还有一个很出色的工具可分析内存和应用程序的性能，它的名字是 PerfView。该工具能收集“Windows 事件跟踪”(Event Tracing for Windows，ETW)日志并处理它们。获取该工具最好的办法是网上搜索 PerfView。最后还应该考虑一下 SOS Debugging Extension(SOS.dll)，它对于内存问题和其他 CLR 问题的调试颇有帮助。对于内存有关的行动，SOS Debugging Extension 允许检查进程中为托管堆分配了多少内存，显示在终结队列中登记终结的所有对象，显示每个 AppDomain 或整个进程的 GCHandle 表中的记录项，并显示是什么根保持对象在堆中存活。



图 21-12 PerfMon.exe 显示了.NET CLR Memory 计数器

21.3 使用需要特殊清理的类型

你现在基本了解了垃圾回收和托管堆的情况，包括垃圾回收器如何回收对象的内存。幸好，大多数类型有内存就能正常工作。但有的类型除了内存还需要本机资源。

例如，**System.IO.FileStream** 类型需要打开一个文件(本机资源)并保存文件的句柄。然后，类型的 **Read** 和 **Write** 方法用句柄操作文件。类似地，**System.Threading.Mutex** 类型要打开一个 Windows 互斥体内核对象(本机资源)并保存其句柄，并在调用 **Mutex** 的方法时使用该句柄。

包含本机资源的类型被 GC 时，GC 会回收对象在托管堆中使用的内存。但这样会造成本机资源(GC 对它一无所知)的泄漏，这当然是不允许的。所以，CLR 提供了称为**终结**(finalization)的机制，允许对象在被判定为垃圾之后，但在对象内存被回收之前执行一些代码。任何包装了本机资源(文件、网络连接、套接字、互斥体)的类型都支持终结。CLR 判定一个对象不可达时，对象将终结它自己，释放它包装的本机资源。之后，GC 会从托管堆回收对象。

终极基类 **System.Object** 定义了受保护的虚方法 **Finalize**。垃圾回收器判定对象是垃圾后，会调用对象的 **Finalize** 方法(如果重写)。Microsoft 的 C#团队认为 **Finalize** 在编程语言中需

要特殊语法(类似于 C#要求用特殊语法定义构造器)。因此，C#要求在类名前添加~符号来定义 **Finalize** 方法，如下例所示：

```
internal sealed class SomeType {
    // 这是一个 Finalize 方法
    ~SomeType() {
        // 这里的代码会进入 Finalize 方法
    }
}
```

编译上述代码，用 ILDasm.exe 检查得到的程序集，会发现 C#编译器实际是在模块的元数据中生成了名为 **Finalize** 的 **protected override** 方法。查看 **Finalize** 的 IL，会发现方法主体的代码被放到一个 **try** 块中，在 **finally** 块中则放入了一个 **base.Finalize** 调用。



重要提示　如果熟悉 C++，会发现 C# **Finalize** 方法的特殊语法非常类似于 C++析构器。事实上，在 C#编程语言规范的早期版本中，真的是将该方法称为**析构器**。但 **Finalize** 方法的工作方式和 C++析构器完全不同，这会使从一种语言迁移到另一种语言的开发人员产生混淆。

问题在于，这些开发人员可能错误地以为使用 C#析构器语法意味着类型的实例会被确定性析构[①]，就像在 C++中那样。但 CLR 不支持确定性析构，而作为面向 CLR 的语言，C#也无法提供这种机制。

被视为垃圾的对象在垃圾回收完毕后才调用 **Finalize** 方法，所以这些对象的内存不是马上被回收的，因为 **Finalize** 方法可能要执行访问字段的代码。可终结对象在回收时必须存活，造成它被提升到另一代，使对象活得比正常时间长。这增大了内存耗用，所以应尽可能避免终结。更糟的是，可终结对象被提升时，其字段引用的所有对象也会被提升，因为它们也必须继续存活。所以，要尽量避免为引用类型的字段定义可终结对象。

另外要注意，**Finalize** 方法的执行时间是控制不了的。应用程序请求更多内存时才可能发生 GC，而只有 GC 完成后才运行 **Finalize**。另外，CLR 不保证多个 **Finalize** 方法的调用顺序。所以，在 **Finalize** 方法中不要访问定义了 **Finalize** 方法的其他类型的对象；那些对象可能已经终结了。但可以安全地访问值类型的实例，或者访问没有定义 **Finalize** 方法的引用类型的对象。调用静态方法也要当心，这些方法可能在内部访问已终结的对象，导致静态方法的行为变得无法预测。

CLR 用一个特殊的、高优先级的专用线程调用 **Finalize** 方法来避免死锁[①]。如果 **Finalize** 方法阻塞(例如进入死循环，或等待一个永远不发出信号的对象)，该特殊线程就调用不了任何更多的 **Finalize** 方法。这是非常坏的情况，因为应用程序永远回收不了可终结对象占用的内存——只要应用程序运行就会一直泄漏内存。如果 **Finalize** 方法抛出未处理的异常，则进程终止，没办法捕捉该异常。

① CLR 将来的版本可能用多个终结器线程提升性能。

综上所述，**Finalize** 方法问题较多，使用须谨慎。记住它们是为释放本机资源而设计的。强烈建议不要重写 **Object** 的 **Finalize** 方法。相反，使用 Microsoft 在 FCL 中提供的辅助类。这些辅助类重写了 **Finalize** 方法并添加了一些特殊的 CLR“魔法”(以后会慢慢讲述)。你从这些辅助类派生出自己的类，从而继承 CLR 的“魔法”。

创建封装了本机资源的托管类型时，应该先从 **System.Runtime.InteropServices.SafeHandle** 这个特殊基类派生出一个类。该类的形式如下(我在方法中添加了注释，指明它们做的事情)：

```
public abstract class SafeHandle : CriticalFinalizerObject, IDisposable {
   // 这是本机资源的句柄
   protected IntPtr handle;

   protected SafeHandle(IntPtr invalidHandleValue, Boolean ownsHandle) {
      this.handle = invalidHandleValue;
      // 如果 ownsHandle 为 true，那么这个从 SafeHandle 派生的对象被回收时，
      // 本机资源会被关闭
   }

   protected void SetHandle(IntPtr handle) {
      this.handle = handle;
   }

   // 可调用 Dispose 显式释放资源
   // 这是 IDisposable 接口的 Dispose 方法
   public void Dispose() { Dispose(true); }

   // 默认的 Dispose 实现(如下所示)正是我们希望的。强烈建议不要重写这个方法
   protected virtual void Dispose(Boolean disposing) {
      // 这个默认实现会忽略 disposing 参数：
      // 如果资源已经释放，那么返回；
      // 如果 ownsHandle 为 false，那么返回；
      // 设置一个标志来指明该资源已经释放；
      // 调用虚方法 ReleaseHandle；
      // 调用 GC.SuppressFinalize(this)方法来阻止调用 Finalize 方法；
      // 如果 ReleaseHandle 返回 true，那么返回；
      // 如果走到这一步，就激活 releaseHandleFailed 托管调试助手(MDA)。
   }

   // 默认的 Finalize 实现(如下所示)正是我们希望的。强烈建议不要重写这个方法
   ~SafeHandle() { Dispose(false); }

   // 派生类要重写这个方法以实现释放资源的代码
   protected abstract Boolean ReleaseHandle();

   public void SetHandleAsInvalid() {
      // 设置标志来指出这个资源已经释放
      // 调用 GC.SuppressFinalize(this)方法来阻止调用 Finalize 方法
   }

   public Boolean IsClosed {
      get {
         // 返回指出资源是否释放的一个标志
      }
   }

   public abstract Boolean IsInvalid {
      // 派生类要重写这个属性
      // 如果句柄的值不代表资源(通常意味着句柄为 0 或-1)，实现应返回 true
      get;
```

```
    }

    // 以下三个方法涉及安全性和引用计数，本节最后会讨论它们
    public void    DangerousAddRef(ref Boolean success) {...}
    public IntPtr  DangerousGetHandle() {...}
    public void    DangerousRelease() {...}
}
```

SafeHandle 类有两点需要注意。其一，它派生自 **CriticalFinalizerObject**；后者在 **System.Runtime.ConstrainedExecution** 命名空间定义。CLR 以特殊方式对待这个类及其派生类。具体地说，CLR 赋予这个类以下三个很酷的功能。

- 首次构造任何 **CriticalFinalizerObject** 派生类型的对象时，CLR 立即对继承层次结构中的所有 **Finalize** 方法进行 JIT 编译。构造对象时就编译这些方法，可确保当对象被确定为垃圾之后，本机资源肯定会得以释放。不对 **Finalize** 方法进行提前编译，那么也许能分配并使用本机资源，但无法保证释放。内存紧张时，CLR 可能找不到足够的内存来编译 **Finalize** 方法，这会阻止 **Finalize** 方法的执行，造成本机资源泄漏。另外，如果 **Finalize** 方法中的代码引用了另一个程序集中的类型，但 CLR 定位该程序集失败，那么资源将得不到释放。

- CLR 是在调用了非 **CriticalFinalizerObject** 派生类型的 **Finalize** 方法之后，才调用 **CriticalFinalizerObject** 派生类型的 **Finalize** 方法。这样，托管资源类就可以在它们的 **Finalize** 方法中成功地访问 **CriticalFinalizerObject** 派生类型的对象。例如，**FileStream** 类的 **Finalize** 方法可以放心地将数据从内存缓冲区 flush[①]到磁盘，它知道此时磁盘文件还没有关闭。

- 如果 **AppDomain** 被一个宿主应用程序(例如 **Microsoft SQL Server** 或者 **Microsoft ASP.NET**)强行中断，**CLR** 将调用 **CriticalFinalizerObject** 派生类型的 **Finalize** 方法。宿主应用程序不再信任它内部运行的托管代码时，也利用这个功能确保本机资源得以释放。

其二，**SafeHandle** 是抽象类，必须有另一个类从该类派生并重写受保护的构造器、抽象方法 **ReleaseHandle** 以及抽象属性 **IsInvalid** 的 **get** 访问器方法。

大多数本机资源都用句柄(32 位系统是 32 位值，64 位系统是 64 位值)进行操作。所以 **SafeHandle** 类定义了受保护 **IntPtr** 字段 **handle**。在 Windows 中，大多数值为 0 或-1 的句柄都是无效的。**Microsoft.Win32.SafeHandles** 命名空间包含 **SafeHandleZeroOrMinusOneIsInvalid** 辅助类，如下所示：

```
public abstract class SafeHandleZeroOrMinusOneIsInvalid : SafeHandle {
   protected SafeHandleZeroOrMinusOneIsInvalid(Boolean ownsHandle)
      : base(IntPtr.Zero, ownsHandle) {
   }

   public override Boolean IsInvalid {
```

① flush 在文档中翻译成“刷新”，本书保留原文未译。其实 flush 在技术文档中的意思和日常生活中一样，即“冲洗(到别处)”。例如，我们会说“冲厕所”，不会说“刷新厕所”。——译注

```
        get {
            if (base.handle == IntPtr.Zero) return true;
            if (base.handle == (IntPtr) (-1)) return true;
            return false;
        }
    }
}
```

SafeHandleZeroOrMinusOneIsInvalid 也是抽象类，所以必须有另一个类从该类派生并重写它的受保护构造器[①]和抽象方法 **ReleaseHandle**。**.NET Framework** 只提供了很少几个从 **SafeHandleZeroOrMinusOneIsInvalid** 派生的公共类，其中包括 **SafeFileHandle**，**SafeRegistryHandle**，**SafeWaitHandle** 和 **SafeMemoryMappedViewHandle**。以下是 **SafeFileHandle** 类：

```
public sealed class SafeFileHandle : SafeHandleZeroOrMinusOneIsInvalid {
    public SafeFileHandle(IntPtr preexistingHandle, Boolean ownsHandle)
        : base(ownsHandle) {
        base.SetHandle(preexistingHandle);
    }
    protected override Boolean ReleaseHandle() {
        // 告诉 Windows 我们希望本机资源关闭
        return Win32Native.CloseHandle(base.handle);
    }
}
```

SafeWaitHandle 类的实现方式与上述 **SafeFileHandle** 类相似。之所以要用不同的类来提供相似的实现，唯一的原因就是为了保证类型安全；编译器不允许将一个文件句柄作为实参传给希望获取一个等待句柄的方法，反之亦然。**SafeRegistryHandle** 类的 **ReleaseHandle** 方法调用的是 Win32 **RegCloseKey** 函数。

如果.NET Framework 提供额外的类来包装各种本机资源，那么肯定会大受欢迎。例如，它似乎还应该提供下面这些类：**SafeProcessHandle**，**SafeThreadHandle**，**SafeTokenHandle**，**SafeLibraryHandle**(其 **ReleaseHandle** 方法调用 Win32 **FreeLibrary** 函数)以及 **SafeLocalAllocHandle**(其 **ReleaseHandle** 方法调用 Win32 **LocalFree** 函数)。

其实，所有这些类(还有许多没有列出)已经和 FCL 一道发布了，只是没有公开。它们全都在定义它们的程序集内部使用。Microsoft 之所以不公开，是因为不想完整地测试它们，也不想花时间编写它们的文档。但如果想在自己的工作中使用这些类，建议用一个工具(比如 ILDasm.exe 或某个 IL 反编译工具)提取这些类的代码，并将代码集成到自己项目的源代码中。所有这些类的实现其实很简单，自己从头写也花不了多少时间。

SafeHandle 派生类非常有用，因为它们保证本机资源在垃圾回收时得以释放。除了前面讨论过的功能，**SafeHandle** 类还有两个功能值得注意。首先，与本机代码互操作时，**SafeHandle** 派生类将获得 CLR 的特殊对待。例如以下代码：

```
using System;
using System.Runtime.InteropServices;
using Microsoft.Win32.SafeHandles;
```

① 构造器不能虚或抽象，自然也不能“重写”。作者的意思是说，派生类会定义一个.ctor 来调用受保护的.ctor，再重写其他抽象成员。——译注

```
internal static class SomeType {
    [DllImport("Kernel32", CharSet=CharSet.Unicode, EntryPoint="CreateEvent")]
    // 这个原型不健壮
    private static extern IntPtr CreateEventBad(
        IntPtr pSecurityAttributes, Boolean manualReset,
            Boolean initialState, String name);

    // 这个原型是健壮的
    [DllImport("Kernel32", CharSet=CharSet.Unicode, EntryPoint="CreateEvent")]
    private static extern SafeWaitHandle CreateEventGood(
        IntPtr pSecurityAttributes, Boolean manualReset,
            Boolean initialState, String name);

    public static void SomeMethod() {
        IntPtr         handle   = CreateEventBad(IntPtr.Zero, false, false, null);
        SafeWaitHandle swh      = CreateEventGood(IntPtr.Zero, false, false, null);
    }
}
```

在上述代码中，**CreateEventBad** 方法的原型是返回一个 **IntPtr**，这会将句柄返回给托管代码。但以这种方式与本机代码交互是不健壮的。调用 **CreateEventBad**(该方法创建本机事件资源)之后，在句柄赋给 **handle** 变量之前，可能抛出一个 **ThreadAbortExcption**。虽然这很少发生，但一旦发生，托管代码将造成本机资源的泄漏。这时只有终止整个进程才能关闭事件。

SafeHandle 类修正了这个潜在的资源泄漏问题。注意，**CreateEventGood** 方法的原型是返回一个 **SafeWaitHandle**(而非 **IntPtr**)。调用 **CreateEventGood** 方法时，CLR 会调用 Win32 **CreateEvent** 函数。**CreateEvent** 函数返回至托管代码时，CLR 知道 **SafeWaitHandle** 是从 **SafeHandle** 派生的，所以会自动在托管堆构造 **SafeWaitHandle** 的实例，向其传递从 **CreateEvent** 返回的句柄值。新 **SafeWaitHandle** 对象的构造以及句柄的赋值现在是在本机代码中发生的，不可能被一个 **ThreadAbortExcption** 打断。托管代码现在不可能泄漏这个本机资源。**SafeWaitHandle** 对象最终会被垃圾回收，其 **Finalize** 方法会被调用，确保资源得以释放。

SafeHandle 派生类最后一个值得注意的功能是防止有人利用潜在的安全漏洞。问题起因是一个线程可能试图使用一个本机资源，而另一个线程试图释放该资源。这可能造成句柄循环使用漏洞。**SafeHandle** 类防范这个安全隐患的办法是使用引用计数。**SafeHandle** 类内部定义了私有字段来维护一个计数器。一旦某个 **SafeHandle** 派生对象被设为有效句柄，计数器就被设为 1。每次将 **SafeHandle** 派生对象作为实参传给一个本机方法(非托管方法)，CLR 就会自动递增计数器。类似地，当本机方法返回到托管代码时，CLR 自动递减计数器。例如，Win32 **SetEvent** 函数的原型如下：

```
[DllImport("Kernel32", ExactSpelling=true)]
private static extern Boolean SetEvent(SafeWaitHandle swh);
```

现在，调用这个方法并传递一个 **SafeWaitHandle** 对象引用，CLR 会在调用前递增计数器，调用后递减计数器。当然，对计数器的操作是以线程安全的方式进行的。那么安全性如何得以保障？当另一个线程试图释放 **SafeHandle** 对象包装的本机资源时，CLR 知道它实际上不能释放资源，因为该资源正在由一个本机(非托管)函数使用。本机函数返回后，计数器

递减为 0，资源才会得以释放。

如果编写或调用代码将句柄作为一个 **IntPtr** 来操作，可以从 **SafeHandle** 对象中访问它，但就要显式操作引用计数器。这是通过 **SafeHandle** 的 **DangerousAddRef** 和 **DangerousRelease** 方法来完成的。另外，可通过 **DangerousGetHandle** 方法访问原始句柄。

System.Runtime.InteropServices 命名空间还定义了一个 **CriticalHandle** 类。该类除了不提供引用计数器功能，其他方面与 **SafeHandle** 类相同。**CriticalHandle** 类及其派生类通过牺牲安全性来换取更好的性能(因为不用操作计数器)。和 **SafeHandle** 相似，**CriticalHandle** 类也有自己的几个派生类型，其中包括 **CriticalHandleMinusOneIsInvalid** 和 **CriticalHandleZeroOrMinusOneIsInvalid**。由于 Microsoft 倾向于建立更安全而不是更快的系统，所以类库中没有提供从这两个类派生的类型。自己写程序时，建议只有在必须追求性能的时候才使用派生自 **CriticalHandle** 的类型。如果降低安全性不会有什么严重的后果，就选择从 **CriticalHandle** 派生的一个类型。

21.3.1 使用包装了本机资源的类型

你现在知道了如何定义包装了本机资源的 **SafeHandle** 派生类，接着说说如何使用它。以常用的 **System.IO.FileStream** 类为例，可利用它打开一个文件，从文件中读取字节，向文件写入字节，然后关闭文件。**FileStream** 对象在构造时会调用 Win32 **CreateFile** 函数，函数返回的句柄保存到 **SafeFileHandle** 对象中，然后通过 **FileStream** 对象的一个私有字段来维护对该对象的引用。**FileStream** 类还提供子几个额外的属性(例如 **Length**，**Position**，**CanRead**)和方法(例如 **Read**，**Write**，**Flush**)。

假定要写代码来创建一个临时文件，向其中写入一些字节，然后删除文件。开始可能会像下面这样写代码：

```
using System;
using System.IO;

public static class Program {
   public static void Main() {
      // 创建要写入临时文件的字节
      Byte[] bytesToWrite = new Byte[] { 1, 2, 3, 4, 5 };

      // 创建临时文件
      FileStream fs = new FileStream("Temp.dat", FileMode.Create);

      // 将字节写入临时文件
      fs.Write(bytesToWrite, 0, bytesToWrite.Length);

      // 删除临时文件
      File.Delete("Temp.dat"); // 抛出 IOException 异常
   }
}
```

遗憾的是，生成并运行上述代码，它也许能工作，但大多数时候都不能。问题在于 **File** 的静态 **Delete** 方法要求 Windows 删除一个仍然打开的文件。所以 **Delete** 方法会抛出 **System.IO.IOException** 异常，并显示以下字符串消息：**文件"Temp.dat"正由另一进程使用，因此该进程无法访问此文件。**

但某些情况下，文件可能“误打误撞”地被删除！如果另一线程不知怎么造成了一次垃圾回收，而且这次垃圾回收刚好在调用 **Write** 之后、调用 **Delete** 之前发生，那么 **FileStream** 的 **SafeFileHandle** 字段的 **Finalize** 方法就会被调用，这会关闭文件，随后 **Delete** 操作也就可以正常运行。但发生这种情况的概率很小，上述代码无法运行的可能性在 99%以上。

类如果想允许使用者控制类所包装的本机资源的生存期，就必须实现如下所示的 **IDisposable** 接口。

```
public interface IDisposable {
   void Dispose();
}
```



重要提示　如果类定义的一个字段的类型实现了 dispose 模式[①]，那么类本身也应实现 dispose 模式。**Dispose** 方法应 dispose[②]字段引用的对象。这就允许类的使用者在类上调用 **Dispose** 来释放对象自身使用的资源。

幸好，**FileStream** 类实现了 **IDisposable** 接口。在实现中、会在 **FileStream** 对象的私有 **SafeFileHandle** 字段上调用 **Dispose**。现在就能修改代码来显式关闭文件，而不是等着未来某个时候 GC 的发生。下面是修改后的源代码。

```
using System;
using System.IO;

public static class Program {
   public static void Main() {
      // 创建要写入临时文件的字节
      Byte[] bytesToWrite = new Byte[] { 1, 2, 3, 4, 5 };

      // 创建临时文件
      FileStream fs = new FileStream("Temp.dat", FileMode.Create);

      // 将字节写入临时文件
      fs.Write(bytesToWrite, 0, bytesToWrite.Length);
```

① 实现了 **IDisposable** 接口，就实现了 dispose 模式。——译注

② 文档将 disposal 和 dispose 翻译成“释放”。这里解释一下为什么不赞成这个翻译。在英语中，这个词的意思是“摆脱”或“除去”(get rid of)一个东西，尤其是在这个东西很难除去的情况下。之所以认为“释放”不恰当，除了和 release 一词冲突，还因为 dispose 强调了“清理”和“处置”，而且在完成(对象中包装的)资源的清理之后，对象占用的内存还暂时不会释放。所以，“dispose 一个对象”真正的意思是：清理或处置对象中包装的资源(比如它的字段引用的对象)，然后等着在一次垃圾回收之后回收该对象占用的托管堆内存(此时才释放)。为避免误解，本书保留了 dispose 和 disposal 的原文。——译注

```
        // 写入结束后显式关闭文件
        fs.Dispose();

        // 删除临时文件
        File.Delete("Temp.dat"); // 总能正常工作
    }
}
```

现在，当调用 **File** 的 **Delete** 方法时，Windows 发现该文件没有打开，所以能成功删除它。

注意，并非一定要调用 **Dispose** 才能保证本机资源得以清理。本机资源的清理最终总会发生，调用 **Dispose** 只是控制这个清理动作的发生时间。另外，调用 **Dispose** 不会将托管对象从托管堆删除。只有在垃圾回收之后，托管堆中的内存才会得以回收。这意味着即使 dispose 了托管对象过去用过的任何本机资源，也能在托管对象上调用方法。

以下代码在文件关闭后调用 **Write** 方法，试图写入更多的字节。这显然不可能。代码执行时，第二个 **Write** 调用会抛出 **System.ObjectDisposedException** 异常并显示以下字符串消息：**无法访问已关闭的文件**。

```
using System;
using System.IO;

public static class Program {
    public static void Main() {
        // 创建要写入临时文件的字节
        Byte[] bytesToWrite = new Byte[] { 1, 2, 3, 4, 5 };

        // 创建临时文件
        FileStream fs = new FileStream("Temp.dat", FileMode.Create);

        // 将字节写入临时文件
        fs.Write(bytesToWrite, 0, bytesToWrite.Length);

        // 结束写入后显式关闭文件
        fs.Dispose();

        // 关闭文件后继续写入
        fs.Write(bytesToWrite, 0, bytesToWrite.Length); // 抛出 ObjectDisposedException

        // 删除临时文件
        File.Delete("Temp.dat");
    }
}
```

这不会造成对内存的破坏，因为 **FileStream** 对象的内存依然“健在”。只是在对象被显式 dispose 之后，它的方法不能再成功执行而已。



重要提示 定义实现 **IDisposable** 接口的类型时，在它的所有方法和属性中，一定要在对象被显式清理之后抛出一个 **System.ObjectDisposedException**。而 **Dispose** 方法永远不要抛出 **ObjectDisposedException**；被多次调用就直接返回。



重要提示　我一般不赞成在代码中显式调用 **Dispose**。理由是 CLR 的垃圾回收器已经写得非常好，应该放心地把工作交给它。垃圾回收器知道一个对象何时不再由应用程序代码访问，而且只有到那时才会回收对象。[①] 而当应用程序代码调用 **Dispose** 时，实际是在信誓旦旦地说它知道应用程序在什么时候不需要一个对象。但许多应用程序都不可能准确知道一个对象在什么时候不需要。

例如，假定在方法 A 的代码中构造了一个新对象，然后将对该对象的引用传给方法 B。方法 B 可能将对该对象的引用保存到某个内部字段变量(一个根)中。但方法 A 并不知道这个情况，它当然可以调用 **Dispose**。但在此之后，其他代码可能试图访问该对象，造成抛出一个 **ObjectDisposedException**。建议只有在确定必须清理资源(例如删除打开的文件)时才调用 **Dispose**。

也可能多个线程同时调用一个对象的 **Dispose**。但 **Dispose** 的设计规范指出 **Dispose** 不一定要线程安全。原因是代码只有在确定没有别的线程使用对象时才应调用 **Dispose**。

前面的例子展示了怎样显式调用类型的 **Dispose** 方法。如果决定显式调用 **Dispose**，强烈建议将调用放到一个异常处理 **finally** 块中。这样可保证清理代码得以执行。因此，前面代码示例可修改成下面这种更好的形式：

```
using System;
using System.IO;

public static class Program {
   public static void Main() {
      // 创建要写入临时文件的字节
      Byte[] bytesToWrite = new Byte[] { 1, 2, 3, 4, 5 };

      // 创建临时文件
      FileStream fs = new FileStream("Temp.dat", FileMode.Create);
      try {
         // 将字节写入临时文件
         fs.Write(bytesToWrite, 0, bytesToWrite.Length);
      }
      finally {
         // 写入字节后显式关闭文件
         if (fs != null) fs.Dispose();
      }

      // 删除临时文件
```

① 垃圾回收系统有许多好处：无内存泄漏、无内存损坏、无地址空间碎片化以及缩小的工作集。现在还增加了一个好处：同步。你没有看错，GC 确实能作为线程同步机制使用。问题是，怎么知道所有线程都不再使用一个对象？答案是，GC 会终结对象。创建自己的应用程序时，利用 GC 的所有功能并不是一件坏事。

```
        File.Delete("Temp.dat");
    }
}
```

添加异常处理代码是正确的，而且应该坚持这样做。幸好，C#语言提供了一个 **using** 语句，它允许用简化的语法来获得和上述代码相同的结果。下面演示了如何使用 C#的 **using** 语句重写上述代码：

```
using System;
using System.IO;

public static class Program {
    public static void Main() {
        // 创建要写入临时文件的字节
        Byte[] bytesToWrite = new Byte[] { 1, 2, 3, 4, 5 };

        // 创建临时文件
        using (FileStream fs = new FileStream("Temp.dat", FileMode.Create)) {
            // 将字节写入临时文件
            fs.Write(bytesToWrite, 0, bytesToWrite.Length);
        }
        // 删除临时文件
        File.Delete("Temp.dat");
    }
}
```

using 语句初始化一个对象，并将它的引用保存到一个变量中。然后在 **using** 语句的大括号内访问该变量。编译这段代码时，编译器自动生成 **try** 块和 **finally** 块。在 **finally** 块中，编译器生成代码将变量转型为一个 **IDisposable** 并调用 **Dispose** 方法。但是很显然，**using** 语句只能用于那些实现了 **IDisposable** 接口的类型。



> **注意** C#的 **using** 语句支持初始化多个变量，只要这些变量的类型相同。另外，**using** 语句还允许只使用一个已初始化的变量。欲知详情，请参见文档中的“**using** 语句”主题。

21.3.2 一个有趣的依赖性问题

System.IO.FileStream 类型允许用户打开文件进行读写。为提高性能，该类型的实现利用了一个内存缓冲区。只有缓冲区满时，类型才将缓冲区中的数据刷入文件。**FileStream** 类型只支持字节的写入。写入字符和字符串可以使用一个 **System.IO.StreamWriter**，如下所示：

```
FileStream fs = new FileStream("DataFile.dat", FileMode.Create);
StreamWriter sw = new StreamWriter(fs);
sw.Write("Hi there");

// 不要忘记写下面这个 Dispose 调用
sw.Dispose();
// 注意：调用 StreamWriter.Dispose 会关闭 FileStream;
// FileStream 对象无需显式关闭
```

注意，**StreamWriter** 的构造器接受一个 **Stream** 对象引用作为参数，可以向它传递一个 **FileStream** 对象引用作为实参。在内部，**StreamWriter** 对象会保存 **Stream** 对象引用。向

一个 **StreamWriter** 对象写入时，它会将数据缓存在自己的内存缓冲区中。缓冲区满时，**StreamWriter** 对象将数据写入 **Stream** 对象。

通过 **StreamWriter** 对象写入数据完毕后应调用 **Dispose**。(由于 **StreamWriter** 类型实现了 **IDisposable** 接口，所以也可使用 C#的 **using** 语句。)这导致 **StreamWriter** 对象将数据 flush 到 **Stream** 对象并关闭该 **Stream** 对象。[①]



注意　不需要在 **FileStream** 对象上显式调用 **Dispose**，因为 **StreamWriter** 会帮你调用。但如果非要显式调用 **Dispose**，**FileStream** 会发现对象已经清理过了，所以方法什么都不做而直接返回。

没有代码显式调用 **Dispose** 会发生什么？在某个时刻，垃圾回收器会正确检测到对象是垃圾，并对其进行终结。但垃圾回收器不保证对象的终结顺序。所以，如果 **FileStream** 对象先终结，就会关闭文件。然后，当 **StreamWriter** 对象终结时，会试图向已关闭的文件写入数据，造成抛出异常。相反，如果是 **StreamWriter** 对象先终结，数据就会安全写入文件。

Microsoft 是如何解决这个问题的呢？让垃圾回收器以特定顺序终结对象是不可能的，因为不同的对象可能包含相互之间的引用，垃圾回收器无法正确猜出这些对象的终结顺序。Microsoft 的解决方案是：**StreamWriter** 类型不支持终结，所以永远不会将它的缓冲区中的数据 flush 到底层 **FileStream** 对象。这意味着如果忘记在 **StreamWriter** 对象上显式调用 **Dispose**，数据肯定会丢失。Microsoft 希望开发人员注意到这个数据一直丢失的问题，并插入对 **Dispose** 的调用来修正代码。



注意　.NET Framework 支持托管调试助手(Managed Debugging Assistant，MDA)功能。启用一个 MDA 后，.NET Framework 就会查找特定种类的常见编程错误，并激活对应的 MDA。在调试器中，激活一个 MDA 感觉就像是抛出一个异常。有一个 MDA 可以检测 **StreamWriter** 对象没有显式 dispose 就作为垃圾被回收的情况。为了在 Visual Studio 中启用这个 MDA，请打开你的项目，选择“调试”|“异常”。在“异常”对话框中，展开“Managed Debugging Assistants”节点并滚动到底部，找到名为 **StreamWriterBufferedDataLost** 的 MDA，勾选“引发”即可让 Visual Studio 调试器在 **StreamWriter** 对象的数据丢失时停止。

21.3.3　GC 为本机资源提供的其他功能

本机资源有时会消耗大量内存，但用于包装它的托管对象只占用很少的内存。一个典型的例子就是位图。一个位图可能占用几兆字节的本机内存，托管对象却极小，只包含一个 **HBITMAP**(一个 4 或 8 字节的值)。从 CLR 的角度看，一个进程可以在执行一次垃圾回收

① 可调用 **StreamWriter** 的获取一个 **Boolean leaveOpen** 参数的构造器来覆盖该行为。——译注

之前分配数百个位图(它们用的托管内存太少了)。但如果进程操作许多位图，进程的内存消耗将以一个恐怖的速度增长。为了修正这个问题，**GC** 类提供了以下两个静态方法：

```
public static void AddMemoryPressure(Int64 bytesAllocated);
public static void RemoveMemoryPressure(Int64 bytesAllocated);
```

如果一个类要包装可能很大的本机资源，就应该使用这些方法提示垃圾回收器实际需要消耗多少内存。垃圾回收器内部会监视内存压力，压力变大时，就强制执行垃圾回收。

有的本机资源的数量是固定的。例如，Windows 以前就限制只能创建 5 个设备上下文。应用程序能打开的文件数量也必须有限制。同样地，从 CLR 的角度看，一个进程可以在执行垃圾回收之前分配数百个对象(每个对象都使用极少的内存)。但是，如果这些本机资源的数量有限，那么一旦试图使用超过允许数量的资源，通常会导致抛出异常。为了解决这个问题，命名空间 **System.Runtime.InteropServices** 提供了 **HandleCollector** 类：

```
public sealed class HandleCollector {
   public HandleCollector(String name, Int32 initialThreshold);
   public HandleCollector(String name, Int32 initialThreshold,
      Int32 maximumThreshold);
   public void Add();
   public void Remove();

   public Int32 Count { get; }
   public Int32 InitialThreshold { get; }
   public Int32 MaximumThreshold { get; }
   public String Name { get; }
}
```

如果一个类要包装数量有限制的本机资源，就应该使用该类的实例来提示垃圾回收器实际要使用资源的多少个实例。该类的对象会在内部监视这个计数，计数太大就强制垃圾回收。



> **注意** 在内部，**GC.AddMemoryPressure** 和 **HandleCollector.Add** 方法会调用 **GC.Collect**，在第 0 代超过预算前强制进行 GC。一般都强烈反对强制开始一次垃圾回收，因为它会对应用程序性能造成负面影响。但是，类之所以调用这些方法，是为了保证应用程序能用上有限的本机资源。本地资源用光了，应用程序就会失败。对于大多数应用程序，性能遭受一些损失总胜于完全无法运行。

以下代码演示了内存压力方法及 **HandleCollector** 类的使用和效果：

```
using System;
using System.Runtime.InteropServices;

public static class Program {
   public static void Main() {
      MemoryPressureDemo(0);                  // 0 导致不频繁的 GC
      MemoryPressureDemo(10 * 1024 * 1024);   // 10MB 导致频繁的 GC

      HandleCollectorDemo();
   }

   private static void MemoryPressureDemo(Int32 size) {
      Console.WriteLine();
      Console.WriteLine("MemoryPressureDemo, size={0}", size);
```

```
        // 创建一组对象，并指定它们的逻辑大小
        for (Int32 count = 0; count < 15; count++) {
            new BigNativeResource(size);
        }

        // 出于演示目的，强制一切都被清理
        GC.Collect();
    }

    private sealed class BigNativeResource {
        private Int32 m_size;

        public BigNativeResource(Int32 size) {
            m_size = size;
            // 使垃圾回收器认为对象在物理上比较大
            if (m_size > 0)  GC.AddMemoryPressure(m_size);
            Console.WriteLine("BigNativeResource create.");
        }

        ~BigNativeResource() {
            // 使垃圾回收器认为对象释放了更多的内存
            if (m_size > 0) GC.RemoveMemoryPressure(m_size);
            Console.WriteLine("BigNativeResource destroy.");
        }
    }

    private static void HandleCollectorDemo() {
        Console.WriteLine();
        Console.WriteLine("HandleCollectorDemo");
        for (Int32 count = 0; count < 10; count++) {
            new LimitedResource();
        }

        // 出于演示目的，强制一切都被清理
        GC.Collect();
    }

    private sealed class LimitedResource {
        // 创建一个 HandleCollector，告诉它当两个或者更多这样的对象
        // 存在于堆中时，就执行回收
        private static readonly HandleCollector s_hc =
            new HandleCollector("LimitedResource", 2);

        public LimitedResource() {
            // 告诉 HandleCollector 堆中增加了一个 LimitedResource 对象
            s_hc.Add();
            Console.WriteLine("LimitedResource create. Count={0}", s_hc.Count);
        }

        ~LimitedResource() {
            // 告诉 HandleCollector 堆中移除了一个 LimitedResource 对象
            s_hc.Remove();
            Console.WriteLine("LimitedResource destroy. Count={0}", s_hc.Count);
        }
    }
}
```

编译并运行上述代码，会得到和下面相似的输出。

```
MemoryPressureDemo, size=0
BigNativeResource create.
BigNativeResource create.
BigNativeResource create.
BigNativeResource create.
BigNativeResource create.
BigNativeResource create.
```

```
BigNativeResource create.
BigNativeResource create.
BigNativeResource create.
BigNativeResource create.
BigNativeResource create.
BigNativeResource create.
BigNativeResource create.
BigNativeResource create.
BigNativeResource create.
BigNativeResource destroy.
BigNativeResource destroy.
BigNativeResource destroy.
BigNativeResource destroy.
BigNativeResource destroy.
BigNativeResource destroy.
BigNativeResource destroy.
BigNativeResource destroy.
BigNativeResource destroy.
BigNativeResource destroy.
BigNativeResource destroy.
BigNativeResource destroy.
BigNativeResource destroy.
BigNativeResource destroy.

MemoryPressureDemo, size=10485760
BigNativeResource create.
BigNativeResource create.
BigNativeResource create.
BigNativeResource create.
BigNativeResource create.
BigNativeResource create.
BigNativeResource create.
BigNativeResource create.
BigNativeResource create.
BigNativeResource create.
BigNativeResource create.
BigNativeResource create.
BigNativeResource destroy.
BigNativeResource destroy.
BigNativeResource destroy.
BigNativeResource destroy.
BigNativeResource destroy.
BigNativeResource destroy.
BigNativeResource destroy.
BigNativeResource create.
BigNativeResource create.
BigNativeResource create.
BigNativeResource destroy.
BigNativeResource destroy.
BigNativeResource destroy.
BigNativeResource destroy.
BigNativeResource destroy.
BigNativeResource destroy.
BigNativeResource destroy.
BigNativeResource destroy.

HandleCollectorDemo
LimitedResource create. Count=1
LimitedResource create. Count=2
LimitedResource create. Count=3
LimitedResource destroy. Count=3
LimitedResource destroy. Count=2
LimitedResource destroy. Count=1
LimitedResource create. Count=1
LimitedResource create. Count=2
LimitedResource create. Count=3
```

```
LimitedResource destroy. Count=2
LimitedResource create. Count=3
LimitedResource destroy. Count=3
LimitedResource destroy. Count=2
LimitedResource destroy. Count=1
LimitedResource create. Count=1
LimitedResource create. Count=2
LimitedResource create. Count=3
LimitedResource destroy. Count=2
LimitedResource destroy. Count=1
LimitedResource destroy. Count=0
```

21.3.4　终结的内部工作原理

终结表面上很简单：创建对象，当它被回收时，它的 **Finalize** 方法得以调用。但一旦深究下去，就会发现终结的门道很多。

应用程序创建新对象时，**new** 操作符会从堆中分配内存。如果对象的类型定义了 **Finalize** 方法，那么在该类型的实例构造器被调用之前，会将指向该对象的指针放到一个**终结列表**(finalization list)中。终结列表是由垃圾回收器控制的一个内部数据结构。列表中的每一项都指向一个对象——回收该对象的内存前应调用它的 **Finalize** 方法。

图 21-13 展示了包含几个对象的堆。有的对象从应用程序的根可达，有的不可达。对象 C，E，F，I 和 J 被创建时，系统检测到这些对象的类型定义了 **Finalize** 方法，所以将指向这些对象的指针添加到终结列表中。

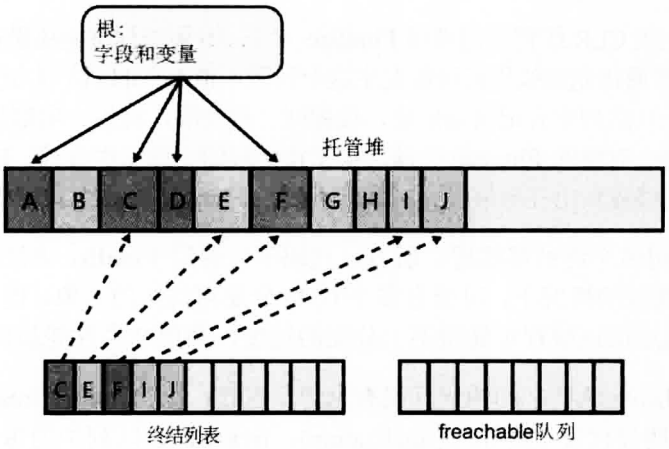


图 21-13　托管堆的络结列表包含了指向对象的指针



注意　虽然 **System.Object** 定义了 **Finalize** 方法，但 CLR 知道忽略它。也就是说，构造类型的实例时，如果该类型的 **Finalize** 方法是从 **System.Object** 继承的，就不认为这个对象是“可终结”的。类型必须重写 **Object** 的 **Finalize** 方法，这个类型及其派生类型的对象才被认为是“可终结”的。

垃圾回收开始时，对象 B，E，G，H，I 和 J 被判定为垃圾。垃圾回收器扫描终结列表以查找对这些对象的引用。找到一个引用后，该引用会从终结列表中移除，并附加到 freachable 队列。freachable 队列(发音是“F-reachable”)也是垃圾回收器的一种内部数据结构。队列中的每个引用都代表其 **Finalize** 方法已准备好调用的一个对象。图 21-14 展示了回收完毕后的托管堆。

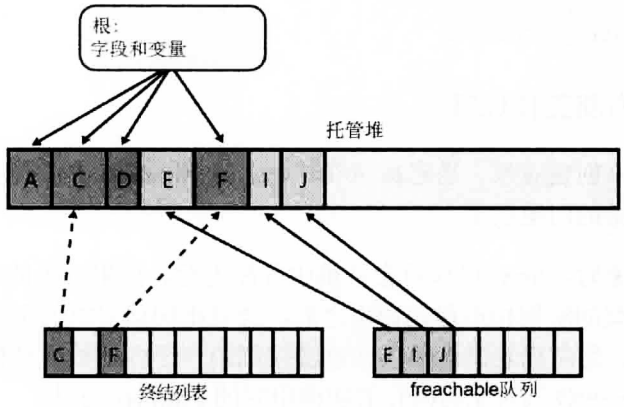


图 21-14　在托管堆中，一些指针从终结列表移至 freachable 队列

从图 21-14 可以看出，对象 B，G 和 H 占用的内存已被回收，因为它们没有 **Finalize** 方法。但对象 E，I 和 J 占用的内存暂时不能回收，因为它们的 **Finalize** 方法还没有调用。

一个特殊的高优先级 CLR 线程专门调用 **Finalize** 方法。专用线程可避免潜在的线程同步问题。(使用应用程序的普通优先级线程就可能发生这个问题。)freachable 队列为空时(这很常见)，该线程将睡眠。但一旦队列中有记录项出现，线程就会被唤醒，将每一项都从 freachable 队列中移除，同时调用每个对象的 **Finalize** 方法。由于该线程的特殊工作方式，**Finalize** 中的代码不应该对执行代码的线程做出任何假设。例如，不要在 **Finalize** 方法中访问线程的本地存储。

CLR 未来可能使用多个终结器线程。所以，代码不应假设 **Finalize** 方法会被连续调用。在只有一个终结器线程的情况下，可能有多个 CPU 分配可终结的对象，但只有一个线程执行 **Finalize** 方法，这造成该线程可能跟不上分配的速度，从而产生性能和伸缩性方面的问题。

终结列表和 freachable 队列之间的交互很有意思。首先，让我告诉你 freachable 队列这个名称的由来。“f”明显代表“终结”(finalization)；freachable 队列中的每个记录项都是对托管堆中应调用其 **Finalize** 方法的一个对象的引用。“reachable”意味着对象是可达的。换言之，可将 freachable 队列看成是像静态字段那样的一个根。所以，freachable 队列中的引用使它指向的对象保持可达，不是垃圾。

简单地说，当一个对象不可达时，垃圾回收器就把它视为垃圾。但是，当垃圾回收器将对象的引用从终结列表移至 freachable 队列时，对象不再被认为是垃圾，不能回收它的内存。对象被视为垃圾又变得不是垃圾，我们说对象被**复活**了。

标记 freachable 对象时，将递归标记对象中的引用类型的字段所引用的对象；所有这些对象也必须复活以便在回收过程中存活。之后，垃圾回收器才结束对垃圾的标识。在这个过

程中，一些原本被认为是垃圾的对象复活了。然后，垃圾回收器压缩(移动)可回收的内存，将复活的对象提升到较老的一代(这不理想)。现在，特殊的终结线程清空 freachable 队列，执行每个对象的 **Finalize** 方法。

下次对老一代进行垃圾回收时，会发现已终结的对象成为真正的垃圾，因为没有应用程序的根指向它们，freachable 队列也不再指向它们。所以，这些对象的内存会直接回收。整个过程中，注意，可终结对象需要执行两次垃圾回收才能释放它们占用的内存。在实际应用中，由于对象可能被提升至另一代，所以可能要求不止进行两次垃圾回收。图 21-15 展示了第二次垃圾回收后托管堆的情况。

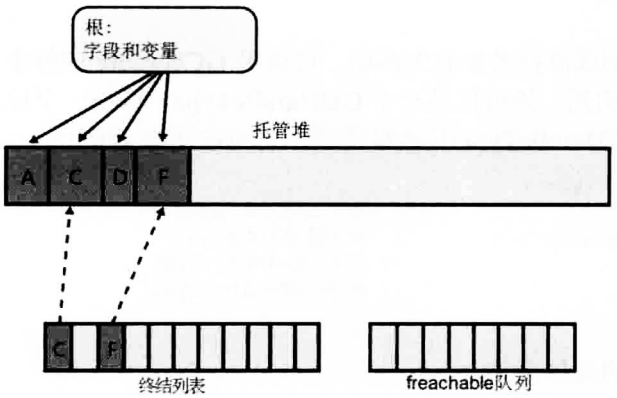


图 21-15　第二次垃圾回收后托管堆的情况

21.3.5　手动监视和控制对象的生存期

CLR 为每个 AppDomain 都提供了一个 **GC 句柄表**(GC Handle table)，允许应用程序监视或手动控制对象的生存期。这个表在 AppDomain 创建之初是空白的。表中每个记录项都包含以下两种信息：对托管堆中的一个对象的引用，以及指出如何监视或控制对象的标志(flag)。应用程序使用如下所示的 **System.Runtime.InteropServices.GCHandle** 类型在表中添加或删除记录项。

```
// 该类型在 System.Runtime.InteropServices 命名空间中定义
public struct GCHandle {
   // 静态方法，用于在表中创建一个记录项
   public static GCHandle Alloc(object value);
   public static GCHandle Alloc(object value, GCHandleType type);

   // 静态方法，用于将一个 GCHandle 转换成一个 IntPtr
   public static explicit operator IntPtr(GCHandle value);
   public static IntPtr ToIntPtr(GCHandle value);

   // 静态方法，用于将一个 IntPtr 转换为一个 GCHandle
   public static explicit operator GCHandle(IntPtr value);
   public static GCHandle FromIntPtr(IntPtr value);

   // 静态方法，用于比较两个 GCHandle
   public static Boolean operator ==(GCHandle a, GCHandle b);
   public static Boolean operator !=(GCHandle a, GCHandle b);

   // 实例方法，用于释放表中的记录项 (索引设为 0)
```

```
    public void Free();

    // 实例属性，用于 get/set 记录项的对象引用
    public object Target { get; set; }

    // 实例属性，如果索引不为 0，就返回 true
    public Boolean IsAllocated { get; }

    // 对于已固定(pinned)的记录项，这个方法返回对象的地址
    public IntPtr AddrOfPinnedObject();

    public override Int32 GetHashCode();
    public override Boolean Equals(object o);
}
```

简单地说，为了控制或监视对象的生存期，可调用 **GCHandle** 的静态 **Alloc** 方法并传递想控制/监视的对象的引用。还可传递一个 **GCHandleType**，这是一个标志，指定了你想如何控制/监视对象。**GCHandleType** 是枚举类型，它的定义如下所示：

```
public enum GCHandleType {
    Weak = 0,                       // 用于监视对象的存在
    WeakTrackResurrection = 1       // 用于监视对象的存在
    Normal = 2,                     // 用于控制对象的生存期
    Pinned = 3                      // 用于控制对象的生存期
}
```

下面解释每个标志的具体含义。

- Weak
  该标志允许监视对象的生存期。具体地说，可检测垃圾回收器在什么时候判定该对象在应用程序代码中不可达。注意，此时对象的 **Finalize** 方法可能执行，也可能没有执行，对象可能还在内存中。

- WeakTrackResurrection
  该标志允许监视对象的生存期。具体地说，可检测垃圾回收器在什么时候判定该对象在应用程序的代码中不可达。注意，此时对象的 **Finalize** 方法(如果有的话)已经执行，对象的内存已经回收。

- Normal
  该标志允许控制对象的生存期。具体地说，是告诉垃圾回收器：即使应用程序中没有变量(根)引用该对象，该对象也必须留在内存中。垃圾回收发生时，该对象的内存可以压缩(移动)。不向 **Alloc** 方法传递任何 **GCHandleType** 标志，就默认使用 **GCHandleType.Normal**。

- Pinned
  该标志允许控制对象的生存期。具体地说，是告诉垃圾回收器：即使应用程序中没有变量(根)引用该对象，该对象也必须留在内存中。垃圾回收发生时，该对象的内存不能压缩(移动)。需要将内存地址交给本机代码时，这个功能很好用。本机代码知道 GC 不会移动对象，所以能放心地向托管堆的这个内存写入。

GCHandle 的静态 **Alloc** 方法在调用时会扫描 **AppDomain** 的 GC 句柄表，查找一个可用的记录项来存储传给 **Alloc** 的对象引用，并将标志设为你为 **GCHandleType** 实参传递的值。然后，**Alloc** 方法返回一个 **GCHandle** 实例。**GCHandle** 是轻量级的值类型，其中包含一个实例字段(一个 **IntPtr** 字段)，它引用了句柄表中的记录项索引。要释放 GC 句柄表中的这个记录项时，可以获取 **GCHandle** 实例，并在这个实例上调用 **Free** 方法。**Free** 方法将 **IntPtr** 字段设为 0，使实例变得无效。

下面展示了垃圾回收器如何使用 GC 句柄表。当垃圾回收发生时，垃圾回收器的行为如下。

1. 垃圾回收器标记所有可达的对象(本章开始的时候已进行了描述)。然后，垃圾回收器扫描 GC 句柄表：所有 **Normal** 或 **Pinned** 对象都被看成是根，同时标记这些对象(包括这些对象通过它们的字段引用的对象)。

2. 垃圾回收器扫描 GC 句柄表，查找所有 **Weak** 记录项。如果一个 **Weak** 记录项引用了未标记的对象，该引用标识的就是不可达对象(垃圾)，该记录项的引用值更改为 **null**。

3. 垃圾回收器扫描终结列表。在列表中，对未标记对象的引用标识的是不可达对象，这些引用从终结列表移至 freachable 队列。这时对象会被标记，因为对象又变成可达了。

4. 垃圾回收器扫描 GC 句柄表，查找所有 **WeakTrackResurrection** 记录项。如果一个 **WeakTrackResurrection** 记录项引用了未标记的对象(它现在是由 freachable 队列中的记录项引用的)，该引用标识的就是不可达对象(垃圾)，该记录项的引用值更改为 **null**。

5. 垃圾回收器对内存进行压缩，填补不可达对象留下的内存“空洞”，这其实就是一个内存碎片整理的过程。**Pinned** 对象不会压缩(移动)，垃圾回收器会移动它周围的其他对象。

理解了机制之后，接着看看在什么情况下使用。最容易理解的标志就是 **Normal** 和 **Pinned**，所以让我们从它们入手。这两个标志通常在和本机代码互操作时使用。

需要将托管对象的指针交给本机代码时使用 **Normal** 标记，因为本机代码将来要回调托管代码并传递指针。但不能就这么将托管对象的指针交给本机代码，因为如果垃圾回收发生，对象可能在内存中移动，指针便无效了。解决方案是调用 **GCHandle** 的 **Alloc** 方法，传递对象引用和 **Normal** 标志。将返回的 **GCHandle** 实例转型为 **IntPtr**，再将这个 **IntPtr** 传给本机代码。本机代码回调托管代码时，托管代码将传递的 **IntPtr** 转型为 **GCHandle**，查询 **Target** 属性获得托管对象的引用(当前地址)。本机代码不再需要这个引用之后，你可以调用 **GCHandle** 的 **Free** 方法，使未来的垃圾回收能够释放对象(假定不存在引用该对象的其他根)。

注意在这种情况下，本机代码并没有真正使用托管对象本身；它只是通过一种方式引用了对象。但某些时候，本机代码需要真正地使用托管对象。这时托管对象必须固定(pinned)，从而阻止垃圾回收器压缩(移动)对象。常见的例子是将托管 **String** 对象传给某个 Win32 函数。这时 **String** 对象必须固定。不能将对托管对象的引用传给本机代码，然后任由垃圾回

收器在内存中移动对象。**String** 对象被移走了，本机代码就会向已经不包含 **String** 对象的字符内容的内存进行读写——导致应用程序的行为无法预测。

使用 CLR 的 P/Invoke 机制调用方法时，CLR 会自动帮你固定实参，并在本机方法返回时自动解除固定。所以，大多数时候都不必使用 **GCHandle** 类型来显式固定任何托管对象。只有在将托管对象的指针传给本机代码，然后本机函数返回，但本机代码将来仍需使用该对象时，才需要显式使用 **GCHandle** 类型。最常见的例子就是执行异步 I/O 操作。

假定分配了一个字节数组，并准备在其中填充来自一个套接字的数据。这时应该调用 **GCHandle** 的 **Alloc** 方法，传递对数组对象的引用以及 **Pinned** 标志。然后，在返回的 **GCHandle** 实例上调用 **AddrOfPinnedObject** 方法。这会返回一个 **IntPtr**，它是已固定对象在托管堆中的实际地址。然后，将该地址传给本机函数，该函数立即返回至托管代码。数据从套接字传来时，该字节数组缓冲区在内存中不会移动；阻止移动是 **Pinned** 标志的功劳。异步 I/O 操作完毕后调用 **GCHandle** 的 **Free** 方法，以后垃圾回收时就可以移动缓冲区了。托管代码应包含一个缓冲区引用，这使你能访问数据。正是由于这个引用的存在，所以才会阻止垃圾回收从内存中彻底释放该缓冲区。

值得注意的是，C#提供了一个 **fixed** 语句，它能在一个代码块中固定对象。以下代码演示了该语句是如何使用的：

```
unsafe public static void Go() {
   // 分配一系列立即变成垃圾的对象
   for (Int32 x = 0; x < 10000; x++) new Object();

   IntPtr originalMemoryAddress;
   Byte[] bytes = new Byte[1000]; // 在垃圾对象后分配这个数组

   // 获取 Byte[]在内存中的地址
   fixed (Byte* pbytes = bytes) { originalMemoryAddress = (IntPtr) pbytes; }

   // 强迫进行一次垃圾回收；垃圾对象会被回收，Byte[]可能被压缩
   GC.Collect();

   // 获取 Byte[]当前在内存中的地址，把它同第一个地址比较
   fixed (Byte* pbytes = bytes) {
      Console.WriteLine("The Byte[] did{0} move during the GC",
         (originalMemoryAddress == (IntPtr) pbytes) ? " not" : null);
   }
}
```

使用 C#的 **fixed** 语句比分配一个固定 GC 句柄高效得多。这里发生的事情是，C#编译器在 **pbytes** 局部变量上生成一个特殊的“已固定”标志。垃圾回收期间，GC 检查这个根的内容，如果根不为 **null**，就知道在压缩阶段不要移动变量引用的对象。C#编译器生成 IL 将 **pbytes** 局部变量初始化为 **fixed** 块起始处的对象的地址。在 **fixed** 块的尾部，编译器还会生成 IL 指令将 **pbytes** 局部变量设回 **null**，使变量不引用任何对象。这样一来，下一次垃圾回收发生时，对象就可以移动了。

现在讨论一下另两个标志，即 **Weak** 和 **WeakTrackResurrection**。它们既可用于和本机代码的互操作，也可在只有托管代码的时候使用。**Weak** 标志使你知道在什么时候一个对象被判定为垃圾，但这时对象的内存不一定被回收。**WeakTrackResurrection** 标志使你知道

在什么时候对象的内存已被回收。两个标志中 **Weak** 更常用。事实上，我还没有见过任何人在真正的应用程序中用过 **WeakTrackResurrection** 标志。

假定 **Object-A** 定时在 **Object-B** 上调用一个方法。但由于 **Object-A** 有一个对 **Object-B** 的引用，所以阻止了 **Object-B** 被垃圾回收。在极少数情况下，这可能并不是你希望的；相反，你可能希望只要 **Object-B** 仍然存活于托管堆中，**Object-A** 就调用 **Object-B** 的方法。为此，**Object-A** 要调用 **GCHandle** 的 **Alloc** 方法，向方法传递对 **Object-B** 的引用和 **Weak** 标志。**Object-A** 现在只需保存返回的 **GCHandle** 实例，而不是保存对 **Object-B** 的引用。

在这种情况下，如果没有其他根引用 **Object-B**(保持其存活状态)，**Object-B** 就可以被垃圾回收。**Object-A** 每次想调用 **Object-B** 的方法时，都必须查询 **GCHandle** 的只读属性 **Target**。如果该属性返回非 **null** 的值，就表明 **Object-B** 依然存活。然后，**Object-A** 的代码会将返回的引用转型为 **Object-B** 的类型，并调用方法。如果 **Target** 属性返回 **null**，表明 **Object-B** 已被回收，**Object-A** 不再尝试调用方法。在这个时候，**Object-A** 的代码也许还要调用 **GCHandle** 的 **Free** 方法来释放 **GCHandle** 实例。

由于使用 **GCHandle** 类型有些繁琐，而且要求提升的安全性才能在内存中保持或固定对象，所以 **System** 命名空间提供了一个 **WeakReference<T>**类来帮助你。

```
public sealed class WeakReference<T> : ISerializable where T : class {
   public WeakReference(T target);
   public WeakReference(T target, Boolean trackResurrection);
   public void SetTarget(T target);
   public Boolean TryGetTarget(out T target);
}
```

这个类其实是包装了一个 **GCHandle** 实例的面向对象包装器(wrapper)：逻辑上说，它的构造器调用 **GCHandle** 的 **Alloc** 方法，**TryGetTarget** 方法查询 **GCHandle** 的 **Target** 属性，**SetTarget** 方法设置 **GCHandle** 的 **Target** 属性，而 **Finalize** 方法(这里未显示，因为是受保护方法)则调用 **GCHandle** 的 **Free** 方法。此外，代码无需特殊权限即可使用 **WeakReference<T>**类，因为该类只支持弱引用；不支持 **GCHandleType** 设为 **Normal** 或 **Pinned** 的 **GCHandle** 实例的行为。**WeakReference<T>**类的缺点在于它的实例必须在堆上分配。所以，**WeakReference** 类比 **GCHandle** 实例更“重”。

开发人员经常需要将一些数据和另一个实体关联。例如，数据可以和一个线程或 AppDomain 关联。另外，可用 **System.Runtime.CompilerServices.ConditionalWeakTable<TKey,TValue>** 类将数据和单独的对象关联，如下所示：

```
public sealed class ConditionalWeakTable<TKey, TValue>
   where TKey : class where TValue : class {
   public ConditionalWeakTable();
   public void Add(TKey key, TValue value);
   public TValue GetValue(TKey key, CreateValueCallback<TKey, TValue>
      createValueCallback);
   public Boolean TryGetValue(TKey key, out TValue value);
   public TValue GetOrCreateValue(TKey key);
   public Boolean Remove(TKey key);

   public delegate TValue CreateValueCallback(TKey key); // 嵌套的委托定义
}
```



重要提示 开发人员刚开始学习弱引用时，会马上想到它们在缓存情形中的用处。例如，他们会想到构造包含大量数据的一组对象，并创建对这些对象的弱引用。程序需要数据时就检查这些弱引用，看看包含这些数据的对象是否依然“健在”。对象还在，程序就直接使用对象；这样程序就会有较好的性能。但如果发生垃圾回收，包含数据的对象就会被销毁。而一旦需要重新创建数据，程序的性能就会受到影响。

这个技术的问题在于：垃圾回收不是在内存满或接近满时才发生的。相反，只要第 0 代满了，垃圾回收就会发生。所以，对象在内存中被抛弃的频率比预想的高得多，应用程序的性能将大打折扣。

弱引用在缓存情形中确实能得到高效应用。但构建良好的缓存算法来找到内存消耗与速度之间的平衡点十分复杂。简单地说，你希望缓存保持对自己的所有对象的强引用，内存吃紧就开始将强引用转换为弱引用。目前，CLR 没有提供一个能告诉应用程序内存吃紧的机制。但已经有人通过定时调用 Win32 **GlobalMemoryStatusEx** 函数并检查返回的 **MEMORYSTATUSEX** 结构的 **dwMemoryLoad** 成员成功做到了这一点。如果该成员报告大于 80 的值，内存空间就处于吃紧状态。然后可以开始将强引用转换为弱引用——可依据的算法包括：最近最少使用算法 (Least-Recently Used algorithm，LRU)、最频繁使用算法 (Most-Frequently Used algorithm，MFU)以及某个时基算法(time-base algorithm)等。

任意数据要和一个或多个对象关联，首先要创建该类的实例。然后调用 **Add** 方法，为 **key** 参数传递对象引用，为 **value** 参数传递想和对象关联的数据。试图多次添加对同一个对象的引用，**Add** 方法会抛出 **ArgumentException**；要更改和对象关联的值，必须先删除 **key**，再用新值把它添加回来。注意这个类是线程安全的，多个线程能同时使用它(虽然这也意味着类的性能并不出众)；应测试好这个类的性能，验证它在是否适合你的实际环境。

当然，表对象在内部存储了对作为 **key** 传递的对象的弱引用(一个 **WeakReference** 对象)；这样可保证不会因为表的存在而造成对象“被迫”存活。但是，**ConditionalWeakTable** 类最特别的地方在于，它保证了只要 **key** 所标识的对象在内存中，值就肯定在内存中。这使其超越了一个普通的 **WeakReference**，因为如果是普通的 **WeakReference**，那么即使 **key** 对象保持存活，值也可能被垃圾回收。**ConditionalWeakTable** 类可用于实现 XAML 的依赖属性(dependency property)机制。动态语言也可在内部利用它将数据和对象动态关联。

以下代码演示了 **ConditionalWeakTable** 类的使用。它允许在任何对象上调用 **GCWatch** 扩展方法并传递一些 **String** 标签(在程序中作为通知消息显示)。然后，在特定对象被垃圾回收时，它通过控制台窗口发出通知：

```
internal static class ConditionalWeakTableDemo {
```

```
   public static void Main() {
      Object o = new Object().GCWatch("My Object created at " + DateTime.Now);
      GC.Collect();     // 此时看不到 GC 通知
      GC.KeepAlive(o); // 确定 o 引用的对象现在还活着
      o = null;         // o 引用的对象现在可以死了

      GC.Collect();     // 此时才会看到 GC 通知
      Console.ReadLine();
   }
}

internal static class GCWatcher {
   // 注意：由于字符串留用(interning)和 MarshalByRefObject 代理对象，所以
   // 使用 String 要当心
   private readonly static ConditionalWeakTable<Object, NotifyWhenGCd<String>> s_cwt =
      new ConditionalWeakTable<Object, NotifyWhenGCd<String>>();

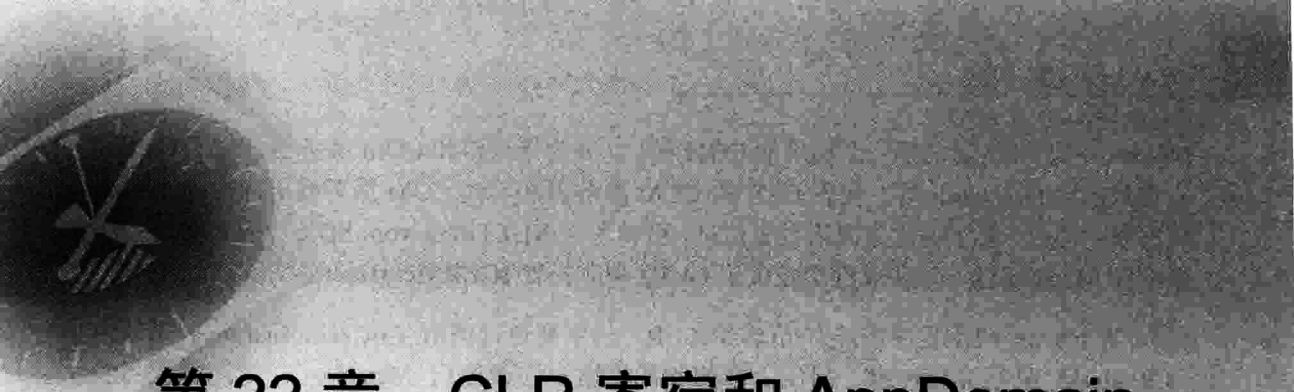
   private sealed class NotifyWhenGCd<T> {
      private readonly T m_value;

      internal NotifyWhenGCd(T value) { m_value = value; }
      public override string ToString() { return m_value.ToString(); }
      ~NotifyWhenGCd() { Console.WriteLine("GC'd: " + m_value); }
   }

   public static T GCWatch<T>(this T @object, String tag) where T : class{
      s_cwt.Add(@object, new NotifyWhenGCd<String>(tag));
      return @object;

   }

}
```



第 22 章　CLR 寄宿和 AppDomain

本章内容：

- CLR 寄宿
- AppDomain
- 卸载 AppDomain
- 监视 AppDomain
- AppDomain FirstChance 异常通知
- 宿主如何使用 AppDomain
- 高级宿主控制

本章主要讨论两个主题：寄宿和 AppDomain。这两个主题充分演示了 Microsoft .NET Framework 的巨大价值。寄宿(hosting)使任何应用程序都能利用 CLR 的功能。特别要指出的是，它使现有的应用程序至少能部分使用托管代码编写。另外，寄宿还为应用程序提供了通过编程来进行自定义和扩展的能力。

允许可扩展性意味着第三方代码可在你的进程中运行。在 Windows 中将第三方 DLL 加载到进程中意味着冒险。DLL 中的代码很容易破坏应用程序的数据结构和代码。DLL 还可能企图利用应用程序的安全上下文来访问它本来无权访问的资源。CLR 的 AppDomain 功能解决了所有这些问题。AppDomain 允许第三方的、不受信任的代码在现有的进程中运行，而 CLR 保证数据结构、代码和安全上下文不被滥用或破坏。

程序员经常将寄宿和 AppDomain 与程序集的加载和反射一起使用。这 4 种技术一起使用，使 CLR 成为一个功能极其丰富和强大的平台。本章重点在于寄宿和 AppDomain。下一章则会将重点放在程序集加载和反射上。学习并理解了所有这些技术后，会发现今天在.NET Framework 上面的投资，将来必会获得丰厚回报。

22.1　CLR 寄宿

.NET Framework 在 Windows 平台的顶部运行。这意味着.NET Framework 必须用 Windows 能理解的技术来构建。首先，所有托管模块和程序集文件都必须使用 Windows PE 文件格式，而且要么是 Windows EXE 文件，要么是 DLL 文件。

开发 CLR 时，Microsoft 实际是把它实现成包含在一个 DLL 中的 COM 服务器。也就是说，

Microsoft 为 CLR 定义了一个标准的 COM 接口，并为该接口和 COM 服务器分配了 GUID。安装.NET Framework 时，代表 CLR 的 COM 服务器和其他 COM 服务器一样在 Windows 注册表中注册。要了解这方面的更多信息，可参考与.NET Framework SDK 一起发布的 C++ 头文件 MetaHost.h。该头文件中定义了 GUID 和非托管 **ICLRMetaHost** 接口。

任何 Windows 应用程序都能寄宿(容纳)CLR。但不要通过调用 **CoCreateInstance** 来创建 CLR COM 服务器的实例，相反，你的非托管宿主应该调用 MetaHost.h 文件中声明的 **CLRCreateInstance** 函数。**CLRCreateInstance** 函数在 MSCorEE.dll 文件中实现，该文件一般在 C:\Windows\System32 目录中。这个 DLL 被人们亲切地称为“垫片”(shim)，它的工作是决定创建哪个版本的 CLR；垫片 DLL 本身不包含 CLR COM 服务器。

一台机器可安装多个版本的 CLR，但只有一个版本的 MSCorEE.dll 文件(垫片)[①]。机器上安装的 MSCorEE.dll 是与机器上安装的最新版本的 CLR 一起发布的那个版本。所以，该版本的 MSCorEE.dll 知道如何查找机器上的老版本 CLR。

包含实际 CLR 代码的文件的名称在不同版本的 CLR 中是不同的。版本 1.0，1.1 和 2.0 的 CLR 代码在 MSCorWks.dll 文件中；版本 4 则在 Clr.dll 文件中。由于一台机器可能安装多个版本的 CLR，所以这些文件安装到不同的目录，如下所示。[②]

- 版本 1.0 在 C:\Windows\Microsoft.Net\Framework\v1.0.3705 中。
- 版本 1.1 在 C:\Windows\Microsoft.Net\Framework\v1.0.4322 中。
- 版本 2.0 在 C:\Windows\Microsoft.Net\Framework\v2.0.50727 中。
- 版本 4 在 C:\Windows\Microsoft.NET\Framework\v4.0.30319[③]中。

CLRCreateInstance 函数可返回一个 **ICLRMetaHost** 接口。宿主应用程序可调用这个接口的 **GetRuntime** 函数，指定宿主要创建的 CLR 的版本。然后，垫片将所需版本的 CLR 加载到宿主的进程中。

默认情况下，当一个托管的可执行文件启动时，垫片会检查可执行文件，提取当初生成和测试应用程序时使用的 CLR 的版本信息。但应用程序可以在它的 XML 配置文件中设置 **requiredRuntime** 和 **supportedRuntime** 这两项来覆盖该默认行为。(XML 配置文件的详情请参见第 2 章和第 3 章。)

GetRuntime 函数返回指向非托管 **ICLRRuntimeInfo** 接口的指针。有了这个指针后，就可

---

① 使用 64 位 Windows 实际会安装两个版本的 MSCorEE.dll 文件。一个是 32 位 x86 版本，在 C:\Windows\SysWOW64 目录中；另一个是 64 位 x64 或 IA64 版本(取决于计算机的 CPU 架构)，在 C:\Windows\System32 目录中。

② 注意，.NET Framework 3.0 和 3.5 是与 CLR 2.0 一起发布的。我没有显示.NET Framework 3.0 和 3.5 的目录，因为 CLR DLL 是从 v2.0.50727 目录加载的。

③ 这是本书出版时的.NET Framework 4 目录。在你的机器上可能有所不同。——译注

利用 **GetInterface** 方法获得 **ICLRRuntimeHost** 接口。宿主应用程序可调用该接口定义的方法来做下面这些事情。

- 设置宿主管理器。告诉 CLR 宿主想参与涉及以下操作的决策：内存分配、线程调度/同步以及程序集加载等。宿主还可声明它想获得有关垃圾回收启动和停止以及特定操作超时的通知。
- 获取 CLR 管理器。告诉 CLR 阻止使用某些类/成员。另外，宿主能分辨哪些代码可以调试，哪些不可以，以及当特定事件(例如 AppDomain 卸载、CLR 停止或者堆栈溢出异常)发生时宿主应调用哪个方法。
- 初始化并启动 CLR。
- 加载程序集并执行其中的代码。
- 停止 CLR，阻止任何更多的托管代码在 **Windows** 进程中运行。



注意　Windows 进程完全可以不加载 CLR，只有在进程中执行托管代码时才进行加载。在.NET Framework 4 之前，CLR 只允许它的一个实例寄宿在 Windows 进程中。换言之，在一个进程中，要么不包含任何 CLR，要么只能包含 CLR v1.0，CLR v1.1 或者 CLR 2.0 之一。每进程仅一个版本的 CLR 显然过于局限。例如，这样 Microsoft Office Outlook 就不能加载为不同版本的.NET Framework 生成和测试的两个加载项了。

但是，随着.NET Framework 4 的发布，Microsoft 支持在一个 Windows 进程中同时加载 CLR v2.0 和 v4.0，为.NET Framework 2.0 和 4.0 写的不同组件能同时运行，不会出现任何兼容性问题。这是一个令人激动的功能，因为它极大扩展了.NET Framework 组件的应用场合。可利用 ClrVer.exe 工具检查给定的进程加载的是哪个(哪些)版本的 CLR。

一个 CLR 加载到 Windows 进程之后，便永远不能卸载；在 **ICLRRuntimeHost** 接口上调用 **AddRef** 和 **Release** 方法是没有作用的。CLR 从进程中卸载的唯一途径就是终止进程，这会造成 Windows 清理进程使用的所有资源。

22.2　AppDomain

CLR COM 服务器初始化时会创建一个 AppDomain。AppDomain 是一组程序集的逻辑容器。CLR 初始化时创建的第一个 AppDomain 称为“默认 AppDomain”，这个默认的 AppDomain 只有在 Windows 进程终止时才会被销毁。

除了默认 AppDomain，正在使用非托管 COM 接口方法或托管类型方法的宿主还可要求 CLR 创建额外的 AppDomain。AppDomain 是为了提供隔离而设计的。下面总结了

AppDomain 的具体功能。

- **一个 AppDomain 的代码不能直接访问另一个 AppDomain 的代码创建的对象**
  一个 AppDomain 中的代码创建了一个对象后，该对象便被该 AppDomain“拥有”。换言之，它的生存期不能超过创建它的代码所在的 AppDomain。一个 AppDomain 中的代码要访问另一个 AppDomain 中的对象，只能使用“按引用封送”(marshal-by-reference)或者“按值封送”(marshal-by-value)的语义。这就强制建立了清晰的分隔和边界，因为一个 AppDomain 中的代码不能直接引用另一个 AppDomain 中的代码创建的对象。这种隔离使 AppDomain 能很容易地从进程中卸载，不会影响其他 AppDomain 正在运行的代码。

- **AppDomain 可以卸载**
  CLR 不支持从 AppDomain 中卸载特定的程序集。但可以告诉 CLR 卸载一个 AppDomain，从而卸载该 AppDomain 当前包含的所有程序集。

- **AppDomain 可以单独保护**
  AppDomain 创建后会应用一个权限集，它决定了向这个 AppDomain 中运行的程序集授予的最大权限。正是由于存在这些权限，所以当宿主加载一些代码后，可以保证这些代码不会破坏(或读取)宿主本身使用的一些重要数据结构。

- **AppDomain 可以单独配置**
  AppDomain 创建后会关联一组配置设置。这些设置主要影响 CLR 在 AppDomain 中加载程序集的方式。涉及搜索路径、版本绑定重定向、卷影复制以及加载器优化。



重要提示 Windows 的一个重要特色是让每个应用程序都在自己的进程地址空间中运行。这就保证了一个应用程序的代码不能访问另一个应用程序使用的代码或数据。进程隔离可防范安全漏洞、数据破坏和其他不可预测的行为，确保了 Windows 系统以及在它上面运行的应用程序的健壮性。遗憾的是，在 Windows 中创建进程的开销很大。Win32 CreateProcess 函数的速度很慢，而且 Windows 需要大量内存来虚拟化进程的地址空间。

但是，如果应用程序完全由托管代码构成(这些代码的安全性可以验证)，同时这些代码没有调用非托管代码，那么在一个 Windows 进程中运行多个托管应用程序是没有问题的。AppDomain 提供了保护、配置和终止其中每一个应用程序所需的隔离。

图 22-1 演示了一个 Windows 进程，其中运行着一个 CLR COM 服务器。该 CLR 当前管理着两个 AppDomain(虽然在一个 Windows 进程中可以运行的 AppDomain 数量没有硬性限制)。每个 AppDomain 都有自己的 Loader 堆，每个 Loader 堆都记录了自 AppDomain 创建以来已访问过哪些类型。这些类型对象已在第 4 章讨论过，Loader 堆中的每个类型对象都有一个方法表，方法表中的每个记录项都指向 JIT 编译的本机代码(前提是方法至少执行过一次)。

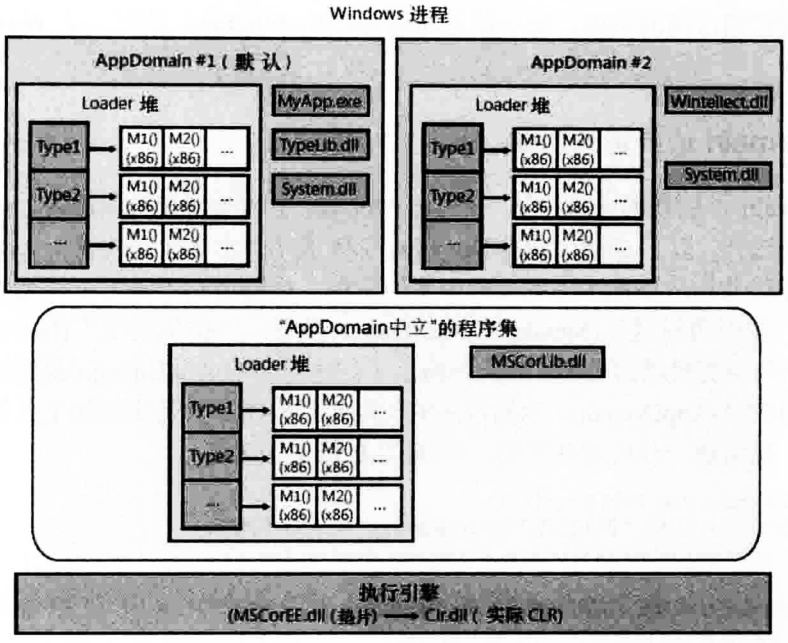


图 22-1　寄宿了 CLR 和两个 AppDomain 的一个 Windows 进程

除此之外，每个 AppDomain 都加载了一些程序集。AppDomain #1(默认 AppDomain)有三个程序集：MyApp.exe，TypeLib.dll 和 System.dll。AppDomain #2 有两个程序集：Wintellect.dll 和 System.dll。

如图 22-1 所示，两个 AppDomain 都加载了 System.dll 程序集。如果这两个 AppDomain 都使用了来自 System.dll 的一个类型，那么两个 AppDomain 的 Loader 堆会为相同的类型分别分配一个类型对象；类型对象的内存不会由两个 AppDomain 共享。另外，一个 AppDomain 中的代码调用一个类型定义的方法时，方法的 IL 代码会进行 JIT 编译，生成的本机代码单独与每个 AppDomain 关联，而不是由调用它的所有 AppDomain 共享。

不共享类型对象的内存或本机代码显得有些浪费。但 AppDomain 的设计宗旨就是提供隔离；CLR 要求在卸载某个 AppDomain 并释放其所有资源时不会影响到其他任何 AppDomain。复制 CLR 的数据结构才能保证这一点。另外，还保证多个 AppDomain 使用的类型在每个 AppDomain 中都有一组静态字段。

有的程序集本来就要由多个 AppDomain 使用。最典型的例子就是 MSCorLib.dll。该程序集包含了 **System.Object**，**System.Int32** 以及其他所有与.NET Framework 密不可分的类型。CLR 初始化时，该程序集会自动加载，而且所有 AppDomain 都共享该程序集中的类型。为了减少资源消耗，MSCorLib.dll 程序集以一种“AppDomain 中立”的方式加载。也就是说，针对以“AppDomain 中立”的方式加载的程序集，CLR 会为它们维护一个特殊的 Loader 堆。该 Loader 堆中的所有类型对象，以及为这些类型定义的方法 JIT 编译生成的所有本机代码，都会由进程中的所有 AppDomain 共享。遗憾的是，共享这些资源所获得的收益并不是没有代价的。这个代价就是，以“AppDomain 中立”的方式加载的所有程序集永远不能

卸载。要回收它们占用的资源，唯一的办法就是终止 Windows 进程，让 Windows 去回收资源。

跨越 AppDomain 边界访问对象

一个 AppDomain 中的代码可以和另一个 AppDomain 中的类型和对象通信，但只能通过良好定义的机制进行。以下 Ch22-1-AppDomains 示例程序演示了如何创建新 AppDomain，在其中加载程序集并构造该程序集定义的类型的实例。代码演示了以下三种类型在构造时的不同行为："按引用封送"(Marshal-by-Reference)类型，"按值封送"(Marshal-by-Value)类型，以及完全不能封送的类型。代码还演示了创建它们的 AppDomain 卸载时这些对象的不同行为。Ch22-1-AppDomains 示例程序的代码实际很少，只是我添加了大量注释。在代码清单之后，我将逐一分析这些代码，解释 CLR 所做的事情。

```
private static void Marshalling() {
   // 获取 AppDomain 引用("调用线程"当前正在该 AppDomain 中执行)
   AppDomain adCallingThreadDomain = Thread.GetDomain();

   // 每个 AppDomain 都分配了友好字符串名称(以便调试)
   // 获取这个 AppDomain 的友好字符串名称并显示它
   String callingDomainName = adCallingThreadDomain.FriendlyName;
   Console.WriteLine("Default AppDomain's friendly name={0}", callingDomainName);

   // 获取并显示我们的 AppDomain 中包含了"Main"方法的程序集
   String exeAssembly = Assembly.GetEntryAssembly().FullName;
   Console.WriteLine("Main assembly={0}", exeAssembly);

   // 定义局部变量来引用一个 AppDomain
   AppDomain ad2 = null;

   // *** DEMO 1: 使用 Marshal-by-Reference 进行跨 AppDomain 通信 ***
   Console.WriteLine("{0}Demo #1", Environment.NewLine);

   // 新建一个 AppDomain(从当前 AppDomain 继承安全性和配置)
   ad2 = AppDomain.CreateDomain("AD #2", null, null);
   MarshalByRefType mbrt = null;

   // 将我们的程序集加载到新 AppDomain 中，构造一个对象，把它
   // 封送回我们的 AppDomain(实际得到对一个代理的引用)
   mbrt = (MarshalByRefType)
      ad2.CreateInstanceAndUnwrap(exeAssembly, "MarshalByRefType");

   Console.WriteLine("Type={0}", mbrt.GetType());  // CLR 在类型上撒谎了

   // 证明得到的是对一个代理对象的引用
   Console.WriteLine("Is proxy={0}", RemotingServices.IsTransparentProxy(mbrt));

   // 看起来像是在 MarshalByRefType 上调用一个方法，实则不然、
   // 我们是在代理类型上调用一个方法，代理使线程切换到拥有对象的
   // 那个 AppDomain，并在真实的对象上调用这个方法
   mbrt.SomeMethod();

   // 卸载新的 AppDomain
   AppDomain.Unload(ad2);
```

```
// mbrt 引用一个有效的代理对象；代理对象引用一个无效的 AppDomain

try {
    // 在代理类型上调用一个方法。AppDomain 无效，造成抛出异常
    mbrt.SomeMethod();
    Console.WriteLine("Successful call.");
}
catch (AppDomainUnloadedException) {
    Console.WriteLine("Failed call.");
}

// *** DEMO 2: 使用 Marshal-by-Value 进行跨 AppDomain 通信 ***
Console.WriteLine("{0}Demo #2", Environment.NewLine);

// 新建一个 AppDomain(从当前 AppDomain 继承安全性和配置)
ad2 = AppDomain.CreateDomain("AD #2", null, null);

// 将我们的程序集加载到新 AppDomain 中，构造一个对象，把它
// 封送回我们的 AppDomain(实际得到对一个代理的引用)
mbrt = (MarshalByRefType)
    ad2.CreateInstanceAndUnwrap(exeAssembly, "MarshalByRefType");

// 对象的方法返回所返回对象的副本;
// 对象按值(而非按引用)封送
MarshalByValType mbvt = mbrt.MethodWithReturn();

// 证明得到的不是对一个代理对象的引用
Console.WriteLine("Is proxy={0}", RemotingServices.IsTransparentProxy(mbvt));

// 看起来是在 MarshalByValType 上调用一个方法，实际也是如此
Console.WriteLine("Returned object created " + mbvt.ToString());

// 卸载新的 AppDomain
AppDomain.Unload(ad2);
// mbvt 引用有效的对象；卸载 AppDomain 没有影响

try {
    // 我们是在对象上调用一个方法；不会抛出异常
    Console.WriteLine("Returned object created " + mbvt.ToString());
    Console.WriteLine("Successful call.");
}
catch (AppDomainUnloadedException) {
    Console.WriteLine("Failed call.");
}

// DEMO 3: 使用不可封送的类型进行跨 AppDomain 通信 ***
Console.WriteLine("{0}Demo #3", Environment.NewLine);

// 新建一个 AppDomain (从当前 AppDomain 继承安全性和配置)
ad2 = AppDomain.CreateDomain("AD #2", null, null);
// 将我们的程序集加载到新 AppDomain 中，构造一个对象，把它
// 封送回我们的 AppDomain(实际得到对一个代理的引用)
mbrt = (MarshalByRefType)
    ad2.CreateInstanceAndUnwrap(exeAssembly, "MarshalByRefType");

// 对象的方法返回一个不可封送的对象；抛出异常
NonMarshalableType nmt = mbrt.MethodArgAndReturn(callingDomainName);
// 这里永远执行不到...
```

```
}

// 该类的实例可跨越 AppDomain 的边界“按引用封送”
public sealed class MarshalByRefType : MarshalByRefObject {
   public MarshalByRefType() {
      Console.WriteLine("{0} ctor running in {1}",
         this.GetType().ToString(), Thread.GetDomain().FriendlyName);
   }

   public void SomeMethod() {
      Console.WriteLine("Executing in " + Thread.GetDomain().FriendlyName);
   }

   public MarshalByValType MethodWithReturn() {
      Console.WriteLine("Executing in" + Thread.GetDomain().FriendlyName);
      MarshalByValType t = new MarshalByValType();
      return t;
   }

   public NonMarshalableType MethodArgAndReturn(String callingDomainName) {
      // 注意: callingDomainName 是可序列化的
      Console.WriteLine("Calling from '{0}' to '{1}'.",
         callingDomainName, Thread.GetDomain().FriendlyName);
      NonMarshalableType t = new NonMarshalableType();
      return t;
   }
}

// 该类的实例可跨越 AppDomain 的边界“按值封送”
[Serializable]
public sealed class MarshalByValType : Object {
   private DateTime m_creationTime = DateTime.Now; // 注意:DateTime 是可序列化的

public MarshalByValType() {
   Console.WriteLine("{0} ctor running in {1}, Created on {2:D}",
      this.GetType().ToString(),
      Thread.GetDomain().FriendlyName,
      m_creationTime);
   }

   public override String ToString() {
      return m_creationTime.ToLongDateString();
   }
}

// 该类的实例不能跨 AppDomain 边界进行封送
// [Serializable]
public sealed class NonMarshalableType : Object {
   public NonMarshalableType() {
      Console.WriteLine("Executing in " + Thread.GetDomain().FriendlyName);
   }
}
```

生成并运行 Ch22-1-AppDomains 应用程序，获得以下输出结果：[①]

① 在本书配套代码中，找到 C22-1 AppDomains 项目，在 **Main()**方法中解除对 **Marshalling()**方法调用的注释。——译注

```
Default AppDomain's friendly name=Ch22-1-AppDomains.exe
Main assembly=Ch22-1-AppDomains, Version=0.0.0.0, Culture=neutral, PublicKeyToken=null

Demo #1
MarshalByRefType ctor running in AD #2
Type=MarshalByRefType
Is proxy=True
Executing in AD #2
Failed call.

Demo #2
MarshalByRefType ctor running in AD #2
Executing in AD #2
MarshalByValType ctor running in AD #2, Created on 2014年5月14日
Is proxy=False
Returned object created 2014年5月14日
Returned object created 2014年5月14日
Successful call.

Demo #3
MarshalByRefType ctor running in AD #2
Calling from 'Ch22-1-AppDomains.exe' to 'AD #2'.
Executing in AD #2

未经处理的异常： System.Runtime.Serialization.SerializationException: 程序集“Ch
22-1-AppDomains, Version=0.0.0.0, Culture=neutral, PublicKeyToken=null”中的类型
“NonMarshalableType”未标记为可序列化。
   在 MarshalByRefType.MethodArgAndReturn(String callingDomainName)
   在 Program.Marshalling()
   在 Program.Main()
```

现在来讨论以上代码以及 CLR 所做的事情。

Marshalling 方法首先获得一个 **AppDomain** 对象引用，当前调用线程正在该 AppDomain 中执行。在 Windows 中，线程总是在一个进程的上下文中创建，而且线程的整个生存期都在该进程的生存期内。但线程和 AppDomain 没有一对一关系。AppDomain 是一项 CLR 功能；Windows 对 AppDomain 一无所知。由于一个 Windows 进程可包含多个 AppDomain，所以线程能执行一个 AppDomain 中的代码，再执行另一个 AppDomain 中的代码。从 CLR 的角度看，线程一次只能执行一个 AppDomain 中的代码。线程可调用 **System.Threading.Thread** 的静态方法 **GetDomain** 向 CLR 询问它正在哪个 AppDomain 中执行。线程还可查询 **System.AppDomain** 的静态只读属性 **CurrentDomain** 获得相同的信息。

AppDomain 创建后可被赋予一个**友好名称**。它是用于标识 AppDomain 的一个 **String**。友好名称主要是为了方便调试。由于 CLR 要在我们的任何代码执行前创建默认 AppDomain，所以使用可执行文件的文件名作为默认的 AppDomain 友好名称。**Marshalling** 方法使用 **System.AppDomain** 的只读 **FriendlyName** 属性来查询默认 AppDomain 的友好名称。

接着，**Marshalling** 方法查询默认 AppDomain 中加载的程序集的强命名标识，这个程序集定义了入口方法 **Main**(其中调用了 **Marshalling**)。程序集定义了几个类型：**Program**，**MarshalByRefType**，**MarshalByValType** 和 **NonMarshalableType**。现在已准备好研究上

面的三个演示(Demo)，它们本质上很相似。

演示 1：使用“按引用封送”进行跨 AppDomain 通信

演示 1 调用 **System.AppDomain** 的静态 **CreateDomain** 方法指示 CLR 在同一个 Windows 进程中创建一个新 AppDomain。**AppDomain** 类型提供了 **CreateDomain** 方法的多个重载版本。建议仔细研究一下它们，并在新建 AppDomain 时选择最合适的一个。本例使用的 **CreateDomain** 接受以下三个参数。

- 代表新 AppDomain 的友好名称的一个 **String**。本例传递的是"AD #2"。
- 一个 **System.Security.Policy.Evidence**，是 CLR 用于计算 AppDomain 权限集的证据。本例为该参数传递 **null**，造成新 AppDomain 从创建它的 AppDomain 继承权限集。通常，如果希望围绕 AppDomain 中的代码创建安全边界，可构造一个 **System.Security.PermissionSet** 对象，在其中添加希望的权限对象(实现了 **IPermission** 接口的类型的实例)，将得到的 **PermissionSet** 对象引用传给接受一个 **PermissionSet** 的 **CreateDomain** 方法重载。
- 一个 **System.AppDomainSetup**，代表 CLR 为新 AppDomain 使用的配置设置。同样，本例为该参数传递 **null**，使新 AppDomain 从创建它的 AppDomain 继承配置设置。如果希望对新 AppDomain 进行特殊配置，可构造一个 **AppDomainSetup** 对象，将它的各种属性(例如配置文件的名称)设为你希望的值，然后将得到的 **AppDomainSetup** 对象引用传给 **CreateDomain** 方法。

CreateDomain 方法内部会在进程中新建一个 AppDomain，该 AppDomain 将被赋予指定的友好名称、安全性和配置设置。新 AppDomain 有自己的 Loader 堆，这个堆目前是空的，因为还没有程序集加载到新 AppDomain 中。创建 AppDomain 时，CLR 不在这个 AppDomain 中创建任何线程；AppDomain 中也不会运行代码，除非显式地让一个线程调用 AppDomain 中的代码。

现在，要在新 AppDomain 中创建类型的实例，首先要将程序集加载到新 AppDomain 中，然后构造程序集中定义的类型的实例。这就是 AppDomain 的公共实例方法 **CreateInstanceAndUnwrap** 所做的事情。调用这个方法时我传递了两个 **String** 实参：第一个标识了想在新 AppDomain(**ad2** 变量引用的那个 AppDomain)中加载的程序集；第二个标识了想构建其实例的那个类型的名称(**"MarshalByRefType"**)。在内部，**CreateInstanceAndUnwrap** 方法导致调用线程从当前 AppDomain 切换新 AppDomain。现在，线程(当前正在调用 **CreateInstanceAndUnwrap**)将指定程序集加载到新 AppDomain 中，并扫描程序集的类型定义元数据表，查找指定类型("MarshalByRefType")。找到类型后，线程调用 **MarshalByRefType** 的无参构造器。现在，线程又切换回默认 AppDomain，使 **CreateInstanceAndUnwrap** 能返回对新 **MarshalByRefType** 对象的引用。



> **注意**　**CreateInstanceAndUnwrap** 方法的一些重载版本允许在调用类型的构造器时传递实参。

所有这一切听起来都很美好，但还有一个问题：CLR 不允许一个 AppDomain 中的变量(根)引用另一个 AppDomain 中创建的对象。如果 **CreateInstanceAndUnwrap** 直接返回对象引用，隔离性就会被打破，而隔离是 AppDomain 的全部目的！因此，**CreateInstanceAndUnwrap** 在返回对象引用前要执行一些额外的逻辑。

我们的 **MarshalByRefType** 类型从一个很特别的基类 **System.MarshalByRefObject** 派生。当 **CreateInstanceAndUnwrap** 发现它封送的一个对象的类型派生自 **MarshalByRefObject** 时，CLR 就会跨 AppDomain 边界按引用封送对象。下面讲述了按引用将对象从一个 AppDomain(源 AppDomain，这是真正创建对象的地方)封送到另一个 AppDomain(目标 AppDomain，这是调用 **CreateInstanceAndUnwrap** 的地方)的具体含义。

源 AppDomain 想向目标 AppDomain 发送或返回对象引用时，CLR 会在目标 AppDomain 的 Loader 堆中定义一个代理类型。代理类型是用原始类型的元数据定义的。所以，它看起来和原始类型完全一样：有完全一样的实例成员(属性、事件和方法)。但是，实例字段不会成为(代理)类型的一部分，我稍后会具体解释这一点。代理类型确实定义了几个(自己的)实例字段，但这些字段和原始类型的不一致。相反，这些字段只是指出哪个 AppDomain“拥有”真实的对象，以及如何在拥有(对象的)AppDomain 中找到真实的对象。(在内部，代理对象用一个 **GCHandle** 实例引用真实的对象。第 21 章“托管堆和垃圾回收”讨论了 **GCHandle** 类型。)

在目标 AppDomain 中定义好这个代理类型之后，**CreateInstanceAndUnwrap** 方法就会创建代理类型的实例，初始化它的字段来标识源 AppDomain 和真实对象，然后将对这个代理对象的引用返回给目标 AppDomain。在 Ch22-1-AppDomains 应用程序中，**mbrt** 变量被设为引用这个代理。注意，从 **CreateInstanceAndUnwrap** 方法返回的对象实际不是 **MarshalByRefType** 类型的实例。CLR 一般不允许将一个类型的对象转换成不兼容的类型。但在当前这种情况下，CLR 允许转型，因为新类型具有和原始类型一样的实例成员。事实上，用代理对象调用 **GetType**，它会向你撒谎，说自己是一个 **MarshalByRefType** 对象。

但可以证明从 **CreateInstanceAndUnwrap** 返回的对象实际是对代理对象的引用。为此，Ch22-1-AppDomains 应用程序调用了 **System.Runtime.Remoting.RemotingService** 的公共静态 **IsTransparentProxy** 方法并向其传递 **CreateInstanceAndUnwrap** 方法返回的引用。从输出结果可知，**IsTransparentProxy** 方法返回 **true**，证明返回的是代理。

接着，Ch22-1-AppDomains 应用程序使用代理调用 **SomeMethod** 方法。由于 **mbrt** 变量引用代理对象，所以会调用由代理实现的 **SomeMethod**。代理的实现利用代理对象中的信息字段，将调用线程从默认 AppDomain 切换至新 AppDomain。现在，该线程的任何行动都在新 AppDomain 的安全策略和配置设置下运行。线程接着使用代理对象的 **GCHandle** 字段查找新 AppDomain 中的真实对象，并用真实对象调用真实的 **SomeMethod** 方法。

有两个办法可证明调用线程已从默认 AppDomain 切换至新 AppDomain。首先，我在 **SomeMethod** 方法中调用了 **Thread.GetDomain().FriendlyName**。这将返回"AD #2"(参见输出)，这是由于线程当前在新 AppDomain 中运行，而这个新 AppDomain 是通过调用 **AppDomain.CreateDomain** 方法，并传递"AD #2"作为友好名称参数来创建的。其次，在调试器中逐语句调试代码，并打开了“调用堆栈”窗口，那么“[外部代码]”行会标注一个线程在什么位置跨越 AppDomain 边界。请参见图 22-2 底部的“调用堆栈”窗口。

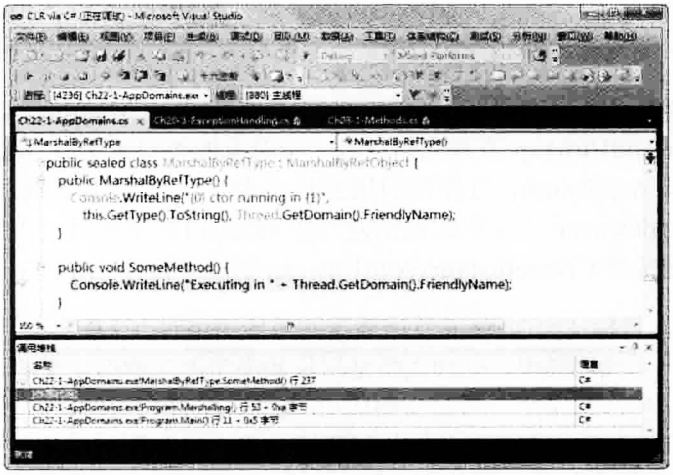


图 22-2　调试器的“调用堆栈”窗口显示了一次 AppDomain 切换

真实的 **SomeMethod** 方法返回后，会返回至代理的 **SomeMethod** 方法，后者将线程切换回默认 AppDomain。线程继续执行默认 AppDomain 中的代码。



注意　一个 AppDomain 中的线程调用另一个 AppDomain 中的方法时，线程会在这两个 AppDomain 之间切换。这意味着跨 AppDomain 边界的方法调用是同步执行的[①]。任何时刻一个线程只能在一个 AppDomain 中，而且要用那个 AppDomain 的安全和配置设置来执行代码。如果希望多个 AppDomain 中的代码并发执行，应创建额外的线程，让这些线程在你希望的 AppDomain 中执行你希望的代码。

Ch22-1-AppDomains 应用程序接着做的事情是调用 **AppDomain** 类的公共静态 **Unload** 方法，这会强制 CLR 卸载指定的 AppDomain(包括加载到其中的所有程序集)，并强制执行一次垃圾回收，以释放由卸载的 AppDomain 中的代码创建的所有对象。这时，默认 AppDomain 的 **mbrt** 变量仍然引用一个有效的代理对象。但代理对象已不再引用一个有效的 AppDomain(因为它已经卸载了)。

当默认 AppDomain 试图使用代理对象调用 **SomeMethod** 方法时，调用的是该方法在代理中

① 一个 AppDomain 的方法执行完毕，才能执行另一个 AppDomain 的方法。不能多个 AppDomain 的代码并发执行。——译注

的实现。代理的实现发现包含真实对象的 **AppDomain** 已卸载。所以，代理的 **SomeMethod** 方法抛出一个 **AppDomainUnloadedException** 异常，告诉调用者操作无法完成。

显然，Microsoft 的 CLR 团队不得不做大量的工作来确保 AppDomain 的正确隔离，但这是他们必须做的。跨 AppDomain 访问对象的功能正在被大量地使用，开发人员对这个功能的依赖性正在日益增加。不过，使用“按引用封送”的语义进行跨 AppDomain 边界的对象访问，会产生一些性能上的开销。所以，一般应尽量少用这个功能。

前面我承诺会更多地讨论实例字段。从 **MarshalByRefObject** 派生的类型可定义实例字段。但这些实例字段不会成为代理类型的一部分，也不会包含在代理对象中。写代码对派生自 **MarshalByRefObject** 的类型的实例字段进行读写时，JIT 编译器会自动生成代码，分别调用 **System.Object** 的 **FieldGetter** 方法(用于读)或 **FieldSetter** 方法(用于写)来使用代理对象(以找到真正的 AppDomain/对象)。这些方法是私有的，而且没有在文档中记录。简单地说，这些方法利用反射机制获取或设置字段值。因此，虽然能访问派生自 **MarshalByRefObject** 的一个类型中的字段，但性能很差，因为 CLR 最终要调用方法来执行字段访问。事实上，即使要访问的字段在你自己的 AppDomain 中，性能也好不到哪里去。[①]

访问实例字段时的性能问题

我用以下代码演示性能损失的程度：

```
private sealed class NonMBRO   : Object            { public Int32 x; }
private sealed class MBRO      : MarshalByRefObject  { public Int32 x; }

private static void FieldAccessTiming(){
   const Int32 count = 100000000;
   NonMBRO nonMbro = new NonMBRO();
   MBRO mbro = new MBRO();

   Stopwatch sw = Stopwatch.StartNew();
   for (Int32 c = 0; c < count; c++) nonMbro.x++;
   Console.WriteLine("{0}", sw.Elapsed); // 00:00:00.4073560

   sw = Stopwatch.StartNew();
   for (Int32 c = 0; c < count; c++) mbro.x++;
   Console.WriteLine("{0}", sw.Elapsed); // 00:00:02.5388665
}
```

我运行以上代码，访问从 **Object** 派生的 **NonMBRO** 类的实例字段只花了约 0.4 秒，但访问从 **MarshalByRefObject** 派生的 **MBRO** 类的实例字段却花了 2.54 秒。也就是说，访问从 **MarshalByRefObject** 派生的一个类的实例字段要多花约 6 倍的时间！

从好不好用(usability)的角度说，派生自 **MarshalByRefObject** 的类型真的应该避免定义任

① 如果 CLR 要求所有字段都必须私有(为了获得好的数据封装，我强烈建议这样做)，那么 **FieldGetter** 和 **FieldSetter** 方法根本没有存在的必要，从方法中总是能够直接访问字段，避免(因为还要经由中间的 getter 或 setter 方法)造成性能损失。

何静态成员。这是因为静态成员总是在调用 AppDomain 的上下文中访问。要切换到哪个 AppDomain 的信息包含在代理对象中，但调用静态成员时没有代理对象，所以不会发生 AppDomain 的切换。让类型的静态成员在一个 AppDomain 中执行，实例成员却在另一个 AppDomain 中执行，这样的编程模型未免太“丑”了！

由于第二个 AppDomain 中没有根，所以代理引用的原始对象可以被垃圾回收。这当然不理想。但另一方面，假如将原始对象不确定地(indefinitely)留在内存中，代理可能不再引用它，而原始对象依然存活；这同样不理想。CLR 解决这个问题的办法是使用一个“租约管理器”(lease manager)。一个对象的代理创建好之后，CLR 保持对象存活 5 分钟。5 分钟内没有通过代理发出调用，对象就会失效，下次垃圾回收会释放它的内存。每发出一次对对象的调用，“租约管理器”都会续订对象的租期，保证它在接下来的 2 分钟内在内存中保持存活。在对象过期后试图通过代理调用它，CLR 会抛出 **System.Runtime.Remoting.RemotingException** 异常。

默认的 5 分钟和 2 分钟租期设定是可以修改的，重写 **MarshalByRefObject** 的虚方法 **InitializeLifetimeService** 即可。详情参见文档的“生存期租约”主题。

演示 2：使用“按值封送”进行跨 AppDomain 通信

演示 2 与演示 1 很相似。和演示 1 一样，演示 2 也创建了新 AppDomain。然后调用 **CreateInstanceAndUnwrap** 方法将同一个程序集加载到新建 AppDomain 中，并在这个新 AppDomain 中创建 **MarshalByRefType** 类型的实例。CLR 为这个对象创建代理，**mbrt** 变量(在默认 AppDomain 中)被初始化成引用这个代理。接着用代理调用 **MethodWithReturn** 方法。这个方法是无参的，将在新 AppDomain 中执行以创建 **MarshalByValType** 类型的实例，并将一个对象引用返回给默认 AppDomain。

MarshalByValType 不从 **System.MarshalByRefObject** 派生，所以 CLR 不能定义一个代理类型并创建代理类型的实例；对象不能按引用跨 AppDomain 边界进行封送。

但由于 **MarshalByValType** 标记了自定义特性**[Serializable]**，所以 **MethodWithReturn** 方法能按值封送对象。下面具体描述了将一个对象按值从一个 AppDomain(源 AppDomain)封送到另一个 AppDomain(目标 AppDomain)的含义。欲知 CLR 序列化和反序列化的详情，请参见第 24 章“运行时序列化”。

源 AppDomain 想向目标 AppDomain 发送或返回一个对象引用时，CLR 将对象的实例字段序列化成一个字节数组。字节数组从源 AppDomain 复制到目标 AppDomain。然后，CLR 在目标 AppDomain 中反序列化字节数组，这会强制 CLR 将定义了“被反序列化的类型”的程序集加载到目标 AppDomain 中(如果尚未加载的话)。接着，CLR 创建类型的实例，并利用字节数组中的值初始化对象的字段，使之与源对象中的值相同。换言之，CLR 在目标 AppDomain 中精确复制了源对象。然后，**MethodWithReturn** 方法返回对这个副本的引用；这样一来，对象就跨 AppDomain 的边界按值封送了。



重要提示　加载程序集时，CLR 使用目标 AppDomain 的策略和配置设置(而 AppDomain 可能设置了不同的 AppBase 目录或者不同的版本绑定重定向)。策略上的差异可能妨碍 CLR 定位程序集。程序集无法加载时会抛出异常，目标 AppDomain 接收不到对象引用。

至此，源 AppDomain 中的对象和目标 AppDomain 中的对象就有了独立的生存期，它们的状态也可以独立地更改。如果源 AppDomain 中没有根保持源对象存活(就像我的 Ch22-1-AppDomains 应用程序所做的那样)，源对象的内存就会在下次垃圾回收时被回收。

为了证明 **MethodWithReturn** 方法返回的不是对代理对象的引用，Ch22-1-AppDomains 应用程序调用了 **System.Runtime.Remoting.RemotingService** 的公共静态 **IsTransparentProxy** 方法，将 **MethodWithReturn** 方法返回的引用作为参数传给它。如输出所示，**IsTransparentProxy** 方法返回 **false**，表明对象是一个真实的对象，不是代理。

现在，程序使用真实的对象调用 **ToString** 方法。由于 **mbvt** 变量引用真实的对象，所以会调用这个方法的真实实现，线程不会在 AppDomain 之间切换。为了证明这一点，可查看调试器的“调用堆栈”窗口，它没有显示一个“[外部代码]”行。

为进一步证实没有涉及代理，Ch22-1-AppDomains 应用程序卸载了 AppDomain，然后尝试再次调用 **ToString**。和演示 1 不一样，这次调用会成功，因为卸载新 AppDomain 对默认 AppDomain“拥有”的对象(其中包括按值封送的对象)没有影响。

演示 3：使用不可封送的类型跨 AppDomain 通信

演示 3 的开始部分与演示 1 和 2 相似，都是新建 AppDomain，调用 **CreateInstanceAndUnwrap** 方法将相同的程序集加载到新 AppDomain 中，在新 AppDomain 中创建一个 **MarshalByRefType** 对象，并让 **mbrt** 引用该对象的代理。

然后，我用代理调用接受一个实参的 **MethodArgAndReturn** 方法。同样地，CLR 必须保持 AppDomain 的隔离，所以不能直接将对实参的引用传给新 AppDomain。如果对象的类型派生自 **MarshalByRefObject**，CLR 会为它创建代理并按引用封送。如果对象的类型用 **[Serializable]**进行了标记，CLR 会将对象(及其子)序列化成一个字节数组，将字节数组封送到新 AppDomain 中，再将字节数组反序列化成对象图，将对象图的根传给 **MethodArgAndReturn** 方法。

在这个特定的例子中，我跨越 AppDomain 边界传递一个 **System.String** 对象。**System.String** 类型不是从 **MarshalByRefObject** 派生的，所以 CLR 不能创建代理。幸好，**System.String** 被标记为**[Serializable]**，所以 CLR 能按值封送它，使代码能正常工作。注意，对于 **String** 对象，CLR 会采取特殊的优化措施。跨越 AppDomain 边界封送一个 **String** 对象时，CLR 只是跨越边界传递对 **String** 对象的引用；不会真的生成 **String** 对象的副本。之所以能提供这个优化，是因为 **String** 对象是不可变的；所以，一个 AppDomain 中的代码不可能破坏

String 对象的字段。[1]**String**“不可变”的详情请参见第 14 章“字符、字符串和文本处理”。

在 **MethodArgAndReturn** 内部，我显示传给它的字符串，证明字符串跨越了 AppDomain 边界。然后，我创建 **NonMarshalableType** 类型的实例，并将对这个对象的引用返回至默认 AppDomain。由于 **NonMarshalableType** 不是从 **System.MarshalByRefObject** 派生的，也没有用**[Serializable]**定制特性进行标记，所以不允许 **MethodArgAndReturn** 按引用或按值封送对象——对象完全不能跨越 AppDomain 边界进行封送！为了报告这个问题，**MethodArgAndReturn** 在默认 AppDomain 中抛出一个 **SerializationException** 异常。由于我的程序没有捕捉这个异常，所以程序终止。

22.3 卸载 AppDomain

AppDomain 很强大的一个地方就是可以卸载它。卸载 AppDomain 会导致 CLR 卸载 AppDomain 中的所有程序集，还会释放 AppDomain 的 Loader 堆。卸载 AppDomain 的办法是调用 **AppDomain** 的静态 **Unload** 方法(参见 Ch22-1-AppDomains 应用程序)。这导致 CLR 执行一系列操作来得体地卸载指定的 AppDomain。

1. CLR 挂起进程中执行过托管代码的所有线程。

2. CLR 检查所有线程栈，查看哪些线程正在执行要卸载的 AppDomain 中的代码，或者哪些线程会在某个时候返回至要卸载的 AppDomain。任何栈上有要卸载的 AppDomain，CLR 都会强迫对应的线程抛出一个 **ThreadAbortException**(同时恢复线程的执行)。这将导致线程展开(unwind)，并执行遇到的所有 **finally** 块以清理资源。如果没有代码捕捉 **ThreadAbortException**，它最终会成为未处理的异常，CLR 会“吞噬”这个异常；线程会终止，但进程可继续运行。这是很特别的一点，因为对于其他所有未经处理的异常，CLR 都会终止进程。



重要提示 如果线程当前正在 **finally** 块、**catch** 块、类构造器、临界执行区域[2]或非托管代码中执行，那么 CLR 不会立即终止该线程。否则，资源清理代码、错误恢复代码、类型初始化代码、关键(critical)代码或者其他任何 CLR 不了解的代码都将无法完成，导致应用程序的行为无法预测，甚至可能造成安全漏洞。线程终止时会等待这些代码块执行完毕。然后，当代码块结束时，CLR 再强制线程抛出一个 **ThreadAbortException** 异常。

① 顺便说一句，这正是为什么 **System.String** 类是密封类的原因。类不密封，就能定义从 **String** 派生的类，并添加自己的字段。如果这样做了，CLR 就没有办法确保你的“字符串”类是不可变的。

② 即 critical execution region，文档翻译为“关键执行区域”，本书翻译为“临界执行区域”或“临界区”。临界区是指线程终止或未处理异常的影响可能不限于当前任务的区域。相反，非临界区中的终止或失败只对出错的任务有影响。——译注

3. 当第 2 步发现的所有线程都离开 AppDomain 后，CLR 遍历堆，为引用了“由已卸载的 AppDomain 创建的对象”的每个代理对象都设置一个标志(flag)。这些代理对象现在知道它们引用的真实对象已经不在了。现在，任何代码在无效的代理对象上调用方法都会抛出一个 **AppDomainUnloadedException** 异常。

4. CLR 强制垃圾回收，回收由已卸载的 AppDomain 创建的任何对象的内存。这些对象的 **Finalize** 方法被调用，使对象有机会正确清理它们占用的资源。

5. CLR 恢复剩余所有线程的执行。调用 **AppDomain.Unload** 方法的线程将继续运行；对 **AppDomain.Unload** 的调用是同步进行的。[①]

在 Ch22-1-AppDomains 应用程序中，所有工作都用一个线程来做。因此，任何时候只要调用 **AppDomain.Unload**，都不可能有另一个线程在要卸载的 AppDomain 中。因此，CLR 不必抛出任何 **ThreadAbortException** 异常。本章后面将进一步讨论 **ThreadAbortException**。

顺便说一句，当一个线程调用 **AppDomain.Unload** 方法时，针对要卸载的 AppDomain 中的线程，CLR 会给它们 10 秒钟的时间离开。10 秒钟后，如果调用 **AppDomain.Unload** 方法的线程还没有返回，CLR 将抛出一个 **CannotUnloadAppDomainException** 异常，AppDomain 将来可能会、也可能不会卸载。



注意　如果调用 **AppDomain.Unload** 方法的线程不巧在要卸载的 AppDomain 中，CLR 会创建另一个线程来尝试卸载 AppDomain。第一个线程被强制抛出 **ThreadAbortException** 并展开(unwind)。新线程将等待 AppDomain 卸载，然后新线程会终止。如果 AppDomain 卸载失败，新线程将抛出 **CannotUnloadAppDomainException** 异常。但是，由于我们没有写由新线程执行的代码，所以无法捕捉这个异常。

22.4　监视 AppDomain

宿主应用程序可监视 AppDomain 消耗的资源。有的宿主根据这种信息判断一个 AppDomain 的内存或 CPU 消耗是否超过了应有的水准，并强制卸载一个 AppDomain。还可利用监视来比较不同算法的资源消耗情况，判断哪一种算法用的资源较少。由于 AppDomain 监视本身也会产生开销，所以宿主必须将 **AppDomain** 的静态 **MonitoringEnabled** 属性设为 **true**，从而显式地打开监视。监视一旦打开便不能关闭；将 **MonitoringEnabled** 属性设为 **false** 会抛出一个 **ArgumentException** 异常。

监视打开后，代码可查询 **AppDomain** 类提供的以下 4 个只读属性。

- **MonitoringSurvivedProcessMemorySize**　这个 **Int64** 静态属性返回由当前 CLR 实例控制的所有 AppDomain 使用的字节数。这个数字只保证在上一次垃圾回收时是准确的。

① 换言之，一旦调用 Unload，只有在它返回之后，线程才能恢复运行。——译注

- **MonitoringTotalAllocatedMemorySize** 这个 **Int64** 实例属性返回特定 AppDomain 已分配的字节数。这个数字只保证在上一次垃圾回收时是准确的。
- **MonitoringSurvivedMemorySize** 这个 **Int64** 实例属性返回特定 AppDomain 当前正在使用的字节数。这个数字只保证在上一次垃圾回收时是准确的。
- **MonitoringTotalProcessorTime** 这个 **TimeSpan** 实例属性返回特定 AppDomain 的 CPU 占用率。

下面这个类演示了如何利用这些属性检查两个时间点之间一个 AppDomain 发生的变化：

```
private sealed class AppDomainMonitorDelta : IDisposable {
   private AppDomain m_appDomain;
   private TimeSpan m_thisADCpu;
   private Int64 m_thisADMemoryInUse;
   private Int64 m_thisADMemoryAllocated;

   static AppDomainMonitorDelta() {
      // 确定已打开了 AppDomain 监视
      AppDomain.MonitoringIsEnabled = true;
   }

   public AppDomainMonitorDelta(AppDomain ad) {
      m_appDomain = ad ?? AppDomain.CurrentDomain;
      m_thisADCpu = m_appDomain.MonitoringTotalProcessorTime;
      m_thisADMemoryInUse = m_appDomain.MonitoringSurvivedMemorySize;
      m_thisADMemoryAllocated = m_appDomain.MonitoringTotalAllocatedMemorySize;
   }

   public void Dispose() {
      GC.Collect();
      Console.WriteLine("FriendlyName={0}, CPU={1}ms", m_appDomain.FriendlyName,
         (m_appDomain.MonitoringTotalProcessorTime - m_thisADCpu).TotalMilliseconds);
      Console.WriteLine(" Allocated {0:N0} bytes of which {1:N0} survived GCs",
         m_appDomain.MonitoringTotalAllocatedMemorySize - m_thisADMemoryAllocated,
         m_appDomain.MonitoringSurvivedMemorySize - m_thisADMemoryInUse);
   }
}
```

以下代码演示了如何使用 **AppDomainMonitorDelta** 类：

```
private static void AppDomainResourceMonitoring() {
   using (new AppDomainMonitorDelta(null)) {
      // 分配在回收时能存活的约 10MB
      var list = new List<Object>();
      for (Int32 x = 0; x < 1000; x++) list.Add(new Byte[10000]);

      // 分配在回收时存活不了的约 20MB
      for (Int32 x = 0; x < 2000; x++) new Byte[10000].GetType();

      // 保持 CPU 工作约 5 秒
      Int64 stop = Environment.TickCount + 5000;
      while (Environment.TickCount < stop) ;
   }
}
```

在我的机器上运行以上代码，得到以下输入：[①]

```
FriendlyName=03-Ch22-1-AppDomains.exe, CPU=5031.25ms
   Allocated 30,159,496 bytes of which 10,085,080 survived GCs
```

22.5　AppDomain FirstChance 异常通知

每个 AppDomain 都可关联一组回调方法；CLR 开始查找 AppDomain 中的 **catch** 块时，这些回调方法将得以调用。可用这些方法执行日志记录操作。另外，宿主可利用这个机制监视 AppDomain 中抛出的异常。回调方法不能处理异常，也不能以任何方式“吞噬”异常(装作异常没有发生)；它们只是接收关于异常发生的通知。要登记回调方法，为 AppDomain 的实例事件 **FirstChanceException** 添加一个委托就可以了。

下面描述了 CLR 如何处理异常：异常首次抛出时，CLR 调用向抛出异常的 AppDomain 登记的所有 **FirstChanceException** 回调方法。然后，CLR 查找栈上在同一个 AppDomain 中的任何 **catch** 块。有一个 **catch** 块能处理异常，则异常处理完成，将继续正常执行。如果 AppDomain 中没有一个 **catch** 块能处理异常，则 CLR 沿着栈向上来到调用 AppDomain，再次抛出同一个异常对象(序列化和反序列化之后)。这时感觉就像是抛出了一个全新的异常，CLR 调用向当前 AppDomain 登记的所有 **FirstChanceException** 回调方法。这个过程会一直持续，直到抵达线程栈顶部。届时如果异常还未被任何代码处理，CLR 只好终止整个进程。

22.6　宿主如何使用 AppDomain

前面已讨论了宿主以及宿主加载 CLR 的方式。同时还讨论了宿主如何告诉 CLR 创建和卸载 AppDomain。为了使这些讨论更加具体，下面将描述一些常见的寄宿和 AppDomain 使用情形。我着重解释了不同应用程序类型如何寄宿(容纳)CLR，以及如何管理 AppDomain。

22.6.1　可执行应用程序

控制台 UI 应用程序、NT Service 应用程序、Windows 窗体应用程序和 Windows Presentation Foundation(WPF)应用程序都是自寄宿(self-hosted，即自己容纳 CLR)的应用程序，它们都有托管 EXE 文件。Windows 用托管 EXE 文件初始化进程时，会加载垫片[②]。垫片检查应用程序的程序集(EXE 文件)中的 CLR 头信息。头信息指明了生成和测试应用程序时使用的 CLR 版本。垫片根据这些信息决定将哪个版本的 CLR 加载到进程中，CLR 加载并初始化好之后，会再次检查程序集的 CLR 头，判断哪个方法是应用程序的入口方(**Main**)。CLR 调用该方法，

---

① 在本书配套代码中，找到C22-1 AppDomains项目，在**Main()**方法中注释掉**Marshalling()**和**FieldAccessTiming()**方法调用，然后解除对 **AppDomainResourceMonitoring()**调用 R 的注释。——译注

② 垫片的问题请参见 22.1 节“CLR 寄宿”。——译注

此时应用程序才真正启动并运行起来。

代码运行时会访问其他类型。引用另一个程序集中的类型时，CLR 会定位所需的程序集，并将其加载到同一个 AppDomain 中。应用程序的 **Main** 方法返回后，Windows 进程终止(销毁默认 AppDomain 和其他所有 AppDomain)。



> **注意** 顺便说一句，要关闭 Windows 进程(包括它的所有 AppDomain)，可调用 **System.Environment** 的静态方法 **Exit**。**Exit** 是终止进程最得体的方式，因为它首先调用托管堆上的所有对象的 **Finalize** 方法，再释放 CLR 容纳的所有非托管 COM 对象。最后，**Exit** 调用 Win32 **ExitProcess** 函数。

应用程序可告诉 CLR 在进程的地址空间中创建额外的 AppDomain。事实上，我的 Ch22-1-AppDomains 应用程序正是这么做的。

22.6.2 Microsoft Silverlight 富 Internet 应用程序

Microsoft Silverlight“运行时”使用了有别于.NET Framework 普通桌面版本的特殊 CLR。安装好 Silverlight“运行时”之后，每次访问使用了 Silverlight 技术的网站，都会造成 Silverlight CLR(CoreClr.dll)加载到浏览器(可能是、也可能不是 Windows Internet Explorer——甚至不一定是 Windows 机器)中。网页上的每个 Silverlight 控件都在它自己的 AppDomain 中运行。用户关闭标签页或切换至另一个网站，不再使用的任何 Silverlight 控件的 AppDomain 都会被卸载。AppDomain 中的 Silverlight 代码在限制了安全权限的沙盒中运行，不会对用户或机器造成任何损害。

22.6.3 Microsoft ASP.NET 和 XML Web 服务应用程序

ASP.NET 作为一个 ISAPI DLL(ASPNet_ISAPI.dll)实现。客户端首次请求由这个 DLL 处理的 URL 时，ASP.NET 会加载 CLR。客户端请求一个 Web 应用程序时，ASP.NET 判断这是不是第一次请求。如果是，ASP.NET 要求 CLR 为该 Web 应用程序创建新 AppDomain；每个 Web 应用程序都根据虚拟根目录来标识。然后，ASP.NET 要求 CLR 将包含应用程序所公开类型的程序集加载到新 AppDomain 中，创建该型的实例，并调用其中的方法响应客户端的 Web 请求。如果代码引用了更多的类型，CLR 将所需的程序集加载到 Web 应用程序的 AppDomain 中。

以后，如果客户端请求已开始运行的 Web 应用程序，ASP.NET 就不再新建 AppDomain 了，而是使用现有的 AppDomain，创建 Web 应用程序的类型的新实例并开始调用方法。这些方法已 JIT 编译成本机代码，所以后续客户端请求的性能会比较出众。

如果客户端请求不同的 Web 应用程序，ASP.NET 会告诉 CLR 创建新 AppDomain。新 AppDomain 通常在和其他 AppDomain 一样的工作进程中创建。这意味着将有大量 Web 应用程序在同一个 Windows 进程中运行，这提升了系统的总体效率。同样地，每个 Web 应

用程序需要的程序集都会加载到一个单独的 AppDomain 中，以隔离不同 Web 应用程序的代码和对象。

ASP.NET 的一个亮点是允许在不关闭 Web 服务器的前提下动态更改网站代码。网站的文件在硬盘上发生改动时，ASP.NET 会检测到这个情况，并卸载包含旧版本文件的 AppDomain(在当前运行的最后一个请求完成之后)，并创建一个新 AppDomain，向其中加载新版本的文件。为了确保这个过程的顺利进行，ASP.NET 使用了 AppDomain 的一个名为“影像复制[①]”(shadow copying)的功能。

22.6.4　Microsoft SQL Server

Microsoft SQL Server 是非托管应用程序，它的大部分代码仍是用非托管 C++写的。SQL Server 允许开发人员使用托管代码创建存储过程。首次请求数据库运行一个用托管代码写的存储过程时，SQL Server 会加载 CLR。存储过程在它们自己的安全 AppDomain 中运行，这避免了存储过程对数据库服务器产生负面影响。

这其实是一项非同寻常的功能！它意味着开发人员可以选择自己喜欢的编程语言来编写存储过程。存储过程可以在自己的代码中使用强类型的数据对象。代码还会被 JIT 编译成本机代码执行，而不是采用解释执行的方式。开发人员可利用 FCL 或任何其他程序集中定义的任何类型。结果是我们的工作变得越来越轻松，但应用程序的表现变得越来越好。作为开发人员，我表示很知足！

22.6.5　更多的用法只局限于想象力

生产型应用程序(比如字处理和电子表格软件)也允许用户使用任何编程语言来编写宏。宏可以访问与 CLR 一起运行的所有程序集和类型。宏会被编译，所以执行得更快。最重要的是，宏在一个安全 AppDomain 中运行，不会骚扰到用户。你自己的应用程序也可利用这个功能。具体怎么用，只局限于你自己的想象力。

22.7　高级宿主控制

本节探讨与寄宿 CLR 有关的高级话题。目的是让你有一个初步的认识，帮你更多地理解 CLR 的能力。如果有兴趣，鼓励你参考其他更有针对性的参考书。

22.7.1　使用托管代码管理 CLR

System.AppDomainManager 类允许宿主使用托管代码(而不是非托管代码)覆盖 CLR 的默认行为。当然，使用托管代码使宿主的实现变得更容易。你唯一要做的就是定义自己的类，让它从 **System.AppDomainManager** 派生，重写想接手控制的任何虚方法。然后，在专用

① 文档如此。参见 **AppDomainSetup.ShadowCopyDirectories** 属性。——译注

的程序集中生成类，并将程序集安装到 GAC 中。这是由于该程序集需要被授予完全信任权限，而 GAC 中的所有程序集都总是被授予完全信任权限。

然后要告诉 CLR 使用你的 **AppDomainManager** 派生类。在代码中，最佳的办法就是创建一个 **AppDomainSetup** 对象，初始化它的 **AppDomainManagerAssembly** 和 **AppDomainManagerType** 属性(均为 **String** 类型)。将前者设为定义了你的 **AppDomainManager** 派生类的那个程序集的强名称字符串，后者设为你的 **AppDomainManager** 派生类的全名。还可在应用程序 **XML** 配置文件中，用 **appDomainManagerAssembly** 和 **appDomainManagerType** 元素来设置 **AppDomainManager**。此外，本机宿主可查询 **ICLRControl** 接口，调用该接口的 **SetAppDomainManagerType** 函数，向它传递 **GAC** 安装的程序集的标识和 **AppDomainManager** 派生类的名称。[①]

现在看看 **AppDomainManager** 派生类能做什么。**AppDomainManager** 派生类的作用是使宿主保持控制权，即使是在加载项(add-in)试图创建自己的 AppDomain 时。进程中的代码试图创建新 AppDomain 时，那个 AppDomain 中的 **AppDomainManager** 派生对象可修改安全性和配置设置。它还可决定阻止一次 AppDomain 创建，或返回对现有 AppDomain 的引用。新 AppDomain 创建好之后，CLR 会在其中创建新的 **AppDomainManager** 派生对象。这个对象也能修改配置设置、决定执行上下文如何在线程之间切换，并决定向程序集授予的权限。

22.7.2 写健壮的宿主应用程序

托管代码出现错误时，宿主可告诉 CLR 采取什么行动。下面是一些例子(按严重性从低到高排序)。

- 如果线程执行时间过长，CLR 可终止线程并返回一个响应。(下一节会更多地讨论这个问题。)
- CLR 可卸载 AppDomain。这会终止该 AppDomain 中的所有线程，导致有问题的代码卸载。
- CLR 可被禁用。这会阻止更多的托管代码在程序中运行，但仍然允许非托管代码运行。
- CLR 可退出 Windows 进程。首先会终止所有线程，并卸载所有 AppDomain，使资源清理操作得以执行，然后才会终止进程。

CLR 可以得体地(gracefully)或者粗鲁地(rudely)终止线程或 AppDomain。得体意味着会执行(资源)清理代码。换言之，**finally** 块中的代码会运行，对象的 **Finalize** 方法也将被执行。而

① 还可使用环境变量和注册表设置来配置一个 **AppDomainManager**。但这些机制比正文描述的方法麻烦得多。除非是在一些测试性的场合中，否则应避免使用。

粗鲁意味着清理代码不会执行。换言之，**finally** 块中的代码可能不会运行，对象的 **Finalize** 方法也可能不会执行。如果得体地终止，当前正在一个 **catch** 块或 **finally** 块中的线程无法终止。但如果粗鲁地终止，就能终止 **catch** 块或 **finally** 块中的线程。遗憾的是，非托管代码或者约束执行区(Constrained Execution Region，CER)中的线程完全无法终止。

宿主可设置所谓的升级策略(escalation policy)，从而告诉 CLR 应该如何处理托管代码的错误。例如，SQL Server 就告诉 CLR 在执行托管代码时，假如遇到未处理的异常应该如何处理。线程遇到未经处理的异常时，CLR 首先尝试将该异常升级成一次得体的线程终止。如果线程在指定时间内没有终止，CLR 就尝试将得体的终止升级成粗鲁的终止。

刚才描述的是最常见的情况。但如果线程在一个临界区(critical region)遇到了未处理的异常，策略就有所不同。位于临界区的线程是指进入线程同步锁的线程，该锁必须由同一个线程释放(例如最初调用 **Monitor.Enter** 方法，调用 **Mutex** 的 **WaitOne** 方法，或者调用 **ReaderWriterLock** 的 **AcquireReaderLock** 或 **AcquireWriteLock** 方法的那个线程[①])。成功等待一个 **AutoResetEvent**，**ManualResetEvent** 或者 **Semaphore** 不会造成线程进入临界区，因为另一个线程可通知这些同步对象。线程在临界区时，CLR 认为线程访问的数据是由同一个 AppDomain 中的多个线程共享的。毕竟，这才是导致线程跑去获取一个锁的原因。直接终止正在正在访问共享数据的线程是不合适的，因为其他线程随后得到的就可能是已损坏的数据，造成 AppDomain 的运行变得无法预测，甚至可能留下安全隐患

所以，位于临界区的线程遭遇未处理的异常时，CLR 首先尝试将异常升级成一次得体的 AppDomain 卸载，从而摆脱(清理)当前正在这个 AppDomain 中的所有线程以及当前正在使用的数据对象。如果 AppDomain 未能在指定时间内卸载，CLR 就将得体的 AppDomain 卸载升级成粗鲁的 AppDomain 卸载。

22.7.3　宿主如何拿回它的线程

宿主应用程序一般都想保持对自己的线程的控制。以一个数据库服务器为例。当一个请求抵达数据库服务器时，线程 A 获得请求，并将该请求派发(dispatch)给线程 B 以执行实际工作。线程 B 可能要执行并不是由数据库服务器的开发团队创建和测试的代码。例如，假定一个请求到达数据库服务器，要执行由运行服务器的公司用托管代码写的存储过程。数据库服务器要求存储过程在自己的 AppDomain 中运行，这个设计自然是极好的，因为能保障安全，防止存储过程访问其 AppDomain 外部的对象，还能防止代码访问不允许访问的资源(比如磁盘文件或剪贴板)。

但是，如果存储过程的代码进入死循环怎么办？在这种情况下，数据库服务器把它的一个线程派发给存储过程代码，但这个线程一去不复返。这便将数据库服务器置于一个危险的

---

① 所有这些锁都在内部调用 **Thread** 的 **BeginCriticalRegion** 和 **EndCriticalRegion** 方法，指出它们要进入和离开临界区。如有必要，你的代码也可调用这些方法。但一般只有在和非托管代码进行互操作时才有必要。

境地：服务器未来的行为变得不可预测了。例如，由于线程进入死循环，所以服务器的性能可能变得很糟。服务器是不是应该创建更多的线程？这样会消耗更多的资源(比如栈空间)，而且这些线程本身也可能进入死循环。

为了解决这些问题，宿主可利用线程中止功能。图 22-3 展示了旨在解决落跑(runway)线程问题的宿主应用程序的典型架构。其具体工作方式如下(编号和图的圆圈中的编号对应)：

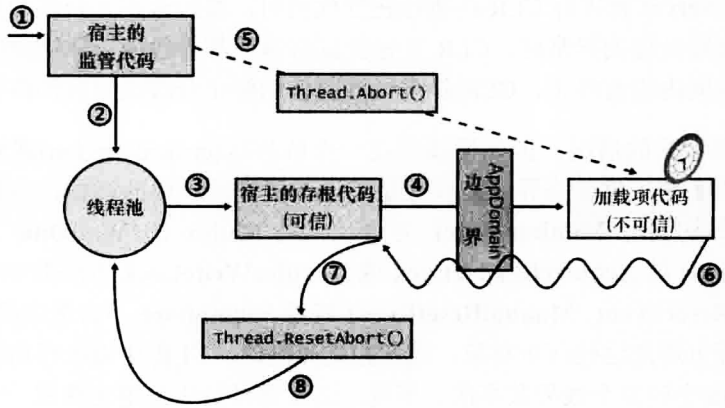


图 22-3　宿主应用程序如何拿回它的线程

1. 客户端向服务器发送请求。

2. 服务器线程获得请求，把它派发给一个线程池线程来执行实际工作。

3. 线程池线程获得客户端的请求，执行由构建并测试宿主应用程序的那个公司写的可信代码(trusted code)。

4. 可信代码进入一个 **try** 块。从这个 **try** 块中，跨越一个 AppDomain 的边界进行调用(通过派生自 **MarshalByRefObject** 的一个类型)。AppDomain 中包含的是不可信代码(可能是存储过程)，这些代码不是由制作宿主应用程序的那个公司生成和测试的。在这个时候，服务器相当于把它的线程的控制权交给了一些不可信的代码。服务器感到有点儿“紧张”了。

5. 宿主会记录接收到客户端请求的时间。不可信代码在管理员设定的时间内没有对客户端做出响应，宿主就会调用 **Thread** 的 **Abort** 方法要求 CLR 中止线程池线程，强制它抛出一个 **ThreadAbortException** 异常。

6. 这时，线程池线程开始展开(unwind)，调用 **finally** 块，使清理代码得以执行。最后，线程池线程穿越 AppDomain 边界返回。由于宿主的存根代码是从一个 **try** 块中调用不可信代码，所以宿主的存根代码有一个 **catch** 块捕捉 **ThreadAbortException**。

7. 为了响应捕捉到的 **ThreadAbortException** 异常，宿主调用 **Thread** 的 **ResetAbort** 方法。稍后将介绍该调用的目的。

8. 现在，宿主的代码已捕捉到 **ThreadAbortException** 异常。因此，宿主可向客户端返回

某种形式的错误，允许线程池线程返回线程池，供未来的客户端请求使用。

澄清一下上述架构中容易被忽视的地方。首先，**Thread** 的 **Abort** 方法是异步的。调用 **Abort** 方法时，会在设置目标线程的 **AbortRequsted** 标志后立即返回。“运行时”检测到一个线程要中止时，会尝试将该线程弄到一个安全地点(safe place)。如果“运行时”认为能安全地停止线程正在做的事情，不会造成灾难性后果，就说线程在安全地点。如果线程正在执行一个托管的阻塞操作(比如睡眠或等待)，它就在一个安全地点。相反，如果线程正在执行类型的类构造器、**catch** 块或 **finally** 块中的代码、CER 中的代码或者非托管代码，线程就不在安全地点。

线程到达安全地点后，“运行时”检测到线程已设置了 **AbortRequsted** 标志。这导致线程抛出一个 **ThreadAbortException**，如果该异常未被捕捉，异常就会成为未处理的异常，所有挂起的 **finally** 块将执行，线程得体地中止。和其他所有异常不同，未处理的 **ThreadAbortException** 不会导致应用程序终止。“运行时”会悄悄地“吞噬”这个异常(假装它没有发生)，线程将“死亡”。但应用程序及其剩余的所有线程都将继续运行。

在本例中，宿主捕捉 **ThreadAbortException**，允许宿主重新获取该线程的控制权，并把它归还到线程池中。但还有一个问题：宿主用什么办法阻止不可信代码自己捕捉 **ThreadAbortException**，从而保持宿主对线程的控制呢？答案是 CLR 以一种非常特殊的方式对待 **ThreadAbortException**。即使代码捕捉了 **ThreadAbortException**，CLR 也不允许代码悄悄地“吞噬”该异常。换言之，在 **catch** 块的尾部，CLR 会自动重新抛出 **ThreadAbortException** 异常。

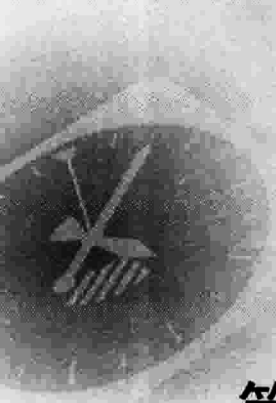
CLR 的这个功能又引起另一个问题：如果 CLR 在 **catch** 块的尾部重新抛出了 **ThreadAbortException** 异常，宿主如何捕捉它并重新获取线程的控制权呢？宿主的 **catch** 块中有一个对 **Thread** 的 **ResetAbort** 方法的调用。调用该方法会告诉 CLR 在 **catch** 块的尾部不要重新抛出 **ThreadAbortException** 异常。

这又引起了另一个问题：宿主怎么阻止不可信代码自己捕捉 **ThreadAbortException** 并调用 **Thread** 的 **ResetAbort** 方法，从而保持宿主对线程的控制呢？答案是 **Thread** 的 **ResetAbort** 方法要求调用者被授予了 **SecurityPermission** 权限，而且其 **ControlThread** 标志已被设为 **true**。宿主为不可信代码创建 AppDomain 时，不会向其授予这个权限，所以不可信代码不能保持对宿主的线程的控制权。

需要指出的是，这里仍然存在一个潜在的漏洞：当线程从它的 **ThreadAbortException** 展开时，不可信代码可执行 **catch** 块和 **finally** 块。在这些块中，不可信代码可能进入死循环，阻止宿主重新获取线程的控制权。宿主应用程序通过设置一个升级策略(前面已进行了讨论)来修正这个问题。要终止的线程在合理的时间内没有完成，CLR 可将线程的终止方式升级成“粗鲁”的线程终止、“粗鲁”的 AppDomain 卸载、禁用 CLR 或者干脆杀死整个进程。还要注意，不可信代码可捕捉 **ThreadAbortException**，并在 **catch** 块中抛出其他种类的一个异常。如果这个其他的异常被捕捉到，CLR 会在 **catch** 块的尾部自动重新抛出 **ThreadAbortException** 异常。

需要指出的是，大多数不可信的代码实际并非故意写成恶意代码；只是根据宿主的标准，它们的执行时间太长了一点。通常，**catch** 块和 **finally** 块只包含极少量代码，这些代码可以很快地执行，不会造成死循环，也不会执行耗时很长的任务。所以，宿主为了重新获取线程的控制权，一般情况都不会动用升级策略(开始各种各样的“粗鲁”行为)。

顺便说一句，**Thread** 类实际提供了两个 **Abort** 方法：一个无参；另一个获取一个 **Object** 参数，允许传递任何东西进来。代码捕捉到 **ThreadAbortException** 时，可查询它的只读 **ExceptionState** 属性。该属性返回的就是传给 **Abort** 的对象。这就允许调用 **Abort** 的线程指定一些额外的信息，供捕捉 **ThreadAbortException** 异常的代码检查。宿主可利用这个功能让自己的处理代码知道它为什么要中止线程。



第 23 章　程序集加载和反射

本章内容：

- 程序集加载
- 使用反射构建动态可扩展应用程序
- 反射的性能
- 设计支持加载项的应用程序
- 使用反射发现类型的成员

本章讨论了在编译时对一个类型一无所知的情况下，如何在运行时发现类型的信息、创建类型的实例以及访问类型的成员。可利用本章讲述的内容创建动态可扩展应用程序。在这种情况下，一般是由一家公司创建宿主应用程序，其他公司创建加载项(add-in)来扩展宿主应用程序。宿主不能基于一些具体的加载项来构建和测试，因为加载项由不同公司创建，而且极有可能是在宿主应用程序发布之后才创建的。这是宿主为什么要在运行时发现加载项的原因。

动态可扩展应用程序可利用第 22 章讲述的 CLR 寄宿和 AppDomain。宿主可以在一个 AppDomain 中运行加载项代码，这个 AppDomain 有它自己的安全性和配置设置。宿主还可通过卸载 AppDomain 来卸载加载项。在本章末尾，将花费一点时间来讨论如何将所有这些功能组合到一起——包括 CLR 寄宿、AppDomain、程序集加载、类型发现、类型实例构造和反射——从而构建健壮、安全而且可以动态扩展的应用程序。



重要提示　.NET Framework 4.5 引入了新的反射 API。旧的 API 缺点太多。例如，它对 LINQ 的支持不好，内建的策略对某些语言来说不正确，有时不必要地强制加载程序集，而且为很少遇到的问题提供了过于复杂的 API。新 API 解决了所有这些问题。但至少就.NET 4.5 来说，新的反射 API 还不如旧 API 完整。利用新 API 和 **System.Reflection.RuntimeReflectionExtensions** 类中的扩展方法，现在所有事情都可以做到。希望在.NET Framework 未来的版本中为新 API 添加更多的方法。当然，对于桌面应用程序，旧 API 是仍然存在的，重新编译现有的代码不会出任何问题。但新 API 是未来的发展方向，这正是本章要全面讨论新 API 的原因。Windows Store 应用由于不用考虑向后兼容，所以必须使用新 API。

23.1 程序集加载

我们知道，JIT 编译器将方法的 IL 代码编译成本机代码时，会查看 IL 代码中引用了哪些类型。在运行时，JIT 编译器利用程序集的 TypeRef 和 AssemblyRef 元数据表来确定哪一个程序集定义了所引用的类型。在 AssemblyRef 元数据表的记录项中，包含了构成程序集强名称的各个部分。JIT 编译器获取所有这些部分——包括名称(无扩展名和路径)、版本、语言文化和公钥标记(public key token)——并把它们连接成一个字符串。然后，JIT 编译器尝试将与该标识匹配的程序集加载到 AppDomain 中(如果还没有加载的话)。如果被加载的程序集是弱命名的，那么标识中就只包含程序集的名称(不包含版本、语言文化及公钥标记信息)。[①]

在内部，CLR 使用 **System.Reflection.Assembly** 类的静态 **Load** 方法尝试加载这个程序集。该方法在.NET Framework SDK 文档中是公开的，可调用它显式地将程序集加载到 AppDomain 中。该方法是 CLR 的与 Win32 **LoadLibrary** 函数等价的方法。**Assembly** 的 **Load** 方法实际有几个重载版本。以下是最常用的重载的原型：

```
public class Assembly {
   public static Assembly Load(AssemblyName assemblyRef);
   public static Assembly Load(String assemblyString);
   // 未列出不常用的 Load 重载
}
```

在内部，**Load** 导致 CLR 向程序集应用一个版本绑定重定向策略，并在 GAC(全局程序集缓存)中查找程序集。如果没找到，就接着去应用程序的基目录、私有路径子目录和 **codebase**[②] 位置查找。如果调用 **Load** 时传递的是弱命名程序集，**Load** 就不会向程序集应用版本绑定重定向策略，CLR 也不会去 GAC 查找程序集。如果 **Load** 找到指定的程序集，会返回对代表已加载的那个程序集的一个 **Assembly** 对象的引用。如果 **Load** 没有找到指定程序集，会抛出一个 **System.IO.FileNotFoundException** 异常。



注意 一些极罕见的情况可能需要加载为特定 CPU 架构生成的程序集。这时在指定程序集的标识时，还可包括一个进程架构部分。例如，假定 GAC 中同时包含了一个程序集的 IL 中立版本和 x86 专用版本，CLR 会默认选择 x86 专用版本(参见第 3 章“共享程序集和强命名程序集”)。但是，为了强迫 CLR 加载 IL 中立的版本，可以向 **Assembly** 的 **Load** 方法传递以下字符串：

```
"SomeAssembly, Version=2.0.0.0, Culture=neutral,
PublicKeyToken=01234567890abcde, ProcessorArchitecture=MSIL"
```

CLR 目前允许 **ProcessorArchitecture** 取 5 个值之一：**MSIL**(Microsoft IL)、**X86**、**IA64**，**AMD64** 以及 **Arm**。

① 强命名程序集和弱命名程序集的区别请参见 3.1 节“两种程序集，两种部署”。——译注
② 要了解 **codeBase** 元素指定的位置，请参见 3.9 节“高级管理控制(配置)”。——译注



重要提示　一些开发人员可能注意到 **System.AppDomain** 提供了 **Load** 方法。和 **Assembly** 的静态 **Load** 方法不同，**AppDomain** 的 **Load** 是实例方法，它允许将程序集加载到指定的 AppDomain 中。该方法设计由非托管代码调用，允许宿主将程序集“注入”特定 AppDomain 中。托管代码的开发人员一般情况下不应调用它，因为调用 **AppDomain** 的 **Load** 方法时需要传递一个标识了程序集的字符串。该方法随后会应用策略，并在一些常规位置搜索程序集。我们知道，AppDomain 关联了一些告诉 CLR 如何查找程序集的设置。为了加载这个程序集，CLR 将使用与指定 AppDomain 关联的设置，而非与发出调用之 AppDomain 关联的设置。

但 **AppDomain** 的 **Load** 方法会返回对程序集的引用。由于 **System.Assembly** 类不是从 **System.MarshalByRefObject** 派生的，所以程序集对象必须按值封送回发出调用的那个 AppDomain。但是，现在 CLR 就会用发出调用的那个 AppDomain 的设置来定位并加载程序集。如果使用发出调用的那个 AppDomain 的策略和搜索位置找不到指定的程序集，就会抛出一个 **FileNotFoundException**。这个行为一般不是你所期望的，所以应该避免使用 **AppDomain** 的 **Load** 方法。

在大多数动态可扩展应用程序中，**Assembly** 的 **Load** 方法是将程序集加载到 AppDomain 的首选方式。但它要求事先掌握构成程序集标识的各个部分。开发人员经常需要写一些工具或实用程序(例如 ILDasm.exe，PEVerify.exe，CorFlags.exe，GACUtil.exe，SGen.exe，SN.exe 和 XSD.exe 等)来操作程序集，它们都要获取引用了程序集文件路径名(包括文件扩展名)的命令行实参。

调用 **Assembly** 的 **LoadFrom** 方法加载指定了路径名的程序集：

```
public class Assembly {
   public static Assembly LoadFrom(String path);
   // 未列出不常用的 LoadFrom 重载
}
```

在内部，**LoadFrom** 首先调用 **System.Reflection.AssemblyName** 类的静态 **GetAssemblyName** 方法。该方法打开指定的文件，找到 **AssemblyRef** 元数据表的记录项，提取程序集标识信息，然后以一个 **System.Reflection.AssemblyName** 对象的形式返回这些信息(文件同时会关闭)。随后，**LoadFrom** 方法在内部调用 **Assembly** 的 **Load** 方法，将 **AssemblyName** 对象传给它。然后，CLR 应用版本绑定重定向策略，并在各个位置查找匹配的程序集。**Load** 找到匹配程序集会加载它，并返回代表已加载程序集的 **Assembly** 对象；**LoadFrom** 方法将返回这个值。如果 **Load** 没有找到匹配的程序集，**LoadFrom** 会加载通过 **LoadFrom** 的实参传递的路径中的程序集。当然，如果已加载了具有相同标识的程序集，**LoadFrom** 方法就会直接返回代表已加载程序集的 **Assembly** 对象。

顺便说一句，**LoadFrom** 方法允许传递一个 URL 作为实参，下面是一个例子：

```
Assembly a = Assembly.LoadFrom(@"http://Wintellect.com/SomeAssembly.dll");
```

如果传递的是一个 Internet 位置，CLR 会下载文件，把它安装到用户的下载缓存中，再从那儿加载文件。注意，当前必须联网，否则会抛出异常。但如果文件之前已下载过，而且 Microsoft Internet Explorer 被设为脱机工作(在 Internet Explorer 中单击“文件”|“脱机工作”)，就会使用以前下载的文件，不会抛出异常。还可以调用 **UnsafeLoadFrom**，它能够加载从网上下载的程序集，同时绕过一些安全检查。



重要提示 一台机器可能同时存在具有相同标识的多个程序集。由于 **LoadFrom** 会在内部调用 **Load**，所以 CLR 有可能不是加载你指定的文件，而是加载一个不同的文件，从而造成非预期的行为。强烈建议每次生成程序集时都更改版本号，确保每个版本都有自己的唯一性标识，确保 **LoadFrom** 方法的行为符合预期。

Microsoft Visual Studio 的 UI 设计人员和其他工具一般用的是 **Assembly** 的 **LoadFile** 方法。这个方法可从任意路径加载程序集，而且可以将具有相同标识的程序集多次加载到一个 AppDomain 中。在设计器/工具中对应用程序的 UI 进行了修改，而且用户重新生成了程序集时，便有可能发生这种情况。通过 **LoadFile** 加载程序集时，CLR 不会自动解析任何依赖性问题；你的代码必须向 **AppDomain** 的 **AssemblyResolve** 事件登记，并让事件回调方法显式地加载任何依赖的程序集。

如果你构建的一个工具只想通过反射(本章稍后进行讨论)来分析程序集的元数据，并希望确保程序集中的任何代码都不会执行，那么加载程序集的最佳方式就是使用 **Assembly** 的 **ReflectionOnlyLoadFrom** 方法或者使用 **Assembly** 的 **ReflectionOnlyLoad** 方法(后者比较少见)。下面是这两个方法的原型：

```
public class Assembly {
   public static Assembly ReflectionOnlyLoadFrom(String assemblyFile);
   public static Assembly ReflectionOnlyLoad(String assemblyString);
   // 未列出不常用的 ReflectionOnlyLoad 重载
}
```

ReflectionOnlyLoadFrom 方法加载由路径指定的文件；文件的强名称标识不会获取，也不会在 GAC 和其他位置搜索文件。**ReflectionOnlyLoad** 方法会在 GAC、应用程序基目录、私有路径和 **codebase** 指定的位置搜索指定的程序集。但和 **Load** 方法不同的是，**ReflectionOnlyLoad** 方法不会应用版本控制策略，所以你指定的是哪个版本，获得的就是哪个版本。要自行向程序集标识应用版本控制策略，可将字符串传给 **AppDomain** 的 **ApplyPolicy** 方法。

用 **ReflectionOnlyLoadFrom** 或 **ReflectionOnlyLoad** 方法加载程序集时，CLR 禁止程序集中的任何代码执行；试图执行由这两个方法加载的程序集中的代码，会导致 CLR 抛出一个 **InvalidOperationException** 异常。这两个方法允许工具加载延迟签名的程序集[①]，这种程序

① 要进一步了解延迟签名的程序集，请参见 3.6 节“延迟签名”。——译注

集正常情况下会因为安全权限不够而无法加载。另外，这种程序集也可能是为不同的 CPU 架构而创建的。

利用反射来分析由这两个方法之一加载的程序集时，代码经常需要向 **AppDomain** 的 **ReflectionOnlyAssemblyResovle** 事件注册一个回调方法，以便手动加载任何引用的程序集(如有必要，还需要调用 **AppDomain** 的 **ApplyPolicy** 方法)；CLR 不会自动帮你做这个事情。回调方法被调用(invoke)时，它必须调用(call) **Assembly** 的 **ReflectionOnlyLoadFrom** 或 **ReflectionOnlyLoad** 方法来显式加载引用的程序集，并返回对该程序集的引用。



注意　经常有人问到程序集卸载的问题。遗憾的是，CLR 不提供卸载单独程序集的能力。如果 CLR 允许这样做，那么一旦线程从某个方法返回至已卸载的一个程序集中的代码，应用程序就会崩溃。健壮性和安全性是 CLR 最优先考虑的目标，如果允许应用程序以这样的一种方式崩溃，就和它的设计初衷背道而驰了。卸载程序集必须卸载包含它的整个 AppDomain。这方面的详情已在第 22 章“CLR 寄宿和 AppDomain”进行了讨论。

使用 **ReflectionOnlyLoadFrom** 或 **ReflectionOnlyLoad** 方法加载的程序集表面上是可以卸载的。毕竟，这些程序集中的代码是不允许执行的。但 CLR 一样不允许卸载用这两个方法加载的程序集。因为用这两个方法加载了程序集之后，仍然可以利用反射来创建对象，以便引用这些程序集中定义的元数据。如果卸载程序集，就必须通过某种方式使这些对象失效。无论是实现的复杂性，还是执行速度，跟踪这些对象的状态都是得不偿失的。

许多应用程序都由一个要依赖于众多 DLL 文件的 EXE 文件构成。部署应用程序时，所有文件都必须部署。但有一个技术允许只部署一个 EXE 文件。首先标识出 EXE 文件要依赖的、不是作为 Microsoft .NET Framework 一部分发布的所有 DLL 文件。然后将这些 DLL 添加到 Visual Studio 项目中。对于添加的每个 DLL，都显示它的属性，将它的“生成操作”更改为“嵌入的资源”。这会导致 C#编译器将 DLL 文件嵌入 EXE 文件中，以后就只需部署这个 EXE。

在运行时，CLR 会找不到依赖的 DLL 程序集。为了解决这个问题，当应用程序初始化时，向 **AppDomain** 的 **ResolveAssembly** 事件登记一个回调方法。代码大致如下：

```
private static Assembly ResolveEventHandler(Object sender, ResolveEventArgs args) {
   String dllName = new AssemblyName(args.Name).Name + ".dll";

   var assem = Assembly.GetExecutingAssembly();
   String resourceName = assem.GetManifestResourceNames().FirstOrDefault(rn =>
rn.EndsWith(dllName));
   if (resourceName == null) return null; // Not found, maybe another handler will find
it
   using (var stream = assem.GetManifestResourceStream(resourceName)) {
      Byte[] assemblyData = new Byte[stream.Length];
```

```
            stream.Read(assemblyData, 0, assemblyData.Length);
            return Assembly.Load(assemblyData);
        }
    }
```

现在，线程首次调用一个方法时，如果发现该方法引用了依赖 DLL 文件中的类型，就会引发一个 **AssemblyResolve** 事件，而上述回调代码会找到所需的嵌入 DLL 资源，并调用 **Assembly** 的 **Load** 方法获取一个 **Byte[]**实参的重载版本来加载所需的资源。虽然我喜欢将依赖 DLL 嵌入程序集的技术，但要注意这会增大应用程序在运行时的内存消耗。

23.2 使用反射构建动态可扩展应用程序

众所周知，元数据是用一系列表来存储的。生成程序集或模块时，编译器会创建一个类型定义表、一个字段定义表、一个方法定义表以及其他表。利用 **System.Reflection** 命名空间中包含的类型，可以写代码来反射(或者说“解析”)这些元数据表。实际上，这个命名空间中的类型为程序集或模块中包含的元数据提供了一个对象模型。

利用对象模型中的类型，可以轻松枚举类型定义元数据表中的所有类型，而针对每个类型都可获取它的基类型、它实现的接口以及与类型关联的标志(flag)。利用 **System.Reflection** 命名空间中的其他类型，还可解析对应的元数据表来查询类型的字段、方法、属性和事件。还可发现应用于任何元数据实体的定制特性(详情参见第 18 章“定制特性”)。甚至有些类允许判断引用的程序集；还有一些方法能返回一个方法的 IL 字节流。利用所有这些信息，很容易构建出与 Microsoft 的 ILDasm.exe 相似的工具。



> **注意** 有的反射类型及其成员是专门由 CLR 编译器的开发人员使用的。应用程序的开发人员一般用不着。FCL 文档没有明确指出哪些类型和成员供编译器开发人员(而非应用程序开发人员)使用，但只要意识到有些反射类型及其成员不适合所有人使用，读文档时就会更清醒一些。

事实上，只有极少数应用程序才需使用反射类型。如果类库需要理解类型的定义才能提供丰富的功能，就适合使用反射。例如，FCL 的序列化机制(详情参见第 24 章“运行时序列化”)就是利用反射来判断类型定义了哪些字段。然后，序列化格式器(serialization formatter)可获取这些字段的值，把它们写入字节流以便通过 Internet 传送、保存到文件或复制到剪贴板。类似地，在设计期间，Microsoft Visual Studio 设计器在 Web 窗体或 Windows 窗体上放置控件时，也利用反射来决定要向开发人员显示的属性。

在运行时，当应用程序需要从特定程序集中加载特定类型以执行特定任务时，也要使用反射。例如，应用程序可要求用户提供程序集和类型名。然后，应用程序可显式加载程序集，构造类型的实例，再调用类型中定义的方法。这种用法在概念上类似于调用 Win32 **LoadLibrary** 和 **GetProcAddress** 函数。以这种方式绑定到类型并调用方法称为**晚期绑定**。(对应的，早期绑定是指在编译时就确定应用程序要使用的类型和方法。)

23.3　反射的性能

反射是相当强大的机制，允许在运行时发现并使用编译时还不了解的类型及其成员。但是，它也有下面两个缺点。

- 反射造成编译时无法保证类型安全性。由于反射严重依赖字符串，所以会丧失编译时的类型安全性。例如，执行 **Type.GetType("int");**要求通过反射在程序集中查找名为 **"int"**的类型，代码会通过编译，但在运行时会返回 **null**，因为 CLR 只知**"System.Int32"**，不知**"int"**。

- 反射速度慢。使用反射时，类型及其成员的名称在编译时未知；你要用字符串名称标识每个类型及其成员，然后在运行时发现它们。也就是说，使用 **System.Reflection** 命名空间中的类型扫描程序集的元数据时，反射机制会不停地执行字符串搜索。通常，字符串搜索执行的是不区分大小写的比较，这会进一步影响速度。

使用反射调用成员也会影响性能。用反射调用方法时，首先必须将实参打包(pack)成数组；在内部，反射必须将这些实参解包(unpack)到线程栈上。此外，在调用方法前，CLR 必须检查实参具有正确的数据类型。最后，CLR 必须确保调用者有正确的安全权限来访问被调用的成员。

基于上述所有原因，最好避免利用反射来访问字段或调用方法/属性。应该利用以下两种技术之一开发应用程序来动态发现和构造类型实例。

- 让类型从编译时已知的基类型派生。在运行时构造派生类型的实例，将对它的引用放到基类型的变量中(利用转型)，再调用基类型定义的虚方法。

- 让类型实现编译时已知的接口。在运行时构造类型的实例，将对它的引用放到接口类型的变量中(利用转型)，再调用接口定义的方法。

在这两种技术中，我个人更喜欢使用接口技术而非基类技术，因为基类技术不允许开发人员选择特定情况下工作得最好的基类。不过，需要版本控制的时候基类技术更合适，因为可随时向基类型添加成员，派生类会直接继承该成员。相反，要向接口添加成员，实现该接口的所有类型都得修改它们的代码并重新编译。

使用这两种技术时，强烈建议接口或基类型在它们自己的程序集中定义，这有助于缓解版本控制问题。欲知详情，请参见稍后的 23.4 节“设计支持加载项的应用程序”。

23.3.1 发现程序集中定义的类型

反射经常用于判断程序集定义了哪些类型。FCL 提供了许多 API 来获取这方面的信息。目前最常用的 API 是 **Assembly** 的 **ExportedTypes** 属性。下例加载一个程序集，并显示其中定义的所有公开导出的类型[①]：

```
using System;
using System.Reflection;

public static class Program {
   public static void Main() {
      String dataAssembly = "System.Data, version=4.0.0.0, " +
         "culture=neutral, PublicKeyToken=b77a5c561934e089";
      LoadAssemAndShowPublicTypes(dataAssembly);
   }

   private static void LoadAssemAndShowPublicTypes(String assemId) {
      // 显式地将程序集加载到这个 AppDomain 中
      Assembly a = Assembly.Load(assemId);

      // 在一个循环中显示已加载程序集中每个公      开导出 Type 的全名
      foreach (Type t in a.ExportedTypes) {
         // 显示类型全名
         Console.WriteLine(t.FullName);
      }
   }
}
```

23.3.2 类型对象的准确含义

注意，上述代码遍历 **System.Type** 对象构成的数组。**System.Type** 类型是执行类型和对象操作的起点。**System.Type** 对象代表一个类型引用(而不是类型定义)。

众所周知，**System.Object** 定义了公共非虚实例方法 **GetType**。调用这个方法时，CLR 会判断指定对象的类型，并返回对该类型的 **Type** 对象的引用。由于在一个 AppDomain 中，每个类型只有一个 **Type** 对象，所以可以使用相等和不等操作符来判断两个对象是不是相同的类型：

```
private static Boolean AreObjectsTheSameType(Object o1, Object o2) {
   return o1.GetType() == o2.GetType();
}
```

除了调用 **Object** 的 **GetType** 方法，FCL 还提供了获得 **Type** 对象的其他几种方式。

- **System.Type** 类型提供了静态 **GetType** 方法的几个重载版本。所有版本都接受一个 **String** 参数。字符串必须指定类型的全名(包括它的命名空间)。注意不允许使用编译器支持的基元类型(比如 C#的 **int**，**string**，**bool** 等)，这些名称对于 CLR 没有任何意义。如果传递的只是一个类型名称，方法将检查调用程序集，看它是否定义了指定名称的类型。如果是，就返回对恰当 **Type** 对象的引用。

① 所谓公开导出的类型，就是程序集中定义的 **public** 类型，它们在程序集的外部可见。——译注

如果调用程序集没有定义指定的类型，就检查 MSCorLib.dll 定义的类型。如果还是没有找到，就返回 **null** 或抛出 **System.TypeLoadException**(取决于调用的是 **GetType** 方法的哪个重载，以及传递的是什么参数)。文档对该方法进行了完整解释。

可向 **GetType** 传递限定了程序集的类型字符串，比如"**System.Int32, mscorlib, Version=4.0.0.0, Culture=neutral, PublicKeyToken=b77a5c561934e089**"。在本例中，**GetType** 会在指定程序集中查找类型(如有必要会加载程序集)。

- **System.Type** 类型提供了静态 **ReflectionOnlyGetType** 方法。该方法与上一条提到的 **GetType** 方法在行为上相似，只是类型会以“仅反射”的方式加载，不能执行。
- **System.TypeInfo** 类型提供了实例成员 **DeclaredNestedTypes** 和 **GetDeclaredNestedType**。
- **System.Reflection.Assembly** 类型提供了实例成员 **GetType**，**DefinedTypes** 和 **ExportedTypes**。



注意　构造传给反射方法的字符串时，要使用类型名称或限定了程序集的类型名称。Microsoft 为这些名称定义了巴克斯-诺尔范式(Backus-Naur Form，BNF)语法。使用反射时，了解这种语法对你很有帮助，尤其是在处理嵌套类型、泛型类型、泛型方法、引用参数或者数组的时候。要想了解完整语法，请参考文档[①]或者 Goolge “BNF 类型名称”。还可参考 **Type** 和 **TypeInfo** 的 **MakeArrayType**, **MakeByRefType**, **MakeGenericType** 和 **MakePointerType** 方法。

许多编程语言都允许使用一个操作符并根据编译时已知的类型名称来获得 **Type** 对象。尽量用这个操作符获得 **Type** 引用，而不要使用上述列表中的任何方法，因为操作符生成的代码通常更快。C#的这个操作符称为 **typeof**，通常用它将晚期绑定的类型信息与早期绑定(编译时已知)的类型信息进行比较。以下代码演示了一个例子：

```
private static void SomeMethod(Object o) {
   // GetType 在运行时返回对象的类型(晚期绑定)
   // typeof 返回指定类的类型(早期绑定)
   if (o.GetType() == typeof(FileInfo)) { ... }
   if (o.GetType() == typeof(DirectoryInfo)) { ... }
}
```



注意　上述代码的第一个 **if** 语句检查变量 **o** 是否引用了 **FileInfo** 类型的对象；它不检查 **o** 是否引用从 **FileInfo** 类型派生的对象。换言之，上述代码测试的是精确匹配，而非兼容匹配。(使用转型或 C#的 **is/as** 操作符时，测试的就是兼容匹配。)

① 参考文档中的“指定完全限定的类型名称”主题。——译注