

CLR（公共语言运行时）是整个.NET平台架构的中心元素。CLR是管理所有.NET程序的软件层。在这里，“管理”一词覆盖了托管应用程序执行时所需要的一组广泛的操作，包括：

- 在一个Windows进程内承载多个应用程序；
- 将IL代码编译为机器语言代码；
- 异常管理；
- 无用对象析构；
- 加载应用程序和程序集；
- 类型解析。

CLR从概念上讲接近于广为人知的Java虚拟机。

4.1 应用程序域（AppDomain）

4.1.1 简介

应用程序域（通常简称为AppDomain）可以视为一种轻量级进程。一个Windows进程内可以包含多个AppDomain。AppDomain这个概念的提出是为了实现在一个物理服务器中承载多个应用程序。ASP.NET利用AppDomain在同一个进程内承载了多组Web应用程序就是一个例子。实际上微软曾进行过在单一进程内承载多达1000个简单Web应用程序的压力测试。使用AppDomain所获得的性能优势主要体现在两方面：

- 创建AppDomain所需要的系统资源比创建一个Windows进程更少。
- 同一个Windows进程内所承载的AppDomain之间可以互相共享资源，如CLR、基本.NET类型、地址空间以及线程。

当一个程序集被执行时，CLR将自动为其创建一个默认AppDomain。每个AppDomain都有一个名字，而默认AppDomain的名字就是所执行的程序集的名称（包含.exe扩展名）。

如果一个程序集被多个AppDomain所加载，那么有两种可能。

- 第一种可能：CLR将多次加载该程序集，为进程中的每一个AppDomain分别加载一次。
- 第二种可能：CLR在所有AppDomain之外只加载一次该程序集，之后该程序集可被同一个进程内的所有AppDomain使用。这种方式加载的程序集称为是域无关（domain neutral）的。

在本章稍后我们将会看到，具体的行为是可以配置的。而默认行为是，如果需要程序集将被多次加载。

4.1.2 线程与 AppDomain

千万不要把“线程”和“应用程序域”这两个概念搞混。

进程的线程和该进程内的AppDomain这两个概念并没有从属关系。有必要提醒一下，线程和进程

确实有从属关系，所有的线程都分别属于某个进程而每个进程都有一个或多个线程。然而事实上，线程并不局限于一个AppDomain之中，而且在任何给定时间内，多个线程可以运行在同一个AppDomain的上下文中。

考虑同一个进程内存在两个AppDomain分别叫做DA和DB。假设有一个对象A来自DA所包含的一个程序集，而对象B来自DB所包含的一个程序集，对象A的某个方法调用了对象B的方法。这种情况下主调方和被调方将在同一个线程中执行。该线程本质上就越过了DA和DB这两个AppDomain的边界。

换句话说，线程的概念和AppDomain的概念是正交的。

4.1.3 卸载 AppDomain

当AppDomain加载了一个程序集之后，就不能再将它从AppDomain中卸载掉。不过可以将整个AppDomain卸载。这个操作会产生十分严重的后果，因为CLR会终止当前正在该AppDomain执行的所有线程，如果此时正在执行非托管代码的话就可能会产生问题。而且该应用程序域中所有托管对象也会被垃圾收集器强制回收。

建议避免使用这种依赖于AppDomain频繁加载和卸载的架构设计。我们会看到，在实现需要保持高度可用时间的服务类程序（如SQL Server 2005，需要在99.999%的时间上保持可用）时，这种架构设计是极其糟糕的。

4.1.4 AppDomain 和孤立性

各个AppDomain之间的孤立性体现为以下这些特征。

- 一个AppDomain可以独立于其他的AppDomain而被卸载。
- 一个AppDomain无法访问其他AppDomain的程序集和对象。
- 若没有发生跨边界的异常抛出，一个AppDomain拥有自己独立的异常管理策略。这意味着一个AppDomain内出现问题不会影响到同一个进程内的其他AppDomain。
- 每个AppDomain可以分别定义各自的程序集代码访问安全策略。
- 每个AppDomain可以分别定义各自的规则以便CLR在加载前定位程序集所在位置。

4.1.5 System.AppDomain 类

System.AppDomain类的实例表示一个到当前进程内某个AppDomain的引用。该类的静态属性CurrentDomain{get;}允许你获得当前AppDomain的引用。下面的例子展示了如何使用该类列举当前AppDomain所加载的程序集：

例4-1

```
using System;
using System.Reflection; // For the Assembly class.
class Program {
    static void Main() {
        AppDomain curAppDomain = AppDomain.CurrentDomain;
        foreach ( Assembly assembly in curAppDomain.GetAssemblies() )
            Console.WriteLine( assembly.FullName );
    }
}
```

4.1.6 在一个进程中承载多个应用程序

AppDomain类有一个CreateDomain()静态方法，可以在当前进程内创建一个新的AppDomain。这个方法提供了多个重载版本。要使用这个方法，需要指定以下内容。

- (必要) AppDomain的名字；

- (可选) 新建AppDomain的CAS安全规则 (使用System.Security.Policy.Evidence类型的对象表示);
- (可选) 该AppDomain的CLR定位机制信息 (使用System.AppDomainSetup类型的对象表示)。System.AppDomainSetup类有两个重要的属性:
 - ApplicationBase, 该属性定义了AppDomain的基础目录位置, CLR在加载该AppDomain的程序集时所用的程序集定位机制会用到这个信息。
 - ConfigurationFile, 该属性表示AppDomain的配置文件。这是一个XML文件, 包含有AppDomain所需的版本信息和定位规则。

现在你已了解到如何创建一个应用程序域, 现在我们来介绍如何使用System.AppDomain.ExecuteAssembly()方法在这个应用程序域中加载和运行程序集。这个程序集必须是可执行文件, 执行流程将从这个程序集的入口点开始。注意, 调用ExecuteAssembly()方法的线程将会是所加载程序集的执行线程。这也正是一个线程跨过AppDomain边界执行的例子。

下面是一个C#例子。最开始一小段代码编译成的程序集将在稍后被第二段代码中定义的程序集加载。

例4-2 AssemblyToLoad.exe

```
using System;
using System.Threading;
public class Program {
    public static void Main() {
        Console.WriteLine(
            "Thread:{0} Hi from the domain: {1}",
            Thread.CurrentThread.Name,
            AppDomain.CurrentDomain.FriendlyName);
    }
}
```

例4-3 AssemblyLoader.exe

```
using System;
using System.Threading;
public class Program {
    public static void Main() {
        // Name the current thread.
        Thread.CurrentThread.Name = "MyThread";
        // Create an AppDomainSetup instance.
        AppDomainSetup info = new AppDomainSetup();
        info.ApplicationBase = "file:///"+ Environment.CurrentDirectory;
        // Create a new appdomain without security parameters.
        AppDomain newDomain = AppDomain.CreateDomain(
            "NewDomain", null, info);
        Console.WriteLine(
            "Thread:{0} Calling ExecuteAssembly() from appdomain {1}",
            Thread.CurrentThread.Name,
            AppDomain.CurrentDomain.FriendlyName );
        // Load the assembly 'AssemblyACharger.exe' inside
        // 'NewDomain' and then execute it.
        newDomain.ExecuteAssembly( "AssemblyToLoad.exe" );
        // Unload the new domain.
        AppDomain.Unload( newDomain );
    }
}
```

这个例子将显示：

Thread:MyThread Calling ExecuteAssembly() from appdomain AssemblyLoader.exe
Thread:MyThread Hi from the domain: NewDomain

注意，指定程序集在本地存储设备上的位置时，需要加“file:///”。如果不是本地位置，可以使用“http://”从Web加载程序集。

这个例子还说明了一个事实，AppDomain默认的名字就是启动程序集的主模块名（在本例中是AssemblyLoader.exe）。

4.1.7 在其他 AppDomain 的上下文中运行代码

在AppDomain.DoCallBack()这样一个实例方法的帮助下，我们可以在其他AppDomain的上下文中运行当前AppDomain中程序集的代码。为了做到这一点，需要将所有代码写在一个方法中，并使用System.CrossAppDomainDelegate委托来引用这个方法。这个过程如下面的例子所示：

例4-4

```
using System;
using System.Threading;
public class Program {
    public static void Main() {
        Thread.CurrentThread.Name = "MyThread";
        AppDomain newDomain = AppDomain.CreateDomain( "NewDomain" );
        CrossAppDomainDelegate deleg = new CrossAppDomainDelegate(Fct);
        newDomain.DoCallBack(deleg);
        AppDomain.Unload( newDomain );
    }
    public static void Fct() {
        Console.WriteLine(
            "Thread:{0} execute Fct() inside the appdomain {1}",
            Thread.CurrentThread.Name,
            AppDomain.CurrentDomain.FriendlyName);
    }
}
```

这个例子的输出为：

Thread:MyThread execute Fct() inside the appdomain NewDomain

有必要了解，这种在AppDomain中“注射”代码可能会引发一个安全异常，除你有所需的安全性权限。

4.1.8 AppDomain 类的事件

AppDomain类有如下事件：

事 件	描 述
AssemblyLoad	当程序集加载时引发
AssemblyResolve	当要加载的程序集没有找到时引发
DomainUnload	当AppDomain要卸载之前引发
ProcessExit	当进程终止时引发（在DomainUnload之前引发）
ReflectionOnlyAssemblyResolve	当通过反射手段加载某个程序集时，发生解析失败时引发

（续）

事 件	描 述
ResourceResolve	当指定资源未找到时引发
TypeResolve	当指定类型未找到时引发
UnhandledException	当AppDomain中有未处理的异常时引发

这些事件可以用来纠正引发这些事件的问题。下面的例子展示了如何使用AssemblyResolve事件来指定一个程序集的加载位置，而这个位置事先没有通过CLR定位机制指定。

例4-5

```
using System;
using System.Reflection; // For the Assembly class.
public class Program {
    public static void Main() {
        AppDomain.CurrentDomain.AssemblyResolve += AssemblyResolve;
        Assembly.Load("AssemblyToLoad.dll");
    }
    public static Assembly AssemblyResolve( object sender,
        ResolveEventArgs e ) {
        Console.WriteLine("Can't find assembly : {0}", e.Name);
        return Assembly.LoadFrom(@"C:\AppDir\ThisAssemblyToLoad.dll");
    }
}
```

如果第二次加载尝试成功了，将不会抛出异常。第二次尝试加载的程序集的名称不必和开始那次相同。

在14.7节，我们将演示一个程序利用UnhandledException这个事件。

在3.10.4小节，我们介绍了使用ClickOnce技术部署的应用程序在第一次运行的时候，System.Deployment命名空间中的类可以利用AssemblyResolve和ResourceResolve事件动态地下载一组文件。

4.1.9 在同一个进程的 AppDomain 之间共享信息

可以用AppDomain类的SetData()和GetData()方法直接在AppDomain中存储数据。下面的例子演示了数据存取的过程，AppDomain中保存的数据是用特征字符串来索引的。

例4-6

```
using System;
using System.Threading;
public class Program {
    public static void Main() {
        AppDomain newDomain = AppDomain.CreateDomain("NewDomain");
        CrossAppDomainDelegate deleg = new CrossAppDomainDelegate(Fct);
        newDomain.DoCallBack(deleg);
        // Fetch from the new appdomain the data named 'AnInteger'.
        int anInteger = (int) newDomain.GetData("AnInteger");
        AppDomain.Unload(newDomain);
    }
    public static void Fct() {
        // This method is ran inside the appdomain 'NewDomain'. It creates
        // an appdomain data named 'AnInteger' which has the value '691'.
        AppDomain.CurrentDomain.SetData("AnInteger", 691);
    }
}
```

```
}
}
```

这个例子演示的是一个简单的情形，我们存储了一个整型数据，而整型是每个AppDomain都能识别的。第22章介绍的.NET Remoting技术允许用更先进、更复杂的方式在AppDomain之间共享数据。

4.2 在 Windows 进程内通过运行时宿主加载 CLR

4.2.1 mscorsvr.dll 和 mscorwks.dll

每个CLR的发行版本都带有两个DLL文件：

- mscorsvr.dll，包含了为多处理器计算机优化版本的CLR（“svr”代表“server（服务器）”）。
- mscorwks.dll，包含了为单处理器计算机优化版本的CLR[“wks”代表“workstation（工作站）”]。

这两个DLL文件不是程序集，因此也不包含任何IL代码。（所以也不能用ildasm.exe工具进行分析。）所有执行一个或多个.NET应用程序的进程都会包含这两个DLL其中的一个。我们称这样的进程为CLR宿主。接下来我们会解释进程如何加载这两个DLL。

4.2.2 mscorlib.dll 程序集

还有一个在.NET应用程序的执行过程中具有非常重要的地位的DLL——mscorlib.dll。这个DLL包含了.NET Framework所有的基本类型（如System.String、System.Object、System.Int32等）。所有其他的.NET程序集都会引用这个程序集，这种引用关系是所有产生IL的编译器自动创建的。用ildasm.exe工具来分析mscorlib程序集是非常有趣的。有必要提及mscorlib是在AppDomain之外执行的，因此在进程的生命期内只能被加载和卸载一次。

4.2.3 运行时宿主介绍

事实上CLR目前并没有和任何操作系统集成，这意味着进程创建时必须由自己完成CLR的加载操作。

将CLR加载到进程的任务中会涉及一个叫运行时宿主（runtime host）的实体。既然运行时宿主需要加载CLR，那么其中一部分代码必然是非托管的，因为托管代码本身需要CLR才能执行。这一部分非托管代码会负责CLR的加载、配置以及将当前线程转交给托管代码。一旦CLR加载完毕，运行时宿主还有其他一些职责，如处理未捕获的异常等。图4-1说明了CLR宿主架构的不同层次。正如你所看到的，CLR和运行时宿主之间通过一个API来交换数据。

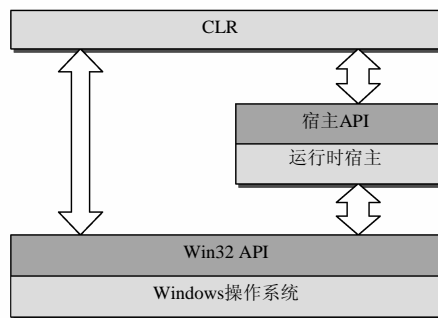


图4-1 CLR的宿主

系统中存在一些现成的运行时宿主，你甚至还可以创建自己的运行时宿主。运行时宿主的选择将

会影响应用程序的性能，还可以用来定义应用程序可用的功能。微软预提供的运行时宿主有：

- **Console和Winform运行时宿主：**执行的程序集在默认AppDomain中加载。隐式加载的程序集将被加载到与引用程序集所在的同一个应用程序域中。这类应用程序通常无需使用除默认AppDomain之外的其他域。
- **ASP.NET运行时宿主：**为每个Web应用程序创建一个AppDomain。Web应用程序通过它的ASP.NET虚拟目录来识别。如果一个Web请求被发送给已经加载的应用程序，那么该请求就会被宿主自动发送到相应的AppDomain。
- **Microsoft Internet Explorer运行时宿主：**默认情况下，这个宿主为每个被访问的Web站点创建一个AppDomain。这样可以允许每个站点使用不同的安全级别。CLR仅在Internet Explorer第一次需要执行程序集的时候加载一次。
- **SQL Server 2005运行时宿主：**允许数据库的查询请求使用IL代码编写。CLR仅在这种请求第一次需要执行的时候加载一次。每个用户/数据库的配对创建一个AppDomain。在这一章我们还会回过头来讨论这种.NET 2已引入的特殊宿主。

4.2.4 在同一台计算机上承载多个版本的 CLR

mscorlib程序集具有强名称，意味着它的多个版本可以共存在同一机器上并行运行。进一步说，就是那些放在“%windir%\Microsoft.NET\Framework”文件夹中的所有CLR版本。某个特定版本.NET Framework的全部文件都放在一个以版本号命名的子文件夹中。正因为每个版本的.NET Framework都有一个独立的文件夹，所以也存在多个版本的mscorsvr.dll和mscorwks.dll。尽管如此，每个进程只能加载一个版本的CLR。

多个版本的CLR可以并存，这客观上就要求存在一个小的软件层，它接受所需CLR的版本为参数，并负责加载相应版本的CLR。这个程序称为shim，保存在mscorlib.dll（MSCOREE的意思是Microsoft Component Object Runtime Execution Engine，组件对象运行时执行引擎）中。

每台计算机只允许有一个shim DLL，它是运行时宿主通过CorBindToRuntimeEx()函数来调用的。mscorlib.dll包含COM接口和类，CorBindToRuntimeEx()基本行为是为请求的CLR版本创建一个CorRuntimeHost COM类的实例。通过这个对象就可以与CLR进行交互了。可以通过CorBindToRuntimeEx()函数返回的ICLRRuntimeHost COM接口来操纵这个对象。

调用CorBindToRuntimeEx()来创建与CLR进行交互的COM对象违背了几条基本的COM准则：不能调用CoCreateInstance()函数创建对象。此外，在这个接口上调用AddRef()和Release()是没有效果的。

4.2.5 使用 CorBindToRuntimeEx()函数加载 CLR

下面是CorBindToRuntimeEx()函数的原型，它负责加载shim DLL，之后由shim 加载CLR。

```
HRESULT CorBindToRuntimeEx(
    LPWSTR    pwszVersion,
    LPWSTR    pwszBuildFlavor,
    DWORD     flags,
    REFCLSID  rclsid,
    REFIID    riid,
    LPVOID *   ppv);
```

这个函数原型的定义在mscorlib.h中，函数的实际代码在mscorlib.dll中。

- **PwszVersion**，指定CLR的版本，这是一个字符串，格式为字母“v”开头（例如“v2.0.50727”）。如果这个字符串未定义（即给函数传了空值），将会使用最新版本的CLR。

- **PwszBuildFlavor**, 这个参数可以指定为字符串“wks”, 表示想要加载工作站版本(mscorwks.dll)的CLR; 或者字符串“svr”, 服务器版本(mscorsvr.dll)的CLR。如果计算机只有一个处理器, 无论指定的是什么版本都将加载工作站版本。微软决定用字符串作为参数类型以便允许将来扩展支持更多CLR种类。

- **Flags**, 此参数由一系列标志组成。

STARTUP_CONCURRENG_GC标志表示希望垃圾收集器以并发模式执行。该模式允许在一个独立的线程中执行垃圾收集的大部分工作, 而不会打扰应用程序的其他线程。如果是在非并发模式, CLR通常会用应用程序的线程来执行垃圾收集。非并发模式的总体性能更好一些, 但是可能导致用户界面偶尔失去响应。

我们还可以用其他标志来指示希望以中性的方式加载与应用程序域有关或无关的程序集。以中性方式加载的意思是程序集的所有资源物理上都只将呈现一次, 即使多个AppDomain加载了同一个程序集(这与在Windows应用程序之间映射DLL的形式类似)。以中性方式加载的程序集并不属于任何一个AppDomain。这种做法带来的最大缺点是除非结束进程, 否则无法卸载该程序集。而且, 每个AppDomain必须应用同一组安全权限, 除非采用多次加载程序集的方式。来自以中性方式加载的程序集的类将会在每个AppDomain中执行一次静态构造函数, 而且所有静态字段也将会为每个域保存一组值。从内部来讲, 这种特性是由CLR所管理的一个间接表来实现的。加入到间接表的操作可能会带来一点点性能损失, 但是想想这个程序集只会被加载一次, 这样消耗的系统资源(如内存)会较少。而且因为程序集只需要经过一次JIT编译, 所以还可以有利于性能的提升。

以上功能涉及的标志如下所示。

- **STARTUP_LOADER_OPTIMIZATION_SINGLE_DOMAIN**, 所有程序集都不按照中性方式加载, 这是默认的行为。
- **STARTUP_LOADER_OPTIMIZATION_MULTI_DOMAIN**, 所有程序集都按照中性方式加载, 目前没有运行时宿主采用这个选项。
- **STARTUP_LOADER_OPTIMIZATION_MULTI_DOMAIN_HOST**, 只有在GAC中共享的程序集按中性方式加载。
- **mscorlib**程序集是个例外, 它总是按中性方式加载。
- **Rclsid**, 实现所需CLR宿主接口的COM类(coclass)的Class ID (CLSID)。合法的参数只有CLSID_CorRuntimeHost、CLSID_CLRRuntimeHost或者null。第二个常数是.NET 2新增的, 因为SQL Server 2005的CLR宿主要求更多新增功能。
- **PwszBuildFlavor**, 你所需要的COM接口的接口ID (IID)。合法的参数只有IID_CorRuntimeHost、IID_CLRRuntimeHost或者null。
- **ppv**, 指向返回COM接口的指针。这个指针的类型要么是ICorRuntimeHost, 要么是ICLRRuntimeHost, 具体按要求而定。

Shim将负责在进程中加载CLR。CLR的生命周期由运行时宿主的非托管代码所控制, 具体就是CorBindToRuntimeEx()函数返回的ICLRRuntimeHost接口。这个接口有两个方法——Start()和Stop(), 其意义正如函数名所描述。

4.2.6 创建一个自定义的运行时宿主

大部分项目里都不需要创建运行时宿主。然而了解一下如何创建是有好处的, 而且创建的过程也十分有意义。

如果想完全了解如何创建运行时宿主，需要对COM技术有深入地了解。别忘了COM仍是Windows 2000/XP 等操作系统的标准通信设施。非托管代码可以通过COM接口操纵CLR。从CorBindToRuntimeEx()获得的ICLRRuntimeHost接口在这个过程中具有决定作用。下面的代码是实现运行时宿主的基本C++代码。

例4-7

```
// You need to link with the static lib mscoree.lib
// to compile this C++ file.
#include <mscoree.h>

// The import must be done on a single line.
#import <mscorlib.tlb> raw_interfaces_only ...
... high_property_prefixes("_get", "_put", "_putref") ...
... rename("ReportEvent", "ReportEventManaged") rename_namespace("CRL")

// We use the namespace ComRuntimeLibrary.
using namespace CRL;
ICLRRuntimeHost * pClrHost = NULL;
void main (void){
    // Get a COM pointer of type ICorRuntimeHost on the CLR.
    HRESULT hr = CorBindToRuntimeEx(
        NULL, // We ask for the most recent version of the CLR.
        NULL, // We ask for the workstation version of the CLR.
        0,
        CLSID_CLRRuntimeHost,
        IID_ICLRRuntimeHost,
        (LPVOID *) &pClrHost);
    if (FAILED(hr)){
        printf("Can't get a pointer of type ICLRRuntimeHost!");
        return;
    }
    printf("We got a pointer of type ICLRRuntimeHost.\n");
    printf("Launch the CLR.\n");
    pClrHost->Start();

    // Here, we can use our COM pointer pClrHost on the CLR.

    pClrHost->Stop();
    printf("CLR stopped.\n");
    pClrHost->Release();
    printf("Bye!\n");
}
```

假设现在有一个程序集，名叫MyManagedLib.dll并放在C:\Test目录下，其代码如下所示。

例4-8 MyManagedLib.cs

```
namespace MyProgramNamespace {
    public class MyClass {
        public static int MyMethod(string s) {
            System.Console.WriteLine(s);
            return 0;
        }
    }
}
```

可以从宿主程序里很容易地调用MyMethod()方法^①:

例4-9

```
...
pClrHost->Start();
DWORD retVal=0;
hr = pClrHost->ExecuteInDefaultAppDomain(
    L"C:\\test\\MyManagedLib.dll", // Path + Asm.
    L"MyProgramNamespace.MyClass", // Full name of the type.
    L"MyMethod",                    // Name of the method to run. It must
                                    // have the signature int XXX(string).
    L"Hello from host!",            // The argument string.
    &retVal);                        // A reference to the returned value.

pClrHost->Stop();
...
```

4.2.7 在自定义运行时宿主中调整 CLR

首先要提到的是,使用COM对象来操纵CLR时,ICLRRuntimeHost并非是该COM对象暴露的唯一接口。实际上可以通过下列任意COM接口操纵CLR。

- ICLRRuntimeHost, 允许创建和卸载AppDomain; 管理CLR的生命周期以及创建代码访问安全性(CAS)机制所需的证据。
- ICorConfiguration, 允许向CLR指定回调接口以接收警告或特定事件。这些回调接口包括: IGThreadControl、IGHostControl和IDebuggerThreadControl等。从这些接口获得的事件在粒度上要比用CLR的剖析API(稍后我们会介绍)得到的差很多。
- ICorThreadPool允许操纵进程中的线程池, 以及调整相应的配置参数。
- IGHost允许获得垃圾收集器行为的信息, 以及调整垃圾收集器的配置参数。
- IValidator允许验证程序集的PE/COFF头(peverify.exe工具使用了这个功能)。
- IMetaDataConverter允许将COM元数据(即tlb/tlh)转换成.NET元数据(tlbexp.exe工具使用了这个功能)。

如果想知道这些接口究竟有哪些方法可用, 只要查看一下文件mscoree.h、ivalidator.h和gchost.h就可以了。如果要在IClrRuntimeHost接口中取得这些接口之一, 只需使用众所周知的QueryInterface()方法, 如下所示。

```
...
ICorThreadPool * pThreadPool = NULL;
hr = pClrHost->QueryInterface( IID_ICorThreadPool,
                              (void**)&pThreadPool);
...
```

4.2.8 SQL Server 2005 运行时宿主的特性

前面我们提到过, SQL Server 2005的运行时宿主有其特殊性, 这是因为RDBMS强加了对可靠性、安全性和性能方面的要求。

这种类型的服务器程序最主要的要求就是可靠性。为了增强.NET应用程序的可靠性, CLR增加了以下三种机制——受约束执行区域(constrained execution region, CER)、关键终结器(critical finalizer)和临界区(critical region, CR)。本章稍后将详细讨论。

^① 原文为Main函数, 但代码中是MyMethod函数。——译者注

第二项要求是安全性。为防止加载恶意用户的代码，所有用户程序集将由运行时宿主从数据库中加载。这表明初始化期间必然会有管理员从数据库中预加载程序集这一动作。在这个期间，管理员可以根据程序集内代码的受信级别为程序集指定所属分类——SAFE、EXTERNAL_ACCESS和UNSAFE。这个受信级别将决定程序集拥有的代码访问安全性权限集合。最后，一些认为在.NET Framework中比较敏感的功能，例如System.Threading命名空间中的某些类，在SQL Server 2005加载的程序集中将无法使用。

第三项要求是性能。只有以最优方式利用资源才能满足该要求。从这个角度出发，最宝贵的资源就是线程和RAM中加载的内存页面。解决方法就是尽量减少线程之间的上下文切换，以及尽量减少虚拟磁盘上的内存页面。

上下文切换通常是由Windows调度器的抢占式多任务处理（preemptive multitasking）来管理的。我们将在5.3.3小节简单介绍抢占式多任务处理。而SQL Server 2005的运行时宿主基于协作式多任务处理模式实现了自己的多任务机制。在这种调度模式下，线程自己决定何时将处理器使用权转移到下一个线程。这样做的优点是线程切换的时机确定得更细致，因为它与程序执行过程的语义结合更紧密。另外一个优点是，这个模型可以使用Windows的纤程（fiber）实现。

纤程是一个逻辑线程或称为轻量级线程。Windows的一个物理线程可以将多个纤程的执行连在一起。优点是当一个纤程转换到另一个纤程的性能消耗小于一次常规的上下文切换。然而，当纤程用在SQL Server 2005的运行时宿主中时，Windows线程和.NET托管线程之间将不再保证具有对应关系。同一个托管线程在其存在期间可能会在不同的物理线程中执行。必须从认识上排除这两个实体类型之间的密切关系。托管线程和控制它的物理线程之间亲缘性的操作，包括线程本地存储、当前区域性设置和派生自WaitHandle类、互斥体、信号量或事件的Windows同步对象。注意，可以通知运行时宿主何时开始或结束一个区域，在此区域内需要使用这种亲缘性操作，只要使用Thread类的BeginThreadAffinity()和EndThreadAffinity()方法即可。使用纤程带来的性能提升一般可达20%。（注意，当前版本SQL Server 2005已经放弃了纤程模型，因为微软的工程师对它的可靠性尚不确定。该产品的将来版本可能会重新引入纤程模型。）

内存页面的存储通常是由Windows的虚拟内存系统管理的，Windows虚拟内存将在5.2.1小节描述。SQL Server 2005的运行时宿主将自己插在CLR的内存请求与操作系统的内存机制之间以便获得尽可能多的可用内存。这同时也能在CLR内存请求失败的时候提供可预知性更好的行为。稍后将会看到，这一特性是确保SQL Server 2005这样的服务器程序可靠性的一项基本功能。

上述所有新功能都可以通过允许CLR和宿主之间通信的API来实现。计划将有大约30个新增接口，现列于下表。宿主有一项职责是提供一个对象通过ICLRRuntimeHost.SetHostControl(IHostControl*)方法实现IHostControl接口。这个接口提供了一个方法GetHostManager(IID, [out]obj)，CLR调用这个方法获得一个宿主的对象，并用这个对象来代理线程管理或程序集加载等职责。更多信息请查找mscorlib.h文件中的接口定义。

职 责	由宿主实现的接口	由CLR实现的接口
加载程序集	IAHostAssemblyManager IAHostAssemblyStore IAHostSecurityManager	ICLRAssemblyReferenceList ICLRAssemblyIdentityManager
安全性	IAHostSecurityContext	ICLRHostProtectionManager
异常管理	IAHostPolicyManager	ICLRPolicyManager
内存管理	IAHostMemoryManager	ICLRMemoryNotification-
垃圾收集器	IAHostMalloc	Callback
线程控制	IAHostGCManager	ICLRGCManager

(续)

职 责	由宿主实现的接口	由CLR实现的接口
线程池	IHostTaskManager IHostTask	ICLRTaskManager ICLRTask
同步	IHostThreadPoolManager IHostSyncManager IHostCriticalSection IHostManualEvent	ICLRSyncManager
I/O完成	IHostAutoEvent IHostSemaphore	
调试	IHostIoCompletionManager	ICLRIOCompletionManager
CLR事件		ICLRDebuggerManager
	IActionOnCLREvent	ICLROnEventManager

4.3 剖析.NET 应用程序的执行状况

本节的目标是了解一个极为有用的功能——细致地剖析CLR的执行状况。换句话说，可以要求CLR在特定事件发生的时候调用非托管代码，比如JIT编译开始或者加载新的程序集时。这些回调方法必须做成非托管代码是有道理的，因为想要通过它们观察CLR的状态，显然就不能让CLR自己来处理它们。

想要使用CLR的剖析功能，必须创建一个COM类来实现ICorProfilerCallback接口。这个接口定义在corprof.h中，可以在Visual Studio安装目录的SDK\v2.0\Include下找到它。这个接口有70个左右的方法。实现了ICorProfilerCallback接口之后，需要通知CLR用此实现作回调。要想把该实现的CLSID传递给CLR，不必创建一个运行时宿主。实际上只需要正确设置Cor_Enable_Profiling和Cor_Profile两个变量即可。Cor_Enable_Profiling必须设为一个非零值以便通知CLR进行剖析回调动作。而Cor_Profile必须设为ICorProfilerCallback接口实现类的CLSID或者ProgID。

我们这里阐述得不够全面，可以参考Matt Pietrek所作的文章“The .NET Profiling API and the DNProfiling Tool”，它刊载在“MSDN Magazine”2001年12月号上（可免费下载）。我们强烈建议下载该文章所附代码，并在你的.NET应用程序进行实验。这样可以使你很好地理解CLR所做工作的范围！这篇文章的代码还会说明如何用标记来启用回调函数的某个特定子集。使用该代码的必要步骤很简单。

- 在机器上注册COM类；
- 打开一个命令行窗口；
- 使用提供的批处理文件设定正确的环境变量；
- 从命令行窗口启动应用程序。

4.4 定位和加载程序集

我们使用的程序集部署模型——私有或共享，都是由CLR定位和加载程序集运行的过程。更精确地讲，这项任务是由CLR的程序集加载器来完成的，程序集加载器通常也称为fusion。总体来说，CLR定位程序集的过程是智能的而且可以配置。

- CLR的可配置性体现在管理员可随时移动程序集的位置而始终确保CLR仍能定位它们。可配置性还意味着我们可以将正在使用的某个版本的程序集重新定向到另一个版本（可以用两个方法做到，用程序集发行者策略或者用配置文件，稍后将会介绍）。
- CLR的智能性体现在当程序集在某个文件夹下没有找到时，CLR会使用某种算法，如试探性地在子文件夹中搜索同名的程序集。CLR的智能性还体现在，如果发现一个程序原来可以正常工作而现在因为找不到某个程序集而出错，它可以轻易地将错误的更改回滚。

以上两点要求可以通过使用fusion的定位算法来达到。

我们在2.1.1节已经介绍过，程序集可以由多个称作模块的文件组成，这时程序集定位意味着定位该程序集的主模块（包含清单的那一个）。提醒读者注意的是，同一程序集的所有模块必须放在同一文件夹下。

4.4.1 CLR 何时尝试定位程序集

当一个执行中的程序集需要加载另一个程序集时，CLR的定位程序就会被调用。如果定位失败，定位程序就会报告`System.IO.FileNotFoundException`异常。发生下列情况时，将使用定位程序。

- 当AppDomain中使用`AppDomain.Load()`方法的某个重载来将某个程序集加载到该AppDomain中时。
 - 当CLR需要隐式加载某个程序集时。这将在下一节，CLR解析类型的部分讨论到。
- 发生下列情况时，将不会使用定位程序。
- 当调用本章前面讨论的`AppDomain.ExecuteAssembly()`方法时。
 - 当调用静态方法`Assembly.LoadFrom`时。

4.4.2 CLR 使用的定位算法

1. 全局程序集缓存

如果程序集的名称是强名称的话，定位算法首先查找全局程序集缓存（GAC）目录中的程序集。搜索GAC时，应用程序的用户可以（通过配置文件）对需要定位的程序集应用发行者策略。

2. CodeBase元素

如果程序集的强名称没有正确指定，或者在GAC中找不到这个程序集，那么这个程序集有可能需要从一个URL（Unique Resource Locator统一资源定位符）加载。这种可能是进行3.11节所描述的“无接触部署”机制的基础。

在应用程序的配置文件中可以有一个`<codeBase>`元素，用以进行有关程序集定位的设置。这个元素可以定义一个URL，之后就会尝试从这个URL加载程序集。如果在指定的URL无法找到所需的程序集，那么定位过程将会失败。如果要寻找的程序集在配置文件中定义有强名称，那么该URL就可以是一个Internet地址、Intranet地址或者当前计算机上的文件夹路径。如果该程序没有使用强名称定义，那么URL就只能是应用程序所在文件夹的子文件夹。

下面是从某个使用了`<codeBase>`元素的应用程序配置文件中抽取出来的片段。

```
...
  <dependentAssembly>
    <assemblyIdentity name="Foo3" publicKeyToken="C64B742BD612D74A"
      culture="en-US"/>
    <codebase version="3.0.0.0"
      href="http://www.smacchia.com/Foo3.dll"/>
  </dependentAssembly>
...
```

注意URL包含的是带有清单的程序集的模块的位置（本例中的`Foo3.dll`）。如果程序集还包含其他模块，那么注意所有模块都必须从同一个位置下载（本例是`http://www.smacchia.com/`）。

如果程序集是由`<codeBase>`元素指定并从Web上下载，那么它们会存放在下载缓存中。同样，当试图从某个URL加载程序集时，fusion会先检查缓存看所需模块是否已经下载过了。

3. 探测机制

如果没有指定程序集的强名称或者在GAC文件夹中无法找到该程序集，而且配置文件中也没有为该程序集定义相应的`<codeBase>`元素的话，定位算法就会尝试对特定文件夹的探测。下面的例子说明

了探测机制的全过程。

- 假设我们需要定位的程序集名叫**Foo**（注意我们没有提供扩展名）。
- 再假设我们在配置文件中为**Foo**程序集加入了一个<probing>元素的定义，指向“Path1”和“Path\Bin”。
- 应用程序的文件夹放在“C:\AppDir\”。
- 最后，假设该程序集没有提供任何区域设置信息。

该程序的搜索将按照以下顺序完成：

```
C:\AppDir\Foo.dll
C:\AppDir\Foo\Foo.dll
C:\AppDir\Path1\Foo.dll
C:\AppDir\Path1\Foo\Foo.dll
C:\AppDir\Path2\Bin\Foo.dll
C:\AppDir\Path2\Bin\Foo\Foo.dll
C:\AppDir\Foo.exe
C:\AppDir\Foo\Foo.exe
C:\AppDir\Path1\Foo.exe
C:\AppDir\Path1\Foo\Foo.exe
C:\AppDir\Path2\Bin\Foo.exe
C:\AppDir\Path2\Bin\Foo\Foo.exe
```

如果该程序集是一个卫星程序集（即不是区域无关的程序集，如“fr-FR”），那么文件夹的搜索顺序如下所示。

```
C:\AppDir\fr-FR\Foo.dll
C:\AppDir\fr-FR\Foo\Foo.dll
C:\AppDir\Path1\fr-FR\Foo.dll
C:\AppDir\Path1\fr-FR\Foo\Foo.dll
C:\AppDir\Path2\Bin\fr-FR\Foo.dll
C:\AppDir\Path2\Bin\fr-FR\Foo\Foo.dll
C:\AppDir\fr-FR\Foo.exe
C:\AppDir\fr-FR\Foo\Foo.exe
C:\AppDir\Path1\fr-FR\Foo.exe
C:\AppDir\Path1\fr-FR\Foo\Foo.exe
C:\AppDir\Path2\Bin\fr-FR\Foo.exe
C:\AppDir\Path2\Bin\fr-FR\Foo\Foo.exe
```

4. AppDomain.AssemblyResolve事件

最后，如果上述步骤都没有找到所需的程序集，CLR会触发AppDomain类的AssemblyResolve事件。这个事件所委托的方法可以返回一个Assembly类型的对象。这样就可以实现程序集定位机制了。这个特性明显地用在了ClickOnce部署技术当中来按需下载文件，这在本书3.10.4小节有所讨论。

可以用fuslogvw.exe来分析fusion机制所产生的日志。这个工具可以用来诊断程序集加载失败的确切原因。

4.4.3 配置文件的<assemblyBinding>元素

程序集ASM的配置文件可以包含在ASM引发对另外一个程序集的定位及加载操作时供fusion机制使用的参数。配置文件可以包含一个<probing>元素来指定探测机制需要搜索的一个或多个文件夹。这个文件还可以为所有潜在地需要定位的程序集定义一个<dependentAssembly>元素。每个<dependentAssembly>元素可以包含下列关于如何定位程序集的信息。

- <publisherPolicy>元素决定在定位程序集时是否需要考虑程序集可能的发行者策略。
- <codeBase>元素，4.4.2节已经描述过它，可以用来定义需要下载程序集的URL。

- **<bindingRedirect>**元素可以让你重定向版本号，它的作用就如同发行者策略程序集一样。发行者策略作用的范围可以是共享程序集的所有客户应用程序，但只有配置文件所对应的应用程序才真正受影响。

下面是一个配置文件的例子（注意和发行者策略程序集XML文件的相似性）。

例4-10

```
<?xml version="1.0" encoding="utf-8"?>
<configuration>
  <runtime>
    <assemblyBinding xmlns="urn:schemas-microsoft-com:asm.v1">
      <probing privatePath="Path1;Path2/bin" />
      <dependentAssembly>
        <assemblyIdentity name="Foo1" culture="neutral"
          publicKeyToken="C64B742BD612D74A" />
        <publiherPolicy apply="no" />
        <bindingRedirect oldVersion="1.0.0.0" newVersion="2.0.0.0"/>
      </dependentAssembly>
      <dependentAssembly>
        <assemblyIdentity name="Foo2" culture="neutral"
          publicKeyToken="C64B742BD612D74A" />
        <codebase version="2.0.0.0"
          href="file:///C:/Code/Foo2.dll"/>
      </dependentAssembly>
      <dependentAssembly>
        <assemblyIdentity name="Foo3" culture="fr-FR"
          publicKeyToken="C64B742BD612D74A" />
        <codebase version="3.0.0.0"
          href="http://www.smacchia.com/Foo3.dll"/>
      </dependentAssembly>
    </assemblyBinding>
  </runtime>
</configuration>
```

如果不想编写XML文件，那么请记住这些信息也可以通过.NET Framework Configuration工具进行配置，可以从开始>控制面板>管理工具>Microsoft .NET Framework 2.0 Configuration>configured assembly菜单找到这个工具。

4.4.4 定位算法示意图

定位算法示意图如图4-2所示。

4.4.5 影子复制机制

当一个程序集加载到一个进程中之后，对应的程序集文件就会自动被Windows加锁。就是说除了可以重命名之外，无法对其进行更新或者删除。ASP.NET服务器的一个进程可能会承载许多应用程序，在这种情况下加锁机制就会带来一个十分麻烦的后果，那就是任何一个应用程序需要更新时，整个ASP.NET进程都需要重新启动。

所幸的是CLR加载程序集时可以采用一种称为影子复制（shadow copy）的机制。在这种机制下，应用程序所需的程序集将在正式加载前被复制到另外一个缓存文件夹中。所以原始文件在执行中也可以进行更新。ASP.NET使用了这个特性并且周期性地检查加载过的程序集是否有更新。如果已经更新，那么应用程序可在不丢失任何请求的前提下重新启动。

使用AppDomainSetup类的string ShadowCopyDirectories{get; set;}属性可以指定包含程序集影子复件的文件夹。

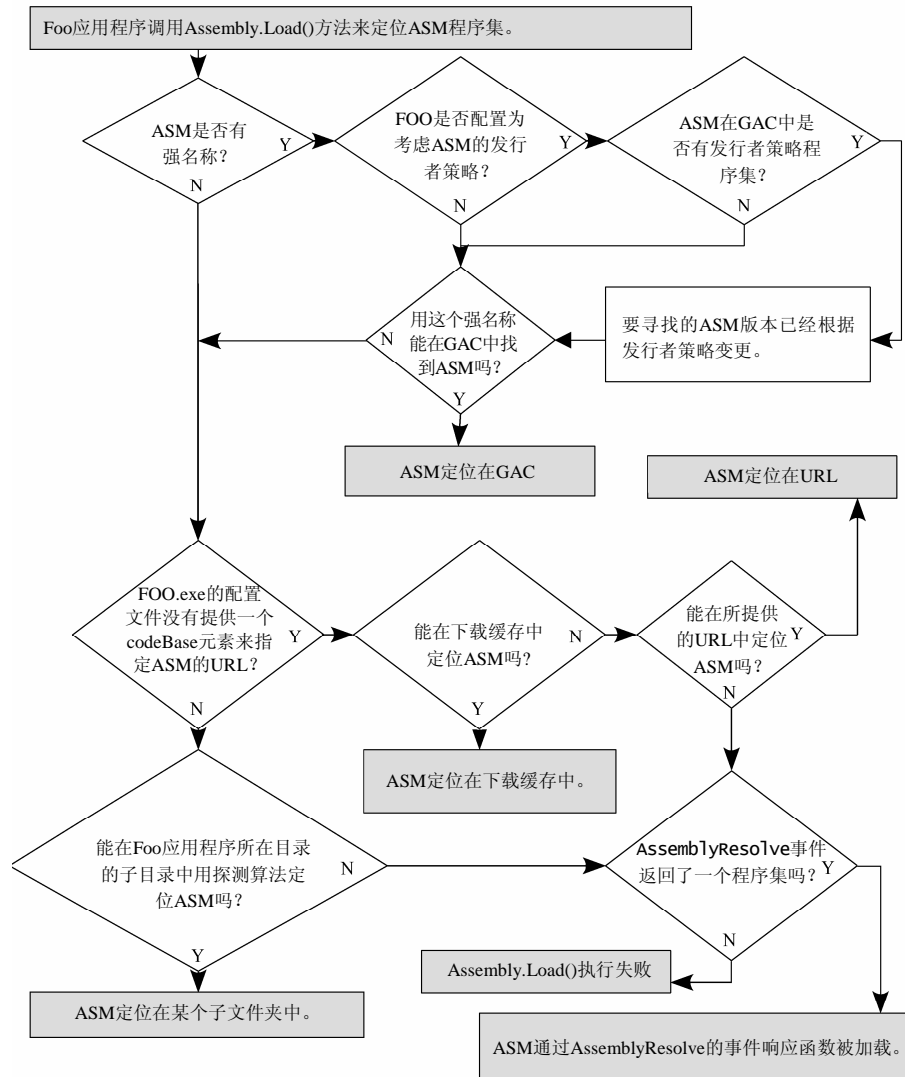


图4-2 定位算法示意图

4.5 运行时类型解析

4.5.1 显式或隐式加载程序集

我们常常需要在代码中使用在其他程序集中声明的类型。要使用定义在其他程序集中的类型有两种方法:

- 可以让编译器早期绑定所需的类型, 然后让CLR在需要的时候自动加载所需的程序集。这称作隐式加载。
- 还可以显式加载程序集, 然后后期绑定所需要的类型。

无论哪一种方法，这个任务都是由CLR中一个称作类加载器的组成部分来完成的。程序集的隐式加载发生在使用了该程序集类型的方法被第一次JIT编译的时候。

隐式加载的概念与DLL的加载机制有关。同样，显式加载的概念类似于COM对象的Automation机制和著名的IDispatch接口。

4.5.2 编译时引用程序集

当程序集A执行并且JIT编译A中一个需要程序集B中某个类型的方法时，若B已经在A编译的时候被A引用，那么程序集B将被CLR隐式加载。A引用B的操作必须在编译时经由以下两种方法之一完成：

- 第一种方法：直接或者通过编译脚本使用 **csc.exe** 命令行编译器编译A。必须正确使用 **/reference**、**/r** 以及 **/lib** 编译选项。
- 第二种方法：用 Visual Studio 中 Reference > AddReference 菜单项设置正确的引用。Visual Studio 环境是在后台使用 **csc.exe** 编译器在 C# 源代码中生成程序集的。环境选项的设置基本上会成为 Visual Studio 使用 **csc.exe** 编译器时所用到的 **/reference**、**/r** 以及 **/lib** 选项。

无论哪一种方法都必须指定要加载的程序集的名称（强名称或普通名称）。被程序集A引用的程序集B的所有类型和成员都会出现在A的TypeRef和MemberRef表中。此外程序集B本身也会出现在A的AssemblyRef表中。一个程序集可以引用多个其他程序集，但是必须要注意避免循环引用（A引用B，B引用C，C回过头来又引用A）Visual Studio可以检测并防止循环引用。还可以用NDepend工具（参见本书附录D最后一则）来检测程序集循环引用。

4.5.3 示例

下面是一个完整的例子，CLR隐式加载程序集所需的工作每一步都做到位了。第一段代码定义了被第二段代码所引用的程序集。

例4-11 被引用程序集的代码AsmLibrary.cs

```
using System;
namespace MyTypes {
    public class FooClass {
        public static int Sum(int a,int b){return a+b;}
    }
}
```

例4-12 引用第一个程序集的代码AsmExecutable.cs

```
using System;
using MyTypes;
class Program {
    static void Main() {
        int i = FooClass.Sum(3,4);
    }
}
```

注意MyType命名空间在第二段代码中的使用。我们还可以将FooClass和Program类放到同一个命名空间，或者放到匿名命名空间。若这样做的话，我们还能说明同一个命名空间可分布到多个程序集中。

用ildasm.exe工具来分析引用程序集的清单部分是很有趣的。可以清晰地看到引用了AsmLibrary程序集。下面的引用信息摘自AssemblyRef表的一项。

```
...
.assembly extern 'AsmLibrary'{
    .ver 0:0:0:0
}
...
```

再看一眼Main()函数的IL代码也是很有趣的。可以发现Sum()这个方法确实是来自另一个叫“AsmLibrary”的程序集。

```
.method private hidebysig static void Main() cil managed
{
    .entrypoint
    // Code size      9 (0x9)
    .maxstack 2
    .locals init (int32 V_0)
    IL_0000: ldc.i4.3
    IL_0001: ldc.i4.4
    IL_0002: call     int32
                ['AsmLibrary']MyTypes.FooClass::Sum(int32,int32)
    IL_0007: stloc.0
    IL_0008: ret
} // end of method Program::Main
```

4.5.4 类型解析算法示意图

类型解析算法示意图如图4-3所示。

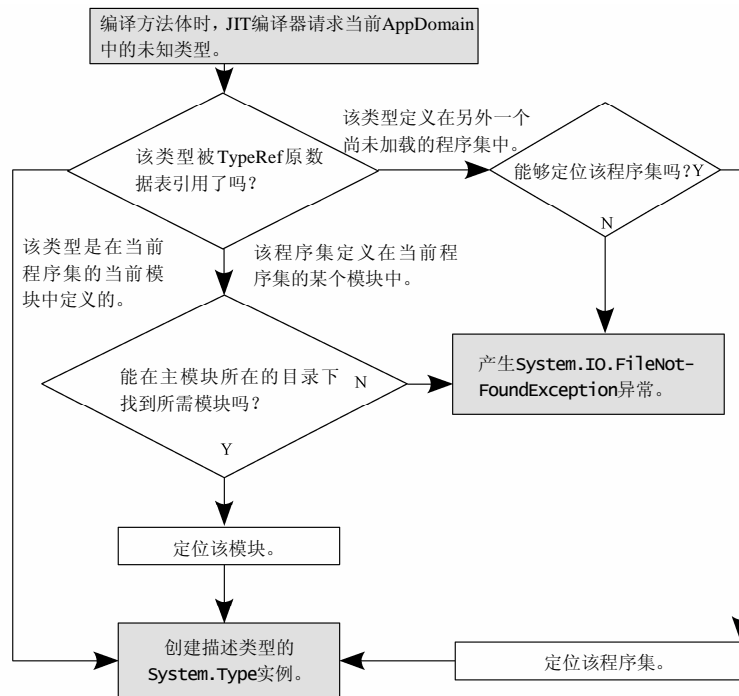


图4-3 类型解析

4.6 JIT（即时）编译

4.6.1 可移植的二进制代码

请注意，不管用何种语言的源代码写成，所有.NET程序都是编译成IL代码的。IL代码储存在程序集或模块的特殊区域。代码执行之前，IL代码都保持这一形态。

正如名称所暗示的, IL (中间语言) 是高级编程语言 (C#、VB.NET等) 和机器语言的中间形态。它的想法是在应用程序执行的时候才将IL代码编译为本地机器代码。这样做就可以让.NET应用程序以一种和计算机 (及操作系统) 无关的形式发布。主要目的是让.NET程序发布时实现跨操作系统平台, 而无需从高级语言代码重新编译。因此我们称.NET应用程序为可移植的二进制代码。

4.6.2 即时编译技术简介

将IL代码编译为机器语言的过程是在程序运行的时候才进行的。我们可以想象两种与IL编译有关的情景:

- 应用程序一启动, 就彻底编译成机器语言。
- IL指令像解释语言那样一条一条地解释执行。

以上任何一种情景都不是IL编译为机器语言的真实做法。实际的做法是介于以上两种做法之间更为高效的做法。按照实际的做法, 方法体的IL语言将在方法第一次调用前夕编译为本地机器语言代码。这就是该运行时编译的过程称作“即时 (Just in Time, JIT) 编译”的原因。方法将在执行时即时编译为机器语言。

1. 用存根截获方法的首次调用

为了完成JIT编译, CLR使用了一个在RPC、DCOM、.NET Remoting及Corba等分布式架构中常用的小技巧。每次当CLR加载一个类时, 为类中的每个方法都分配一个存根 (stub)。在分布式架构中, 存根是客户端程序的一个软件层, 用来截获方法调用并将它们转化为远程调用 (DCOM用的是代理)。而在JIT编译的过程中, 存根是一个用来截获方法首次调用的代码层。

存根代码的功能基本上是启动JIT编译器来验证IL代码, 并且将IL代码翻成本地机器指令。本地代码将会存储在进程的内存空间中, 之后将被执行。当然, 之后存根会被替换成跳转到生成代码所在内存地址的代码, 确保每个方法仅被编译一次。

2. IL代码的验证

在方法编译成本地机器语言之前, JIT编译器会对其进行一系列的检查以确保方法体是合法的。要检查一个方法是否是合法的, 编译器会检查IL指令流并验证运算栈的内容以找出内存非法访问等问题。如果代码验证失败, JIT编译器会报告一个异常。

在安全模式下编写的C#代码将自动被翻译成可验证的IL代码。然而在14.1节我们会介绍特殊条件下C#编译器能够产生不能被JIT编译器所验证的特殊IL代码。

3. JIT编译器进行的优化

JIT编译器将会对代码实施一些优化。以下是一些例子来告诉你怎么做:

- 为了避免传递参数带来的损耗, JIT编译器可能会将方法体的代码插入到调用这个方法的另一个方法体内。这种优化称为内联化 (inlining)。为了确保该优化本身的损耗不超过优化所得的性能收益, 内联化的方法必须满足一些条件。这种方法编译后的大小必须小于32字节; 该方法还不能捕获异常, 不能包含循环, 也不能是虚函数。
- JIT编译器能将所有引用类型的本地变量在最后一次使用之后设为空引用。这样可以减少对象的引用计数, 以便有机会让垃圾收集器尽早收集, 给其他对象留出更大空间。注意该优化会在调用System.GC.KeepAlive()方法之后局部失效。
- JIT编译器可以将频繁使用的本地变量直接存放到处理器的寄存器中, 而不用栈来储存它们。这会带来显著的优化效果, 因为访问寄存器要比访问栈的效率高得多。这项优化通常称作寄存器化 (enregistration)。

4. 抛弃机制介绍

JIT编译方法是需要消耗内存的，因为方法的本地代码需要存储空间。如果CLR检测到应用程序的内存面临用尽，就可以通过释放一些方法的本地代码存储空间来恢复内存。很自然，这些被释放方法的存根将重新生成。这种功能称作抛弃机制（pitching）。

抛弃机制的实现比看上去要复杂一些。为了更加高效，释放的代码必须存储在连续空间以免产生内存碎片。还有一个重要的问题出现在存有指向本地代码内存地址的线程栈上，所有这些复杂因素对开发人员来说都是暗箱操作。

JIT产生的方法本地代码将在应用程序的AppDomain卸载时销毁。

5. JIT和JITA缩写

那些从COM/DCOM/COM+世界过来的读者见到JIT这个缩写可能会想到JITA（Just In Time Activation，在8.7.1节描述）这个缩写。这两项机制是不同的，它们有不同的目的而且所用的技术基本上也不一样。但是底层的想法是一样的：都是让一种实体的动员（在JIT中为方法IL的编译；在JITA中是COM对象的激活）仅当实体被请求时（即在使用之前）才进行。我们通常可以用“惰性”这个词来描述此类机制。

4.6.3 ngen.exe 工具

微软提供了一个名叫ngen.exe的工具，能够在程序集执行之前将它们编译。此工具的功能就像一个标准编译器，用以取代JIT机制。ngen.exe的真名是“本地映像生成器”。当发现JIT机制引入了严重的性能下降，通常是导致引用程序启动缓慢时就可以使用ngen.exe这个工具。不过要注意正常的编译器能进行许多ngen.exe所不能进行的优化。所以通常更倾向于使用JIT编译器。

通常在应用程序安装的时候完成ngen.exe的编译。你无需操纵存有本地代码的新文件——本地映像文件。本地映像文件自动储存在计算机的一个特殊文件夹中，该文件夹称为本地映像缓存。它可以通过ngen.exe的/show和/delete选项来访问。这个文件夹同样可以在用户查看全局程序集缓存的时候看到，因为本地映像包含在这个文件夹中。查看此文件夹的方法请见3.5.5节。

ngen.exe选项	描 述
/show [程序集名 文件夹名]	允许查看“本地映像缓存”中的本地映像列表 如果在此选项之后接程序集的名字，将只列出同名的映像 如果在此选项之后接文件夹名，将只列出此文件夹中的映像
/delete [程序集名 文件夹名]	删除“本地映像缓存”中的所有映像 如果在此选项之后接程序集的名字，将只删除同名的映像 如果在此选项之后接文件夹名，将只删除此文件夹中的映像
/debug	产生可供调试器使用的映像

此编译器工具还有许多其他选项，MSDN文档中以“Native Image Generator (Ngen.exe)”为标题的文档具有这些选项的完整列表。注意，.NET2.0的版本支持许多新功能，可支持使用反射技术的程序集以及在程序集依赖项改变之后自动更新其编译版本，等等。要获取更多信息，可阅读在线文章——Reid Wilkes的“NGen Revs Up Your Performance with Powerful New Features”，刊载在2005年4月号的“MSDN Magazine”上。

4.6.4 性能计数器与 JIT 编译

在“.NET CLR Jit”分类下有6个与JIT有关性能计数器。

计数器名称字符串	描 述
"# of IL Methods JITted"	已编译方法的计数。该计数不包含用ngen.exe编译的方法数
"# of IL Bytes JITted"	已编译的IL代码字节数。已抛弃的方法不会从这个总数中减去
"Total # of IL Bytes Jitted"	减去已抛弃方法的已编译IL代码字节数
"% Time in Jit"	JIT编译器花费时间所占的百分比。这个计数在每次JIT编译之后更新
"IL Bytes Jitted / sec"	平均每秒所编译的IL代码量
"Standard Jit Failures"	检测到非法方法并且没有进行JIT编译的次数

可以选择查看所有托管应用程序自计算机启动以来执行的性能计数。只要通过PerformanceCounterCategory.GetCounters()方法的参数即可进行选择。第一种做法是用"_Global_"字符串作为这个方法的参数；第二种做法是用需要观察的进程名作为这个方法的参数。

在评估JIT编译过程的性能消耗时，这些性能计数器十分有用。如果这项消耗看起来太高就可以考虑使用ngen.exe工具来部署应用程序。下面的例子演示了这些性能计数器的用法（注意这里的程序集叫MyAssembly.exe）。

例4-13 MyAssembly.cs

```
using System.Diagnostics;
class Program {
    static void DisplayJITCounters() {
        PerformanceCounterCategory perfCategory
            = new PerformanceCounterCategory(".NET CLR Jit");
        PerformanceCounter[] perfCounters;
        perfCounters = perfCategory.GetCounters("MyAssembly");
        foreach(PerformanceCounter perfCounter in perfCounters)
            System.Console.WriteLine("{0}:{1}",
                perfCounter.CounterName,
                perfCounter.NextValue());
    }
    static void f() {
        System.Console.WriteLine("----> Calling f().");
    }
    static void Main() {
        DisplayJITCounters();
        f();
        DisplayJITCounters();
    }
}
```

该程序将显示：

```
# of Methods Jitted:2
# of IL Bytes Jitted:108
Total # of IL Bytes Jitted:108
IL Bytes Jitted / sec:0
Standard Jit Failures:0
% Time in Jit:0
Not Displayed:0
----> Calling f().
# of Methods Jitted:3
# of IL Bytes Jitted:120
Total # of IL Bytes Jitted:120
IL Bytes Jitted / sec:0
```

```
Standard Jit Failures:0
% Time in Jit:0,02865955
Not Displayed:0
```

更确切地说,我们还可以通过`perfmon.exe`工具来查看性能计数器,这个工具可以这样打开:开始菜单 > 运行……> `perfmon.exe`。

4.7 垃圾收集器和托管堆

4.7.1 垃圾收集技术简介

在.NET语言中,析构托管对象的职责并不是由程序员来完成。值类型的析构问题容易解决,分配在当前线程栈上的对象,在退栈清空时自动销毁。而真正的问题出在引用类型身上。为了简化开发人员的任务,.NET平台提供了垃圾收集器(或简称GC),它将自动承担回收为对象分配的内存。这就是所谓的托管堆。

如果一个对象不再有别的引用,它对程序来说就是不可访问的,也就不再需要它了。GC会标记所有可访问的对象。当一个对象没有标记为可访问,那就意味着它在程序中不能被访问而且应当销毁。不过存在一个问题就是实际上GC仅在决定这样做时(一般是当程序需要内存时)才会解除分配给对象的内存。开发者可以对GC进行有限的控制。与其他语言,如C++等相比,这在程序的执行期间引入了一层不定因素。一个直接结果就是程序员感到不能拥有彻底的控制权而失望。然而在垃圾收集器的帮助下,以往经常出现的诸如内存泄露等问题确实大大减少了。



尽管已经有GC,内存泄露还是有可能发生。实际上,开发人员可能因为保持无用对象的引用而造成内存泄露(比如没有清空的集合)。不过,.NET程序大多数常见的内存泄露都是因为删除非托管资源引起的。

请记住GC是CLR中的一个软件层,每个进程中都存在一个独立的实例。

4.7.2 垃圾收集算法遇到的问题

我们此处的目标不是探讨各种现存的垃圾收集算法,而是希望你认识到GC的设计人员需要面对的问题。这样,就可以更好地了解为什么下面各节所描述的算法被微软选中来实现.NET的垃圾收集机制。

你可能会以为这是一件很简单的事情,或者等对象不再被引用时释放掉不就完了。但是这种过分单纯的方法很容易导致算法失败。比如,设想有两个对象A和B,A中有一个指向B的引用,B中有一个指向A的引用,除了它们之间的交叉引用之外这两个对象没有任何其他引用。很明显,程序不再需要这两个对象,即使它们仍有引用GC也应当销毁它们。GC必须使用引用树和循环引用检测算法来解决这些问题。

除了对象的销毁之外还有另外一个问题。就是GC必须要避免堆碎片。大小不同的内存区域经过一段时间的分配和解除分配后,堆便会成为碎片。许多程序不再使用的内存空间使得整个内存很乱,导致内存的浪费。GC的一个任务就是清理堆碎片从而减少内存碎片的产生。有多个算法来完成这项任务。

最后,常识和经验告诉我们一个规则:越老的对象生命期越长,越新的对象生命期就越短。如果将这个规则用于GC的回收算法设计,就是要优先解除分配更新的对象而不是更老的对象,那么就可获得性能的提升。

.NET 2.0的GC在实现时已考虑到上述问题和经验规则。

4.7.3 .NET 的 GC

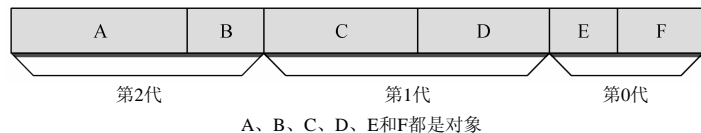
GC对对象的收集动作是由CLR监测到内存不足时自动引发的。微软的文档并没有介绍判断这一时机具体算法。

术语代（generation）定义了两收集动作的时间间隔。每个对象都属于创建它的代。此外，下面这个公式总是成立的。

$$\{\text{进程中代的数目}\} = 1 + \{\text{进程中收集动作的数目}\}$$

代是以数字的方式表现的，这个数字在每次收集动作之后都会增加，直到达到代的上限为止（目前CLR的实现中，该上限为2）。按照约定，第0代总是最“年轻”的对象，就是说刚分派到堆中的对象会成为第0代对象的一员。

这一机制的结果就是属于同一个代的对象在内存中都是紧邻存放的，如图4-4所示。



A、B、C、D、E和F都是对象

图4-4 对象的代

4.7.4 第一步：寻找根对象

活动对象（即不能解除分配的对象）的引用树的根是那些由静态字段引用的对象、线程栈中引用的对象以及物理地址（或物理地址的偏移量）保存在处理器的某个寄存器里的对象。

4.7.5 第二步：建立活动对象树

GC会从根对象开始建立这棵树，方法是把所有已经在树上的对象所引用的对象添加到树中，并重复这一过程直到无法再加入新的引用。所有被这棵树所引用的对象都标记为活动对象。如果一个对象已经被标记为活动的，这个算法就不会再次考虑这个对象以免在树中出现循环引用。GC从类型元数据中获得类型的定义，并根据它寻找对象中的引用。

在图4-5中，对象A和B为对象树的根，而且A和B同时引用了C。B还引用了E；C引用了F。对象D则没有标记为活动对象。

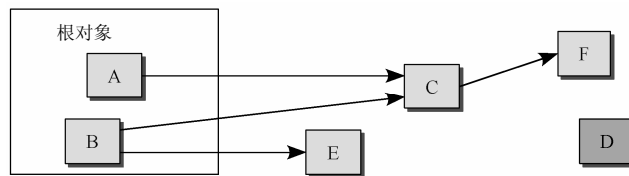


图4-5 引用对象树

4.7.6 第三步：解除分配非活动对象

GC线性遍历堆，将所有未标记为活动的对象解除分配。某些对象的Finalize()方法需被调用以便物理销毁。这个方法也称作终结器（finalizer），定义在Object类中。如果对象所属的类重写了这个方法，那么就必须必须在对象被销毁之前调用这个方法。调用终结器的任务由另一个专用的线程来完成，以便不重载负责对象收集的线程。因此，需要调用终结器的对象的内存分配将存在于一次收集动作执行的整个过程中进行。

在以下两种情况中GC并不需要遍历整个堆：第一种是GC已经收集到足够的内存；第二种是需要进行部分收集时。部分收集对树的遍历算法有很大影响，因为有可能老对象（第2代）引用了较新的对象（第0代或第1代）。尽管收集动作的触发频率与具体应用程序有关，但可以基于以下数量级考虑：每秒进行一次第0代不完全收集；每进行10次0代收集才进行一次第1代不完全收集；而每进行10次1代收集才进行一次完整的收集。

图4-6中展示对象D没有被标记为活动的，因此它将被销毁。



图4-6 解除分配非活动对象

4.7.7 第四步：清理堆碎片

GC清理堆碎片，意思是GC会将活动对象移动到堆底部，以填补上一步解除分配对象所留下的内存空洞。堆顶的地址将重新计算，而每一代的数字均增加。越老的对象越靠近堆底，而越新的对象则越靠近堆顶。进一步说，两个同时创建的活动对象，在内存中也会紧挨着存放。GC常常仅检查最新的对象，也就是说要探测的几乎总是同一个内存页，这样能获得最大限度的性能。

图4-7（继续图4-6）中，GC已经提升了代的数字。假设对象D的类没有终结器，这意味着对象D的内存已经解除分配，而且堆碎片已经清理过了（E和F已经移动并填充了D留下的内存空洞）。注意A和B所在的代并没有增加，因为它们已经是在第2代了。

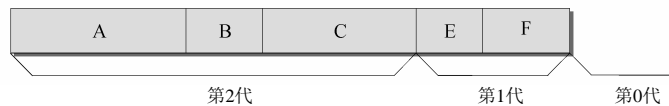


图4-7 清理堆碎片

有一些不能移动的特殊对象（这些对象也称为定址对象），它们的物理地址不能被GC所修改。如果一个对象被未受保护的代码或非托管代码引用，那么可以考虑将其定址。这个问题的具体信息可参见14.2.8节。

4.7.8 第五步：重新计算托管引用所使用的物理地址

某些对象的内存地址在上一个步骤中已经改变了，接下来GC必须遍历活动对象树，并用新的物理地址更新每个对象的引用。

4.7.9 推荐做法

下面是一些从上述算法中总结出来的推荐做法。

- 尽早释放对象，以免它们提升到更高的代中。
- 找出所有生命期很长的对象，分析它们生命期过长的原因以便尝试减少它们的生命期。我们建议用微软提供的CLR Profiler工具来进行此项分析，也可以使用Visual Studio Team System所提供的剖析工具。
- 只要可能，就避免在生命期较长的对象内引用生命期短的对象。
- 避免实现终结器，以免对象推迟收集。
- 尽早将引用设为null，特别是在调用一个大方法之前。

4.7.10 针对大对象的特殊堆

所有大小在特定临界值之下的对象都将按照前面各节所介绍的那样储存在托管堆中。不过微软没有有在文档中明确这个具体临界值，我们估计它的数量级大约是20到100KB。因性能原因，大小超过这个临界值的对象会储存在一个特殊堆中。GC 不会移动这个堆中的对象。Windows内存页面的大小根据处理器不同可能是4K或者8K。在这个特殊堆中的对象总是储存在页面大小整数倍的空间中，即使对象的真实大小并不确切是页面大小的整数倍。这样会导致少许内存浪费，但一般不会对性能造成很大影响。至于多大的对象才会储存在大对象堆的具体实现对开发人员来说是透明的。

4.7.11 多线程环境下的垃圾收集

如果手动触发，GC的收集动作可以在一个应用程序的线程中执行；或者也可以在CLR决定需要垃圾收集时在CLR的线程中执行。开始收集以前，CLR必须先将其他应用程序线程挂起，以免在收集过程中出现栈的改变。为了做到这一点，现有好几种技术。我们说JIT编译器可以在代码中插入安全点(safe point)，允许程序在此查看是否有收集动作等待执行。注意第0代的堆通常划分为若干部分（称为arena），每个线程负责一个，以避免对堆的并发访问带来的同步问题。最后提醒，终结器是由另一个专门的CLR线程来执行的。

4.7.12 弱引用

1. 潜在的问题

在应用程序执行期间，每个对象在任意时刻要么是活动的，即表示引用程序有指向它的引用；要么它就是非活动的。当程序释放了指向对象的最后一个引用，那么该对象就从活动状态转为非活动状态，就没有可能继续访问这个对象了。

事实上，在活动与非活动之间还存在第三种中间状态。如果对象处在这种状态，应用程序还可以访问该对象，而GC也可以随时释放它。这明显是一种矛盾的状态，因为对象还可以访问就表示它还有引用，而还有引用的话就应该不能销毁它。为了解决这个矛盾，我们需要引入弱引用(weak reference)的概念。当对象是通过弱引用而引用的，就既可以被应用程序访问，又可以被GC回收。

2. 使用弱引用的原因和时机

如果满足以下所有条件，开发人员就可以使用弱引用。

- 对象稍后可能会被使用，但是不确定。如果确定稍后一定会使用，那就应该用强引用。
- 如果需要，对象可以重新构造出来（比如从数据库构造）。如果稍后有可能用到对象，但是无法重建它，就不能让GC把它销毁。
- 对象占相对比较大的内存（若干KB）。如果对象十分轻量，就可以保持在内存里。不过若前两个条件适用于大量的轻量对象，也最好对每个对象都使用弱引用。

所有这些条件都是十分理论化的。实践当中我们说弱引用相当于对象的缓存。实际上这些条件对缓存中的对象来说都是完全能够满足的（我们这里讲的缓存是指概念上的缓存，而不是指某种具体实现）。

使用缓存可以看成是一种在内存消耗、处理器资源和网络带宽的使用之间自然而且自动的平衡。当缓存消耗太多内存时，就可以销毁其中一部分对象。而假如我们稍后又访问这些对象，就需要使用处理器和带宽资源将所需的对象重新构造出来。

综上所述，如果需要一个缓存，我们推荐使用弱引用。

3. 如何使用弱引用

弱引用需要通过System.WeakReference类的帮助才能使用。与其进行冗长的讨论不如给出一个例子，请看以下C#代码。

例4-14

```

class Program {
    public static void Main() {
        // 'obj' is a strong reference on the object
        // created by this line.
        object obj = new object();

        // 'wobj' is a weak reference on our object.
        System.WeakReference wobj = new System.WeakReference(obj);

        obj = null; // Discard the strong reference 'obj'.
        // ...
        // Here, our object might potentially be deallocated by the GC.
        // ...
        // Build a strong reference from the weak reference.
        obj = wobj.Target;
        if (obj == null) {
            // If the thread pass here, it means that the object
            // has been deallocated by the GC!
        }
        else {
            // If the thread pass here, it means that the object
            // hasn't been deallocated by the GC. We can thus use it.
        }
    }
}

```

WeakReference类提供了一个**bool IsAlive()**方法, 如果弱引用所指的對象已被銷毀, 这个方法就返回**false**。此外还推荐在弱引用创建之后尽快将该对象所有强引用都置为**null**以确保銷毀它的所有强引用。

4. 短弱引用和长弱引用

WeakReference类有两个构造函数。

```

WeakReference(object target);
WeakReference(object target, bool trackResurrection);

```

如果使用第一个构造函数, 那么**trackResurrection**参数将默认设为**false**。如果这个参数设为**true**, 那么应用程序仍能在对象的**Finalize()**方法调用之后, 而对象的内存还没有真正发生改变(通常是指清理堆碎片时将其他对象复制到这一对象所在的内存位置)的这段时间内访问这个对象。这种情况我们称之为长弱引用。如果该参数设为**false**, 那么一旦**Finalize()**方法调用完毕, 引用程序就不能再访问这个对象。这种情况称为短弱引用。

尽管使用长弱引用可能会获得一些便利, 但最好避免使用它们, 因为长弱引用维护起来十分困难。实际上, 若**Finalize()**方法的实现没有对长弱引用作出解释, 那么它将可能导致处于无效状态的对象复苏。

4.7.13 使用 **System.GC** 类影响 GC 的行为

我们可以使用**System.GC**类的静态方法来更改或分析GC的行为。主要的目的是提升应用程序的性能。不过微软早已为优化.NET的GC投入了很大心血。所以我们推荐仅在确信可以获得性能优势的时候才使用**System.GC**类提供的功能。下面列出了该类的一些静态属性和方法。

- **static int MaxGeneration{ get; }**, 该属性返回托管堆最高代的数字。默认情况下在当前版本的.NET中该属性返回2, 而且在整个应用程序生命期内保持为常数。

□ `static void WaitForPendingFinalizers()`，该方法将挂起当前线程，直到所有等待执行的终结器全部执行完毕为止。

□ `static void Collect()`和`static void Collect(int generation)`，该方法指示GC开始一次收集动作。用这个方法可能触发一次部分收集，因为GC收集第0代到第`generation`代之间的对象。这个`generation`参数不能超过`MaxGeneration`。如果调用没有参数的重载方法，那么垃圾收集器将收集所有各代的对象。

调用这个方法通常都预示着不好的行为，因为开发人员希望它能解决糟糕设计所带来的内存问题（比如分配太多对象，对象之间太多引用以及内存泄露，等等）。

不过在调用一些关键函数之前引发一次收集动作也不失为一个好主意，这样就不会让这个关键函数被内存耗尽或者意料之外的收集动作所打扰。这种情况下，推荐按照下面的顺序调用以确保获取最大限度的可用内存。

```
// Trigger a first collection.
GC.Collect()
// Wait for all pending finalisers.
GC.WaitForPendingFinalizers()
// Trigger a second collection to get back the memory
// used by discarded objects.
GC.Collect()
```

□ `static int CollectionCount(int generation)`，返回对指定的代所进行的收集次数。这个方法可以用来监测某段特定的时间内有没有发生过收集动作。

□ `static int GetGeneration(object obj)`和`static int GetGeneration(WeakReference wo)`，返回由强引用或弱引用所引对象的代号。

□ `static void AddMemoryPressure(long pressure)`和`static void RemoveMemoryPressure(long pressure)`提供这两个方法是因为GC的算法实际上并没有考虑到非托管内存。想象一下，假如有32个`Bitmap`类的实例，每个实例占32字节；而每个实例又含有一个指向6MB大小的位图的引用。如果不用这两个方法，GC会认为只分配了32×32字节内存，因而可能不会觉得有必要进行一次收集动作。比较好的做法是，如果一个类需要维护较大块的内存，那么就在它的构造函数和终结器中分别调用这两个函数。还要提到一点，就是在8.3.2节将要讨论的`HandleCollector`类也提供类似的服务。

□ `static long GetTotalMemory(bool forceFullCollection)`，返回托管堆当前预计的大小，以字节为单位。可以通过指定`forceFullCollection`参数为`true`来获得更精确的结果，这样该方法就会在有收集动作时阻塞执行。当收集动作完成或者等待时间超过一定限度之后，该方法就会返回。GC完成自己的任务之后，这个函数所返回的值就会更准确。

□ `static void KeepAlive(object obj)`，确保调用`KeepAlive`的方法执行期间，`obj`对象不会被GC销毁。`KeepAlive()`必须在方法体的结尾调用。你可能认为这个方法没什么用，因为它要求传入一个对象的强引用，因此在调用之前对象必定存在。实际上JIT编译器会对本地代码进行优化，使得所有本地引用变量在最后一次使用之后都置为空值。这个`KeepAlive()`就是简单地关闭这一优化。

□ `static void SuppressFinalize(object obj)`，当对象作为参数传递给这个方法之后，它的`Finalize()`方法就不会再被GC调用。本来GC确保在进程结束之前，所有支持终结器的对象其`Finalize()`方法都会被调用的。

`Finalize()`方法包含的代码逻辑上应当是解除对象所分配的资源。不过，由于我们无法控制

调用`Finalize()`方法的时机,所以经常会另外创建一个专门的方法来解除资源的分配,以便在我们想调用的时候调用。通常我们用`IDisposable.Dispose()`方法来达到这个目的。在这个方法里应当调用`SuppressFinalize()`方法,因为一旦调用了`Dispose()`,再调用`Finalize()`就没有意义了。这个话题还将在11.13.1节继续讨论。

❑ `static void ReRegisterForFinalize(object obj)`, 传入该方法的对象,其`Finalize()`方法将会在垃圾收集时被GC调用。这个方法主要在以下两种情况下使用。

- 第一种情况: 如果已经调用过`SuppressFinalize()`方法而我们又改变主意了(这种情况应该避免,因为这预示着不好的设计)。
- 第二种情况: 如果在`Finalize()`方法中调用`ReRegisterForFinalize()`方法,对象就能免于在GC中被销毁。这可以用来分析GC的行为。不过,若不停地调用`Finalize()`,程序将无法结束。因此必须考虑到在一定的条件下不调用`ReRegisterForFinalize()`方法,以便程序可以终止运行。此外,如果在这样的`Finalize()`方法中引用了其他对象,那么必须小心这些对象可能在不留意的情况下被GC销毁。实际上这种“无法销毁”的对象并不当成活动对象来考虑,因此它引用的其他对象不会在建立引用树的时候算进去。

4.8 提高代码可靠性的机制

4.8.1 异步异常及托管代码可靠性

我们希望本章的内容能让你相信,托管代码的执行在技术上是一个很大的进步。现在我们要看一看托管环境的黑暗面,运行时CLR可能会不易察觉地在任何时候引发代价昂贵的操作。这个事实指出我们无法准确预测何时会发生资源耗尽。下面列有一些CLR引发昂贵操作的典型例子(而且该列表远没有列全所有情况)。

- ❑ 加载程序集。
- ❑ 执行一个类的静态构造函数。
- ❑ GC收集对象。
- ❑ JIT编译类或者方法。
- ❑ 遍历CAS栈。
- ❑ 隐式装箱。
- ❑ 创建进程中所有AppDomain共享的程序集中类的静态字段。

资源耗尽一般会转化为CLR在执行请求的线程上所引发的`OutOfMemoryException`、`StackOverflowException`或`ThreadAbortException`等异常。我们要讨论的这些异常都是异步异常,与这个概念相对应的是应用程序异常。引发应用程序异常时,当前代码有义务捕获和处理该异常。比如访问文件时,就应该准备好捕捉`FileNotFoundException`异常。但是CLR引发异步异常的时候,当前正在执行的代码通常无法负责处理。因此,建议不要试图用`try/catch/finally`块来捕捉异步异常带来的副作用,这样只会让代码变得更糟。负责处理这些异常的应当是本章已经介绍过的运行时宿主。

如果是Console或者Windows应用程序,异步异常很少发生,而且通常都来自算法错误(内存泄露,递归误用等)。因此,若此类应用程序没有捕捉到异常,则此应用程序的运行时宿主就会终止整个进程。

ASP.NET上的行为与此类似。事实上,我们注意到多种非正常行为检测算法常常可以在发生异步异常之前就检测到问题并将进程回收。这些机制将在23.7.3节进行讨论。

直到CLR集成到SQL Server 2005之前,异步异常的问题都不是十分显著。SQL Server的2005版要求可靠性达到99.999%。为了更有效率,SQL Server 2005的进程需要最大限度地数据加载到内存中,并限制从硬盘的内存页面中加载。该进程常常会遇到2G或3G的内存上限问题。最后,执行请求的超

时机制是通过引发`ThreadAbortException`异常来完成的。综上所述，处理这种将系统逼到极限的服务器程序时，异步异常将变得很常见，而且绝对不能让进程崩溃。

面对这些要求，CLR的设计人员需要提出一些新技术。这些技术就是本节即将讨论的内容。一定要记住，这些技术需要明智地使用，只有在开发大规模的服务器程序，需要创建服务器自己的运行时宿主时才用到它们，而这时很可能就要面对异步异常的问题。

4.8.2 受约束执行区域

为了防止整个进程被异步异常打断，我们需要在这种异常产生时卸载特定的进程。这里提到的方法叫做应用程序域回收(application domain recycling)。回收操作的主要难点在于必须恰当地进行而不能造成内存泄露或者破坏进程的基本状态。因此就需要一种机制来保护我们自己免受异步异常的影响。如果没有这样的机制，就几乎无法确保在这种异常发生时正确释放所有非托管资源。

.NET 2 Framework允许你通知CLR在某一段代码上引发异步异常会产生灾难性后果。如果必须发生某个异步异常，这种方法能强迫CLR在这段代码执行之前或之后发生异常，而不是在代码执行期间引发。我们称这段代码为受约束执行区域(Constrained Execution Region, CER)。

为了确保不会在CER执行期间发生异步异常，CLR必须在开始执行这段代码之前进行充分的准备。主要目的就是在不得不发生资源耗尽错误时，提前到执行CER之前触发。典型的做法是CLR先将所有可能执行的方法都编译为本地代码。为了知晓哪些方法可能执行，CLR静态遍历以CER为根的方法的调用图。此外CLR还会抑制CER执行期间可能会抛出的`ThreadAbortException`异常，直到CER执行结束。

开发人员必须注意不要在CER中分配内存。这个要求极其严格，包括所有可能隐式触发的内存分配。例如应当注意装箱指令、多维数组的访问以及同步对象的操作等。

4.8.3 如何定义 CER

CER 是通过紧跟在 `System.Runtime.CompilerServices.RuntimeHelpers` 类的 `PrepareConstrainedRegion()` 静态方法调用之后的 `try/catch/finally` 块定义的。所有 `catch` 块和 `finally` 块能到达的代码就组成了 CER。`PrepareConstrainedRegion()` 方法在执行期间最后会调用该类的 `ProbeForSufficientStack()` 静态方法。这依赖于一个事实：运行时宿主在实现 CER 的时候必须处理执行期间超过当前线程栈上限的情况(栈溢出)。该方法在 x86 型处理器上将会尝试保留 48KB 的内存。

即使保留了内存，还是可能发生栈溢出。还可以在调用 `PrepareConstrainedRegion()` 之后紧接着调用 `ExecuteCodeWithGuaranteedCleanup()` 来指定一个包含清理代码的方法，它会在发生溢出的时候调用。这种方法以 `PrePrepareMethodAttribute` 为标记，以便向 `ngen.exe` 指定其特殊用途。

`RuntimeHelpers` 类还提供了一些方法供开发人员来帮助 CLR 进行 CER 的执行准备。可以在诸如类的构造函数等处调用这些方法。

- `static void PrepareMethod(RuntimeMethodHandle m)`，CLR 会遍历 CER 中方法的调用图，将他们编译成本地代码。但这种遍历无法确定虚方法的哪个重写版本将被调用。强制指定 CER 中要调用的方法具体编译哪个版本也就成了开发人员的职责。
- `static void PrepareDelegate(Delegate d)`，需要在 CER 执行期间调用的委托，必须事先用这个方法进行准备。
- `static void RunClassConstructor(RuntimeTypeHandle t)`，这个方法可以强制执行一个类的静态构造函数。很自然，只有之前没有运行过该静态构造函数，那么在调用时它才会执行。

4.8.4 内存门

与 `ProbeForSufficientStack()` 静态方法的思想一样，需要时可以使用 `System.Runtime.Memory-`

FailPoint类来检测内存是否够用，使用方法如下所示。

```
// We are about to complete an operation which required
// at most 15MB of memory.
using( new MemoryFailPoint(15) ) {
    // Complete the operation ...
}
```

和ProbeForSufficientStack()方法不同，这个类并不会保留所给的内存量。该类的构造函数仅在执行期间计算操作系统是否可以满足所请求的内存量，而不会抛出OutOfMemoryException异常。注意，具体情况在这个请求的操作和你的代码执行之间可能会发生变化，比如另外一个线程请求了一大块内存。鉴于存在这种缺点，该技术只能认为是从效率方面考虑的。

如果要请求的内存量真的无法满足，MemoryFailPoint的构造函数会引发一个System.InsufficientMemoryException异常。因此，我们常常称这种特性为内存门（memory gate）。应当理解，这里关键的优点就是能通过InsufficientMemoryException异常获知内存不足而不会产生破坏性的后果，因此可以在应用程序中安全地捕捉这个异常并继续运行。

4.8.5 可靠性契约

.NET 2 Framework提供了一个System.Runtime.ConstrainedExecution.ReliabilityContractAttribute，用于标记方法。用这个attribute可以为标记的方法在发生异步异常时后果的严重程度定级。这个严重程度的等级是通过System.Runtime.ConstrainedExecution.Consistency枚举中的来指定的。

可靠性强度	描 述
MayCorruptProcess	标记的方法可能会破坏进程的状态，并因此使程序崩溃
MayCorruptAppDomain	标记的方法可能会破坏AppDomain的状态，并导致其被卸载
MayCorruptInstance	标记的方法可能会破坏它所在实例的状态，并且导致该对象析构
WillNotCorruptState	标记的方法不会破坏任何状态。

System.Runtime.ConstrainedExecution.Consistency类型的第二个参数适用于每一个人ReliabilityContractAttribute。这个值用于证明方法在最后可能执行它的CER的保证之下是否会执行失败。当然，如果一个方法会破坏进程或者AppDomain的状态，那么它必然能够违反这些保证，那么你必须使用Cer.None这个值。

可靠性契约制定了一套对代码进行描述的依据。同时CLR在遍历方法调用图以准备执行CER时也会利用到它。如果遍历时遇到一个没有足够可靠性契约的方法，树的遍历就会停止（因为我们知道代码是不可靠的）。不过，CER仍然会执行。微软的工程师决定允许这种潜在的危险行为，因为目前.NET Framework中相当多的方法都没有打上可靠性契约的标记。注意，CER的执行只有具备以下三种可靠性契约才能满足：

```
[ReliabilityContract(Consistency.MayCorruptInstance, Cer.MayFail)]
[ReliabilityContract(Consistency.WillNotCorruptState, Cer.MayFail)]
[ReliabilityContract(Consistency.WillNotCorruptState, Cer.Success)]
```

4.8.6 关键终结器

CLR能将继承自System.Runtime.ConstrainedExecution.CriticalFinalizerObject的类的终结器当作CER来执行。我们称之为关键终结器（critical finalizer）。除了作为CER执行，一组对象的终结器执行期间，所有的关键终结器都将在所有普通终结器之后执行。这保证托管资源所依赖的最关键

资源到最后才释放。

.NET Framework 中掌管 Win32 句柄（`System.Runtime.InteropServices.CriticalHandle` 和 `System.Runtime.InteropServices.SafeHandle`，将在 8.3 节描述）的类都使用了关键终结器机制。

.NET Framework 中还有几个继承自 `CriticalFinalizerObject` 的类，类 `System.Security.SecureString`（见 6.14.4 节）和类 `System.Threading.ReaderWriterLocker`（见 5.8 节）。

4.8.7 临界区

除了 CER 之外还有另外一种可靠性机制。其主要思想是向 CLR 提供信息，在一个被多线程共享访问的资源有所更新时通知 CLR。这里更新资源的代码段就称为临界区（Critical Region, CR）。只需要在代码段前后分别调用 `Thread` 类的 `BeginCriticalRegion()` 和 `EndCriticalRegion()` 就可以定义临界区。

如果在临界区发生异步异常，共享资源的状态就有可能被破坏。当前线程将被销毁，但是这并不足以保证应用程序可以继续执行。事实上，其他线程可能会访问已被破坏的数据而产生无法预料的后果。唯一可能的解决方法就是将当前 `AppDomain` 完全卸载，而事实正是这样。这种将线程本地的问题传递到整个 `AppDomain` 的作法称作扩大化策略（escalation policy）。临界区带来的另一个继承效应就是在内存请求失败时强制 CLR 卸载当前 `AppDomain`。

如果使用了 .NET Framework 提供的加锁类（`Monitor` 类、`lock` 关键字或 `ReaderWriterLock` 等）来对共享资源的访问进行同步，就不必在代码中显式定义临界区，因为这些类已经使用了 `BeginCriticalRegion()` 和 `EndCriticalRegion()` 方法。因此，临界区大都用在开发新的、不怎么常见的同步机制。

4.9 CLI 和 CLS

这两个缩写词的后面隐藏了 .NET 支持多语言开发的秘密。CLI（Common Language Infrastructure，公共语言基础设施）是一种规范，描述了 CLR 和程序集所遵循的要求。满足 CLI 要求的软件层有能力运行 .NET 应用程序。该规范是作为 ECMA 标准而发布的，可以在 <http://www.ecmainternational.org/publications/standards/ECMA-335.HTM> 上在线查看。

4.9.1 .NET 语言必须满足的要求

为了让所有语言编译的程序集都能通过 CLR 托管执行（或者通过某个实现 CLI 的软件层），并且能够使用所有 .NET Framework 的类和工具，编程语言和编译器必须遵守一组称为 CLS（Common Language Specification，公共语言规范）的要求。这些要求包括但不限于支持 CTS 类型。语言和编译器必须满足的要求十分之多，这里列出几个最一般的要求。

- 该语言必须提供一种语法，来解决类因实现两个接口而导致的重复方法所带来的冲突。在这种情况下，所实现的两个接口中各有一个方法，这两个方法具有一样的名字和签名式。CLS 规定类必须实现两种不同的方法。
- 只有少数基元类型与 CLS 兼容，比如 C# 提供的 `ushort` 类型就不与 CLS 兼容。
- 公有方法的参数类型必须与 CLS 兼容。与 CLS 兼容的概念将在稍后介绍。
- 作为异常抛出的对象必须是 `System.Exception` 或其子类的实例。

完整的要求列表可见 MSDN 上名为“Common Language Specification”的文章。

编程语言并不需要完全兼容 CLS，主要有两个级别的兼容性。

- “CLS 使用者”的语言。能够实例化 CLS 兼容程序集中的类并且使用类的公有成员。
- “CLS 扩展者”的语言。能够继承 CLS 兼容程序集中的公有类。该兼容性同时还隐含了 CLS 使

用者兼容性。

C#、VB.NET和C++/CLI能满足以上所有兼容性要求。

4.9.2 从开发人员的观点看 CLI 和 CLS

如果说一个程序集与CLS兼容，那么以下要点都要与CLS兼容。

- 公有类型的定义；
- 公有类型的公有和保护成员的定义；
- 公有和保护方法的参数。

开发人员对开发具有CLS兼容性的类库都有既定的兴趣，因为这样可以确保将来更容易地重用这些类库。幸运的是，开发人员没有必要将CLS的要求完全掌握以便检查类是否与CLS兼容。只要用 `System.CLSCompliantAttribute` 标记代码，就能让编译器来检查应用程序中的元素（程序集、类、方法等）是否与CLS兼容。例如：

例4-15

```
using System;
[assembly : CLSCompliantAttribute(true)]
namespace CLSCompliantTest {
    public class Program {
        public static void Fct(ushort i ) { ushort j = i; }
        static void Main(){}
    }
}
```

编译器会对 `ushort` 类型提出警告，它并不是与CLS兼容的类型，但却用作公有类的公有方法参数。不过在 `Fct()` 方法体内部使用 `ushort` 类型并不会产生警告。

还可以用 `CLSCompliantAttribute` 标记不想被编译器CLS测试兼容性的代码，而不会影响到程序集中其他代码的兼容性测试。如不想让CLS测试 `Fct()` 方法：

例4-16

```
using System;
[assembly : CLSCompliantAttribute(true)]
namespace CLSCompliantTest {
    public class Program {
        [CLSCompliantAttribute(false)]
        public static void Fct(ushort i ) { ushort j = i; }
        static void Main(){}
    }
}
```

应当理解，`Fct()` 在不支持 `ushort` 类型的代码中可能无法调用。