DateTime** 是值类型，所以 **obj** 参数引用一个 **DateTime** 的已装箱实例。大多数值类型中的字段都无法更改(值类型本来就设计成“不可变”)，所以我的 **SetObjectData** 方法会忽略 **obj** 参数，并返回一个新的 **DateTime** 对象，其中已装好了期望的值。

此时，你肯定会问，序列化/反序列化一个 **DateTime** 对象时，格式化器怎么知道要用这个 **ISerializationSurrogate** 类型呢？以下代码对 **UniversalToLocalTimeSerializationSurrogate** 类进行了测试：

```
private static void SerializationSurrogateDemo() {
    using (var stream = new MemoryStream()) {
        // 1. 构造所需的格式化器
        IFormatter formatter = new SoapFormatter();

        // 2. 构造一个 SurrogateSelector(代理选择器)对象
        SurrogateSelector ss = new SurrogateSelector();

        // 3. 告诉代理选择器为 DateTime 对象使用我们的代理
        ss.AddSurrogate(typeof(DateTime), formatter.Context,
            new UniversalToLocalTimeSerializationSurrogate());
        // 注意: AddSurrogate 可多次调用来登记多个代理

        // 4. 告诉格式化器使用代理选择器
        formatter.SurrogateSelector = ss;

        // 创建一个 DateTime 来代表机器上的本地时间，并序列化它
        DateTime localTimeBeforeSerialize = DateTime.Now;
        formatter.Serialize(stream, localTimeBeforeSerialize);
```

```
            // stream将Universal时间作为一个字符串显示，证明能正常工作
            stream.Position = 0;
            Console.WriteLine(new StreamReader(stream).ReadToEnd());

            // 反序列化Universal时间字符串，并且把它转换成本地DateTime
            stream.Position = 0;
            DateTime localTimeAfterDeserialize = (DateTime)formatter.Deserialize(stream);

            // 证明它能正确工作
            Console.WriteLine("LocalTimeBeforeSerialize ={0}", localTimeBeforeSerialize);
            Console.WriteLine("LocalTimeAfterDeserialize={0}", localTimeAfterDeserialize);
        }
}
```

步骤 1 到步骤 4 执行完毕后，格式化器就准备好使用已登记的代理类型。调用格式化器的 **Serialize** 方法时，会在 **SurrogateSelector** 维护的集合(一个哈希表)中查找(要序列化的)每个对象的类型。如果发现一个匹配，就调用 **ISerializationSurrogate** 对象的 **GetObjectData** 方法来获取应该写入流的信息。

格式化器的 **Deserialize** 方法在调用时，会在格式化器的 **SurrogateSelector** 中查找要反序列化的对象的类型。如果发现一个匹配，就调用 **ISerializationSurrogate** 对象的 **SetObjectData** 方法来设置要反序列化的对象中的字段。

SurrogateSelector 对象在内部维护了一个私有哈希表。调用 **AddSurrogate** 时，**Type** 和 **StreamingContext** 构成了哈希表的键(key)，对应的值(value)就是 **ISerializationSurrogate** 对象。如果已经存在和要添加的 **Type/StreamingContext** 相同的一个键，**AddSurrogate** 会抛出一个 **ArgumentException**。通过在键中包含一个 **StreamingContext**，可以登记一个代理类型对象，它知道如何将 **DateTime** 对象序列化/反序列化到一个文件中；再登记一个不同的代理对象，它知道如何将 **DateTime** 对象序列化/反序列化到一个不同的进程中。



> **注意** **BinaryFormatter** 类有一个 bug，会造成代理无法序列化循环引用的对象。为了解决这个问题，需要将对自己的 **ISerializationSurrogate** 对象的引用传给 **FormatterServices** 的静态 **GetSurrogateForCyclicalReference** 方法。该方法返回一个 **ISerializationSurrogate** 对象。然后，可以将对这个对象的引用传给 **SurrogateSelector** 的 **AddSurrogate** 方法。但要注意，使用 **GetSurrogateForCyclicalReference** 方法时，代理的 **SetObjectData** 方法必须修改 **SetObjectData** 的 **obj** 参数所引用的对象中的值，而且最后要向调用方法返回 **null** 或 **obj**。在本书的配套资源中，有一个例子展示了如何修改 **UniversalToLocalTimeSerializationSurrogate** 类和 **SerializationSurrogateDemo** 方法来支持循环引用。

代理选择器链

多个 **SurrogateSelector** 对象可链接到一起。例如，可以让一个 **SurrogateSelector** 对象维护

一组序列化代理，这些序列化代理(surrogate)用于将类型序列化成代理(proxy)[①]，以便通过网络传送，或者跨越不同的 AppDomain 传送。还可以让另一个 **SurrogateSelector** 对象维护一组序列化代理，这些序列化代理用于将版本 1 的类型转换成版本 2 的类型。

如果有多个希望格式化器使用的 **SurrogateSelector** 对象，必须把它们链接到一个链表中。**SurrogateSelector** 类型实现了 **ISurrogateSelector** 接口，该接口定义了三个方法。这些方法全部跟链接有关。下面展示了 **ISurrogateSelector** 接口是如何定义的：

```
public interface ISurrogateSelector {
    void ChainSelector(ISurrogateSelector selector);
    ISurrogateSelector GetNextSelector();
    ISerializationSurrogate GetSurrogate(Type type, StreamingContext context,
        out ISurrogateSelector selector);
}
```

ChainSelector 方法紧接在当前操作的 **ISurrogateSelector** 对象(**this** 对象)之后插入一个 **ISurrogateSelector** 对象。**GetNextSelector** 方法返回对链表中的下一个 **ISurrogateSelector** 对象的引用；如果当前操作的对象是链尾，就返回 **null**。

GetSurrogate 方法在 **this** 所代表的 **ISurrogateSelector** 对象中查找一对 **Type/StreamingContext**。如果没有找到 **Type/StreamingContext** 对，就访问链中的下一个 **ISurrogateSelector** 对象，以此类推。如果找到一个匹配项，**GetSurrogate** 将返回一个 **ISerializationSurrogate** 对象，该对象负责对找到的类型进行序列化/反序列化。除此之外，**GetSurrogate** 还会返回包含匹配项的 **ISurrogateSelector** 对象；一般都用不着这个对象，所以一般会将其忽略。如果链中所有 **ISurrogateSelector** 对象都不包含匹配的一对 **Type/StreamingContext**，**GetSurrogate** 将返回 **null**。



注意　FCL 定义了一个 **ISurrogateSelector** 接口，还定义了一个实现了该接口的 **SurrogateSelector** 类型。然而，只有在一些非常罕见的情况下，才需要定义自己的类型来实现 **ISurrogateSelector** 接口。实现 **ISurrogateSelector** 接口的唯一原因就是将类型映射到另一个类型时需要更大的灵活性。例如，你可能希望以一种特殊方式序列化从一个特定基类继承的所有类型。**System.Runtime.Remoting.Messaging.RemotingSurrogateSelector** 类就是一个很好的例子。出于远程访问(remoting)目的而序列化对象时，CLR 使用 **RemotingSurrogateSelector** 来格式化对象。这个代理选择器(surrogate selector)以一种特殊方式序列化从 **System.MarshalByRefObject** 派生的所有对象，确保反序列化会造成在客户端创建代理对象(proxy object)。

① 两个“代理”是不同的概念。**surrogate** 对象负责序列化，而 **proxy** 对象负责跨越 **AppDomain** 边界访问对象(参见 22.2.1 节“跨越 **AppDomain** 边界访问对象”)。——译注

24.9 反序列化对象时重写程序集/类型

序列化对象时，格式化器输出类型及其定义程序集的全名。反序列化对象时，格式化器根据这个信息确定要为对象构造并初始化什么类型。前面讨论了如何利用 **ISerializationSurrogate** 接口来接管特定类型的序列化和反序列化工作。实现了 **ISerializationSurrogate** 接口的类型与特定程序集中的特定类型关联。

但有的时候，**ISerializationSurrogate** 机制的灵活性显得有点不足。在下面列举的情形中，有必要将对象反序列化成和序列化时不同的类型。

- 开发人员可能想把一个类型的实现从一个程序集移动到另一个程序集。例如，程序集版本号的变化造成新程序集有别于原始程序集。
- 服务器对象序列化到发送给客户端的流中。客户端处理流时，可以将对象反序列化成完全不同的类型，该类型的代码知道如何向服务器的对象发出远程方法调用。
- 开发人员创建了类型的新版本，想把已序列化的对象反序列化成类型的新版本。

利用 **System.Runtime.Serialization.SerializationBinder** 类，可以非常简单地将一个对象反序列化成不同类型。为此，要先定义自己的类型，让它从抽象类 **SerializationBinder** 派生。在下面的代码中，假定你的版本 1.0.0.0 的程序集定义了名为 **Ver1** 的类，并假定程序集的新版本定义了 **Ver1ToVer2SerializationBinder** 类，还定义了名为 **Ver2** 的类：

```
internal sealed class Ver1ToVer2SerializationBinder : SerializationBinder {
    public override Type BindToType(String assemblyName, String typeName) {
        // 将任何 Ver1 对象从版本 1.0.0.0 反序列化成一个 Ver2 对象

        // 计算定义 Ver1 类型的程序集名称
        AssemblyName assemVer1 = Assembly.GetExecutingAssembly().GetName();
        assemVer1.Version = new Version(1, 0, 0, 0);

        // 如果从 v1.0.0.0 反序列化 Ver1 对象，就把它转变成一个 Ver2 对象
        if (assemblyName == assemVer1.ToString() && typeName == "Ver1")
            return typeof(Ver2);

        // 否则，就只返回请求的同一个类型
        return Type.GetType(String.Format("{0}, {1}", typeName, assemblyName));
    }
}
```

现在，在构造好格式化器之后，构造 **Ver1ToVer2SerializationBinder** 的实例，并设置格式化器的可读/可写属性 **Binder**，让它引用绑定器(binder)对象。设置好 **Binder** 属性后，调用格式化器的 **Deserialize** 方法。在反序列化期间，格式化器发现已设置了一个绑定器。每个对象要反序列化时，格式化器都调用绑定器的 **BindToType** 方法，向它传递程序集名称以及格式化器想要反序列化的类型。然后，**BindToType** 判断实际应该构建什么类型，并返回这个类型。



注意　SerializationBinder 类还可重写 **BindToName** 方法，从而在序列化对象时更改程序集/类型信息，这个方法看起来像下面这样：

```
public virtual void BindToName(Type serializedType,
    out string assemblyName, out string typeName)
```

序列化期间，格式化器调用这个方法，传递它想要序列化的类型。然后，你可以通过两个 out 参数返回真正想要序列化的程序集和类型。如果两个 out 参数返回 **null** 和 **null**(默认实现就是这样的)，就不执行任何更改。

[redacted]

第 25 章　与 WinRT 组件互操作

本章内容：

- CLR 投射与 WinRT 组件类型系统规则
- 框架投射
- 用 C#定义 WinRT 组件

Windows 8/8.1 带来了一个新类库，应用程序可通过它访问操作系统功能。类库正式名称是 **Windows 运行时**(Windows Runtime，WinRT)，其组件通过 WinRT 类型系统访问。首次发布时，WinRT 的许多目标都和 CLR 相同，例如简化应用程序开发，以及允许代码用不同编程语言实现以简化互操作。特别是，Microsoft 支持在原生 C/C++、JavaScript(通过 Microsoft 的“Chakra” JavaScript 虚拟机)和 C#/Visual Basic(通过 CLR)中使用 WinRT 组件。

图 25-1 展示了 Windows 的 WinRT 组件所公开的功能，以及可以访问它们的 Microsoft 语言。对于用原生 C/C++实现的应用程序，开发人员必须为每种 CPU 架构(x86，x64 和 ARM)单独编译代码。相比之下，.NET 开发人员只需编译一次(编译成 IL，CLR 自行将其编译成与主机 CPU 对应的本机代码)。JavaScript 应用程序则自带了源代码，“Chakra”虚拟机解析这些源代码，把它编译成与主机 CPU 对应的本机代码。其他公司也可制作能与 WinRT 组件互操作的语言和环境。

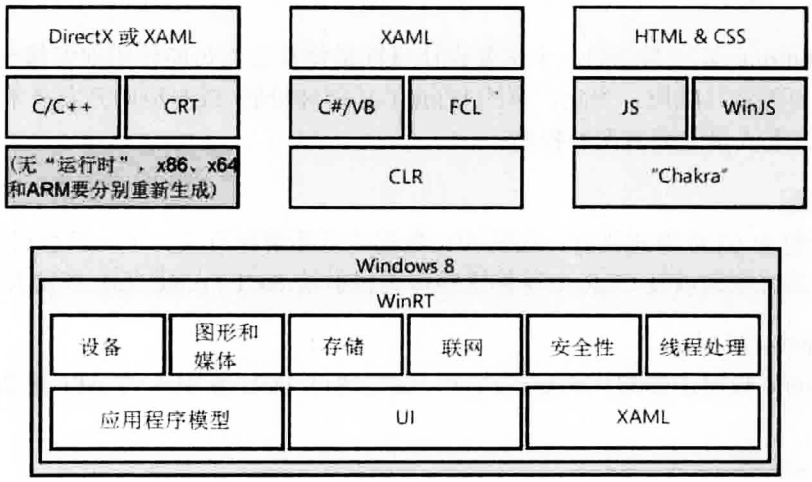


图 25-1　Windows 的 WinRT 组件所公开的功能，以及访问它们的各种语言

Windows Store 应用和桌面应用程序可通过 WinRT 组件来利用操作系统的功能。Windows 配套提供的 WinRT 组件数量比.NET Framework 类库小多了。但设计就是这样的，组件的目的是公开操作系统最擅长的事情，也就是对硬件和跨应用程序的功能进行抽象。所以，大多数 WinRT 组件都只是公开了功能，比如存储、联网、图形、媒体、安全性、线程处理等。而其他核心语言服务(比如字符串操作)和较复杂的框架(比如 LINQ)不是由操作系统提供，而是由访问 WinRT 组件的语言提供。

WinRT 组件内部作为“组件对象模型”(Component Object Model，COM)组件来实现，后者是 Microsoft 1993 年推出的技术。COM 当年被认为过于复杂，规则过于难解，是一个很让人头疼的编程模型。但 COM 实际是有许多亮点的。多年来，Microsoft 对其进行了大量修订，显著地进行了简化。Microsoft 对 WinRT 组件进行了一个很大的调整，不是使用类库来描述 COM 组件的 API，而是使用元数据。你没有看错，WinRT 组件使用由 ECMA 协会标准化的.NET 元数据格式(ECMA-335)来描述其 API。

这个元数据格式正是本书一直在讨论的。元数据比类库更有表现力，而且 CLR 已经对元数据有了全面理解。此外，CLR 一开始就通过**运行时可调用包装器**(Runtime Callable Wrapper，RCW)和 **COM 可调用包装器**(COM Callable Wrapper，CCW)实现了与 COM 组件的互操作。这使在 CLR 顶部运行的语言(如 C#)能无缝地与 WinRT 类型和组件进行互操作。

在 C#中引用 WinRT 对象，实际获得的是对一个 RCW 的引用，RCW 内部引用了 WinRT 组件。类似地，将一个 CLR 对象传给 WinRT API，实际传递的是一个 CCW 引用，CCW 内部容纳了对 CLR 对象的引用。

WinRT 组件将元数据嵌入扩展名为.winmd 的文件中(winmd 代表 Windows MetaData)。Windows 搭载的 WinRT 组件将元数据存储到各种 Windows.*.winmd 文件中，这些文件可在%WinDir%\System32\WinMetadata 目录中找到。生成应用时要引用安装到以下目录的 Windows.winmd 文件[①]：

```
%ProgramFiles(x86)%\Windows Kits\8.1\References\CommonConfiguration\Neutral\Windows.winmd
```

Windows Runtime 类型系统的一个主要设计目标是使开发人员能使用他们擅长的技术、工具、实践以及约定写应用。为此，有的 WinRT 功能被投射[②]成对应的开发技术。针对.NET Framework 开发人员主要有两种投射：

- **CLR 投射**
  CLR 投射由 CLR 隐式执行，通常和元数据的重新解释有关。下一节会讨论 WinRT 组件类型系统规则以及 CLR 如何将这些规则投射给.NET Framework 开发人员。

- **Framework 投射**
  Framework 投射由你的代码显式执行，这是通过 FCL 新引入的 API 来执行的。如果

① 针对 Windows 8.1。——译注

② 本章所说的投射(projection)和映射(mapping)是一回事。——译注

WinRT 类型系统和 CLR 类型系统差异太大，造成 CLR 不能隐式地投射，就需要用到 Framework 投射。本章稍后会讨论这种投射。

25.1　CLR 投射与 WinRT 组件类型系统规则

就像 CLR 强制遵循一个类型系统，WinRT 组件也遵循自己的类型系统。CLR 看到一个 WinRT 类型时，通常允许通过 CLR 的一般化 COM 互操作技术来使用该类型。但有时 CLR 会隐藏 WinRT 类型(将其动态设为私有)。然后，CLR 通过一个不同的类型来公开该类型。在内部，CLR 会查找特定的类型(通过元数据)，然后将这些类型映射成 FCL 的类型。要获得 CLR 隐式投射到 FCL 类型的完整 WinRT 类型列表，请访问 *http://msdn.microsoft.com/en-us/library/windows/apps/hh995050.aspx*。

WinRT 类型系统的核心概念

WinRT 类型系统在功能上不如 CLR 类型系统丰富。下面总结了 WinRT 类型系统的核心概念以及 CLR 如何投射它们。

- **文件名和命名空间**
  .winmd 文件本身的名称必须和包含 WinRT 组件的命名空间匹配。例如，Wintellect.WindowsStore.winmd 文件必须在 **Wintellect.WindowsStore** 命名空间或者它的子命名空间中定义 WinRT 组件。由于 Windows 系统不区分大小写，所以仅大小写不同的命名空间是不允许的。另外，WinRT 组件不能与命名空间同名。

- **通用基类型**
  WinRT 组件不共享一个通用基类。CLR 投射一个 WinRT 类型时，感觉 WinRT 就像是从 **System.Object** 派生，因此所有 WinRT 类型都会继承所有公共方法，包括 **ToString**、**GetHashCode**、**Equals** 和 **GetType**。所以，在通过 C#使用 WinRT 对象时，对象看起来是从 **System.Object** 派生 ，可在代码中到处传递 WinRT 对象而不会出任何问题。还可调用“继承”的方法，例如 **ToString**。

- **核心数据类型**
  WinRT 类型系统支持核心数据类型，包括 Boolean，无符号字节、16/32/64 位有符号和无符号整数、单精度和双精度浮点数、16 位字符、字符串和 void[①]。和 CLR 一样，其他所有数据类型都由这些核心数据类型合成。

- **类**
  WinRT 是面向对象的类型系统，这意味着 WinRT 组件支持数据抽象、继承和多态[②]。但有的语言(如 JavaScript)不支持类型继承。为了迎合这些语言，几乎没有 WinRT 组件

① WinRT 不支持有符号字节。

② 数据抽象实际是被强制的，因为 WinRT 类不允许有公共字段。

会利用继承。这意味着它们也没有利用多态。事实上，只有除 JavaScript 之外的其他语言所用的 WinRT 组件才会利用继承和多态。在随 Windows 发布的 WinRT 组件中，只有 XAML 组件(用于创建 UI)才利用了继承和多态。用 JavaScript 写的应用程序使用 HTML 和 CSS 来创建 UI。

- **结构**

  WinRT 支持结构(值类型)，它们的实例跨越 COM 互操作边界按值封送。和 CLR 的值类型不同，WinRT 结构只能包含核心数据类型或其他 WinRT 结构类型的公共字段[①]。另外，WinRT 结构不能有任何构造器或辅助方法(helper method)。为方便起见，CLR 将某些操作系统 WinRT 结构投射成原生 CLR 类型，后者确实提供了构造器和辅助方法。CLR 开发人员会觉得这些投射的类型更亲切。例子包括 **Windows.Foundation** 命名空间中定义的 **Point**、**Rect**、**Size** 和 **TimeSpan** 结构。

- **可空结构**

  WinRT API 可公开可空结构(值类型)。CLR 将 WinRT 的 **Windows.Foundation.IReference<T>**接口投射成 CLR 的 **System.Nullable<T>**类型。

- **枚举**

  枚举值作为有符号或无符号 32 位整数传递。用 C#定义枚举类型，基础类型要么是 **int**，要么是 **uint**。另外，有符号 32 位枚举被看成是离散值，而无符号 32 位枚举被看成是可以 OR 到一起的标志值。

- **接口**

  对于 WinRT 接口的成员，其参数和返回类型只能是 WinRT 兼容的类型。

- **方法**

  WinRT 提供了对方法重载的有限支持。具体地说，由于 JavaScript 使用动态类型，所以它分辨不了仅参数类型有区别的方法。例如，JavaScript 允许向原本期待字符串的方法传递数字。但 JavaScript 确实能区分获取一个参数和获取两个参数的方法 。此外，WinRT 不支持操作符重载方法和默认参数值。另外，实参只能在封送进入或外出(marshal in or out)之间选择一个，永远都不能两者同时进行(marshal in and out)。这意味着不能向方法实参应用 **ref**，但应用 **out** 就是可以的。欲知详情，请参考下个列表的“数组”项目。

- **属性**

  WinRT 属性的数据类型只能指定 WinRT 兼容类型。WinRT 不支持有参属性或只写属性。

- **委托**

  WinRT 委托类型只能为参数类型和返回类型指定 WinRT 组件。向 WinRT 组件传递一个委托时，委托对象会用一个 CCW 包装，在使用它的 WinRT 组件释放 CCW 之前，

① 枚举也可以，但枚举本质上是 32 位整数。

该委托对象不会被垃圾回收。WinRT 委托无 **BeginInvoke** 和 **EndInvoke** 方法。

- **事件**

  WinRT 组件可通过一个 WinRT 委托类型公开事件。由于大多数 WinRT 组件都密封(无继承)，WinRT 定义了一个 **TypedEventHandler** 委托，其 **sender** 参数是泛型类型(而不是 **System.Object**)。

```
public delegate void TypedEventHandler<TSender, TResult>(TSender sender, TResult args);
```

  还有一个 **Windows.Foundation.EventHandler<T>** WinRT 委托类型，CLR 把它投射成你熟悉的.NET Framework 的 **System.EventHandler<T>**委托类型。

- **异常**

  和 COM 组件一样，WinRT 组件幕后用 HRESULT 值(具有特殊语义的 32 位整数)指明其状态。CLR 将 **Windows.Foundation.HResult** 类型的 WinRT 值投射成异常对象。WinRT API 返回已知的、代表错误的 **HRESULT** 值时，CLR 会抛出对应的 **Exception** 派生类实例。例如，**HRESULT 0x8007000e(E_OUTOFMEMORY)** 映射成 System.OutOfMemoryException。其他 **HRESULT** 值造成 CLR 抛出 System.Exception 对象，其 HResult 属性将包含 HRESULT 值。用 C#实现的 WinRT 组件可以直接抛出所需类型的异常，CLR 会把它自动转换成恰当的 HRESULT 值。要获得对 HRESULT 值的完全控制，可构造异常对象，将你想要的 HRESULT 值赋给对象的 HResult 属性，再抛出对象。

- **字符串**

  当然可以在 WinRT 和 CLR 类型系统之间传递不可变的字符串。但 WinRT 类型系统不允许字符串为 **null** 值。向 WinRT API 的字符串参数传递 null，CLR 会检测到这个动作并抛出 **ArgumentNullException**。相反，应该用 **String.Empty** 向 WinRT API 传递空字符串。字符串以传引用的方式传给 WinRT API：传入时被固定(pinned)，返回时解除固定(unpinned)。从 WinRT API 返回 CLR 时，字符串总是被拷贝。将 CLR 字符串数组(**String[]**)传入或传出 WinRT API 时，会生成包含其所有字符串元素的数组拷贝。传入或返回的是这个拷贝。

- **日期和时间**

  WinRT **Windows.Foundation.DateTime** 结构代表的是一个 UTC 日期/时间。CLR 将 WinRT **DateTime** 结构投射成.NET Framework 的 **System.DateTimeOffset** 结构，这是因为 **DateTimeOffset** 优于.NET Framework 的 System.DateTime 结构。在生成的 DateTimeOffset 实例中，CLR 将从 WinRT 组件返回的 UTC 日期/时间转换成本地时间。相反，CLR 将一个 **DateTimeOffet** 作为 UTC 时间传给 WinRT API。

- **URI**

  CLR 将 WinRT **Windows.Foundation.Uri** 类型投射成.NET Framework 的 **System.Uri** 类型。向 WinRT API 传递一个.NET Framework URI 时，CLR 发现它是相对 URI 会抛出一个 **ArgumentException**。WinRT 只支持绝对 URI。URI 总是跨越互操作边界进行

拷贝。

- IClosable/IDisposable

  CLR 将 WinRT **Windows.Foundation.IClosable** 接口(仅有一个 Close 方法)投射成.NET Framework 的 **System.IDisposable** 接口(及其 **Dispose** 方法)。注意，执行 I/O 操作的所有 WinRT API 都是异步实现的。由于 IClosable 接口的方法称为 **Close** 而不是 **CloseAsync**，所以 Close 方法绝对不能执行任何 I/O 操作。这在语义上有别于.NET Framework 的 **Dispose** 方法。对于.NET Framework 实现的类型，调用 Dispose 是可以执行 I/O 操作的。而且事实上，它经常导致先将缓冲数据写入再关闭设备。但 C#代码在某个 WinRT 类型上调用 **Dispose** 时，I/O(比如将缓冲数据写入)是不会执行的，所以有丢失数据的风险。这一点务必引起重视。写包装了输出流的 WinRT 组件时，必须显式调用方法来防止数据丢失。例如，使用 **DataWriter** 时必须记得调用它的 **StoreAsync** 方法。

- 数组

  WinRT API 支持一维零基数组。WinRT 能将数组元素封送进入方法，或者从方法中封送出去(marshal in or out)，永远不能两者同时进行(marshal in and out)。所以，不能将数组传入 WinRT API，让 API 修改数组元素，再在 API 返回后访问修改的元素。[①]但我说的只是一个应该遵守的协定。该协定没有得到强制贯彻，所以有的投射可能同时封送传入传出数组内容。这一般是为了改善性能。例如，如果数组包含结构，CLR 会直接固定(pin)数组，把它传给 WinRT API，返回后再解除固定(unpin)。在这个过程中，实际是传入数组内容，由 WinRT API 修改内容，然后返回修改的内容。但在这个例子中，WinRT API 违反了协定，其行为是得不到保证的。事实上，在进程外运行的一个 WinRT 组件上调用 API 就肯定行不通。

- 集合

  向 WinRT API 传递集合时，CLR 用一个 CCW 来包装集合对象，然后将 CCW 引用传给 WinRT API。WinRT 代码在 CCW 上调用一个成员时，调用线程要跨越互操作边界，造成一定的性能损失。和数组不同，这意味着将集合传给 WinRT API 后，API 可以现场操作集合，不会创建集合元素的拷贝。表 25-1 总结了 WinRT 集合接口以及 CLR 如何把它们投射到.NET 应用程序代码。

表 25-1 WinRT 集合接口和投射的 CLR 集合类型

WinRT 集合类型 (Windows.Foundation.Collections 命名空间)	投射的 CLR 集合类型 (System.Collections.Generic 命名空间)
IIterable<T>	IEnumerable<T>
IVector<T>	IList<T>

① 这意味着无法实现像 **System.Array** 的 **Sort** 方法这样的 API。有趣的是，所有语言(C，C++，C#，Visual Basic 和 JavaScript)都支持传入和传出数组元素，但 WinRT 类型系统就是不允许。

续表

WinRT 集合类型 (Windows.Foundation.Collections 命名空间)	投射的 CLR 集合类型 (System.Collections.Generic 命名空间)
IVectorView<T>	IReadOnlyList<T>
IMap<K, V>	IDictionary<TKey, TValue>
IMapView<K, V>	IReadOnlyDictionary<TKey, TValue>
IKeyValuePair<K, V>	KeyValuePair<TKey, TValue>

从上述列表可以看出，CLR 团队进行了大量工作，尽量保证了 WinRT 类型系统和 CLR 类型系统之间的无缝互操作，使托管代码的开发人员能在代码中更好地利用 WinRT 组件。[①]

25.2　框架投射

在 CLR 不能将一个 WinRT 类型隐式投射给.NET Framework 开发人员的时候，开发人员就必须显式使用框架投射。主要有三种需要进行框架投射的技术：异步编程、WinRT 流和.NET Framework 流之间的互操作以及需要在 CLR 和 WinRT API 之间传输数据块的时候。后续的小节将分别讨论这三种框架投射。由于许多应用程序都要使用这些技术，所以有必要很好地理解和高效地使用它们。

25.2.1　从.NET 代码中调用异步 WinRT API

线程以同步方式执行 I/O 操作时，线程可能阻塞不确定的时间。GUI 线程等待一个同步 I/O 操作时，应用程序 UI 会停止响应用户的输入，比如触摸、鼠标和手写笔事件，造成用户对应用程序感到厌烦。为了防止应用程序出现不响应的情况，执行 I/O 操作的 WinRT 组件通过异步 API 公开其功能。事实上，凡是 CPU 计算时间可能超过 50 毫秒的功能，WinRT 组件都通过异步 API 来公开该功能。本书第 V 部分“线程处理”将详细讨论如何构建响应灵敏的应用程序。

由于如此多的 WinRT API 都是异步的，所以为了高效地使用它们，你需要理解如何通过 C#与它们互操作。例如以下代码：

```
public void WinRTAsyncIntro() {
   IAsyncOperation<StorageFile> asyncOp = KnownFolders.MusicLibrary.GetFileAsync("Song.mp3");
   asyncOp.Completed = OpCompleted;
   // 可选：在以后某个时间调用 asyncOp.Cancel()
}

// 注意：回调方法通过 GUI 或线程池线程执行
private void OpCompleted(IAsyncOperation<StorageFile> asyncOp, AsyncStatus status) {
   switch (status) {
      case AsyncStatus.Completed: // 处理结果
```

① 欲知详情，请访问 *http://msdn.microsoft.com/en-us/library/windows/apps/hh995050.aspx* 并下载 CLRandtheWindowsRuntime.docx 文档。

```
            StorageFile file = asyncOp.GetResults(); /* Completed... */ break;

         case AsyncStatus.Canceled: // 处理取消
            /* Canceled... */ break;

         case AsyncStatus.Error: // 处理异常
            Exception exception = asyncOp.ErrorCode; /* Error... */ break;
      }
      asyncOp.Close();
   }
```

WinRTAsyncIntro 方法调用 WinRT **GetFileAsync** 方法在用户的音乐库中查找文件。执行异步操作的所有 WinRT API 名称都要以 **Async** 结尾，而且都要返回类型实现了 WinRT **IAsyncXxx** 接口的对象；本例使用的接口是 **IAsyncOperation<TResult>**，其中 **TResult** 是 WinRT **StorageFile** 类型。我将对该对象的引用放到名为 **asyncOp** 的变量中，代表等待进行的异步操作。代码必须通过某种方式接收操作完成通知。为此，必须实现一个回调方法(本例是 **OpCompleted**)，创建对它的委托，并将委托赋给 **asyncOp** 的 **Completed** 属性。现在，一旦操作结束，就会由某个线程(不一定是 GUI 线程)调用回调方法。如果操作在将委托赋给 **Completed** 属性之前便结束了，系统会尽快安排对回调方法的调用。换言之，这里发生了竞态条件(race condition)，但实现 **IAsyncXxx** 接口的对象帮你解决了竞态，确保代码能正常工作。

就像 **WinRTAsyncIntro** 方法最后的注释所说的，要取消正在等待进行的操作，可选择调用所有 **IAsyncXxx** 接口都有提供的 **Cancel** 方法。所有异步操作结束都是因为三个原因之一：操作成功完成，操作被显式取消，或者操作出错。异步操作因为上述任何原因而结束时，系统都会调用回调方法，向其传递和原始 **XxxAsync** 方法返回的一样的对象引用，同时传递一个 **AsyncStatus**。在 **OpCompleted** 方法中，我检查 **status** 参数，分别处理成功完成、显式取消和出错的情况[①]。还要注意，处理好操作因为各种原因而结束的情况之后，应该调用 **IAsyncXxx** 接口对象的 **Close** 方法进行清理。

图 25-2 展示了各种 WinRT **IAsyncXxx** 接口。主要的 4 个接口都从 **IAsyncInfo** 接口派生。其中，两个 **IAsyncAction** 接口使你知道操作在什么时候结束，但这些操作没有返回值(**GetResults** 的返回类型是 **void**)。而两个 **IAsyncOperation** 接口不仅使你知道操作在什么时候结束，还能获取它们的返回值(**GetResults** 方法具有泛型 **TResult** 返回类型)。

两个 **IAsyncXxxWithProgress** 接口允许代码接收异步操作期间的定期进度更新。大多数异步操作都不提供进度更新，但有的会(比如后台下载和上传)。接收定时进度更新要求定义另一个回调方法，创建引用它的委托，并将委托赋给 **IAsyncXxxWithProgress** 对象的 **Progress** 属性。回调方法被调用时，会向其传递类型与泛型 **TProgress** 类型匹配的实参。

---

① **IAsyncInfo** 接口提供了一个 **Status** 属性，其中包含的值和传给回调方法的 **status** 参数的值是一样的。但由于参数以传值方式传递，所以访问参数的速度比查询 **IAsyncInfo** 的 **Status** 属性快（查询属性要求通过一个 RCW 来调用 WinRT API）。

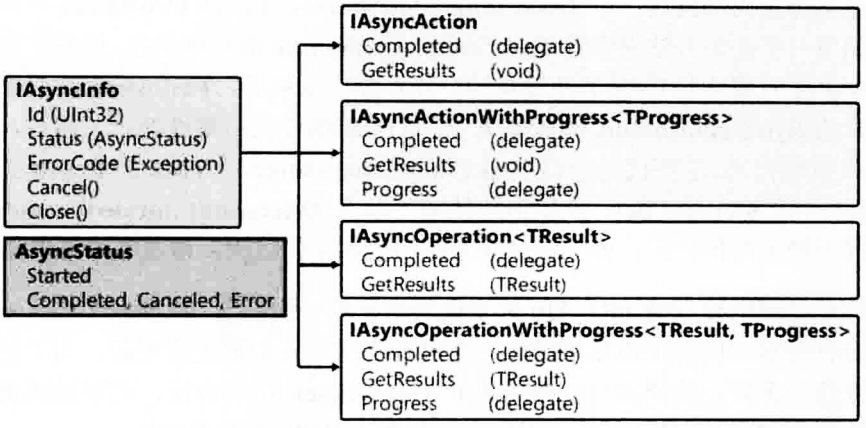


图 25-2　和执行异步 I/O 与计算操作有关的 WinRT 接口

.NET Framework 使用 **System.Threading.Tasks** 命名空间的类型来简化异步操作。我将在第 27 章“计算限制的异步操作”解释这些类型以及如何用它们执行计算操作，在第 28 章“I/O 限制的异步操作”解释如何用它们执行 I/O 操作。除此之外，C#提供了 **async** 和 **await** 关键字，允许使用顺序编程模型来执行异步操作，从而大幅简化了编码。

以下代码重写了之前的 **WinRTAsyncIntro** 方法。这个版本利用了.NET Framework 提供的一些扩展方法，将 WinRT 异步编程模型转变成更方便的 C#编程模型。

```
using System;      // 为了使用 WindowsRuntimeSystemExtensions 中的扩展方法，
                   // 这些扩展方法称为框架投射扩展方法
.
.
.
public async void WinRTAsyncIntro() {
   try {
      StorageFile file = await KnownFolders.MusicLibrary.GetFileAsync("Song.mp3");
      /* Completed... */
   }
   catch (OperationCanceledException) { /* Canceled... */ }
   catch (SomeOtherException ex) { /* Error... */ }
}
```

C#的 **await** 操作符导致编译器在 **GetFileAsync** 方法返回的 **IAsyncOperation<StorageFile>** 接口上查找 **GetAwaiter** 方法。该接口没有提供 **GetAwaiter** 方法，所以编译器查找扩展方法。幸好，.NET Framework 团队在 System.Runtime.WindowsRuntime.dll 中提供了能在任何一个 WinRT **IAsyncXxx** 接口上调用的大量扩展方法。

```
namespace System {
   public static class WindowsRuntimeSystemExtensions {
      public static TaskAwaiter GetAwaiter(
         this IAsyncAction source);
      public static TaskAwaiter GetAwaiter<TProgress>(
         this IAsyncActionWithProgress<TProgress> source);
      public static TaskAwaiter<TResult> GetAwaiter<TResult>(
         this IAsyncOperation<TResult> source);
      public static TaskAwaiter<TResult> GetAwaiter<TResult, TProgress>(
         this IAsyncOperationWithProgress<TResult, TProgress> source);
   }
}
```

所有这些方法都在内部构造一个 **TaskCompletionSource**，并告诉 **IAsyncXxx** 对象在异步操作结束后调用一个回调方法来设置 **TaskCompletionSource** 的最终状态。这些扩展方法返回的 **TaskAwaiter** 对象才是 C#最终所等待的。异步操作结束后，**TaskAwaiter** 对象通过与原始线程关联的 **SynchronizationContext**(第 28 章讨论)确保代码继续执行。然后，线程执行 C#编译器生成的代码。这些代码查询 **TaskCompletionSource** 的 **Task** 的 **Result** 属性来返回结果(本例是一个 **StorageFile**)，在取消的情况下抛出 **OperationCanceledException**，或在出错的情况下抛出其他异常。要了解这些方法的内部工作过程，请参考本节末尾的代码。

刚才展示的只是调用异步 WinRT API 并发现其结果的一般情况。我展示了如何知道发生了取消，但没有展示如何真正地取消操作，也没有展示如何处理进度更新。为了正确处理取消和进度更新，不要让编译器自动调用某个 **GetAwaiter** 扩展方法，而要显式调用同样在 **WindowsRuntimeSystemExtensions** 类中定义的某个 **AsTask** 扩展方法。

```
namespace System {
   public static class WindowsRuntimeSystemExtensions {
      public static Task AsTask<TProgress>(this IAsyncActionWithProgress<TProgress> source,
         CancellationToken cancellationToken, IProgress<TProgress> progress);

      public static Task<TResult> AsTask<TResult, TProgress>(
         this IAsyncOperationWithProgress<TResult, TProgress> source,
         CancellationToken cancellationToken, IProgress<TProgress> progress);

      // 未显示更简单的重载
   }
}
```

现在就可以对程序进行最后的完善了。下面展示了如何调用异步 WinRT API，并在需要的时候正确使用取消和进度更新功能。

```
using System; // 为了 WindowsRuntimeSystemExtensions 的 AsTask
using System.Threading; // 为了 CancellationTokenSource

internal sealed class MyClass {
   private CancellationTokenSource m_cts = new CancellationTokenSource();

   // 注意：如果由 GUI 线程调用，所有代码都通过 GUI 线程执行
   private async void MappingWinRTAsyncToDotNet(WinRTType someWinRTObj) {
      try {
         // 假定 XxxAsync 返回 IAsyncOperationWithProgress<IBuffer, UInt32>
         IBuffer result = await someWinRTObj.XxxAsync(...)
            .AsTask(m_cts.Token, new Progress<UInt32>(ProgressReport));
         /* Completed... */
      }
      catch (OperationCanceledException) { /* Canceled... */ }
      catch (SomeOtherException) { /* Error... */ }
   }

   private void ProgressReport(UInt32 progress) { /* Update progress... */ }

   public void Cancel() { m_cts.Cancel(); } // 以后某个时间调用
}
```

有的读者想知道这些 **AsTask** 方法内部如何将一个 WinRT **IAsyncXxx** 转换成最终可以等待的一个.NET Framework **Task**。以下代码展示了最复杂的 **AsTask** 方法在内部如何实现。当然，更简单的重载实现起来更简单。

```
public static Task<TResult> AsTask<TResult, TProgress>(
   this IAsyncOperationWithProgress<TResult, TProgress> asyncOp,
   CancellationToken ct = default(CancellationToken),
   IProgress<TProgress> progress = null) {

   // 在 CancellationTokenSource 取消时取消异步操作
   ct.Register(() => asyncOp.Cancel());

   // 在异步操作报告进度时，报告给进度回调
   asyncOp.Progress = (asyncInfo, p) => progress.Report(p);

   // 这个 TaskCompletionSource 监视异步操作结束
   var tcs = new TaskCompletionSource<TResult>();

   // 在异步操作结束时通知 TaskCompletionSource①。
   // 届时，正在等待 TaskCompletionSource 的代码重新获得控制权
   asyncOp.Completed = (asyncOp2, asyncStatus) => {
      switch (asyncStatus) {
         case AsyncStatus.Completed: tcs.SetResult(asyncOp2.GetResults()); break;
         case AsyncStatus.Canceled: tcs.SetCanceled(); break;
         case AsyncStatus.Error: tcs.SetException(asyncOp2.ErrorCode); break;
      }
   };

   // 调用代码等待这个返回的 Task 时，它调用 GetAwaiter，后者
   // 用一个 SynchronizationContext 包装 Task，确保异步操作
   // 在 SynchronizationContext 对象的上下文中结束
   return tcs.Task;
}
```

25.2.2　WInRT 流和.NET 流之间的互操作

许多.NET Framework 类都要求操作 **System.IO.Stream** 派生类型，包括序列化和 LINQ to XML 等。只有使用 **System.IO.WindowsRuntimeStorageExtensions** 类定义的扩展方法，实现了 WinRT **IStorageFile** 或 **IStorageFolder** 接口的 WinRT 对象才能和要求 **Stream** 派生类型的.NET Framework 类一起使用。

```
namespace System.IO { // 在 System.Runtime.WindowsRuntime.dll 中定义
   public static class WindowsRuntimeStorageExtensions {
      public static Task<Stream> OpenStreamForReadAsync(this IStorageFile file);
      public static Task<Stream> OpenStreamForWriteAsync(this IStorageFile file);

      public static Task<Stream> OpenStreamForReadAsync(this IStorageFolder rootDirectory,
         String relativePath);
      public static Task<Stream> OpenStreamForWriteAsync(this IStorageFolder rootDirectory,
      String relativePath, CreationCollisionOption creationCollisionOption);
   }
}
```

下例使用扩展方法打开一个 WinRT **StorageFile**，将内容读入一个.NET Framework **XElement** 对象。

```
async Task<XElement> FromStorageFileToXElement(StorageFile file) {
   using (Stream stream = await file.OpenStreamForReadAsync()) {
      return XElement.Load(stream);
   }
}
```

① 或者说向这个 **TaskCompletionSource** 发信号。——译注

最后，**System.IO.WindowsRuntimeStreamExtensions** 类提供了一些扩展方法能将 WinRT 流接口(例如 **IRandomAccessStream**，**IInputStream** 和 **IOutputStream**)“转型”为.NET Framework 的 **Stream** 类型，或者反向转换。

```
namespace System.IO { // 在 System.Runtime.WindowsRuntime.dll 中定义
   public static class WindowsRuntimeStreamExtensions {
      public static Stream AsStream(this IRandomAccessStream winRTStream);
      public static Stream AsStream(this IRandomAccessStream winRTStream, Int32 bufferSize);

      public static Stream AsStreamForRead(this IInputStream winRTStream);
      public static Stream AsStreamForRead(this IInputStream winRTStream, Int32 bufferSize);

      public static Stream AsStreamForWrite(this IOutputStream winRTStream);
      public static Stream AsStreamForWrite(this IOutputStream winRTStream, Int32 bufferSize);

      public static IInputStream AsInputStream (this Stream clrStream);
      public static IOutputStream AsOutputStream(this Stream clrStream);
   }
}
```

下例使用扩展方法将一个 WinRT **IInputStream** “转型”为.NET Framework **Stream** 对象。

```
XElement FromWinRTStreamToXElement(IInputStream winRTStream) {
   Stream netStream = winRTStream.AsStreamForRead();
   return XElement.Load(netStream);
}
```

注意，.NET Framework 提供的“转型”扩展方法幕后不仅仅是执行转型。具体地说，将 WinRT 流转换成.NET Framework 流时，会在托管堆中为 WinRT 流隐式创建一个缓冲区。结果是大多数操作都向这个缓冲区写入，不需要跨越互操作边界，这提升了性能。涉及大量小的 I/O 操作(比如解析 XML 文档)时，性能的提升尤其明显。

使用.NET Framework 流投射的好处是，在同一个 WinRT 流实例上多次执行一个 **AsStreamXxx** 方法，不用担心会创建多个相互没有连接的缓冲区，造成向一个缓冲区写入的数据在另一个那里看不见。.NET Framework 的 API 确保每个流对象都有唯一的适配器实例，所有用户共享同一个缓冲区。

虽然默认缓冲区大多数时候都能在性能与内存使用之间获得较好的平衡，但有时还是希望调整缓冲区的大小(而不是默认的 16 KB)。这时可以使用 **AsStreamXxx** 方法的重载版本来达到目的。例如，如果知道要长时间操作一个很大的文件，同时不会使用其他太多的缓冲流，就可为自己的流请求一个很大的缓冲区来获得进一步的性能提升。相反，有的应用程序要求低网络延迟，这时可能希望确保除非应用程序显式请求，否则不要从网络读取更多的字节。这时可考虑完全禁用缓冲区。为 **AsStreamXxx** 方法指定零字节的缓冲区，就不会创建缓冲区对象了。

25.2.3 在 CLR 和 WinRT 之间传输数据块

要尽量使用上一节讨论的框架投射，因为它们有不错的性能。但有时需要在 CLR 和 WinRT 组件之间传递原始数据块(raw blocks)。例如，WinRT 的文件和套接字流组就要求读写原始数据块。另外，WinRT 的加密组件要对数据块进行加密和解密，位图像素也要用原始数据块来维护。

.NET Framework 获取数据块的方式一般是通过字节数组(**Byte[]**)，或者通过流(比如在使用 **MemoryStream** 类的时候)。当然，字节数组和 **MemoryStream** 对象都不能直接传给 WinRT 组件。所以，WinRT 定义了 **IBuffer** 接口，实现该接口的对象代表可传给 WinRT API 的原始数据块。WinRT **IBuffer** 接口是这样定义的：

```
namespace Windows.Storage.Streams {
   public interface IBuffer {
      UInt32 Capacity { get; }        // 缓冲区最大大小(以字节为单位)
      UInt32 Length { get; set; }     // 缓冲区当前使用的字节数
   }
}
```

如你所见，**IBuffer** 对象定义了缓冲区的最大大小和实际长度。但奇怪的是，它没有提供实际在缓冲区中读写数据的方式。这主要是由于 WinRT 类型不能在其元数据中表示指针，因为指针不能很好地映射到部分语言(比如 JavaScript 和安全 C#代码)。所以，**IBuffer** 对象实际只是在 CLR 和 WinRT API 之间传递内存地址的一种方式。为了访问内存地址处的字节，需要使用一个名为 **IBufferByteAccess** 的内部 COM 接口。注意这是 COM 接口(因为返回指针)而不是 WinRT 接口。.NET Framework 团队为这个 COM 接口定义了一个内部 RCW，如下所示：

```
namespace System.Runtime.InteropServices.WindowsRuntime {
   [Guid("905a0fef-bc53-11df-8c49-001e4fc686da")]
   [InterfaceType(ComInterfaceType.InterfaceIsIUnknown)]
   [ComImport]
   internal interface IBufferByteAccess {
      unsafe Byte* Buffer { get; }
   }
}
```

CLR 内部获取 **IBuffer** 对象，查询其 **IBufferByteAccess** 接口，再查询 **Buffer** 属性获来得指向缓冲区中的字节数据的不安全指针。利用该指针就能直接访问字节。

为防止开发人员写不安全代码来操作指针，FCL 包含一个 **WindowsRuntimeBufferExtensions** 类，它定义了大量扩展方法，.NET Framework 开发人员可显式调用这些方法在 CLR 字节数组和传给 WinRT **IBuffer** 对象的流之间传递数据块。调用这些扩展方法要求在源代码中添加 **using System.Runtime.InteropServices.WindowsRuntime;**指令。

```
namespace System.Runtime.InteropServices.WindowsRuntime {
   public static class WindowsRuntimeBufferExtensions {
      public static IBuffer AsBuffer(this Byte[] source);
      public static IBuffer AsBuffer(this Byte[] source, Int32 offset, Int32 length);
      public static IBuffer AsBuffer(this Byte[] source, Int32 offset, Int32 length,
            Int32 capacity);
      public static IBuffer GetWindowsRuntimeBuffer(this MemoryStream stream);
      public static IBuffer GetWindowsRuntimeBuffer(this MemoryStream stream, Int32
            position, Int32 length);
   }
}
```

所以，要将一个 **Byte[]**传给需要一个 **IBuffer** 的 WinRT API，只需在 **Byte[]**数组上调用 **AsBuffer**。这实际是将对 **Byte[]**的引用包装到实现了 **IBuffer** 接口的对象中；**Byte[]**数组的内容不会被复制，所以效率很高。类似地，对于包装了公共 **Byte[]**数组缓冲区的一个 **MemoryStream** 对象，只需在它上面调用 **GetWindowsRuntimeBuffer** 就可以将对

MemoryStream 的缓冲区的引用包装到一个实现了 **IBuffer** 接口的对象中。缓冲区内容同样不会被复制，所以效率很高。以下方法演示了这两种情况。

```
private async Task ByteArrayAndStreamToIBuffer(IRandomAccessStream winRTStream, Int32 count)
{
   Byte[] bytes = new Byte[count];
   await winRTStream.ReadAsync(bytes.AsBuffer(), (UInt32)bytes.Length, InputStreamOptions.None);
   Int32 sum = bytes.Sum(b => b); // 访问从 Byte[]读取的字节

   using (var ms = new MemoryStream())
   using (var sw = new StreamWriter(ms)) {
      sw.Write("This string represents data in a stream");
      sw.Flush();
      UInt32 bytesWritten = await winRTStream.WriteAsync(ms.GetWindowsRuntimeBuffer());
   }
}
```

WinRT 的 **IRandomAccessStream** 接口实现了 WinRT 的 **IInputStream** 接口。

```
namespace Windows.Storage.Streams {
   public interface IInputStream : IDisposable {
      IAsyncOperationWithProgress<IBuffer, UInt32> ReadAsync(
          IBuffer buffer,
          uint count,
          InputStreamOption options)
   }
}
```

在代码中调用 **AsBuffer** 或 **GetWindowsRuntimeBuffer** 扩展方法时，这些方法将来源对象包装到实现了 **IBuffer** 接口的一个类的对象中。然后，CLR 为该对象创建一个 CCW，并将 CCW 传给 WinRT API。一旦 WinRT API 查询 **IBufferByteAccess** 接口的 **Buffer** 属性，获取指向基础字节数组的指针，字节数组就会被固定(pinned)，其地址返回给 WinRT API 以便访问数据。一旦 WinRT API 内部在 **IBufferByteAccess** 接口上调用 COM 的 **Release** 方法，字节数组就会解除固定(unpinned)。

调用返回一个 **IBuffer** 的 WinRT API 时，数据本身可能在本机(native)内存中，需要以某种方式从托管代码中访问这些数据。这时需要借助于 **WindowsRuntimeBufferExtensions** 类定义的其他扩展方法。

```
namespace System.Runtime.InteropServices.WindowsRuntime {
   public static class WindowsRuntimeBufferExtensions {
      public static Stream AsStream(this IBuffer source);
      public static Byte[] ToArray(this IBuffer source);
      public static Byte[] ToArray(this IBuffer source, UInt32 sourceIndex, Int32 count);

      // 未显示：CopyTo 方法在一个 IBuffer 和一个 Byte[]之间传输字节
      // 未显示：GetByte，IsSameData 方法
   }
}
```

AsStream 方法创建包装了来源 **IBuffer** 的一个 **Stream** 派生对象。调用 **Stream** 的 **Read**、**Write** 和其他类似的方法，即可访问该 **IBuffer** 的数据。**ToArray** 方法在内部分配一个 **Byte[]**，将来源 **IBuffer** 中的所有字节复制到 **Byte[]**中；注意该扩展方法可能会占用大量内存和 CPU 时间。

WindowsRuntimeBufferExtensions 类还提供了 **CopyTo** 方法的几个重载版本，能在一个

IBuffer 和一个 **Byte[]**之间复制数据。还提供了一个 **GetByte** 方法，能每次从 **IBuffer** 中获取一个字节。**IsSameData** 方法则比较两个 **IBuffer** 对象的内容，判断内容是否完全一样。大多数应用程序都不需要调用这些方法。

.NET Framework 定义了一个 **System.Runtime.InteropServices.WindowsRuntimeBuffer** 类，允许创建字节在托管堆中的 **IBuffer** 对象。对应地，WinRT 组件 **Windows.Storage.Streams.Buffer** 允许创建字节在本机(native)堆中的 **IBuffer** 对象。大多数.NET Framework 开发人员都不需要在代码中显式使用这两个类。

25.3　用 C#定义 WinRT 组件

本章一直在讲述如何在 C#中使用现有的 WinRT 组件。但也可定义自己的 WinRT 组件，以便原生 C/C++、C#/Visual Basic、JavaScript 和其他语言使用这些组件。但有的时候是不为也，非不能也。例如，如果唯一的使用者就是在 CLR 顶部运行的其他托管语言，那么用 C#定义的 WinRT 组件就没有什么意义。这是由于 WinRT 类型系统功能少得多，相比 CLR 的类型系统限制太大。

用 C#定义能由原生 C/C++代码使用的 WinRT 组件也没有多大意义。一般情况下，应用程序关心性能和内存消耗时才会用原生 C/C++来实现。这时不太可能使用由托管代码实现的 WinRT 组件，否则就要被迫在进程中加载 CLR，增大内存消耗和降低性能(因为要进行垃圾回收和 JIT 编译)。所以，大多数 WinRT 组件(比如随同 Windows 提供的那些)都是用原生代码实现的。当然，如果原生 C++应用的某些部分对性能不敏感，就可考虑利用.NET Framework 的功能来提高开发效率。例如，必应地图 Bing Maps 用原生 C++和 DirectX 绘制 UI，但业务逻辑用 C#实现。

所以，我认为用 C#实现的 WinRT 组件最佳应用场合就是：Windows Store 应用的开发人员用 HTML 和 CSS 构建 UI。然后，使用 JavaScript 代码将 UI 和用 C# WinRT 组件实现的业务逻辑“粘合”起来。还有一个应用场合是在 HTML/JavaScript 应用中使用现有的 FCL 功能(比如 WCF)。HTML/JavaScript 开发人员已习惯了浏览器引擎造成的性能损失和内存消耗，所以基本上能接受使用 CLR 造成的额外性能损失和内存消耗。

用 C#定义 WinRT 组件首先要在 Microsoft Visual Studio 中创建“Windows 运行时组件”项目。创建的其实是一个普通的类库项目，但 C#编译器会自动添加**/t:winmdobj** 命令行开关来生成.winmdobj 文件。文件中会插入一些和平时不同的 IL 代码。例如，WinRT 组件采用和 CLR 不同的方式为事件添加和删除委托。所以，如果指定了这个编译器开关，编译器就会为事件的添加和删除方法生成不同的代码。本节稍后会展示如何显式地实现事件的添加和删除方法。

编译器生成.winmdobj 文件后将启动 WinMD 实用程序(WinMDExp.exe)[①]，向它传递由编译

① WinMDExp 是 Windows Metadata Exporter 的简称。——译注