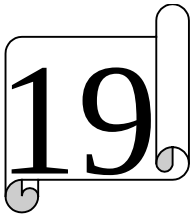

自动内存管理(垃圾收集)

本章讨论有关托管对象的一些核心问题，它们包括新对象的创建、生存期的管理、以及内存资源的回收。换言之，本章将向大家解释 Microsoft .NET 框架中垃圾收集器的工作原理，以及与之相关的各种性能问题。

19.1　垃圾收集平台基本原理解析

每个程序都要使用这样或那样的资源，比如文件、内存缓冲、屏幕空间、网络连接、数据库资源，等等。实际上，在一个面向对象的环境里，每一个类型都标识着某些为程序所用的资源。要想使用这些资源，我们必须为相应的类型实例分配一定的内存空间。下面是访问一个资源所需要的几个步骤。

(1) 调用中间语言(IL)中的 newobj指令，为表示某个特定资源的类型实例分配一定的内存空间。当我们在C#或者Visual Basic以及其他一些编程语言中调用new操作符时，编译器将产生newobj指令。

(2) 初始化上一步所得的内存，设置资源的初始状态，从而使其可以为程序所用。一个类型的实例构造器负责做这样的初始化工作。

(3) 通过访问类型成员来使用资源，这根据需要会有一些反复。

(4) 销毁资源状态，执行清理工作。这将在本章19.4一节中予以探讨。

(5) 释放内存。这一步由垃圾收集器全权负责。(译注：注意这里的内存指的是分配在托管堆上的引用类型实例所占的内存资源。除了托管堆中的内存，系统运行时还有一类内存，即值类型实例所占的内存，它们位于当前运行线程的堆栈上，垃圾收集器并不负责这些内存资源的回收。当值类型实例变量所在的方法执行结束时，它们的内存将随着堆栈空间的消亡而自动消失，也就无所谓回收。)

这种看起来简单的模式却是导致许多编程错误的主要原因之一。想想看，有多少次我们忘记了释放无用的内存？又有多少次我们试图访问已经被释放的内存，恐怕谁也说不清楚。

因为这两类 bug 发生的时间和次序都难以预料，所以它们对于应用程序来说比其他大多数 bug 的危害都要大得多。对于其他大多数 bug 来讲，程序的异常行为一经发现，一般问题都能很快得到解决。但是这两类 bug 的直接后果是资源泄漏(内存消耗)和对象损毁(状态不稳定)，这使得应用程序的行为变得不可预期，发生泄漏和损毁的次数也不可预期。实际上，有许多工具(如微软的 Windows 任务管理器、系统监视器 ActiveX 控件，以及来自 Compuware 公司的 NuMega BoundsChecker 和 Rational 公司的 Purify)就是专门用来帮助开发人员查找此类 bug 的。

正确无误的资源管理通常是一件比较困难和单调的工作，它们会极大地分散开发人员解决实际问题的注意力。如果有一种机制能够简化这种容易遗漏的内存管理任务，那将是一件令人愉快的事。值得庆幸的是，确实存在这样的机制，这就是垃圾收集(garbage collection)。

垃圾收集完全将开发人员从繁杂的内存管理工作中解脱了出来，而这在以前需要开发人员追踪每一块内存的使用情况，并知晓何时释放它们。但是垃圾收集器对内存中的类型表示着何种资源却一无所知，这意味着垃圾收集器并不清楚怎样执行前面列表中的第 4 步，即“销毁资源状态，执行清理工作”。为了使资源得到正确的清理，开发人员必须自己编写执行这部分工作的代码。这些代码一般被放在 Finalize、Dispose 以及 Close 方法中，本章稍后将会谈到它们。其实在后面大家将会看到，垃圾收集器在这个问题上也能提供一些帮助，从而允许我们在许多情形下可以跳过上面第 4 步的工作。

另外，大多数类型，例如 Int32、Point、Rectangle、String、ArrayList 以及 SerializationInfo，表示的资源并不需要任何特殊的清理操作。例如，一个 Point 资源完全可以通过销毁对象内存中的 x 字段和 y 字段来完成清理工作。

另一方面，对于一个表示(或者说封装)着非托管(操作系统)资源(例如文件、数据库连接、套接字、互斥体、位图、图标，等等)的类型，在其对象被销毁时，就必须执行一些清理代码。

本章稍后将向大家解释怎样正确定义那些需要明确清理资源的类型，以及怎样正确使用这些类型。我们目前首先来探讨内存分配和资源初始化问题。

通用语言运行时(CLR)要求所有的内存资源(译注：这里仅限于分配给引用类型实例的内存资源)都从一个称作**托管堆**(managed heap)的地方分配而得。除了不需要从托管堆中释放对象外——当应用程序不再需要它们时，对象会被自动释放——托管堆和 C 语言运行时中的堆非常相似。这自然会提出一个问题，“托管堆是如何知道应用程序何时不再使用某个对象的？”这个问题稍后回答。

目前应用于实践中的垃圾收集算法有好几种。每一种算法都是针对某一特定环境而量身定做的最优化方案。本章中，我们将把注意力集中在.NET 框架中 CLR 提供的垃圾收集算法上。我们先从最基本的概念开始。

当应用程序进程完成初始化后，CLR 将保留(reserve)一块连续的地址空间，这段空间最初并不对应任何的物理内存(backing storage)。(译注：在进程的可用地址空间上为其分配内存的行为称作“保留”，进程因此分配而得的内存空间称作“保留地址空间”。由于保留地址空间是一段虚拟地址空间，所以要真正使用它，还必须为其“提交”物理内存。有关物理内存与虚拟内存、保留与提交的更详细的知识可参见 Jeffrey Richter 先生的 Programming Applications for Microsoft Windows 第 4 版。)该地址空间即为托管堆。托管堆上维护着一个指针，我们暂且称之为 NextObjPtr。该指针标识着下一个新建对象分配时在托管堆中所处的位置。刚开始的时候，NextObjPtr 被设为 CLR 保留地址空间的基地址。

中间语言(IL)指令 newobj 负责创建新的对象。如前所述，许多语言(包括 C#和 Visual Basic)都提供有一个 new 操作符，该操作符会使编译器在相应方法的 IL 代码中产生一个 newobj 指令。在代码运行时，newobj 指令将导致 CLR 执行以下几步操作：

(1) 计算类型所有字段(以及其基类所有的字段)(译注：这里所说的字段应为类型的实例字段)所需要的字节总数。

(2) 在前面所得字节总数的基础上再加上对象额外的附加成员所需的字节数。每个对象包括两个附加字段：一个方法表指针和一个SyncBlockIndex。在32位的系统中，这两个字段各占32位，合起来将为每个对象增加8个字节。在64位的系统中，每个字段将占用64位空间，两个合起来将为每个对象增加16个字节。

(3) CLR检查保留区域中的空间是否满足分配新对象所需的字节数——如果需要则提交(commit)物理内存。如果满足，对象将被分配在NextObjPtr指针所指示的地方。接着，类型的实例构造器被调用(NextObjPtr指针会被传递给this参数)，IL指令newobj(或者说new操作符)返回为其分配的内存地址。就在newobj指令返回新对象的地址之前，NextObjPtr指针会越过新对象所处的内存区域，并指示出下一个新建对象在托管堆中的地址。

图19.1演示了一个包括3个对象的托管堆：A、B和C。如果再分配新的对象，它将被放在NextObjPtr指针所指示的位置(紧接着对象C之后)。

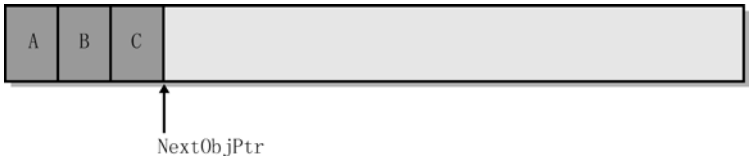

图 19.1 包含 3 个对象的托管堆

作为对比，我们来看一下 C 语言运行时中的堆分配内存时的情况。C 语言运行时中的堆在为对象分配内存时首先需要遍历一个链表数据结构，一旦找到了一个足够大的内存块，该内存块就会被拆分开来，同时链表相应节点上的指针也会得到适当的调整。但是对于托管堆来说，分配对象仅仅意味着在指针上增加一个数值——这显然要比操作链表的做法快许多。实际上，在托管堆中分配对象几乎可以达到和在线程堆栈中分配对象一样快的速度！另外，大多数的堆(如 C 语言运行时中的堆)都是在所找到的自由空间中为对象分配内存。因此，如果我们连续地创建了几个对象，它们将很有可能被分散在地址空间的各个角落。但是在托管堆中，连续分配的对象可以保证它们在内存中也是连续的。

在许多应用程序中，同一时间分配的对象彼此之间大多都有着较强的联系，它们经常会在同一时间被访问。例如，通常我们会在分配一个 FileStream 对象之后紧接着再分配一个 BinaryWriter 对象。然后，当应用程序使用 BinaryWriter 对象时，BinaryWriter 对象内部也会使用到 FileStream 对象。在一个垃圾收集环境中，对象在内存中的连续分配会由于引用的本地化而获得一些性能方面的提升。具体而言，这意味着应用程序的进程工作集将变得更小，方法中使用的对象也更有可能全部驻留在 CPU 缓存中。这样以来，应用程序只需使用 CPU 缓存中的数据(不会出现因缓存遗漏而导致的强制 RAM 访问)就可以执行大多数的操作，访问对象的速度自然会变得更快。(译注：进程的工作集指进程在运行时被频繁地访问的内存页面集合。如果进程的工作集比较大，进程在运行时将出现频繁的页面换入或换出现象。这种页面的换入或换出会发生在硬盘和内存之间，也会发生在内存和 CPU 缓存之间，这里主要指后一种情况。)

到目前为止，看起来好像托管堆在实现的简单性和速度方面要远优于 C 语言运行时中的堆。但是，在为此兴奋之前我们还需要清楚一个细节，即托管堆之所以能够获得这些好处是因为它实际上做了一个相当大胆的假设——那就是应用程序的地址空间和存储空间是无限的。显然，这种假设是荒谬的。托管堆必须应用某种机制来允许它做这样的假设。这种机制就是垃圾收集器。下面是其工作原理。

当应用程序调用 new 操作符创建对象时，托管堆中可能没有足够的地址空间来分配该对象。托管堆通过将对象所需要的字节总数添加到 NextObjPtr 指针表示的地址上来检测这种情况。如果得到的结果超出了托管堆的地址空间范围，那么托管堆将被认为已经充满，这时就需要执行垃圾收集。

重要　上面的描述有些过于简单。实际上，垃圾收集在第 0 代对象充满时就出现了。一些垃圾收集器使用一种称作**代龄**(generation)的机制，该机制的惟一目的就是提高垃圾收集的性能。其基本思想是将应用程序生存期中新创建的对象认为是较新的代的一部分，而将早先创建的对象认为是较老的代的一部分。将对象划分为各个代龄使得垃圾收集器能够将执行对象限定在某个特定的代龄中，从而避免每次都对托管堆中所有的对象执行垃圾收集。本章稍后会对垃圾收集中代龄这一机制进行详细的探讨。在此之前，大家可以先简单地认为在托管堆空间充满的时候才会出现垃圾收集。

19.2　垃圾收集算法

垃圾收集器通过检查在托管堆中是否有应用程序不再使用的对象来回收内存。如果有这样的对象，它们占用的内存将可以被回收。(如果托管堆中没有可用的内存，new 操作符将会抛出一个 OutOfMemoryException 异常。)那么垃圾收集器是怎样知道应用程序是否正在使用一个对象呢？这不是一个三言两语就能说的清楚的问题。

每个应用程序都有一组根(root)。一个根是一个存储位置，其中包含着一个指向引用类型的内存指针。该指针或者指向一个托管堆中的对象，或者被设为 null。例如，所有的全局引用类型变量或静态引用类型变量都被认为是根。另外，一个线程堆栈上所有引用类型的本地变量或者参数变量也被认为是一个根。最后，在一个方法内，指向引用类型对象的 CPU 寄存器也被认为是一个根。

当 JIT 编译器编译一个方法的 IL 代码时，除了产生本地 CPU 代码外，JIT 编译器还会创建一个内部的表。从逻辑上来讲，该表中的每一个条目都标识着一个方法的本地 CPU 指令的字节偏移范围，以及该范围中一组包含根的内存地址(或者 CPU 寄存器)。表 19.1 形象地描述了一个这样的内部表。

表 19.1 一个 JIT 编译器生成的表，其中展示了本地代码偏移和方法中的根的映射关系

起始字节偏移	结尾字节偏移	根
0x00000000	0x00000020	this, arg1, arg2, ECX, EDX
0x00000021	0x00000122	this, arg2, fs, EBX
0x00000123	0x00000145	fs

如果在 0x00000021 和 0x00000122 之间的代码执行时开始了垃圾收集，那么垃圾收集器将知道参数 this、参数 arg2、本地变量 fs、以及寄存器 EBX 都是根，它们引用的托管堆中的对象将不会被认为是可收集的垃圾对象。除此之外，垃圾收集器还可以遍历线程的调用堆栈，通过检测其中每一个方法的内部表来确定所有调用方法中的根。最后，垃圾收集器使用其他一些手段来获得存储在全局引用类型变量和静态引用类型变量中保存的根。

注意 在表 19.1 中，方法的 arg1 参数在偏移为 0x00000020 处的指令执行完毕后就不再被引用了。这意味着 arg1 引用的对象在该指令执行后的任何时刻都可以被执行垃圾收集(假设应用程序中没有其他的根再引用该对象)。换句话说，只要一个对象不再可达，它就是垃圾收集的候选对象——CLR 并不保证对象在一个方法的整个生存期内都能一直存活。

但是，当应用程序运行在一个调试器中，或者当程序集包含有 System.Diagnostics.DebuggableAttribute 特性、且其构造器的参数 isJITOptimizerDisabled 被设为 true 时，JIT 编译器将会把所有变量（值类型和引用类型）的生存期扩展到它们的作用范围末端——通常为方法的结尾之处。（顺便提一句，微软的 C#编译器提供了一个/debug 命令行开关选项，它可以为程序集添加 DebuggableAttribute 特性，并将其参数 isJITOptimizerDisabled 设为 true。）该扩展会阻止垃圾收集器当代码在引用类型对象的作用范围内运行时对它们执行垃圾收集，这在做代码调试时将十分有用。想想看，如果我们调用了一个对象的方法时得到了错误的结果，但随后却看不到该对象，这无论如何都是一件让人感到奇怪的事情。

当垃圾收集器开始执行时，它首先假设托管堆中所有的对象都是可收集的垃圾。换句话说，垃圾收集器假设应用程序中没有一个根引用着托管堆中的对象。然后，垃圾收集器遍历所有的根，构造出一个包含所有可达对象的图。例如，垃圾收集器可能会定位出一个引用着托管对象的全局变量。图 19.2 展示了一个分配有几个对象的托管堆，其中对象 A、C、D 和 F 为应用程序的根所直接引用。所有这些对象都是可达对象图的一部分。当对象 D 被添加到该图中时，垃圾收集器注意到它还引用着对象 H，于是对象 H 也被添加到该图中。垃圾收集器就这样以递归的方式来遍历应用程序中所有的可达对象。

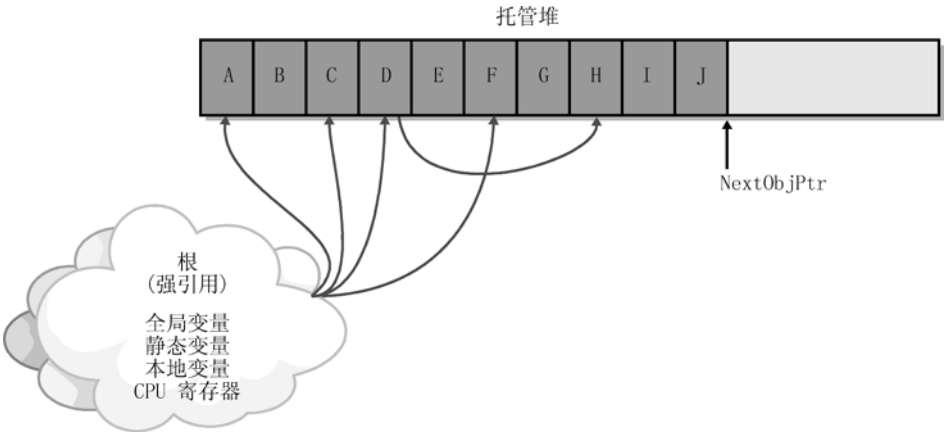

图 19.2　垃圾收集执行前的托管堆

一旦该部分的可达对象图完成以后，垃圾收集器将检查下一个根，并遍历其引用的对象。当垃圾收集器在对象之间进行遍历时，如果它试图将一个先前已经添加过的对象再一次添加到可达对象图中时，它会停止沿着该对象标识的路径方向上的遍历活动。这种行为有两个目的。首先，这可以避免垃圾收集器对一些对象执行多次遍历，这在客观上提高了性能。其次，如果对象之间出现了循环引用，这可以避免遍历陷入无限循环。

垃圾收集器一旦检查完所有的根，其得到的可达对象图将包含所有从应用程序的根可以访问的对象。任何不在该图中的对象都将是应用程序不可访问的对象，因此也是可以被执行垃圾收集的对象。垃圾收集器接着线性地遍历托管堆以寻找包含可收集垃圾对象的连续区块(这些区块现在被认为是自由空间)。一些容量较小的内存块将被垃圾收集器忽略不计。

如果找到了较大的连续区块，垃圾收集器将会把内存中的一些非垃圾对象搬移到这些连续区块中(使用大家非常熟悉的 memcpy 函数)以压缩托管堆。显然，搬移内存中的对象将使所有指向这些对象的指针变得无效。所以垃圾收集器必须修改应用程序的根以使它们指向这些对象更新后的位置。另外，如果任何对象包含有指向这些对象的指针，那么垃圾收集器也会负责矫正它们。在托管堆中的内存被压缩之后，托管堆上的 NextObjPtr 指针将被设为指向最后一个非垃圾对象之后。图 19.3 演示了垃圾收集执行后的托管堆。

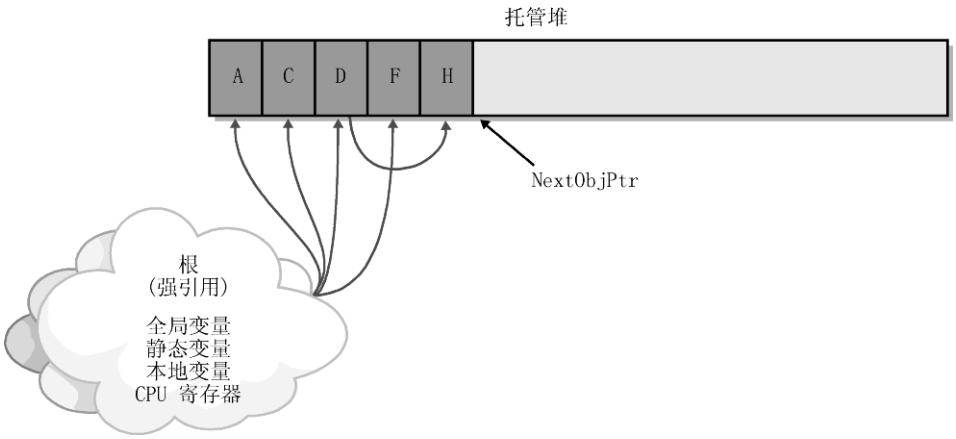

图 19.3 垃圾收集执行后的托管堆

如我们所见，垃圾收集会给应用程序带来不小的性能损伤，这也是使用托管堆时主要的负面影响。但是，我们要清楚垃圾收集只有在第 0 代对象充满时才会出现。在这之前，托管堆的性能要比 C 语言运行时中的堆的性能高许多。最后，CLR 的垃圾收集器还提供了一些特殊的优化设计可以大幅度地提高垃圾收集的性能。本章稍后将在 19.7 和 19.9 两节中讨论这些优化设计。

作为应用程序开发人员，大家应该从上面的讨论中得出以下两点重要的认识。首先，我们不必再自己实现代码来管理应用程序中对象的生存期。其次，本章开始描述的两种 bug 将不复存在。因为任何不可从应用程序的根中访问的对象都会在某个时刻被收集，所以应用程序将不可能再发生内存泄漏的情况。另外，应用程序也不可能再访问已经被释放的对象。因为如果对象可达，它将不可能被释放；而如果对象不可达，应用程序必将无法访问到它。

下面的代码演示了垃圾收集器是怎样分配和管理对象的。

```
class App {
    static void Main() {

        // 在托管堆中创建一个 ArrayList 对象，a 现在就是一个根
        ArrayList a = new ArrayList();

        // 在托管堆中创建 10,000 个对象
        for (Int32 x = 0; x < 10000; x++) {
            a.Add(new Object());    // 对象被创建在托管堆中
        }
        // 现在 a 是一个根(位于线程的堆栈上)。所以 a 是一
        // 个可达对象，a 引用的 10,000 个对象也是可达对象
        Console.WriteLine(a.Count);

        // 在 a.Count 返回后，a 便不再被 Main 中的代码所引用，
        // 因此也就不再是一个根。如果另一个线程在 a.Count
        // 的结果被传递给 WriteLine 之前启动了垃圾收集，那
        // 么这 10001 个对象的内存将被回收。
        // (译注：这 10001 个对象包括 a 引用的一个对象和其内
        // 引用的 10000 个对象。变量 x 虽然在此后的代码中也不
        // 再被引用，但由于它是一个值类型，并不存在于托管堆
        // 中，所以它不受垃圾收集器的管理，它在 Main 方法执
        // 行完毕后会随着堆栈的消失而自动被系统回收。)
        Console.WriteLine("End of method");
    }
}
```

注意　如果垃圾收集的功能是如此强大，大家可能会奇怪它为什么没有被 ANSI C++所采用。这是因为垃圾收集器必须能够识别出应用程序的根，并且还要找到所有的对象指针。非托管 C++的问题在于它允许我们将一个指针从一个类型转换为另一个类型，而我们无从知道它真正引用的对象是什么。但在 CLR 中，托管堆总能知道一个对象的实际类型，从而使用其元数据信息来判断一个对象的哪些成员引用着其他的对象。

19.3　终止化操作

到目前为止，大家应该已经对垃圾收集和托管堆的情况(包括垃圾收集器怎样回收一个对象的内存)有了一个基本的了解。大多数类型仅需要内存即可进行操作.

例如，Int32、Point、Rectangle、String 以及 ArrayList 等这些类型只需要操作内存中的字节数据。但是，有些类型要发挥作用仅有内存是不够的。

例如，System.IO.FileStream 类型就需要打开一个文件，并保存文件的句柄。然后该类型的 Read 和 Write 方法才能使用该句柄来操作文件。类似地，System.Threading.Mutex 类型也需要打开一个 Windows 互斥体内核对象，并保存其句柄以供随后 Mutex 的方法被调用时使用。

任何封装了非托管资源的类型，例如文件、网络链接、套接字、互斥体等，都必须支持一种称作**终止化**(finalization)的操作。终止化操作允许一种资源在它所占用的内存被回收之前首先执行一些清理工作。要提供终止化操作，我们必须为类型实现一个名为 Finalize 的方法。当垃圾收集器判定一个对象为可收集的垃圾时，它便会调用该对象的 Finalize 方法(如果存在的话)。Finalize 方法的实现通常便是调用 CloseHandle 函数(译注：CloseHandle 为一个 Win32 函数，其主要用于关闭一个打开的对象句柄)，该函数接受非托管资源的句柄作为参数。因为 FileStream 类型定义了一个 Finalize 方法(译注：这里准确地讲应该为“重写了 System.Object 的 Finalize 方法”，因为 System.Object 中已经定义了一个 Finalize 虚方法，任何需要释放非托管资源的类型都要重写这个 Finalize 方法。如果一个类型以及其所有的基类型都没有重写 Object 的 Finalize 方法，那么垃圾收集器会认为该类型不需要终止化操作，它不会调用 Object 的 Finalize 方法。本章后面会多次提到“某某类型定义了一个 Finalize 方法”，如未做特殊说明，大家应该将其理解为“某某类型或者是其某个基类型重写了 System.Object 的 Finalize 方法”)，所以每一个 FileStream 对象都会保证在其内存被回收时能够释放它所占用的非托管资源。如果一个封装了非托管资源的类型没有定义 Finalize 方法，那么这些非托管资源将得不到关闭，从而会导致某种程度的资源泄漏(译注：这里的前提是我们没有显式关闭对象所封装的非托管资源)。直到进程结束，这些非托管资源才会被操作系统回收。

下面的 OSHandle 类型演示了如何定义一个封装着非托管资源的类型。当垃圾收集器判定一个 OSHandle 对象是可收集的垃圾时，它会调用其上的 Finalize 方法，Finalize 方法在内部调用了 Win32 函数 CloseHandle，从而确保非托管资源得到了释放。OSHandle 类可以用于任何使用 CloseHandle 函数释放的非托管资源。如果大家处理的非托管资源需要其他不同的清理函数，则可以在下面 Finalize 方法实现的基础上做相应的改动。

```
public sealed class OSHandle {

   // 下面的字段保存着一个非托管资源的 Win32 句柄
   private IntPtr handle;

   // 下面的构造器用于初始化 handle 句柄
   public OSHandle(IntPtr handle) {
      this.handle = handle;
   }
```

```
    // 当垃圾收集执行时，下面的 Finalize 方法将被调用，
    // 它将关闭非托管资源句柄
    protected override void Finalize() {
        try {
            CloseHandle(handle);
        }
        finally {
            base.Finalize();
        }
    }

    // 下面的公有方法用于返回所封装的 handle 句柄
    public IntPtr ToHandle() { return handle; }

    // 下面的公有隐式转型操作符也用于返回所封装的 handle 句柄
    public static implicit operator IntPtr(OSHandle osHandle) {
        return osHandle.ToHandle();
    }

    // 下面的私有方法用于释放非托管资源
    [System.Runtime.InteropServices.DllImport("Kernel32")]
    private extern static Boolean CloseHandle(IntPtr handle);
}
```

当 OSHandle 对象被执行垃圾收集时，垃圾收集器会调用它的 Finalize 方法。该方法首先执行必要的清理操作，然后调用基类型的 Finalize 方法以使基类型也能够有机会释放其内的非托管资源。对基类型的 Finalize 方法的调用被放在一个 finally 块中，这可以确保即使在 OSHandle 的清理代码抛出异常的情况下它也能够得到调用。在上面的例子中，System.Object 的 Finalize 方法将被调用。由于 Object 的 Finalize 方法并没有任何实现代码，所以我们可以去掉上面代码中的异常处理和对 base.Finalize 的调用，这可以在不失正确性的前提下提高代码的性能。

然而，C#编译器实际上并不会编译上面的代码。C#编译器组发现许多开发人员对 Finalize 方法的实现都不够正确。具体而言，许多开发人员都会忘记在 Finalize 方法中对异常情况进行处理、以及调用基类型的 Finalize 方法。为了减轻开发人员的负担，C#为定义 Finalize 方法提供了特殊的语法。下面的 C#代码所做的工作实际上和上面的代码是等同的，只是这段代码会成功通过编译，因为它使用了 C#提供的特殊语法来定义 Finalize 方法。

```
public sealed class OSHandle {

   // 下面的字段保存着一个非托管资源的 Win32 句柄
   private IntPtr handle;

   // 下面的构造器用于初始化 handle 句柄
   public OSHandle(IntPtr handle) {
      this.handle = handle;
   }

   // 当垃圾收集执行时，下面的析构器(Finalize)方法将被
   // 调用，它将关闭非托管资源句柄
   ~OSHandle() {
      CloseHandle(handle);
   }

   // 下面的公有方法用于返回所封装的 handle 句柄
   public IntPtr ToHandle() { return handle; }

   // 下面的公有隐式转型操作符也用于返回所封装的 handle 句柄
   public static implicit operator IntPtr(OSHandle osHandle) {
      return osHandle.ToHandle();
   }

   // 下面的私有方法用于释放非托管资源
   [System.Runtime.InteropServices.DllImport("Kernel32")]
   private extern static Boolean CloseHandle(IntPtr handle);
}
```

编译上面的代码，并用 ILDasm.exe 查看生成的程序集，我们将会看到 C#编译器实际上在托管模块的元数据中产生了一个名为 Finalize 的方法。如果我们再查看 Finalize 方法的 IL 代码，我们会看到 C#编译器将 CloseHandle 函数调用放在了一个 try 块内，同时又在和该 try 块相匹配的 finally 块中添加了一个 base.Finalize 调用。

要创建一个 OSHandle 的对象实例，我们首先必须调用一个 Win32 函数来返回一个非托管资源的句柄，这样的函数有 CreateFile、CreateMutex、CreateSemaphore、CreateEvent、socket、CreateFileMapping，等等。然后，我们便可以使用 C#的 new 操作符来构造一个 OSHandle 的实例(将上面得到的非托管资源句柄作为构造器参数)。

这样当未来某个时刻垃圾收集器判定该对象为可收集的垃圾时，它会看到该类型定义有一个 Finalize 方法，于是它便会调用该方法，从而允许 CloseHandle 函数来关闭其中的非托管资源。在 Finalize 方法返回之后的某个时刻，该 OSHandle 对象在托管堆中所占的内存才会被回收。

重要　如果大家熟悉 C++，大家会注意到 C#定义 Finalize 方法的语法非常类似于 C++中定义析构器的语法。实际上 C#语言规范中甚至就将该方法称为析构器。但是，Finalize 方法的工作原理和非托管 C++中析构器的工作原理完全不同。

我个人认为 C#编译器组选择模仿 C++析构器的语法，并将该方法称为析构器是一个错误的决定。因为这会搞乱那些从 C++背景转向 C#和.NET 框架的开发人员。这些开发人员可能会错误地认为使用 C#的析构器语法意味着类型实例就可以被确定性地析构（就象 C++中的一样）。但是 CLR 不支持确定性的析构，因此 C#也不能提供这种机制。实际上，没有哪个支持 CLR 的编程语言可以提供这样的能力。即使我们使用托管扩展 C++来实现一个托管类型，定义一个“析构器”也会导致编译器产生一个只有在垃圾收集执行时才能被调用的 Finalize 方法。

不要被一个语言的析构器语法所迷惑——Finalize 方法只有在垃圾收集执行时才会被调用。它不会在一个方法退出、或者对象超出作用范围时被调用。

在设计一个类型时，我们应该尽可能地避免使用 Finalize 方法，原因如下：

- 终止化对象(译注：终止化对象是指那些类型本身，或者其基类型重写了 System.Object 的 Finalize 方法的对象。反之，非终止化对象是指那些类型本身及其所有的基类型都没有重写 System.Object 的 Finalize 方法的对象)的代龄会被提升，这会增加内存的压力，并会在垃圾收集器判定对象为可收集的垃圾时阻止回收对象的内存。另外，所有被终止化对象直接引用或者间接引用的对象的代龄也将被提升。(本章稍后将讨论对象的代龄和代龄的提升等相关问题。)

- 终止化对象的分配花费的时间较长，因为指向它们的指针必须被放在终止化链表上(这将在 19.3.2 节中阐述)。

- 强制垃圾收集器执行 Finalize 方法会极大地损伤应用程序的性能。记住，每个对象都会被终止化。所以，如果我们有一个含有 10,000 个对象的数组，那么每个对象的 Finalize 方法都必须被调用。

- 一个终止化对象可能会引用其他的对象(包括终止化对象和非终止化对象)，从而不必要地延长它们的生存期。实际上，我们可以考虑将一个类型拆分成两个不同的类型，一个是包含 Finalize 方法的轻量级类型，其中不要引用任何其他的对象(就象前面展示的 OSHandle 类型一样)。另一个是不包含 Finalize 方法的类型，其中可以引用任何其他对象。

- 我们并不能控制 Finalize 方法何时执行。对象可能会一直占有着非托管资源，直到出现垃圾收集。

- CLR 不对 Finalize 方法的执行顺序做任何保证。例如，假设一个对象中包含着一个指向另一个对象(下面称为内部对象)的指针。现在，垃圾收集器检测到两个对象都是可被收集的垃圾，假设内部对象的 Finalize 方法首先被调用。让我们再假设外部对象的 Finalize 方法要访问内部对象，并调用其上的方法。但是内部对象已经执行了终止化操作，因此结果将变得不可预期。出于这种原因，微软强烈建议大家不要在 Finalize 方法中访问任何内部成员对象。本章稍后谈论的 Dispose 模式会为我们提供另一种清理资源的方式，它就不会受到这里谈到的约束。

如果我们决定为自己的类型实现 Finalize 方法，我们要确保其中的代码能够尽可能快地执行，而避免那些有可能阻塞 Finalize 方法的行为，包括任何线程同步操作。另外，如果有任何异常在 Finalize 方法中未经捕获而逃脱，那么 CLR 会忽略它，并继续调用其他对象的 Finalize 方法。

注意 到目前为止，实现一个 Finalize 方法最常见的原因便是释放对象所占有的非托管资源。实际上，在终止化操作中，我们应该避免编写代码访问其他的托管对象或者托管静态方法。避免访问其他托管对象的原因是这些对象的类型也可能实现了 Finalize 方法，而它们有可能首先被调用，从而将这些对象置于一个不可预期的状态。避免调用托管静态方法的原因是这些方法内部可能会访问已经执行了终止化操作的对象，从而也会把这些方法置于一种不可预期的状态。

Finalize 方法也可以用于其他目的。下面演示的类会在每次执行垃圾收集时使计算机发出“嘟嘟”的响声。

```
public sealed class GCBeep {
    ~GCBeep() {
        // 进入终止化操作，使计算机发出响声
        MessageBeep(-1);

        // 如果应用程序域(AppDomain)不是正在执行卸载，
        // 那么创建一个新的对象以备下一次垃圾收集时执行
        // 终止化，本章下一节将讨论 IsFinalizingForUnload
        if (!AppDomain.CurrentDomain.IsFinalizingForUnload())
            new GCBeep();
    }

    [System.Runtime.InteropServices.DllImport(“User32.dll”)]
    private extern static Boolean MessageBeep(Int32 uType);
}
```

要使用上面的类，我们只需要构造一个 GCBeep 类的实例。然后不管何时出现垃圾收集，GCBeep 对象的 Finalize 方法都会被调用，Finalize 方法内部在调用 MessageBeep 之后会构造一个新的 GCBeep 对象。这样当下一次垃圾收集出现时，新构造的 GCBeep 对象的 Finalize 方法又会被调用。下面的例子演示了 GCBeep 类的使用方法。

```
class App {
    static void Main() {
        // 构造一个 GCBeep 对象以使每次执行
        // 垃圾收集时计算机都能发出一次响声
        new GCBeep();

        // 构造多个含有 100 个字节的对象
        for (Int32 x = 0; x < 10000; x++) {
            Console.WriteLine(x);
            Byte[] b = new Byte[100];
        }
    }
}
```

另外需要注意的是即使一个类型的实例构造器抛出了异常，它的 Finalize 方法也会被调用。所以我们实现的 Finalize 方法不应想当然地认为对象会处在一个良好的、一致的状态。下面的代码演示了这一点。

```
class TempFile {
    String filename = null;
    public FileStream fs;

    public TempFile(String filename) {
        // 下面一行代码可能会抛出异常
        fs = new FileStream(filename, FileMode.Create);

        // 保存文件名称
        this.filename = filename;
    }

    ~TempFile() {
        // 因为我们并不能确信 filename 在构造器中是否完成了初始
        // 化，所以这里正确的做法是首先测试 filename 是否为 null
        if (filename != null)
            File.Delete(filename);
    }
}
```

作为选择，我们也可以用如下的代码来实现同样的功能。

```
class TempFile {
    String filename;
    public FileStream fs;

    public TempFile(String filename) {
        try {
            // 下面一行代码可能会抛出异常
            fs = new FileStream(filename, FileMode.Create);

            // 保存文件名称
            this.filename = filename;
        }
        catch {
            // 如果出现了问题，告诉垃圾收集器不要调用 Finalize
            // 方法。本章稍后会讨论 SuppressFinalize 方法
            GC.SuppressFinalize(this);

            // 告诉调用者发生了异常
            throw;
        }
    }

    ~TempFile() {
        // 没有必要再用 if 语句，因为这段代码只会在构造器运行
        // 成功的情况下才会执行
        File.Delete(filename);
    }
}
```

19.3.1　调用 Finalize 方法的条件

有 4 种事件会导致一个对象的 Finalize 方法被调用：

- **第 0 代对象充满**　该事件是目前导致 Finalize 方法被调用的最常见的一种方式。该事件通常在应用程序代码运行过程中分配新对象的时候发生。
- **显式调用 System.GC 的静态方法 Collect**　我们的代码可以显式地请求 CLR 执行垃圾收集。虽然微软强烈建议不要这样做，但某些时候强制执行垃圾收集对应用程序来说还是有意义的。(本章稍后将探讨这一点。)
- **CLR 卸载应用程序域**　当一个应用程序域(AppDomain)卸载时，CLR 会认为该应用程序域中不存在任何根，因此会调用该应用程序域中创建的所有对象(译注：准确地讲应该是“所有的终止化对象”)上的 Finalize 方法。本书第 20 章将讨论应用程序域。
- **CLR 被关闭**　当一个进程正常中断时，它会试图关闭 CLR。这时 CLR 会认为该进程中不存在任何根，因此会调用托管堆中所有对象(译注：同样应该是“所有的终止化对象”)上的 Finalize 方法。

CLR 使用一个特殊的专用线程来调用 Finalize 方法。对于上面列出的第 1 种、第 2 种和第 3 种事件来说，如果有 Finalize 方法进入了一个无限循环，那么这个特殊的线程将被阻塞，其他的 Finalize 方法将得不到调用。这种情况非常糟糕，因为应用程序将不能够再回收其他终止化对象占用的内存——只要应用程序还在运行，它就存在泄漏内存的可能。

对于第 4 种事件来说，每个 Finalize 方法会有大约 2 秒钟的运行时间。如果一个 Finalize 方法没有在 2 秒钟内返回，那么 CLR 将中断该进程——其他的 Finalize 方法将得不到调用。另外，如果调用所有对象的 Finalize 方法超过了 40 秒钟，那么 CLR 也会中断该进程。

> **注意**　这些为超时所设定的值在本书写作的时候是正确的，但是微软可能会在将来改变它们。

Finalize 方法中的代码也可能会构造新的对象。如果这发生在 CLR 关闭期间，CLR 会继续收集对象，并调用它们的 Finalize 方法，直到不再存在对象、或者它们的执行时间超过 40 秒钟为止。

回顾本章前面的 GCBeep 类型。如果一个 GCBeep 对象由于第 1 种或第 2 种事件被执行了终止化，那么将有一个新的 GCBeep 对象被构造。因为应用程序会继续运行，后面还会有更多的垃圾收集出现，所以这种做法没什么问题。但是，如果一个 GCBeep 对象由于第 3 种或第 4 种事件被执行了终止化，那么将不会有新的 GCBeep 对象被构造，因为这时应用程序域正在执行卸载、或者 CLR 正在关闭。如果在这时构造了这些新的对象，那么 CLR 将面临许多无用的工作，因为它还要调用它们的 Finalize 方法。

为了阻止构造新的 GCBeep 对象，GCBeep 的 Finalize 方法中调用了 AppDomain 的 IsFinalizingForUnload 方法。如果 GCBeep 对象的 Finalize 方法是由于应用程序域卸载而被调用的，那么该方法将返回 true。这种解决方案对于应用程序域卸载是可行的，但如果是 CLR 正在关闭呢？

为了获知 CLR 是否正在关闭，微软为 System.Environment 类添加了一个 HasShutdownStarted 只读属性。GCBeep 的 Finalize 方法应该读取该属性，如果其返回的值为 true，那么就不应该再构造新的对象。不幸的是，GCBeep 的 Finalize 方法并没有这样做，为什么呢？这是因为该属性根本无法访问！微软的开发人员在这里犯了一个错误，他们把 HasShutdownStarted 属性实现成了一个实例属性，但是 Environment 类却只有一个私有构造器，因此我们没有办法创建它的实例，从而也就没有办法访问该属性——这实在是太糟糕了！希望微软能够在.NET 框架类库(FCL)的后续版本中修复这一 bug。

目前我们还没有办法解决这一问题。GCBeep 的 Finalize 方法仍然会在 CLR 关闭的时候构造新的对象。新构造的对象继续被执行终止化，再构造新的对象……直到过了 40 秒钟后，CLR 才会放弃这样的操作并中断进程。

19.3.2 终止化操作的内部机理

从表面上来看，终止化操作好像很简单。我们创建一个对象，当该对象被执行垃圾收集时，垃圾收集器会自动调用它的 Finalize 方法。但是一旦深入研究下去，大家会发现终止化操作远非这么简单。

当应用程序创建一个新对象时，new 操作符会为对象从托管堆上分配内存。如果该对象的类型定义了 Finalize 方法，那么在该类型的实例构造器被调用之前，指向该对象的一个指针将被放到一个称作终止化链表(finalization list)的数据结构里面。终止化链表是一个由垃圾收集器控制的内部数据结构。链表上的每一个条目都引用着一个对象，这实际上是在告诉垃圾收集器在回收这些对象的内存之前要首先调用它们的 Finalize 方法。

图 19.4 展示了一个托管堆，其中包含着几个对象。它们中有些是从应用程序的根可达的对象，有些则不是。当对象 C、E、F、I 和 J 被创建时，系统会检测到这些对象的类型定义了 Finalize 方法，于是便会将指向这些对象的指针添加到终止化链表中。

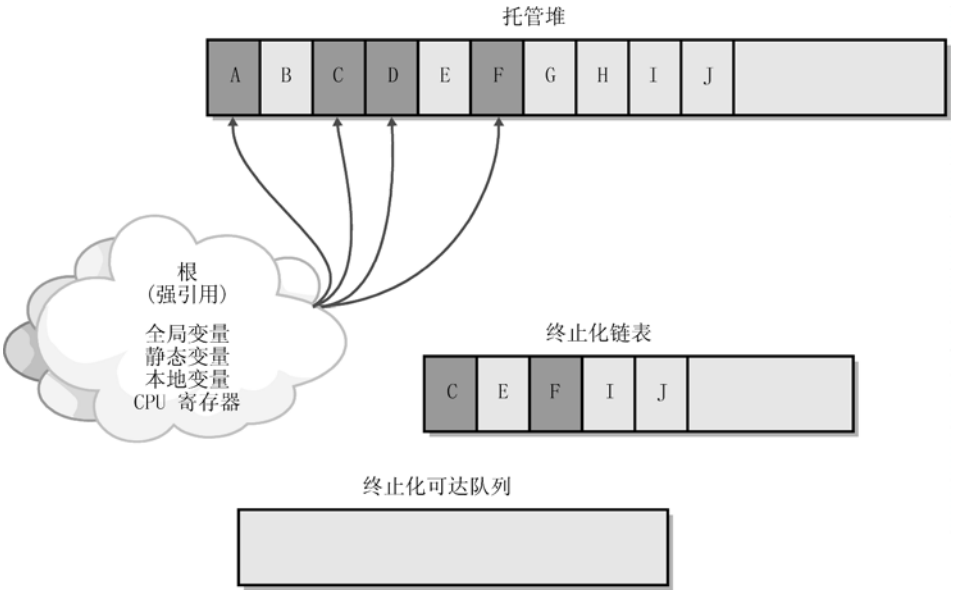

图 19.4　终止化链表上含有一些指针

> **注意**　虽然 System.Object 定义了一个 Finalize 方法，但是 CLR 会忽略它。也就是说，当构造一个类型的实例时，如果该类型的 Finalize 方法是从 System.Object 继承的，那么该对象将不被认为是终止化对象。一个对象要成为终止化对象，那么在它的类型及其基类型中（除 Object 之外），必须至少有一个重写 Object 的 Finalize 方法。

当垃圾收集开始时，对象 B、E、G、H、I 和 J 将被判定为可收集的垃圾。垃圾收集器然后扫描终止化链表以查找其中是否有指向这些对象的指针。当找到这样的指针时，它们会被从终止化链表上移除，并添加到一个称作终止化可达队列(freachable queue，其英文名称中的 freachable 可念作“F-reachable”)的数据结构上。终止化可达队列是另一个由垃圾收集器控制的内部数据结构。在终止化可达队列中出现的对象表示该对象的 Finalize 方法即将被调用。当垃圾收集执行完毕后，托管堆的情况将如图 19.5 所示。

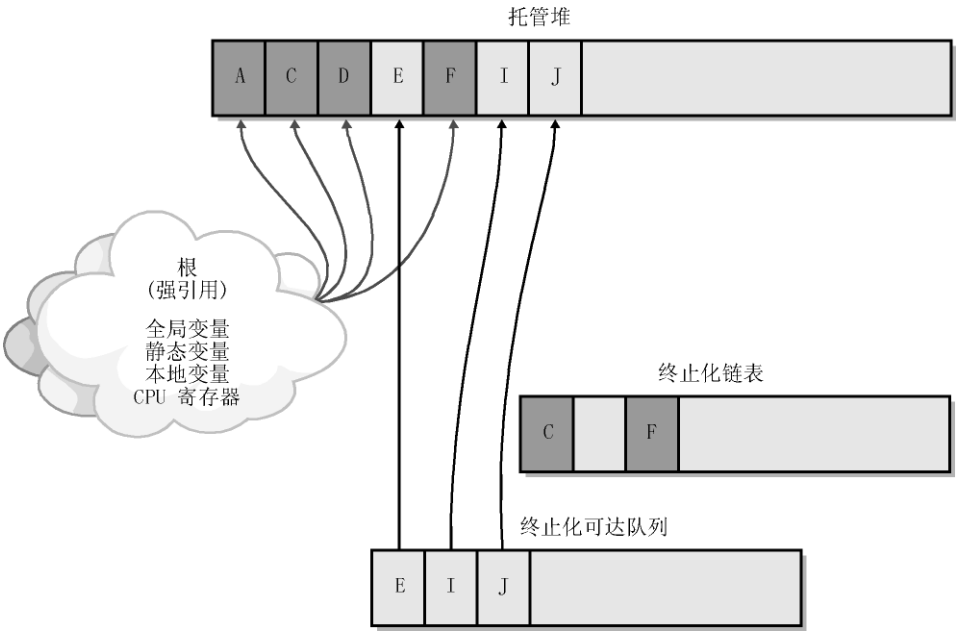

图 19.5 终止化链表中的一些指针已经转移到终止化可达队列中

在上面的图中，我们看到对象 B、G、和 H 占用的内存已经被回收了，因为它们没有 Finalize 方法需要调用。但是对象 E、I 和 J 占用的内存却不能被回收，因为它们的 Finalize 方法还没有被调用。

CLR 中有一个特殊的高优先级的线程专门用于调用 Finalize 方法。使用专用的线程可以避免潜在的线程同步问题。如果使用的是应用程序的一个线程，线程同步问题就有可能造成一些麻烦。当终止化可达队列为空时(通常的情况)，该线程将处于睡眠状态。但当终止化可达队列中有条目出现时，该线程将被唤醒，开始把每个条目从终止化可达队列中移除，并调用每个对象的 Finalize 方法。因为该线程特殊的工作方式，我们在 Finalize 方法中的代码不应该对正在执行的线程做任何假设。例如，我们应该避免在 Finalize 方法中访问线程本地存储数据(Thread Local Storage，简称 TLS)。

终止化链表和终止化可达队列之间的交互非常有意思。首先，我们来了解一下终止化可达队列这一名称的由来。其中“freachable”的“f”显然代表“终止化”(finalization)，其含义为终止化可达队列中每一个条目的 Finalize 方法都应该被调用。而“reachable”部分又意味着对象是可达的。换一种方式来看，我们可以把终止化可达队列视作和全局变量、静态变量一样的根。所以如果一个对象位于终止化可达队列上，那么该对象将是一个可达对象，因此也就不是一个可被收集的垃圾对象。

简而言之，如果一个对象是不可达的，那么垃圾收集器将把它视作可收集的垃圾。当垃圾收集器将一个对象从终止化链表转移到终止化可达队列上时，该对象将不再被认为是可收集的垃圾对象，它的内存也就不可能被回收。到此为止，垃圾收集器已经完成了垃圾对象的鉴别工作。一些原先被认为是垃圾的对象现在又被认为不是垃圾，从某种意义上讲，对象又“复苏”了。当第一次垃圾收集执行完毕后，特殊的 CLR 线程将会清空终止化可达队列中的对象，同时执行其中每个对象的 Finalize 方法。

等下一次垃圾收集执行时，它会看到终止化对象已经成为真正的垃圾对象，因为应用程序的根不再指向它，终止化可达队列也不再指向它。这时对象的内存才会被回收。在整个过程中，我们需要认识到的最重要的一点就是终止化对象需要执行两次垃圾收集才能释放掉它所占用的内存。实际上，由于对象的代龄可能会被提升，所以释放一个对象占用的内存所需要的垃圾收集的次数可能会超过两次，本章稍后将谈论这一问题。图 19.6 展示了第 2 次垃圾收集执行后托管堆的情况。

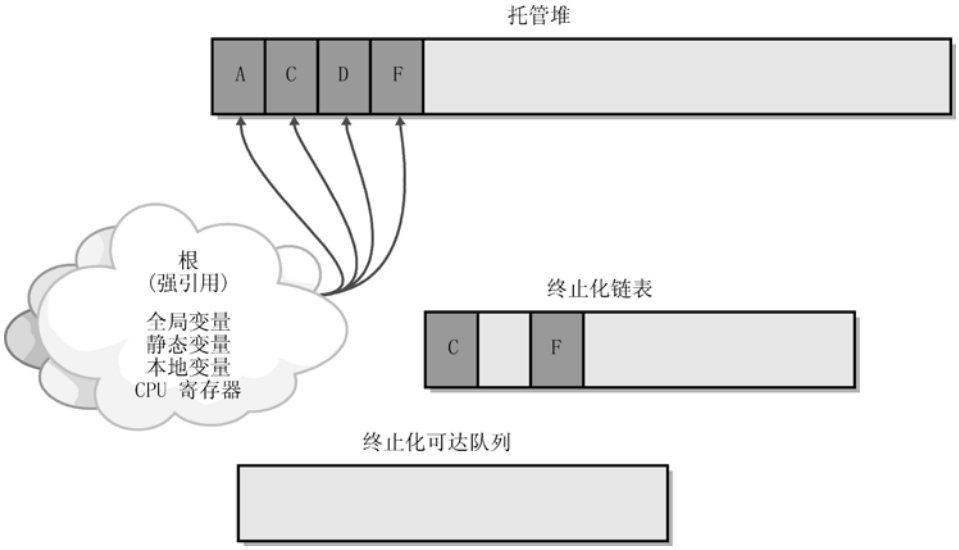

图 19.6　第 2 次垃圾收集执行后托管堆的情况

19.4　Dispose 模式：强制对象清理资源

Finalize 方法非常有用，因为它可以确保托管对象在释放内存的同时不会泄漏非托管资源。但是 Finalize 方法的问题在于我们并不能确定它会在何时被调用，而且由于它并不是一个公有方法，我们也不能显式地调用它。

当我们使用互斥体和位图这样的非托管资源时，能够显式地释放或者关闭对象通常是很有用的。使用文件和数据库连接更是如此。例如，我们可能希望打开一个数据库连接，查询一些记录，然后再关闭该数据库连接——我们并不希望让该数据库连接一直打开直到下一次垃圾收集出现，因为下一次垃圾收集完全有可能在我们获取数据库记录后的几个小时、甚至几天后才出现。

要提供显式释放或者关闭对象的能力，一个类型通常要实现一种被称为 Dispose 的模式。Dispose 模式定义了开发人员在实现类型的显式资源清理功能时所要遵循的一些约定。如果一个类型实现了 Dispose 模式，使用该类型的开发人员将能够知道当对象不再被使用时如何显式地释放掉它所占用的资源。

注意 所有定义了 Finalize 方法的类型都应该实现本节所描述的 Dispose 模式以给用户更多的控制权。但是，一个类型也可以实现 Dispose 模式，而不用定义 Finalize 方法。例如，System.IO.BinaryWriter 就是这样的类型。本章后面 19.4.3 一节将解释其中的原因。

前面曾向大家演示了一个 OSHandle 类型，其中就实现了一个 Finalize 方法。这样当 OSHandle 对象被执行垃圾收集时，其封装的非托管资源将自动被关闭。但使用 OSHandle 对象的开发人员却无法显式关闭其中的非托管资源。

下面的代码演示了一个新版的 OSHandle 实现，其中就应用了 Dispose 模式。

```
using System;

// 实现 IDisposable 接口以表明该类提供了 Dispose 模式
public sealed class OSHandle : IDisposable {

   // 下面的字段保存着一个非托管资源的 Win32 句柄
   private IntPtr handle;
```

```
// 下面的构造器用于初始化 handle 句柄
public OSHandle(IntPtr handle) {
    this.handle = handle;
}

// 当垃圾收集执行时，下面的 Finalize 方法将被调用，
// 它将关闭非托管资源句柄
~OSHandle() {
    Dispose(false);
}

// 下面的公有方法可以被显式调用来关闭非托管
// 资源句柄
public void Dispose() {
    // 因为对象的资源被显式清理，所以在这里阻止
    // 垃圾收集器调用 Finalize 方法
    GC.SuppressFinalize(this);

    // 进行实际的资源清理工作
    Dispose(true);
}

// 下面的公有方法可以用来替换 Dispose 方法
public void Close() {
    Dispose();
}

// 下面的方法用于进行实际的清理工作。Finalize、Dispose
// 和 Close 都要调用该方法。因为该类是一个密封类，所以
// 该方法被定义为私有方法。如果该类不是一个密封类，
// 那么该方法应该是一个受保护方法
private void Dispose(Boolean disposing) {
    // 同步那些同时调用 Dispose/Close 方法的线程
    lock (this) {
        if (disposing) {
            // 对象正在被显式释放/关闭，而非执
            // 行终止化。因此在该 if 语句中访问
            // 那些引用其他对象的字段是安全的，
            // 因为这些对象的 Finalize 方法还没有
            // 被调用

            // 对于 OSHandle 来说这里没什么可做的
        }

        // 对象正在被释放/关闭、或者被执行终止化
        if (IsValid) {
            // 如果 handle 有效，那么关闭非托管资源
```

```
                // 注意：将 CloseHandle 替换成大家自己
                // 关闭非托管资源的函数
                CloseHandle(handle);

                // 将 handle 字段设置为某个标记值用来防
                // 止多次调用 CloseHandle
                handle = InvalidHandle;
            }
        }
    }

    // 下面的公有属性用于返回一个无效的句柄值。注意：使
    // 该属性返回一个表示正在使用的非托管资源无效的值
    public IntPtr InvalidHandle { get { return IntPtr.Zero; } }

    // 下面的公有方法用于返回所封装的 handle 句柄
    public IntPtr ToHandle() { return handle; }

    // 下面的公有隐式转型操作符也用于返回所封装的 handle 句柄
    public static implicit operator IntPtr(OSHandle osHandle) {
        return osHandle.ToHandle();
    }

    // 下面的公有属性用于表示所封装的句柄是否有效
    public Boolean IsValid { get { return (handle != InvalidHandle); } }
    public Boolean IsInvalid { get { return !IsValid; } }

    // 下面的私有方法用于释放非托管资源
    [System.Runtime.InteropServices.DllImport("Kernel32")]
    private extern static Boolean CloseHandle(IntPtr handle);
}
```

上面的 OSHandle 实现看起来有些琐碎，但实际却不然，其中有许多代码是用来支持 Dispose 模式的。虽然该类只封装了一个简单的非托管资源，但是它却几乎可以用于任何场合。如果大家需要设计一个封装有非托管资源的类型，完全可以将上面的代码拷贝到自己的项目中并做适当的修改即可使用。实际上，如果微软能够将上面的 OSHandle 类引入到 FCL 中就更好了。

下面对新版的 OSHandle 实现代码做一解释。首先，OSHandle 类实现了 System.IDisposable 接口。该接口在 FCL 中定义如下：

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

任何实现了该接口的类型都是在声明自己遵循了 Dispose 模式。简而言之，这意味着该类型提供了一个公有、无参的 Dispose 方法。我们可以通过显式调用 Dispose 方法来释放对象所封装的非托管资源。注意调用该方法并不会释放对象在托管堆中占用的内存，释放对象内存的工作仍由垃圾收集器负责，而且释放的时间仍不确定。

注意　大家可能注意到了 OSHandle 类型还提供了一个公有的 Close 方法。该方法只是调用了一下 Dispose 方法。一些遵循 Dispose 模式的类型为方便起见大都提供了 Close 方法，但这对于 Dispose 模式来说并非必须。例如，System.IO.FileStream 类在遵循 Dispose 模式的同时也提供了一个 Close 方法，这是因为很多开发人员都认为“关闭(close)”一个文件要比“释放(dispose)”一个文件讲起来更自然一些。但是，System.Threading.Timer 类就没有提供 Close 方法，虽然它也实现了 Dispose 模式。

无参的 Dispose 方法和 Close 方法都应该是公有的非虚方法。但是，因为 Dispose 方法是在实现一个接口的方法，所以它默认情况下是一个虚方法。所以最好的做法是将 Dispose 方法定义为密封方法。幸运的是，当我们在 C#中实现一个接口方法时，编译器将默认该虚方法为公有密封方法。但是如果大家使用的是另外不同的编程语言，则应该确保 Dispose 方法被定义为密封方法。

现在我们有了 3 种方法可以用来清理 OSHandle 对象封装的非托管资源：第 1 种是显式调用 Dispose 方法，第 2 种是显式调用 Close 方法，第 3 种是让垃圾收集器调用 Finalize 方法。不管选用哪一种方法，清理代码都是一样的，所以我们将其放在一个单独的、私有非虚 Dispose 方法中，该方法接受一个 Boolean 类型的参数 disposing。

带 Boolean 参数的 Dispose 方法中包含着实际进行非托管资源清理工作的代码。在 OSHandle 例子中，该方法只是简单地调用了一下 CloseHandle 函数，并将 handle 字段设置为一个无效的句柄。将 handle 字段设置为一个无效的句柄可以确保它所表示的句柄不至于被多次关闭，因为无参的 Dispose 方法和 Close 方法都是公有方法，应用程序有可能多次调用它们。另外，由于多个线程可能会同时调用这些方法，所以带 Boolean 参数的 Dispose 方法中使用了 C#的 lock 语句来确保该方法中的代码是线程安全的。

当一个 OSHandle 对象的 Finalize 方法被调用时，Dispose 方法的 disposing 参数将被设为 false。这将告诉 Dispose 方法它不应该执行任何引用其他托管对象(译注：这里的托管对象准确地讲应该为“终止化对象”)的代码。

假设 CLR 正在关闭，而在一个 Finalize 方法中我们却试图往一个 FileStream 中写入数据。由于 FileStream 的 Finalize 方法可能已经被调用了，所以这种操作就有可能失败。

另一方面，当调用无参的 Dispose 方法和 Close 方法时，disposing 参数将被设为 true。这表示对象正在被显式执行资源清理，而非由垃圾收集器执行终止化。在这种情况下，Dispose 方法就可以执行引用其他对象(例如 FileStream)的代码。因为这时应用程序的逻辑是由我们控制，所以我们知道 FileStream 对象是否仍然还打开。

顺便提一句，如果 OSHandle 类型没有标识 sealed，那么带 Boolean 参数的 Dispose 方法应该被实现为一个受保护的虚方法，而不是一个私有的非虚方法。任何继承自 OSHandle 的类都可以重写该方法，但是不要去重写无参的 Dispose 方法和 Close 方法，也不要去重写 Finalize 方法，这 3 个方法应该直接被继承到派生类型中。

重要 我们需要清楚使用 Dispose 模式时可能出现的一些版本问题。如果第 1 个版本中的基类型没有实现 IDisposable 接口，那么它将不可以在后续的版本中再实现该接口。如果该基类型在后续的版本中添加了 IDisposable 接口，那么所有的派生类型将不会知道去调用基类型中的 Dispose 方法，因此基类型中封装的非托管资源将不能得到正确的清理。类似地，如果第 1 个版本中的基类型实现了 IDisposable 接口，那么它将不可以在后续的版本中再删除该接口，因为派生类型会试图去调用一个在基类型中不存在的方法。(译注: 这显然是接口实现本身所固有的困境，而非 Dispose 模式所专有，只是因为 Dispose 模式负有释放资源这样的重大责任而对这个问题比较敏感而已。)

实际上，如果派生类型本身没有资源清理的工作要做的话，那么它也不必再做任何与 Dispose 模式相关的事情。但是，如果派生类型需要进行一些资源清理工作的话，我们只需重写带 Boolean 参数的 Dispose 虚方法即可。在重写该方法时，我们首先要实现自身的资源清理逻辑，然后再去调用基类的带 Boolean 参数的 Dispose 方法。看下面的例子(假设 OSHandle 不是密封类)：

```
class SomeType : OSHandle {

   // 下面的字段保存着一个非托管资源的 Win32 句柄
   private IntPtr handle;

   protected override void Dispose(Boolean disposing) {
      // 同步那些同时调用 Dispose/Close 方法的线程
      lock (this) {
```

```
            try {
                if (disposing) {
                    // 对象正在被显式释放/关闭，而非执行终止化。
                    // 因此在该 if 语句中访问那些引用其他对象的
                    // 字段是安全的，因为这些对象的 Finalize，
                    // 方法还没有被调用。
                    // 对于 SomeType 类来说这里没什么可做的
                }
                // 对象正在被释放/关闭、或者被执行终止化
                if (IsValid) {
                    // 如果 handle 是有效的，那么关闭非托管资
                    // 源。注意：将 CloseHandle 替换成大家自
                    // 己关闭非托管资源的函数
                    CloseHandle(handle);
                    // 将 handle 字段设置为某个标记值用来防止
                    // 多次调用 CloseHandle
                    handle = InvalidHandle;
                }
            }
            finally {
                // 让基类执行自己的清理工作
                base.Dispose(disposing);
            }
        }
    }
}
```

另一个值得注意的地方是我们在无参的 Dispose 方法中调用了 GC 的静态方法 SuppressFinalize。这是因为如果使用 OSHandle 对象的代码显式调用了 Dispose 方法或 Close 方法，那么该对象的 Finalize 方法就不应该再被执行。如果再执行 Finalize 方法，CloseHandle 函数将被多次调用。(译注：显然这不是调用 SuppressFinalize 方法的原因，因为带 Boolean 参数的 Dispose 方法中后一个 if 判断语句已经很好地解决了这个问题。调用 SuppressFinalize 方法的主要原因是它避免了终止化对象给垃圾收集器带来的沉重负担。)调用 GC 的 SuppressFinalize 方法会打开与 this 所引用的对象相关的一个位标记。当该位标记被打开时，CLR 将不会再把对象的指针从终止化链表转移到终止化可达队列上，从而阻止对象的 Finalize 方法被调用。当垃圾收集器判定该对象是可收集的垃圾时，它的内存会被立即回收。

19.4.1　使用实现了 Dispose 模式的类型

既然已经知道了怎样为一个类型实现 Dispose 模式，下面就让我们来看一下怎样使用实现了 Dispose 模式的类型。这里不再讨论前面的 OSHandle 类，而是讨论更为常见的 System.IO.FileStream 类。利用 FileStream 类，我们可以打开一个文件，从中读取字节或者向其写入字节，并最终将其关闭。

FileStream 类的内部实现和 OSHandle 类有些相似，不同的是它的构造器中调用了 Win32 函数 CreateFile，并将函数返回的结果保存在一个私有句柄字段中。FileStream 类还提供有几个额外的属性(例如 Length、Position、CanRead、Handle 等)和方法(例如 Read、Write、Flush 等)。

假设我们希望编写代码来创建一个临时文件，并向其中写入一些字节，然后再删除该文件。我们开始可能会像下面这样编写代码。

```
using System;
using System.IO;

class App {
   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");   // 这里会抛出一个 IOException
   }
}
```

不幸的是，如果我们编译并运行上面的代码，它可能会工作，但大多数时候并不能。问题在于 File 的静态方法 Delete 在请求 Windows 删除一个打开的文件，Delete 会因此抛出一个 System.IO.IOException，同时还附带有一个异常的消息文本“该进程无法访问文件“Temp.dat”，因为该文件正由另一进程使用”。

但是我们要注意在有些情况下，文件实际上可能会被删除。这是因为如果在 Write 调用之后 Delete 调用之前垃圾收集线程由于某种情况而被启动，那么 FileStream 对象的 Finalize 方法将被调用，这时文件就会被关闭，随后的 Delete 操作也就可以正常运行。然而产生这种情况的可能性非常小，上面的代码在绝大多数情况下都会失败。

幸运的是，FileStream 类实现了 Dispose 模式，它允许我们显式地关闭文件。下面是修改后的代码。

```
using System;
using System.IO;

class App {
    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);

        // 在写入字节后显式关闭文件
         ((IDisposable) fs).Dispose();

        // 删除临时文件
        File.Delete("Temp.dat");    // 这里总会正常运行
    }
}
```

这段代码和前面的代码惟一的不同在于其中调用了 FileStream 的 Dispose 方法。Dispose 方法调用带 Boolean 参数的 Dispose 方法，带 Boolean 参数的 Dispose 方法又调用 CloseHandle 函数，Windows 于是关闭文件。这样，当 File 的 Delete 方法被调用时，Windows 会发现该文件已经关闭，因此文件删除会成功进行。

注意 FileStream 对象仍然存在于托管堆中，所以我们仍然可以调用其上的方法。最后，垃圾收集器会运行，并将该 FileStream 对象判定为可收集的垃圾。这时，垃圾收集器本来应该调用 FileStream 对象上的 Finalize 方法，但是因为 Dispose 方法调用了 GC 的 SuppressFinalize 方法，所以 Finalize 方法不会再被调用，对象的内存将直接被回收。

注意　前面的代码在调用 Dispose 方法之前首先将 fs 变量转型成了一个 IDisposable 接口。大多数实现了 Dispose 模式的类型并不需要这样的转型，但是 FileStream 对象则需要这样做，这是因为微软的开发人员将 Dispose 方法实现成了一个显式接口方法(本书第 15 章对此有描述)。我个人认为这是一种令人遗憾的做法，因为这只能使事情更复杂，也不会增添任何价值。一般情况下，只有在具有多个同名方法时，我们才应该使用显式接口方法实现。因为 Dispose 和 Close 方法的名称并不相同，所以它们都应该被实现为公有方法，从而避免调用时的转型操作。

幸运的是，FileStream 类还提供了一个公有的 Close 方法，借助该方法我们可以用更简便的代码来实现同样的功能。

```
using System;
using System.IO;

class App {
   static void Main() {

      // 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);

      // 在写入字节后显式关闭文件
      fs.Close();

      // 删除临时文件
      File.Delete("Temp.dat");    // 这里总会正常运行
   }
}
```

注意 记住 Close 方法并不是 Dispose 模式正式定义的一部分，有些类型提供了 Close 方法，而有些类型则没有。

需要注意的是调用 Dispose 或 Close 方法只是在一个确定的时刻对对象占用的非托管资源执行清理操作而已，它们并不会控制托管堆中对象所使用的内存的生存期。这意味着我们在调用过 Dispose 或 Close 方法之后，仍然可以调用对象上的方法。下面的代码在文件关闭之后又调用了 FileStream 的 Write 方法试图向文件中写入更多的字节。显然，这样的操作不可能成功。当代码执行时，该 Write 方法调用将抛出一个 System.ObjectDisposedException 异常，同时还附带有一个异常的消息文本“无法访问已关闭的文件”。

```
using System;
using System.IO;

class App {
    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);

        // 在写入字节后显式关闭文件
        fs.Close();

        // 试图在文件关闭之后写入字节。下面一行
        // 会抛出一个 ObjectDisposedException 异常
        fs.Write(bytesToWrite, 0, bytesToWrite.Length);

        // 删除临时文件
        File.Delete("Temp.dat");    // 这里总会正常运行
    }
}
```

注意这里不会出现内存崩溃的情况，因为 FileStream 对象的内存仍然存在。只是由于对象被执行了资源清理之后，我们便不能再成功执行某些方法而已。

重要　在为我们自己的类型实现 Dispose 模式时，如果对象占用的非托管资源被执行了清理，那么所有的方法调用都要抛出一个 System.ObjectDisposedException 异常(译注：要所有的方法调用都抛出 ObjectDisposedException 异常是不合适的，只有那些在对象占用的非托管资源有效的情况下才能成功执行的方法才应该这么做)。但是 Dispose 和 Close 方法在多次调用的情况下则不应该抛出 ObjectDisposedException 异常，遇到这种情况它们应该不执行任何操作而直接返回(就像前面 OSHandle 类型中实现的那样)。

重要 一般情况下，我个人建议大家不要使用 Dispose 或者 Close 方法。理由是 CLR 的垃圾收集器已经实现的很好了，我们完全可以将工作交给它来做。垃圾收集器知道一个对象何时不再被应用程序代码所访问，直到在做出这样的判定之后它才会考虑收集对象。而当应用程序代码调用 Dispose 或 Close 方法时，它实际上是在表明自己知道应用程序已经不再需要使用该对象了。对于许多应用程序来说，这往往是不可能的。

例如，我们的代码构造了一个对象，然后又将其传递给了另外一个方法，而该方法可能会将该对象引用保存在自己的某个内部字段中(成为一个根)。我们的代码可能并不知道这些情况。如果我们的代码调用了该对象的 Dispose 或者 Close 方法，那么当其他的代码再试图操作该对象时，系统将很有可能抛出 ObjectDisposedException 异常。

但是，在需要显式清理对象占用的非托管资源(例如试图删除一个打开的文件)、或者确信操作是安全的情况下，我仍然建议大家调用 Dispose 或者 Close 方法，因为这可以阻止由于 Finalize 方法运行所导致的对象代龄的提升，从而提高应用程序的性能。

19.4.2 C#的 using 语句

前面的示例代码向大家展示了怎样显式调用一个类型的 Dispose 或者 Close 方法。如果大家决定显式调用这两个方法，强烈建议把它们放在一个异常处理的 finally 块中。因为这样可以保证它们被执行。因此，上面的示例代码更好的写法如下：

```
using System;
using System.IO;

class App {
   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 = null;
      try {
         fs = new FileStream(“Temp.dat”, FileMode.Create);
```

```
            // 将字节写入临时文件
            fs.Write(bytesToWrite, 0, bytesToWrite.Length);
        }
        finally {
            // 在写入字节后显式关闭文件
            if (fs != null)
             ((IDisposable) fs).Dispose();
        }

        // 删除临时文件
        File.Delete("Temp.dat");    // 这里总会正常运行
    }
}
```

像上面这样添加异常处理代码是必要。如果大家使用的是 C#，则还可以利用其中的 using 语句，该语句为我们提供了一种简化的语法来产生和上述代码相同的结果。看下面的代码：

```
using System;
using System.IO;

class App {
    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");    // 这里总会正常运行
    }
}
```

我们首先在 using 语句内初始化一个对象，并将其引用保存在一个变量中。然后我们在和 using 语句匹配的大括号内访问该变量。当我们编译这段代码时，编译器会自动产生一个 try 块和一个 finally 块。在 finally 块中，编译器会产生代码将变量转型为一个 IDisposable 接口，并调用其上的 Dispose 方法。很明显，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的能力，前提是这些变量的类型相同。另外，using 语句还支持使用一个已经初始化的变量。有关该话题的更多信息，请参见《C#程序员参考》。

我目前还没有见过其他哪一门编程语言为实现 Dispose 模式的类型提供了如此方便的语法。在这些编程语言中，大家还必须手动添加 try 和 finally 异常处理代码。

重要 如前面一节的末尾所指出的那样，我们只应该在需要资源清理的地方调用 Dispose 或者 Close 方法。因此，我们也应该谨慎地使用 C#的 using 语句。我看到许多开发人员经常随意地使用 C#的 using 语句，但后来他们才发现自己过早地执行了资源清理，从而导致应用程序的其他部分抛出 ObjectDisposedException 异常。

19.4.3 一个有趣的依赖问题

System.IO.FileStream 类型允许用户打开一个文件进行读写操作。为了提高性能，该类型的实现使用了一个内存缓冲。只有在内存缓冲充满时，FileStream 对象才会将缓冲区中的数据填充到文件中。FileStream 类型只支持字节的读写操作。如果我们希望操作更复杂的数据类型(例如 Int32、Double、String 等)，则可以使用 System.IO.BinaryWriter 类型，下面的代码演示了可能的做法：

```
FileStream fs = new FileStream(“DataFile.dat”, FileMode.Create);
BinaryWriter bw = new BinaryWriter(fs);
bw.Write(“Hi there”);

// 下面对 Close 的调用是我们应该做的
bw.Close()
// 注意：调用 BinaryWriter.Close 会同时关闭其使用的 FileStream
// 对象。FileStream 对象在这种情况下不必显式关闭。
```

注意 BinaryWriter 的构造器接受一个 FileStream 对象作为参数。BinaryWriter 对象内部会保存该 FileStream 对象的一个引用。当我们向一个 BinaryWriter 对象写入数据时，它会将数据缓存在自己的内存缓冲区中。当其内存缓冲区充满时，BinaryWriter 对象才会将数据写入 FileStream 对象。

当我们通过 BinaryWriter 对象进行数据写入的操作执行完毕时，我们应该调用其上的 Dispose 方法或 Close 方法。(由于 BinaryWriter 实现了 Dispose 模式，所以我们也可以使用 C#的 using 语句。)这两个方法的行为相同，都会导致 BinaryWriter 对象将其内存缓冲区中的数据填充到 FileStream 对象中，同时关闭该 FileStream 对象。当 FileStream 对象被关闭时，它会首先将自己缓冲区中的内容填充到磁盘文件中，然后才会调用 CloseHandle 函数。

注意　我们不必显式调用 FileStream 对象的 Dispose 或 Close 方法，因为 BinaryWriter 已经为我们做了这件事。如果我们调用了 Dispose 或 Close 方法，FileStream 对象会发现它已经被执行了资源清理，Dispose 或 Close 方法将不执行任何操作而直接返回。

那么如果我们没有显式调用 BinaryWriter 对象的 Dispose 或 Close 方法会出现什么情况呢？我们知道垃圾收集器能够正确地检测出对象是否已成为可收集的垃圾，如果是，垃圾收集器将对它们执行终止化操作。但是垃圾收集器并不能保证多个对象上的 Finalize 方法的执行顺序，所以如果首先被执行终止化操作的是 FileStream 对象的话，它将会关闭文件。然后，当 BinaryWriter 对象被执行终止化时，它将试图向一个已关闭的文件中写入数据，这自然会抛出异常。但是，如果首先被执行终止化的是 BinaryWriter 对象的话，其中的数据将会被安全地写入到文件中。

那么微软是如何解决这个问题的呢？使垃圾收集器以特定的顺序来执行对象终止化操作显然不可能，因为对象之间可能包含着对彼此的引用，垃圾收集器根本无法正确地推断出对它们执行终止化的顺序。微软的解决方法是不让 BinaryWriter 类重写 Finalize 方法。这意味着如果我们忘记了显式关闭 BinaryWriter 对象，其内存缓冲区中的数据必然会丢失。微软希望开发人员能够认识到这一点，并在自己的代码中显式调用 Dispose 或 Close 方法。

19.5　弱 引 用

我们知道，当一个根指向一个对象时，该对象不可能被执行垃圾收集，因为应用程序代码还可以访问该对象。在这种情况下，我们通常说存在一个该对象的强引用(strong reference)。垃圾收集器还支持弱引用(weak reference)的概念。弱引用允许垃圾收集器收集对象，同时也允许应用程序访问该对象，结果是哪一个要取决于时间。

如果只有对象的弱引用存在，那么在垃圾收集执行后，该对象的内存将被回收，应用程序再试图访问对象时会告失败。另一方面，要访问一个弱引用对象，应用程序必须获取该对象的一个强引用。如果应用程序在对象被执行垃圾收集之前获得了它的强引用，那么垃圾收集器将不能对该对象执行垃圾收集，因为这时存在一个该对象的强引用。

是否对上面的叙述感到有些困惑？我们来看一段代码，这会帮助大家搞清楚弱引用的真正含义。

```
void SomeMethod() {
    // 创建一个新对象的强引用
    Object o = new Object();

    // 创建一个短弱引用对象。
    // 该对象负责追踪对象 o 的生存期
    WeakReference wr = new WeakReference(o);

    o = null;          // 移除对象的强引用

    o = wr.Target;

    if (o == null) {
        // 出现过垃圾收集，对象的内存已经被回收
    } else {
        // 未出现过垃圾收集，所以我们可以成功地
        // 使用变量 o 来访问对象
    }
}
```

那么我们为什么需要弱引用呢？我们经常会用到这样一些数据结构，它们很容易创建但是却需要大量的内存。例如，我们可能会有一个应用程序需要知道用户硬盘中所有的目录和文件。我们可以很容易地构造一个树来反映这些信息，当应用程序运行时，它就可以引用内存中的树，而不必再访问用户的硬盘。这显然会极大地提高应用程序的性能。

但问题在于这个树可能会非常庞大，需要许多内存。如果用户转而访问应用程序的其余部分，那么这个树可能变得不再必要，但却仍然浪费着许多内存。我们可能会放弃对这个树的根对象的引用，但是如果用户又切换回到应用程序的第一部分，那么我们又需要重新构造这个树。弱引用使得我们可以方便、高效地处理这种情况。

当用户离开应用程序的第一部分时，我们可以创建一个弱引用对象来引用这个树的根对象，而放弃其所有的强引用。如果应用程序其他组件的内存负载比较低，垃圾收集器将不会回收这个树的内存。

当用户重新切换回应用程序的第一部分时，应用程序可以尝试获得对这个树的根对象的强引用。如果成功，应用程序将不再需要遍历用户的硬盘。

System.WeakReference 类型提供有两个公有构造器：

```
public WeakReference(Object target);
public WeakReference(Object target, Boolean trackResurrection);
```

参数 target 表示 WeakReference 要追踪的对象。参数 trackResurrection 表示 WeakReference 是否要追踪对象复苏，换句话说它表示在对象的 Finalize 方法被调用之后，WeakReference 是否还应该追踪对象。通常情况下，我们为参数 trackResurrection 传递的值为 false，实际上第一个构造器在内部就是这样调用第二个构造器的。

为方便起见，我们将不追踪对象复苏的 WeakReference 称作一个短弱引用(short weak reference)，而将追踪对象复苏的 WeakReference 称作长弱引用(long weak reference)。如果一个对象的类型没有重写 Finalize 方法，那么短弱引用和长弱引用的行为是一样的。强烈建议大家不要使用长弱引用，因为长弱引用在一个对象被执行终止化后仍允许我们使该对象重新复苏，而这会导致对象状态的不可预期。

一旦我们创建了一个对象的弱引用，我们通常要将该对象的强引用设置为 null。如果有任何该对象的强引用存在，垃圾收集器都不会对该对象执行垃圾收集。

为了再次使用对象，我们必须将弱引用转换为一个强引用，这可以通过查询 WeakReference 对象的 Target 属性，并将结果赋值给应用程序的一个根来完成。如果 Target 属性的返回值为 null，那么对象已经被执行了垃圾收集。如果 Target 属性的返回值不为 null，那么我们将得到对象的一个强引用，应用程序代码也就可以继续操作该对象。只要存在对象的强引用，它就不可能被执行垃圾收集。

19.5.1　弱引用的内部机理

从前面的讨论中我们可以看出 WeakReference 对象和其他对象的行为有很大差别。通常情况下，如果我们的应用程序有一个根引用了一个对象，而该对象又引用了另一个对象，那么这两个对象都将是可达对象，二者都不可能被执行垃圾收集。但是，如果我们的应用程序有一个根引用了一个 WeakReference 对象，那么被该 WeakReference 对象引用的对象将不再被认为是可达对象，并且可以被执行垃圾收集。

为了完全理解弱引用的工作原理，我们再来深入探究一下托管堆。托管堆中包含有两个内部数据结构专门用来管理弱引用，即短弱引用表和长弱引用表。这两个表中只是包含着一些指针，它们引用着托管堆中的对象。

刚开始的时候，这两个表都为空。当我们创建一个 WeakReference 对象时，垃圾收集器并不会从托管堆中为其分配内存。相反，它会在两个弱引用表中选择一个(短弱引用使用短弱引用表，长弱引用使用长弱引用表)，并在其中寻找一个空白插槽。

一旦找到一个空白插槽，该插槽的值将被设为我们希望追踪的对象的地址——也就是传递给 WeakReference 构造器的那个对象指针，new 操作符返回的值就是相应弱引用表中插槽的地址。显然，这两个弱引用表不会被认为是应用程序的根，否则垃圾收集器将不能收集它们中的指针引用的对象。

下面是垃圾收集器运行时发生的一系列事情：

(1) 垃圾收集器构造一个包含所有可达对象的图。前面对此已经做过详细的介绍。

(2) 垃圾收集器扫描短弱引用表。如果该表中有指针引用的对象不是前面构造的可达对象图的一部分，那么该指针标识的将是一个不可达对象，短弱引用表中对应插槽的值将被设为null。

(3) 垃圾收集器扫描终止化链表。如果该链表中有指针引用的对象不是前面构造的可达对象图的一部分，那么该指针标识的将是一个不可达对象，它将被从终止化链表转移到终止化可达队列上。这时，对象又成为可达对象图的一部分。

(4) 垃圾收集器扫描长弱引用表。如果该表中有指针引用的对象不是可达对象图(该图现在包括终止化可达队列中引用的对象)的一部分，那么该指针标识的将是一个不可达对象，长弱引用表中对应插槽的值将被设为null。

(5) 垃圾收集器压缩内存，填充不可达对象空出的位置。注意有时候如果垃圾收集器判定不可达对象空出的内存碎片数量不值得耗费时间去压缩，那么它将不会执行这一步。

一旦大家理解了垃圾收集器背后的工作原理，理解弱引用也就很容易了。查询 WeakReference 的 Target 属性会导致系统返回相应弱引用表中插槽的值。如果返回的插槽中的值为 null，那就证明对象已经被执行了垃圾收集。

短弱引用并不追踪对象复苏。这意味着只要垃圾收集器判定对象成为不可达对象，它就会把短弱引用表中对应的指针设为 null。如果对象的类型重写了 Finalize 方法，那么这时该方法还没有被调用，所以对象应该仍然存在。如果应用程序访问 WeakReference 对象的 Target 属性，返回值将为 null，虽然这时对象仍然存在。

长弱引用追踪对象复苏。这意味着只有在垃圾收集器认为对象的存储空间可以被回收时，它才会将长弱引用表中对应的指针设为 null，这时的对象也不可能再重新复苏。如果对象的类型重写了 Finalize 方法，那么这时该方法应该已经被调用。

19.6　对 象 复 苏

相信很多人都对终止化操作很感兴趣。实际上，终止化的内容远不止前面所描述的那些。大家可能已经注意到了这样一个现象，当一个需要终止化的对象被认为“死亡”时，垃圾收集器可以强制使该对象获得“重生”，因为只有这样才能调用它的 Finalize 方法。在它的 Finalize 方法被调用之后，它才算真正地“死亡”了。总的来说，一个需要终止化的对象会经历“死亡”、“重生”、然后再“死亡”的过程。这个有趣的现象被称为复苏(resurrection)。顾名思义，复苏就是使一个对象“死而复生”。

准备调用一个对象的 Finalize 方法的行为就是一种复苏的形式。当垃圾收集器将对象的一个引用放到终止化可达队列中时，对象就成为一个可达对象，可以说获得了“重生”。但在对象的 Finalize 方法被调用之后，再没有任何根指向对象，这时对象才算真正地“死亡”了。但是如果我们在一个对象的 Finalize 方法中将该对象的指针再放入一个全局变量或静态变量中又会出现什么情况呢？看下面的代码。

```
class SomeType {
    ~SomeType() {
        Application.ObjHolder = this;
    }
}

class Application {
    public static Object ObjHolder;    // 默认值为 null
    …
}
```

在上面的代码中，当 SomeType 对象的 Finalize 方法被执行时，该对象的一个引用将被放入一个根中，从而使其又成为应用程序中的一个可达对象。对象重新复苏以后，垃圾收集器不会再将其认为是可收集的垃圾，应用程序也可以自由地使用该对象。但是我们必须记住该对象已经被执行了终止化，所以使用它可能会导致不可预期的结果。另外需要注意的是如果 SomeType 中包含有引用其他对象的字段(直接或者间接)的话，所有被引用的对象都将重新复苏，因为它们现在也都是可达对象。但是，我们需要注意这些重新复苏的对象中的一些可能也已经被执行了终止化。

注意 我们定义的任何类型都可能在某一点因重新复苏而脱离我们的控制。换句话说一个对象在其 Finalize 方法被调用之后，它的成员仍然可能会被访问。在一个理想的环境中，我们可能会考虑添加代码来检查一个对象的 Finalize 方法是否已经执行，然后根据检查结果来对成员访问做相应的处理(比如抛出适当的异常)。但是在实践中我们必须为此编写许多繁冗的代码，我怀疑很多开发人员是否对每一个类型都能设计的如此周到。

如果有代码将 Application.ObjHolder 设置为 null，那么对象就又成为不可达对象。最后，垃圾收集器将对象判定为可收集的垃圾，并回收其在托管堆中的内存。因为终止化链表上不再有对象的指针，所以其 Finalize 方法将不会再被调用。

对象复苏听起来很酷，但是能够很好地使用它却并不容易，我们应该尽可能地避免使用这项技术。当开发人员使用对象复苏时，他们通常希望对象在每次“死亡”时都能够顺利地清理自己的非托管资源。为了做到这一点，GC 类提供了一个名为 ReRegisterForFinalize 的静态方法，该方法接受一个类型为 System.Object 的参数。下面是前面代码改进后的一个版本。

```
class SomeType {
    ~SomeType() {
        Application.ObjHolder = this;
        GC.ReRegisterForFinalize(this);
    }
}
```

当 Finalize 方法被调用时，它首先用一个应用程序的根来引用对象，从而使其重新复苏。然后，Finalize 方法又调用了静态方法 ReRegisterForFinalize，将指定对象(this)的地址添加到终止化链表的末端。当未来某个时刻垃圾收集器检测到对象不可达时，它会将对象的指针转移到终止化可达队列上，对象的 Finalize 方法将再一次被调用。

上面的例子向大家展示了如何创建一个不断复苏、永不“死亡”的对象——但是我们通常不会希望对象有这样的行为，更常见的做法是在 Finalize 方法内部根据一定的条件来使对象重新复苏。

注意　我们要确保每次对象复苏时 ReRegisterForFinalize 方法都只能被调用一次，否则对象的 Finalize 方法将会被多次调用。因为每调用 ReRegisterForFinalize 方法一次，它就会在终止化链表上添加一个条目。当对象被判定为可收集的垃圾时，终止化链表上所有这些条目都将被转移到终止化可达队列上，从而导致对象的 Finalize 方法被多次调用。

19.6.1　利用复苏设计一个对象池

下面的示例可以向大家展示对象复苏的应用价值。假设我们希望创建一个 Expensive 对象池。由于这些对象非常耗费资源，创建它们需要花费很多时间。处于性能考虑，我们打算在应用程序启动之时就能创建一组 Expensive 对象，然后在应用程序的整个生存期中重复地使用它们。

下面是管理 Expensive 对象池的示例代码。

```
using System;
using System.Collections;

// 下面这个类的实例构造起来比较耗费资源
class Expensive {

    // 下面的静态 Stack 中包含着
    // 对象池中可用对象的引用
    static Stack pool = new Stack();

    // 下面的静态方法返回从对象池中取得的对象
    public static Expensive GetObjectFromPool() {
        // 从对象池中获取一个对象，并将其引用从对象池中删除
        return (Expensive) pool.Pop();
    }

    // 当应用程序关闭时，调用下面的静态方法销毁对象池
    public static void ShutdownThePool() {

        // 阻止终止化对象将自己添加到对象池中
        pool = null;
    }
```

```
    // 下面的构造器创建一个对象并将其添加到对象池中
    public Expensive() {
        // 构造对象要花费比较长的时间
        …

        // 在对象构造完之后，将其添加到对象池中
        pool.Push(this);
    }

    // 当应用程序不再需要对象时，下面的 Finalize 方法将被调用
    ~Expensive() {

        // 如果应用程序没有关闭，就将对象重新
        // 添加到对象池中
        if (pool != null) {

            // 调用 ReRegisterForFinalize 方法以使对象能够以
            // 正确的状态加入到对象池中
            GC.ReRegisterForFinalize(this);

            // 将对象重新加入到对象池中
            pool.Push(this);
        }
    }
}

class App {
    static void Main() {
        // 构造一组 Expensive 对象填充对象池
        for (Int32 i = 0; i < 10; i++)
            new Expensive();
        …

        // 当我们需要对象时，就从对象池中获取
        Expensive e = Expensive.GetObjectFromPool();
        // 应用程序下面就可以使用 e 了
        …

        // 要关闭应用程序，首先关闭对象池
        Expensive.ShutdownThePool();
    }
}
```

Expensive 中定义了一个类型为 System.Collections.Stack 的私有静态字段 pool，该字段负责管理对象池中可用的对象。在 Main 中，我们用一个循环构造了 10 个 Expensive 对象。当每个对象被构造时，它会将自己添加到对象池中。Expensive 的静态字段 pool 本身是一个应用程序的根，它引用着一组 Expensive 对象，这些对象一经创建便不可能被执行垃圾收集。

当应用程序需要使用 Expensive 对象时，它就调用 Expensive 的静态方法 GetObjectFromPool。该方法从对象池中返回一个对象引用，同时将其从对象池中删除。应用程序自此便可以使用 Expensive 对象了。

随着时间的推移，应用程序将不再持有前面返回的 Expensive 对象引用。如果在这之后出现了垃圾收集，那么没有被任何根所引用的 Expensive 对象将被执行终止化。当 Expensive 对象的 Finalize 方法被调用时，它会将自己再次加入到对象池中，从而使其重新复苏，阻止垃圾收集器回收其内存。另外，Finalize 方法还会调用 GC 的 ReRegisterForFinalize 方法。在未来的某个时刻，该 Expensive 对象还会再次被应用程序所获取，并在随后的某个时刻再次成为不可达对象，垃圾收集器也会再次被启动。由于我们前面调用了 ReRegisterForFinalize 方法，所以 Finalize 方法还会再次被调用，从而再次将对象加入到对象池中。

在 Main 退出之前，它会调用 Expensive 的静态方法 ShutdownThePool，该方法会将 pool 字段设为 null。当应用程序关闭时，CLR 会为在托管堆中的所有对象调用 Finalize 方法。这时我们不应该再去调用 ReRegisterForFinalize 方法，因为这样做会导致一个无限循环(CLR 会在 40 秒后强制中断进程)。因此我们会在 Expensive 的 Finalize 方法中检测 pool 字段。如果该字段为 null，那么 Expensive 对象将不会再被添加到终止化链表上，同时也不会被添加到对象池中，在这之后，Expensive 对象的内存将可以被回收。

如我们所见，对象复苏为实现对象池提供了一种简单有效的方式。

19.7　对象的代龄

如本章开始所言，代龄是旨在提高垃圾收集器性能的一种机制。一个基于代龄的垃圾收集器(又称季节性垃圾收集器，ephemeral garbage collector，本书不拟使用这一术语)有以下几点假设：

- 对象越新，其生存期越短。
- 对象越老，其生存期越长。
- 对托管堆的一部分执行垃圾收集要比对整个托管堆执行垃圾收集速度更快。

这些假设经过了大量的研究，已经在很多现存的应用程序上得到了验证，它们影响着垃圾收集器的实现。本节将解释垃圾收集器中代龄的工作机制。

在托管堆初始化时，其中不包括任何对象。这时添加到托管堆中的对象被称为第 0 代对象。简单地说，第 0 代对象就是那些新构造的对象，垃圾收集器还没有对它们执行过任何检查。图 19.7 向我们展示了一个新启动的应用程序中托管堆的情况，其中分配有 5 个对象(A 到 E)。经过一段时间后，对象 C 和对象 E 将变为不可达对象。

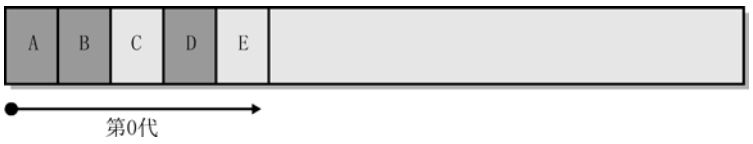

图 19.7 一个刚经过初始化的托管堆，其中包含着一些对象。
这时所有的对象都处于第 0 代。垃圾收集还没有执行过。

当 CLR 初始化时，它会为第 0 代对象选择一个阈值容量，假定为 256 KB。(实际的容量可能会与此不同。)当分配新对象导致第 0 代对象超过了为其设定的阈值容量时，垃圾收集器就必须启动了。假设从对象 A 到对象 E 总共占用了 256 KB，那么当对象 F 被分配时，垃圾收集器就会启动。垃圾收集器判定对象 C 和 E 为垃圾对象，因此会压缩对象 D 使其邻接于对象 B。在此次垃圾收集中存活下来的对象(对象 A、B、和 D)将被认为是第 1 代对象，它们经过了一轮垃圾收集检查。这时托管堆的情况如图 19.8 所示。

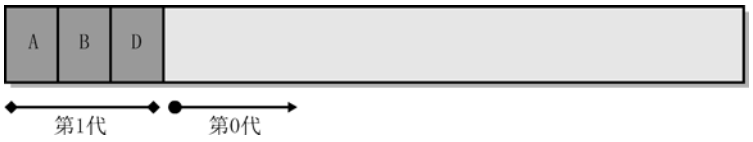

图 19.8 经过一次垃圾收集之后，第 0 代对象中的存活者被
提升为第 1 代对象，第 0 代对象暂时空缺

在第一次垃圾收集执行后，第 0 代对象暂时空缺。但很快，随着应用程序的运行，又有新的对象被分配而成为第 0 代对象。图 19.9 中展示的对象 F 到对象 K 就是这样的情况。应用程序继续运行，对象 B、H 和 J 又成为不可达对象，它们的内存也要在某一点被回收。

图 19.9 第 0 代中又分配了新的对象，第 1 代中某些对象
已经成为垃圾对象

现在假设应用程序又试图分配对象 L，这将再一次使第 0 代对象超过它的阈值容量，于是又必须开始执行第二次垃圾收集。在这之前，垃圾收集器必须判定要收集哪些代的对象。前面曾说过，CLR 初始化的时候会为第 0 代对象选择一个阈值容量。实际上，它也会为第 1 代对象选择一个阈值容量。我们假设其为第 1 代对象选择的阈值容量为 2 MB。

当第二次垃圾收集开始执行时，它也会查看第 1 代对象占用了多少内存。在本例中，由于第 1 代对象占用的内存远少于 2 MB，所以垃圾收集器只会去检查第 0 代对象。在继续讨论之前，我们再来回顾一下一个基于代龄的垃圾收集器所做的几点假设。其第一个假设是新创建的对象的生存期比较短。所以第 0 代对象中成为垃圾对象的数量会比较多，因此对第 0 代对象执行垃圾收集将有可能回收比较多的内存，忽略掉第 1 代对象会提高垃圾收集的速度。

然而忽略第 1 代对象的意义不仅仅在于不对它们执行垃圾收集。实际上，更重要的是垃圾收集器不用再遍历整个托管堆中的所有对象了，这对垃圾收集器的性能提升具有重大的意义。如果一个根或者一个对象引用了一个代龄较大的对象，那么垃圾收集器就可以忽略一些这样的内部引用，从而减少构造可达对象图所需的时间。说忽略代龄较大的对象中的一些内部引用，而不是全部内部引用是因为一个代龄较大的对象有可能引用一个新的对象。为了确保这些位于代龄较大的对象中的一些新对象也能被检查到，垃圾收集器使用 JIT 编译器内部的一种机制来在对象的内部引用字段改变时设置一个相应的位标记。这使得垃圾收集器可以知道自从上次垃圾收集执行以来，哪些代龄较大的对象已经被应用程序所改写。这样垃圾收集器只需要检查它们是否引用了第 0 代对象就可以了。

注意　微软的性能测试显示在 Pentium 200-MHZ 的机器上对第 0 代对象执行一次垃圾收集所花费的时间不超过 1 毫秒。微软的目标是使垃圾收集所花费的时间不超过一个普通的内存页面错误所花费的时间。

一个基于代龄的垃圾收集器所做的第二个假设是生存期比较长的对象将倾向于继续存活，所以第 1 代对象继续成为可达对象的可能性比较大。因此，如果垃圾收集器检查第 1 代对象，很有可能找不到很多垃圾对象，能够回收的内存也就有限。这样以来，收集第 1 代对象也就很有可能会浪费大量的时间。所以如果有垃圾对象位于第 1 代，那么它仍然会呆在那里。在第二次垃圾收集执行后，托管堆的情况将如图 19.10 所示。

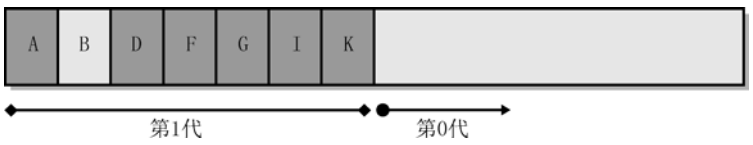

图 19.10 经过两次垃圾收集之后，第 0 代对象中的存活者被提升为第 1 代(第 1 代内存总量有所增长)，第 0 代暂时空缺

如我们所见，所有第 0 代对象中的存活者现在又成了第 1 代的一部分。因为垃圾收集器没有检查第 1 代，所以对象 B 的内存并没有被回收，虽然它在第二次垃圾收集执行时已经成为不可达对象。同样，在第二次垃圾收集执行后，第 0 代对象暂时空缺，只有等着分配新的对象。接着，应用程序继续运行，并分配对象 L 到对象 O。过一段时间后，对象 G、L 和 M 又成为不可达对象。图 19.11 展示了这时托管堆的情况。

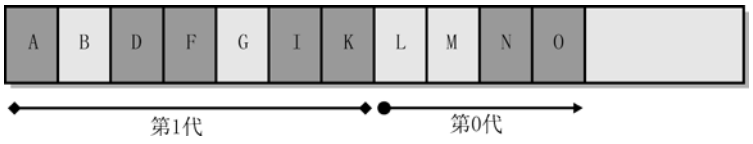

图 19.11 新对象被分配成为第 0 代，第 1 代中有了更多的垃圾

假设分配对象 P 又导致了第 0 代对象超过它的阈值容量，于是垃圾收集又被启动。因为第 1 代中所有对象的内存总量仍小于 2 MB，所以垃圾收集器仍会决定只收集第 0 代对象，而忽略第 1 代中的不可达对象(对象 B 和 G)。在这次垃圾收集执行完毕后，托管堆的情况将如图 19.12 所示。

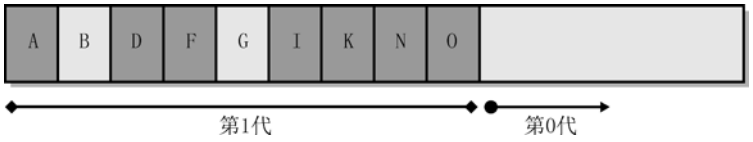

图 19.12 经过三次垃圾收集之后，第 0 代对象中的存活者被提升为第 1 代(第 1 代内存总量再次增长)，第 0 代暂时空缺

在图 19.12 中，我们看到第 1 代对象在缓慢增长。现在，让我们假设第 1 代对象的增长导致其内存总量到达了 2 MB 这一阈值容量。这时，应用程序继续运行(因为垃圾收集刚刚完成)，并分配对象 P 到对象 S，这使得第 0 代也达到它的阈值容量。这时托管堆的情况将如图 19.13 所示。

图 19.13　新对象被分配成为第 0 代，第 1 代中有了更多的垃圾

当应用程序试图分配对象 T 时，因为第 0 代对象已经充满(即达到设定的阈值容量)，所以必须执行垃圾收集。但是这一次垃圾收集器会发现第 1 代对象的内存总量已经超过了 2 MB 这一阈值——因为在前面几次对第 0 代对象的收集中，许多第 1 代对象已经成为了垃圾对象。所以这次垃圾收集器会同时收集第 0 代和第 1 代中所有的垃圾对象。在这次垃圾收集执行完毕后，托管堆的情况将如图 19.14 所示。

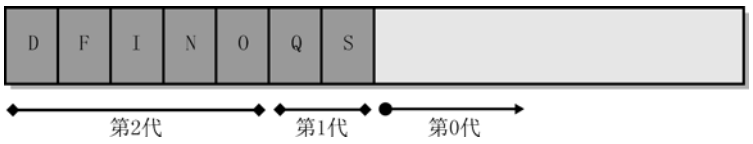

图 19.14 经过四次垃圾收集之后，第 1 代中的存活者被提升为第 2 代，第 0 代中的存活者被提升为第 1 代，第 0 代暂时空缺

和前面一样，在这次垃圾收集执行完毕后，所有第 0 代中的存活者将被提升为第 1 代，而第 1 代中的存活者将被提升为第 2 代，第 0 代对象则暂时空缺。其中第 2 代中的对象至少经过了两次垃圾收集的检查。在产生第 2 代对象之前，系统有可能已经执行了多次垃圾收集，但只有在第 1 代对象的内存总量达到它的阈值容量时，垃圾收集器才会检查第 1 代对象，而在这之前系统可能已经对第 0 代对象执行了好几次垃圾收集。

CLR 的托管堆只支持 3 个代龄：第 0 代、第 1 代和第 2 代。没有第 3 代。当 CLR 初始化时，它会为这三代选择 3 个阈值容量。如前所述，第 0 代的阈值容量大约为 256 KB，第 1 代的阈值容量大约为 2 MB。第 2 代的阈值容量大约为 10 MB。选择阈值容量是为了提高系统性能，阈值容量越大，垃圾收集执行的频率也就越低。性能提升源于下面的初始假设：新对象的存活时间比较短，而老对象则倾向于继续存活。

另外，CLR 的垃圾收集器还是一个自调节的垃圾收集器。这意味着垃圾收集器会在执行垃圾收集的过程中学习应用程序的行为。例如，如果我们的应用程序构造了许多对象，并在很短的时间里使用它们，那么垃圾收集器很有可能在第 0 代就能回收掉许多内存。实际上，所有第 0 代的对象都有可能被回收掉。

如果垃圾收集器发现在第 0 代对象被收集以后存活下来的对象很少，那么它可能会决定将第 0 代的阈值容量从 256 KB 减少到 128 KB。这将使得垃圾收集执行的频率变得更高，但是每次需要做的收集工作将更少，这样以来应用程序进程的工作集便不会增长的过大。实际上，如果第 0 代中所有的对象都是垃圾对象的话，垃圾收集器将不必压缩任何内存，它可以直接将 NextObjPtr 指针移到第 0 代对象的开始之处来完成垃圾收集——这恐怕是回收内存最快的方法了。

注意 垃圾收集器在 ASP.NET Web 窗体和 XML Web 服务应用程序中工作的情况相当出色。对于 ASP.NET 应用程序来说，当客户请求到达时，会有许多新的对象被构造，这些对象会根据客户的行为执行某些操作，然后将结果返回给客户。之后，所有用来响应客户请求的对象都将成为垃圾对象。换句话说，每一次 ASP.NET 应用程序请求都会导致产生许多垃圾对象。因为这些对象几乎在它们被创建之后立即就成为了不可达对象，所以每次垃圾收集能够回收掉许多内存。这将使进程的工作集保持在非常低的状态，垃圾收集器的性能自然也更好。

另一方面，如果垃圾收集器收集了第 0 代对象之后看到还有很多对象存活(也就是说回收的内存并不多)的话，垃圾收集器将把第 0 代的阈值容量向上调整，比如调整到 512 KB。这样，垃圾收集执行的频率将降低，但是每一次回收的内存将比较多。

上面仅仅讨论了每次垃圾收集执行后动态调整第 0 代阈值容量的情况，但实际上，垃圾收集器也使用类似的启发式算法来调整第 1 代和第 2 代的阈值容量。当位于这些代龄中的对象被执行垃圾收集时，垃圾收集器会查看回收的内存总量和存活的对象总量。基于这些结果，垃圾收集器会调整各个代的阈值容量，从而达到提高应用程序性能的目的。

19.8　编程控制垃圾收集器

System.GC 类型为我们的应用程序提供了直接控制垃圾收集器的一些方法。例如我们可以通过读取 GC.MaxGeneration 属性来查询托管堆支持的最大代龄。该属性目前的返回值为 2。

我们也可以通过调用下面两个静态方法中的任何一个来强制执行垃圾收集：

```
void GC.Collect(Int32 Generation)
void GC.Collect()
```

其中第一个方法允许我们指定收集某一特定代龄的对象。我们可以为其传递在 0 和 GC.MaxGeneration 之间(包括 GC.MaxGeneration)的任何整数。传递 0 将导致第 0 代对象被执行垃圾收集，传递 1 将导致第 0 代和第 1 代对象被执行垃圾收集，传递 2 将导致第 0 代、第 1 代和第 2 代对象被执行垃圾收集。无参的 Collect 方法将强制对所有代龄的对象执行垃圾收集，它等同于下面的方法调用：

```
GC.Collect(GC.MaxGeneration);
```

大多数情况下，我们应该避免调用这些 Collect 方法，而让垃圾收集器根据自己的判断来执行，并根据应用程序的实际行为调整各个代龄的阈值容量。但是，如果我们正在编写的是 CUI 或 GUI 应用程序，我们的应用程序代码将“拥有”整个进程以及进程中的 CLR。对于这些应用程序，我们可能会希望在某些时候能够强制执行垃圾收集。

例如，在用户保存了一个数据文件之后，我们可能希望能够强制执行一次完全的垃圾收集(收集所有代龄的对象)。我们也可能希望 Web 浏览器在每次页面卸载后也能执行一次完全的垃圾收集。我们还可能希望应用程序在执行一些较费时的操作时能强制执行一次垃圾收集。这里的基本思想是当应用程序执行较为耗时的操作时，垃圾收集的执行时间可以被掩盖起来。在上面 3 个应用场景中，由于应用程序的其他一些操作会占用大量的时间，所以用户将感觉不到垃圾收集的存在。

GC 类型还为我们提供了一个 WaitForPendingFinalizers 方法。该方法会挂起调用线程，直到处理终止化可达队列的线程清空了该队列，并完成每个对象的 Finalize 方法调用为止。在大多数应用程序中，我们一般没有必要调用该方法。但是，我个人却看到过如下的代码：

```
GC.Collect();
GC.WaitForPendingFinalizers();
GC.Collect();
```

上面的代码首先强制执行一次垃圾收集。当第一轮垃圾收集完成后，不需要终止化的那些对象的内存将被回收。但是终止化对象的内存还没有被回收。在第一次 Collect 调用返回后，一个特殊的、专门用于终止化的线程将采用异步的方式来调用所有终止化对象的 Finalize 方法。WaitForPendingFinalizers 方法将使应用程序线程处于睡眠状态，直到所有的 Finalize 方法调用完成为止。当 WaitForPendingFinalizers 方法返回时，所有的终止化对象将成为真正的可收集垃圾。这时，第二次 Collect 调用将强制执行第二轮垃圾收集，所有终止化对象的内存将在这一轮垃圾收集中被完全回收。

最后，GC 还提供有两个静态方法供我们确定一个对象所处的代龄：

```
Int32 GetGeneration(Object obj)
Int32 GetGeneration(WeakReference wr)
```

第 1 个版本的 GetGeneration 接受一个对象引用作为参数，第 2 个版本的 GetGeneration 接受一个 WeakReference 引用作为参数。两个方法的返回值在 0 和 GC.MaxGeneration 之间。

下面的代码可以帮助大家理解代龄的工作原理，其中也演示了如何使用上面提到的一些 GC 的方法。

```
using System;

class GenObj {
    ~GenObj() {
        Console.WriteLine("In Finalize method");
    }
}

class App {
    static void Main() {
        Console.WriteLine("Maximum generations: " + GC.MaxGeneration);

        // 在托管堆中创建一个新的 GenObj 对象
        Object o = new GenObj();

        // 因为该对象为新创建的对象，所以它的代龄应该为 0
        Console.WriteLine("Gen " + GC.GetGeneration(o));     // 0

        // 执行垃圾收集提升对象的代龄
        GC.Collect();
        Console.WriteLine("Gen " + GC.GetGeneration(o));     // 1
```

```
        GC.Collect();
        Console.WriteLine("Gen " + GC.GetGeneration(o));     // 2

        GC.Collect();
        Console.WriteLine("Gen " + GC.GetGeneration(o));     // 2 (最大值)

        o = null;      // 销毁对象的强引用

        Console.WriteLine("Collecting Gen 0");
        GC.Collect(0);                       // 收集第 0 代对象
        GC.WaitForPendingFinalizers();       // 不会调用 Finalize

        Console.WriteLine("Collecting Gen 1");
        GC.Collect(1);                       // 收集第 1 代对象
        GC.WaitForPendingFinalizers();       // 不会调用 Finalize

        Console.WriteLine("Collecting Gen 2");
        GC.Collect(2);                       // 等同于 Collect()
        GC.WaitForPendingFinalizers();       // 调用 Finalize
    }
}
```

编译并运行上面的代码，我们将会看到以下输出：

```
Maximum generations: 2
Gen 0
Gen 1
Gen 2
Gen 2
Collecting Gen 0
Collecting Gen 1
Collecting Gen 2
In Finalize method
```

19.9　其他一些与垃圾收集器性能相关的问题

本章前面在讨论垃圾收集算法时做了一个大胆的假设，即应用程序中只有一个线程在运行。但在现实世界中，经常会出现多个线程同时访问托管堆的情况(至少会有多个线程同时操作托管堆中分配的对象)。如果因为某个线程的执行而导致了垃圾收集的运行，那么其他线程将不能再访问任何对象(包括各线程自己堆栈上的对象引用)，因为垃圾收集器随时都有可能搬移这些对象、改变它们的内存地址。

因此当垃圾收集器开始运行时，所有执行托管代码的线程都必须被挂起。CLR 使用几种不同的机制来在执行垃圾收集时保证安全地挂起线程。存在多种机制的目的是为了尽可能地让线程保持运行，从而减少不必要的开销。本书不打算过多地深入探讨这里的一些细节，大家只需清楚微软在这方面确实已经做了大量的工作。随着时间的推移，这些机制将会得到持续的改进，垃圾收集器也会变得更加高效。

当 CLR 开始执行垃圾收集时，它会立即挂起进程中所有已经执行过托管代码的线程。接着，CLR 会检查每个线程的指令指针以判断线程执行到了哪里。然后，CLR 将指令指针地址和 JIT 编译器产生的表做比较，从而找出线程正在执行的代码。

如果线程的指令指针位于某个表中标识的偏移之处，那么该线程将被认为是到达了一个安全点(safe point)。一个安全点就是可以在垃圾收集执行完毕之前一直挂起线程的地方。如果线程的指令指针没有位于某个内部方法表中标识的偏移之处，那么该线程就没有到达一个安全点，CLR 也就不能执行垃圾收集。在这种情况下，CLR 会劫持(hijack)该线程——也就是改变线程的堆栈，以使其返回地址指向 CLR 内部实现的一个特殊函数。然后，该线程将被允许继续执行。当目前执行的方法返回时，CLR 内部的特殊函数将执行，从而挂起该线程。

然而，线程有时候并不能很快就从当前的方法中返回。所以在线程继续执行后，CLR 会留出大约 250 毫秒的时间等待线程被劫持。过了这个时间之后，CLR 再次挂起线程，并检测其指令指针。如果线程到达了一个安全点，那么垃圾收集就可以开始运行。如果线程仍然没有到达一个安全点，那么 CLR 将检测看是否该方法内部又调用了其他的方法。如果确实调用了其他的方法，那么 CLR 将再一次改变线程的堆栈，以使线程在最近执行的方法中返回时就能被劫持。然后，线程被允许继续执行，CLR 会再等一段时间，之后再次尝试劫持线程。

当所有的线程都到达了一个安全点、或者被劫持后，垃圾收集才可以开始运行。当垃圾收集完成后，所有的线程将被允许继续执行，应用程序也将恢复运行。被劫持的线程也会返回到原来调用它们的方法中。

注意　上面提到的算法中还有一个额外的细节需要大家注意。如果 CLR 挂起了一个线程，并检测到该线程正在执行非托管代码，那么该线程的返回地址也会被劫持，并被允许继续执行。但是，在这种情况下，即使线程仍在执行，垃圾收集器也可以启动。这样做不会出现问题是因为非托管代码不会访问未被固定(pinned)的托管对象。而一个固定对象的内存不允许被垃圾收集器任意搬移。如果一个正在执行非托管代码的线程返回到了托管代码中，那么该线程将被劫持，然后挂起，直到垃圾收集完成。

除了上面提到的一些机制(代龄、安全点、劫持)外，垃圾收集器还提供了几种额外的机制来提高对象分配和垃圾收集的性能。

19.9.1　省却同步控制的多线程分配

在一个运行着工作站版本执行引擎(MSCorWks.dll)或者服务器版本执行引擎(MSCorSvr.dll)的多处理器系统上，托管堆中的第 0 代对象会被划分到多个内存区域中，每个线程会有一个独立的区域。这使得多个线程可以同时分配内存，从而避免了托管堆的独占访问。

19.9.2　可扩展并行收集

在一个运行着服务器版本执行引擎(MSCorSvr.dll)的多处理器系统上，托管堆会被划分成几个区间，每个 CPU 会有一个独立的区间。当垃圾收集初始启动时，垃圾收集器将在每个 CPU 上有一个线程，各个线程只负责自己区间中对象的收集工作。并行垃圾收集对于服务器应用程序有着很出色的表现，因为各个工作线程的行为将趋向一致。工作站版本的执行引擎(MSCorWks.dll)不支持此项特性。

19.9.3 并发收集

在一个运行着工作站版本执行引擎(MSCorWks.dll)的多处理器系统上，垃圾收集器有一个额外的背景线程可以在应用程序运行时并发地执行垃圾收集。当一个线程因为分配新对象而使第 0 代对象的内存空间超过预设的阈值时，垃圾收集器会首先挂起所有线程，然后判定需要收集哪些代的对象。如果垃圾收集器需要收集第 0 代或者第 1 代的对象，那么它的处理方式和往常一样。但是，如果垃圾收集器需要收集的是第 2 代对象，它会放弃本应做的收集工作而继续分配新对象(换一个角度来看，也可以说第 0 代对象的阈值容量增加了)，应用程序的线程则会被允许继续运行。

当应用程序线程运行时，垃圾收集器有一个处于正常优先级的背景线程会去构造不可达对象图。该线程将和应用程序线程竞争 CPU 的时间，这会导致应用程序的执行变慢。但是，由于并发垃圾收集只运行在多处理器系统上，所以我们不会看到太大的性能损失。一旦不可达对象图构造完毕，垃圾收集器将挂起所有线程，并判断是否压缩内存。如果垃圾收集器判断需要压缩内存，那么托管堆内存将被压缩，根引用也将被修正，然后应用程序线程被允许继续运行——这样的垃圾收集花费的时间要比通常的少，因为不可达对象图已经构造好了。但是，垃圾收集器也可能决定不压缩内存。实际上，垃圾收集器更喜欢后一种方式。如果我们的系统有很多空闲内存，那么垃圾收集器将不会压缩托管堆，这会在某种程度上提高应用程序的性能，虽然这会导致应用程序进程工作集的增长。当使用并发垃圾收集时，大家一般会发现应用程序消耗的内存要比使用非并发垃圾收集时的多。

并发垃圾收集会为用户带来更好的交互体验，对于交互性的 CUI 和 GUI 应用程序来说是最好的选择。但是，对于某些应用程序，并发垃圾收集可能会损伤应用程序的性能，导致使用过多的内存。在测试应用程序时，我们应该使用并发垃圾收集和非并发垃圾收集两种方案分别进行测试，然后根据测试结果选择较好的方案。

我们可以通过创建一个包含 gcConcurrent 元素的配置文件(本书第 2 章和第 3 章曾对配置文件有过描述)来告诉 CLR 使用并发垃圾收集。下面是一个配置文件的示例：

```
<configuration>
   <runtime>
      <gcConcurrent enabled="false"/>
   </runtime>
</configuration>
```

我们也可以使用 Microsoft .NET Framework Configuration 管理工具来创建一个包含 gcConcurrent 元素的应用程序配置文件。首先打开【控制面板】，选择【管理工具】，然后调用 Microsoft .NET Framework Configuration 工具。在左边树状面板中的【应用程序】节点上，添加一个应用程序或者选择一个现存的应用程序。然后在应用程序上右击并选择【属性】，将出现如图 19.15 所示的对话框。

图 19.15　使用 Microsoft .NET Framework Configuration 管理工具为应用程序配置并发垃圾收集

点击【垃圾回收模式】下的单选按钮即可将 gcConcurrent 元素的 enabled 属性设为 true 或 false。

19.9.4　大尺寸对象

垃圾收集器还有一种提升性能的机制，即针对大尺寸对象的特殊收集策略。任何占用内存等于或超过 85,000 字节的对象都被认为是大尺寸对象(large object)。大尺寸对象从一个特殊的大尺寸对象托管堆中分配。该托管堆中对象的终止化和内存释放行为与前面描述的小对象相同。但是，大尺寸对象不会被压缩，因为在托管堆中搬移 85,000 字节以上的内存块会浪费 CPU 比较多的时间。

另外，大尺寸对象总被认为是第 2 代对象，因此我们只应该为那些需要保存很长时间的资源创建大尺寸对象。分配生存期比较短的大尺寸对象将导致垃圾收集器频繁地收集第 2 代对象，这无疑会损伤应用程序的性能。

所有这些机制对于我们的应用程序代码都是透明的。对于应用程序开发人员来说，系统看起来就好像只有一个托管堆。这些机制存在的惟一目的就是提高应用程序的性能。

19.10 监视垃圾收集

当我们安装.NET框架时，它会安装一组性能计数器为CLR的一些操作提供许多实时的统计数据。这些统计数据可以通过和 Windows 一起发布的 PerfMon.exe 工具或者 System Monitor ActiveX 控件看到。访问 System Monitor 控件最简单的方式就是运行 PerfMon.exe，然后选择＋工具栏按钮，我们将会看到如图 19.16 所示的【添加计数器】对话框。

图 19.16 PerfMon.exe 中显示的.NET CLR 内存计数器

要监视 CLR 的垃圾收集器，首先选择.NET CLR Memory 性能对象。然后再从实例列表框中选择一个应用程序。最后，选择我们感兴趣的计数器集合，并单击【添加】按钮。若想了解某一特定计数器的含义，大家可以选择该计数器，并单击【说明】按钮。关闭该对话框后，【系统监视器】会图示化我们所选择的实时统计数据。