[**CLR和.Net对象生存周期**](http://www.cnblogs.com/Wddpct/p/5547765.html)

标签：GC .Net C# CLR

* [前言](http://www.cnblogs.com/Wddpct/p/5547765.html#0.)
* [1. 基础概念明晰](http://www.cnblogs.com/Wddpct/p/5547765.html#1.)  
  \* [1.1 公告语言运行时](http://www.cnblogs.com/Wddpct/p/5547765.html#1.1)  
  \* [1.2 托管模块](http://www.cnblogs.com/Wddpct/p/5547765.html#1.2)  
  \* [1.3 对象和类型](http://www.cnblogs.com/Wddpct/p/5547765.html#1.3)  
  \* [1.4 垃圾回收器](http://www.cnblogs.com/Wddpct/p/5547765.html#1.4)
* [2. 垃圾回收模型](http://www.cnblogs.com/Wddpct/p/5547765.html#2.)  
  \* [2.1 为什么需要垃圾回收](http://www.cnblogs.com/Wddpct/p/5547765.html#2.1)  
  \* [2.2 什么时候进行垃圾回收](http://www.cnblogs.com/Wddpct/p/5547765.html#2.2)  
  \* [2.3 垃圾回收时发生了什么](http://www.cnblogs.com/Wddpct/p/5547765.html#2.3)  
  \* [2.4 GC为我们解决了什么问题](http://www.cnblogs.com/Wddpct/p/5547765.html#2.4)  
  \* [2.5 代数的概念（Generation）](http://www.cnblogs.com/Wddpct/p/5547765.html#2.5)  
  \* [2.6 使用System.GC类控制垃圾回收](http://www.cnblogs.com/Wddpct/p/5547765.html#2.6)  
  \* [2.7 非托管对象资源回收](http://www.cnblogs.com/Wddpct/p/5547765.html#2.7)
* [3. 总结](http://www.cnblogs.com/Wddpct/p/5547765.html#3.)
* [4. 参考资料](http://www.cnblogs.com/Wddpct/p/5547765.html#4.)

**前言**

对象的生存周期和垃圾回收一直是容易被我们忽略的知识点，因为我们现在高级语言编程平台太“智能”了，自动的异常处理，内存管理，线程同步，以至于我们中的大部分人只需要按部就班面向对象编程就能完成大部分的工作——写接口的时候继承一个IDisposable，释放文件占用的时候强制Close一下，异步编程就用Async和Await……

比如最近结合ABP框架写Web Api项目的时候，对于最重要的两个消息处理对象[HttpRequestMessaga和HttpResponseMessage的释放过程](http://www.cnblogs.com/artech/p/asp-net-web-api-active-release-controller.html)，我几乎完全不用知道他们的生存环境，只要在后台写好对应的逻辑代码即可。如果我们不了解这些东西，只是遵循规范在使用的话，或许也能写出好看的代码，但这和程序员钻研的精神就不符合了。所以趁着小组内的讲课机会，我整理了下以前积累的一些读书和博客笔记，将我对于这些基础知识点的理解概括了一下，主要讨论下.Net平台上的一些常见概念，以及应用程序如何构造新对象，包括对象的生命周期和回收工作。希望能够为大家写出更优雅的代码，更深入地理解.Net平台提供一点微小的帮助

*Tips1*：因为本人水平有限，同时也是为了社区的和谐发展，本博文将尽量不涉及不同语言和平台之争，最多只是比较下不同语言间的异同。不过有兴趣的JRs可以看看赵三本的[《Why Java Sucks and C# Rocks》](http://blog.zhaojie.me/2010/04/why-java-sucks-and-csharp-rocks-1-thoughts-and-goals.html)系列，至少对理解C#的一些特性还是挺有帮助的。

外站引用图片点击可跳转源链接，其他所有图示都由Visio作出。

**1. 基础概念明晰**

**1.1 公共语言运行时**

顾名思义，公共语言运行时（Common Language Runtime，CLR）是一个可以由多种编程语言使用的运行时，如同java的JVM（Java Virtual Machine）。CLR的核心功能包括**内存管理，程序集加载，类型安全，异常处理和线程同步**，而且还负责对代码实施严格的类型安全检查，保证代码的准确性，这些功能都可以提供给面向CLR的所有语言（C#，F#等）使用。

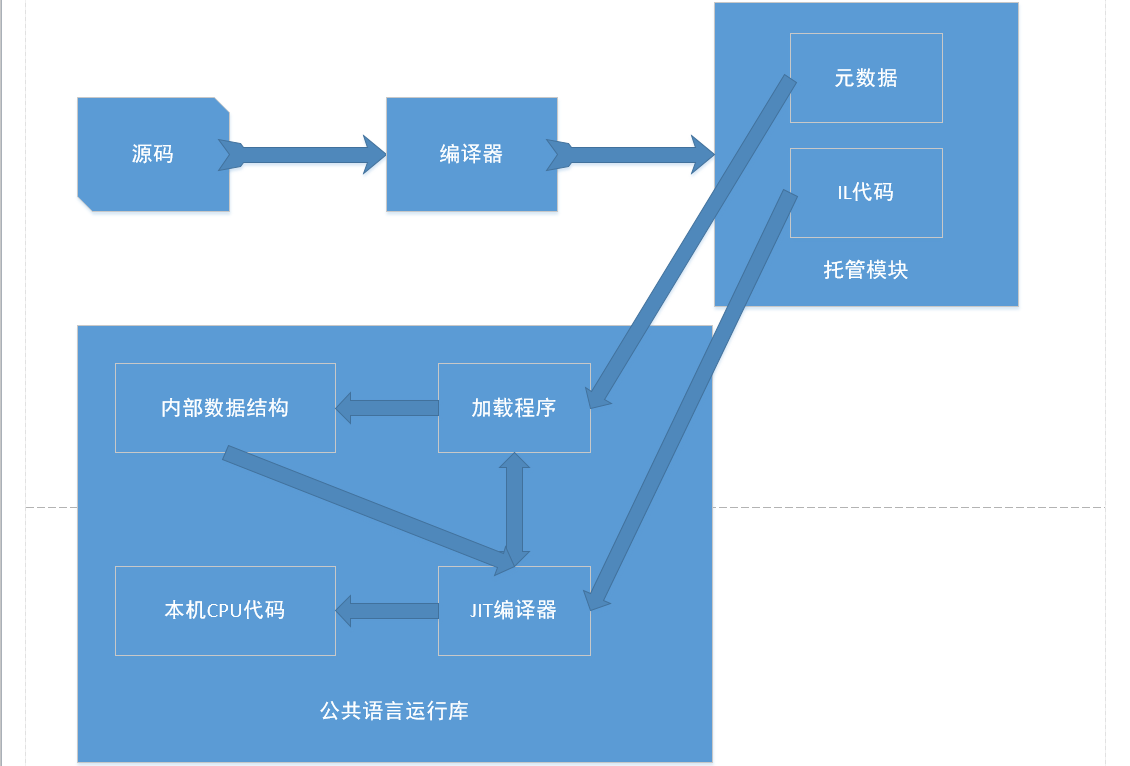
.NET Framework 的版本号无需对应于它所包含的 CLR 的版本号。以下给出两个版本号关联表，详情参阅[.NET Framework 版本和依赖关系](https://msdn.microsoft.com/zh-cn/library/bb822049.aspx)

| **.NET Framework** | **CLR** |
| --- | --- |
| 1.0 | 1.0 |
| 1.1 | 1.1 |
| 2.0 | 2.0 |
| 3.0 | 2.0 |
| 3.5 | 2.0 |
| 4 | 4 |
| 4.5.x | 4 |
| 4.6.x | 4 |

涉及到.Net Core当中的CoreCLR和目前.Net Framework上的CLR的比较，大家可以参见  
[.NET Core has two major components. It includes a small runtime that is built from the same codebase as the .NET Framework CLR. The .NET Core runtime includes the same GC and JIT (RyuJIT), but doesn’t include features like Application Domains or Code Access Security. The runtime is delivered on NuGet, via the Microsoft.CoreCLR package.](https://link.zhihu.com/?target=http%3A//stackoverflow.com/questions/29820947/whats-the-difference-between-asp-net-5-net-core-and-asp-net-core-5/29821705%2329821705)  
以及  
[CoreCLR started as a copy of CLR. It has been modified to support different OSes. They're maintained separately and in parallel.](http://stackoverflow.com/questions/34665026/whats-the-difference-between-net-coreclr-corert-roslyn-and-llilc)  
可以看到两者并没有什么特别变化，内存管理，GC，线程同步的机制也都是类似的（毕竟CoreCLR原先就是由CLR的版本分支出去的，详见[CoreCLR官方Git](https://github.com/dotnet/coreclr)），更多的其实是在服务器OS的优化（GC，GIT等）下了功夫。特别是在当前CoreCLR学习资料比较少的情况下，开发人员把.Net Framework实现的CLR搞搞懂也就差不多了。

**1.2 托管模块**

CLR并不关心开发人员使用什么语言来进行编程，只要我们使用的编译器（充当语法检查器和‘正确代码’分析器）是面向CLR的就行。常见的语言编译器包括C++/CLI，C#，F#，VB和一个中间语言汇编器（Intermediate Language，IL） ，以下是编译器编译代码的过程，可以看到最终都是生成包含中间代码（IL）和托管数据（可进行垃圾回收的数据类型）的**托管模块**。

下图代表CLR将源代码编译成托管模块并最终运行，其中JIT将IL代码转换成本机CPU指令  


那托管模块是标准的32位或64位Microsoft Windows可移植执行体文件，主要由以下几部分组成

* PE32或PE32+
* CLR头
* **元数据**
* **IL代码（基于栈，也称为托管代码）**

**什么是托管代码和非托管代码**  
托管代码：由公共语言运行库环境（而不是直接由操作系统）执行的代码。托管代码应用程序可以获得公共语言运行库服务，例如自动垃圾回收、运行库类型检查和安全支持等。这些服务帮助提供独立于平台和语言的、统一的托管代码应用程序行为。  
非托管代码：在公共语言运行库环境的外部，由操作系统直接执行的代码。非托管代码必须提供自己的垃圾回收、类型检查、安全支持等服务；它与托管代码不同，后者从公共语言运行库中获得这些服务。例如COM/COM++组件,ActiveX控件,API函数,指针运算,自制的资源文件，一般情况下我们会采取手动回收，如调用Dispose接口或使用using包裹逻辑块，

**1.3 对象和类型**

CLR支持两种类型，引用类型和值类型。  
引用类型总是从托管堆分配，每次我们通过使用new操作符返回对象内存地址——即指向对象数据的内存地址，而后把这个内存地址pop进线程栈中。为了避免每次实例化对象都要进行一次内存分配，CLR也为我们提供了另一种轻量级类型——值类型，值类型的实例一般在线程栈上直接分配，不同于引用类型变量中包含指向实例的地址，值类型变量中直接就包含了实例本身的字段。  
两种类型具体的比较和扩展就不在这里延伸了，唯一要重申的就是引用类型总是处于已装箱状态。

下图描述了值类型和引用类型关系图（字符串不可变的原理）

*Tips*：进程初始化时，CLR会自动划出一个地址空间区域作为托管堆（相对于本机堆的说法，是由一个由CLR访问的随即内存块）。每个托管进程都有一个托管堆，进程中的所有线程都在同一堆上分配对象记忆。这里还涉及到一个重要的指针，Jeffrey将称为[NextObjPtr](https://blogs.msdn.microsoft.com/microsoft_press/2012/11/29/new-book-clr-via-c-fourth-edition/)，由CLR进行维护，该指针指向下一个对象在堆中的分配位置。  
对于托管堆而言，分配一个对象只是修改NextObjPtr指针的指向，这个速度是非常快的。事实上，在托管堆上分配一个对象和在线程栈上分配内存的速度很接近。

**不妨把托管堆想象成是一间房子，入住的对象一开始都是有门卡（和引用类型的变量关联证明）的房客，后来因为不交钱了（失去了关联证明）就被赶出来了**，详细的交互过程会在之后说明。

CLR要求所有对象（主要指引用类型）都用new操作符创建，new操作符在完成[四步操作](http://www.cnblogs.com/guochenkai/p/3880070.html)以后，会返回指向托管堆上新建对象的一个引用（或指针，视情况而定）,在使用完以后，C#并没有如C++对应的delete操作符来删除对象，也就是说，开发人员是没有办法显示释放为对象分配的内存，但是CLR采用了垃圾回收机制，能够自动检测到一个对象是否可达，并且自动释放资源。

**1.4 垃圾回收器**

垃圾回收器（Garbage Collector）简称GC，采用引用跟踪算法，在CLR中用作自动内存管理器，用于控制的分配和释放的托管内存。刚才的堆比作是房子的话，GC就是堆的清洁工。它主要为开发人员提供以下作用

* 开发应用程序时不必释放内存。
* 有效分配托管堆上的对象。
* 回收不再使用的对象，清除它们的内存，并保留内存以用于将来分配。托管对象会自动获取干净的内容来开始，因此，它们的构造函数不必对每个数据字段进行初始化。
* 通过确保对象不能使用另一个对象的内容来提供内存安全。

垃圾回收器跟踪并回收托管内存中分配的对象。垃圾回收器会定期执行垃圾回收来回收内存分配给对象没有有效的引用。当无法满足内存要求，使用可用的可用内存（如new 时发现内存占满），垃圾回收时会自动发生。或者，应用程序可以强制垃圾收集使用 Collect 方法。  
整个垃圾回收过程包括以下步骤 ︰

* 垃圾回收器搜索托管代码中引用的托管对象。
* 垃圾回收器尝试完成未被引用的对象。
* 垃圾回收器释放未被引用的对象，并回收它们的内存。

结合托管堆，.Net已经为开发人员提供了一个很简便的编程模型：分配并初始化内存直接使用。大多数类型并不需要我们进行资源清理，GC会自动释放内存。只是针对于一些特殊对象时，如文件占用，数据库连接，开发人员才需要手动销毁资源占用空间。

**2. 垃圾回收模型**

经过了上面基础概念明晰的讲解，想必大家已经对整个.Net平台上的代码编写，编译和运行过程有了一个简单的认识，接下来就让我们更加深入地了解下整个回收模型。

**2.1 为什么需要垃圾回收**

我们始终要明确一个概念，为什么我们需要垃圾回收——这是因为我们的运行环境内存总是有限的。当CLR在托管堆上为非垃圾对象分配地址空间时，总是分配出新的地址空间，且呈连续分配。也正因为这种引用的“局部化”（工作集的集中+对象驻留在内存中），托管堆的性能是极快的，但这毕竟是基于“内存无限”而言。实际环境中内存总是有限的（或者期待Intel和Google实现内存无限的黑科技），所以CLR才通过GC的技术删除托管堆中不再使用的数据对象。

**2.2 什么时候进行垃圾回收**

当满足以下条件之一时CLR将发生垃圾回收：

1. 系统具有低的物理内存。
2. **由托管堆上已分配的对象使用的内存超出了可接受的阈值（即将涉及到代的概念）**。随着进程的运行，此阈值会不断地进行调整。
3. 强制调用 GC.Collect 方法。
4. CLR正在卸载应用程序域（AppDomain）
5. CLR正在关闭。

*Tips*：对于未装箱的值类型对象而言，由于其不在堆上分配，一旦定义了该类型的一个实例的方法不再活动，为它们分配的存储资源就会被释放，而不是等着进行垃圾回收

**2.3 垃圾回收时发生了什么**

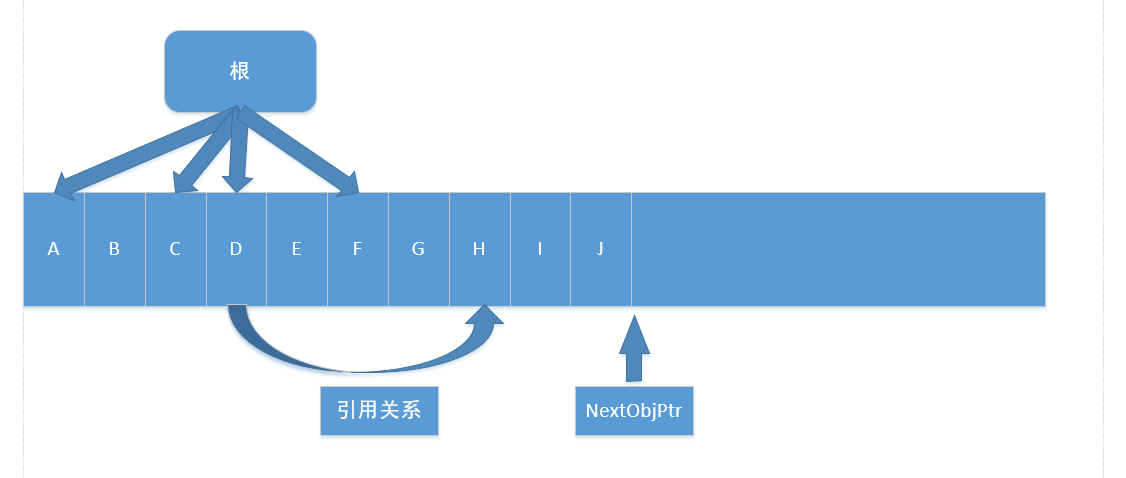
上文提到GC是一种**分代式垃圾回收器**（[同JVM](http://wenku.baidu.com/view/8fce26f6ba0d4a7302763a27.html)，具体处理上有差异)，使用引用计数算法，该算法只关心引用类型变量，下文中统一将该类变量称为**根**。

*Tips*：所有的全局和静态对象指针是应用程序的根对象，另外在线程栈上的局部变量/参数也是应用程序的根对象，还有CPU寄存器中的指向托管堆的对象也是根对象。

具体流程如下：

* GC的准备阶段  
  在这个阶段，CLR会暂停进程中的所有线程，这是为了防止线程在CLR检查根期间访问堆。
* GC的标记阶段  
  当GC开始运行时，它会假设托管堆上的所有对象都是垃圾。也就是说，假定没有根对象，也没有根对象引用的对象，然后GC开始遍历根对象并构建一个由所有和根对象之间有引用关系对象构成的**对象图**，然后，GC会挨个遍历根对象和引用对象，假如一个根包含null，GC会忽略这个根并继续检查下个根（这很关键）。反之，假如根引用了堆上的对象，GC就会标记那个对象并加入对象图中。如果GC发现一个对象已经在图中就会换一个路径继续遍历。这样做有两个目的：一是提高性能，二是避免无限循环。

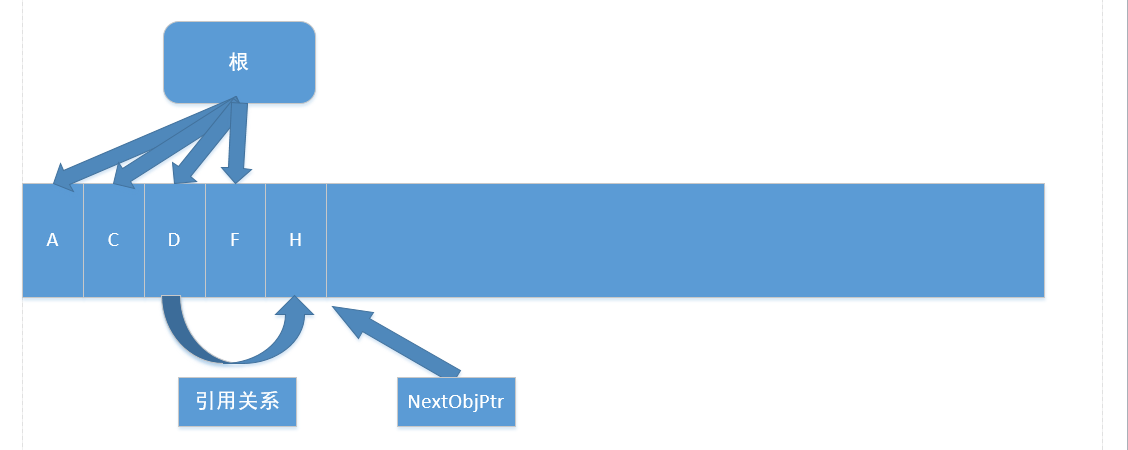
*Tips*：将引用赋值为null并不意味着强制GC立即启动并把对象从堆上移除，唯一完成的事情是显式取消了引用和之前 引用所指向对象之间的连接。

如下图所示，根直接引用了对象A，C，D，F。标记对象D时，垃圾回收器发现这个对象含有一个引用对象H的字段，所以H也会被标记，整个过程一直持续到所有根检查完毕。下图是**回收之前的托管堆模型**  


这里我们还注意到了NextObjPtr对象始终保持指向最后一个对象放入托管堆的地址。

*Tips*：等标记过程结束后，堆中的对象只有标记和未标记两种状态，由上文标记规则我们可以知道，被标记的对象至少被一个根引用，我们把这种对象称为**可达**（也称为幸存），反之称为**不可达**。

* GC的碎片整理阶段  
  所有的根对象都检查完之后，GC构建的对象图中就有了应用程序中所有的可达对象。托管堆上所有不在这个图上的对象就是要做回收的垃圾对象了。同时，CLR会对堆中非垃圾对象进行位置上的整理，使它们覆盖占用连续的内存空间（这个动作还伴随着对根返回新的内存地址的行为），这样一方面恢复了引用的“局部化”，压缩了工作集，同时空出了空间给其他对象入住，另外也解决了本机堆的空间碎片化问题。
* GC恢复阶段  
  完成了综上的所有操作后，CLR也恢复了原先暂停的所有线程，使这些线程可以继续访问对象。

下图是**回收之后的托管堆模型**  


可以看到不可达的BEGIJ对象都已经被回收了，并且可达对象的位置也重新排列了，NextObjPtr依然指向最后一个可达对象之后的位置，为CLR下一次操作对象标识分配位置。

**2.4 GC为我们解决了什么问题**

通过以上描述可知，不同于C/C++需要手动管理内存，GC的自动垃圾回收机制为我们解决了可能存在的**内存泄漏**和因为访问被释放内存而造成的**内存损坏**的问题。

**2.5 代数的概念（Generation）**

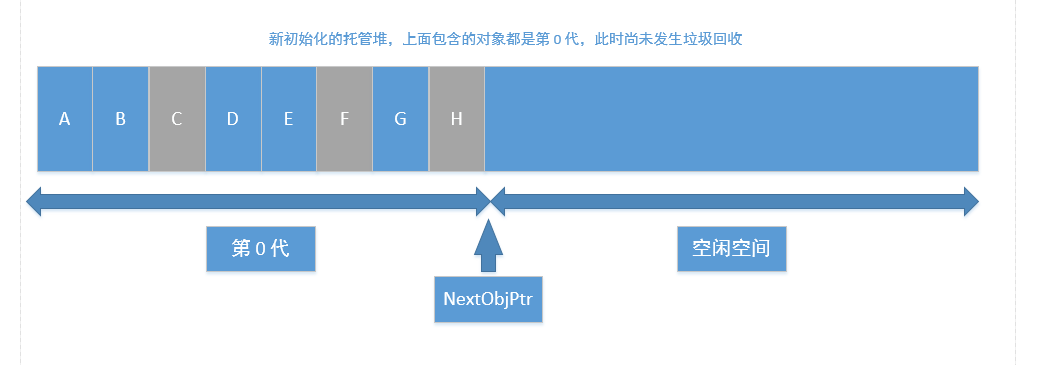
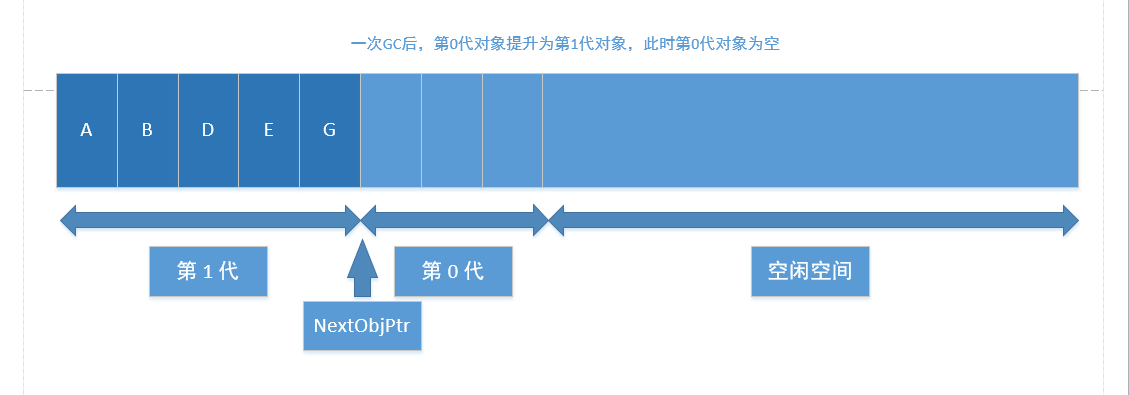
如流程描述一样，垃圾回收会有显著的性能损失，这是使用托管堆的一个明显的缺点。上文中曾提到CLR的GC是基于代的分代式垃圾回收器，而代就是一种为了降低GC对性能影响的机制，代的设计思路也很简单：

* 对象越新，生命周期越短，反之也成立
* 回收托管堆的一部分，速度快于回收整个堆

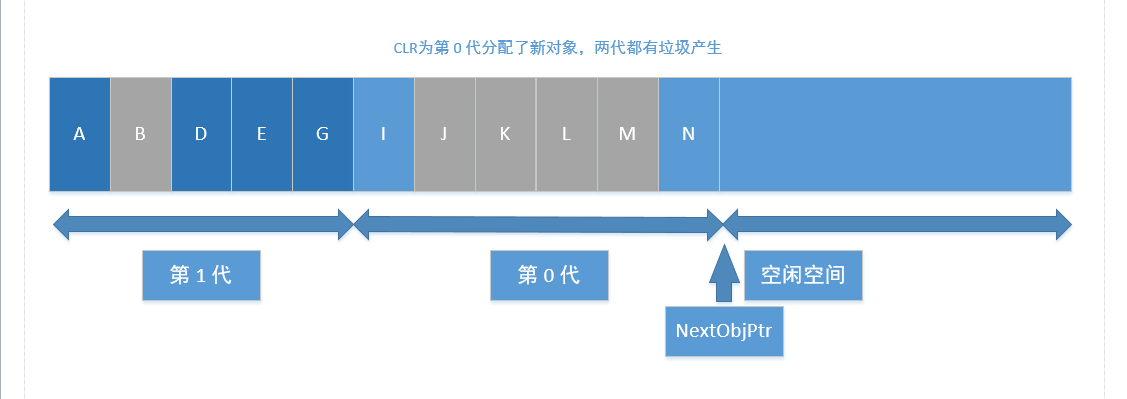
基于以上假设，托管堆中的每个对象都可以被分为0、1、2三个代（System.GC.MaxGeneration=2）：

* 第 0 代： 从没有被标记为回收的新分配对象
* 第 1 代： 在上一次垃圾回收中没有被回收的对象
* 第 2 代： 在一次以上的垃圾回收后仍然没有被回收的对象.

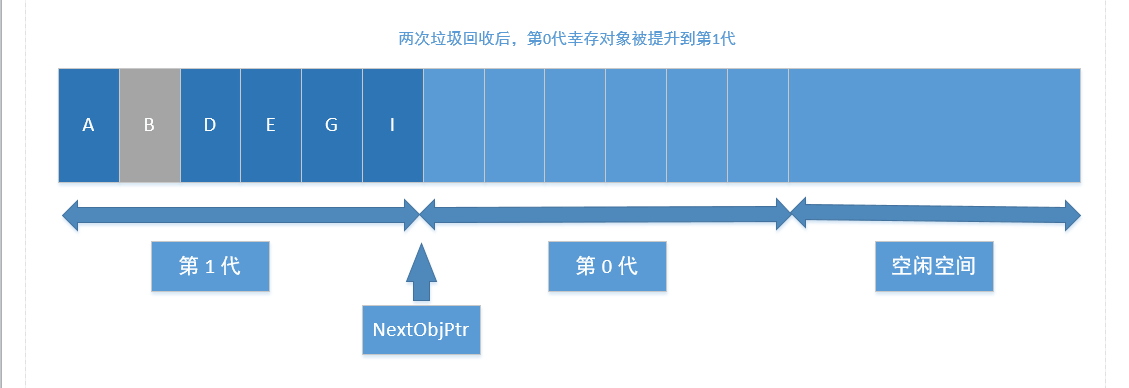
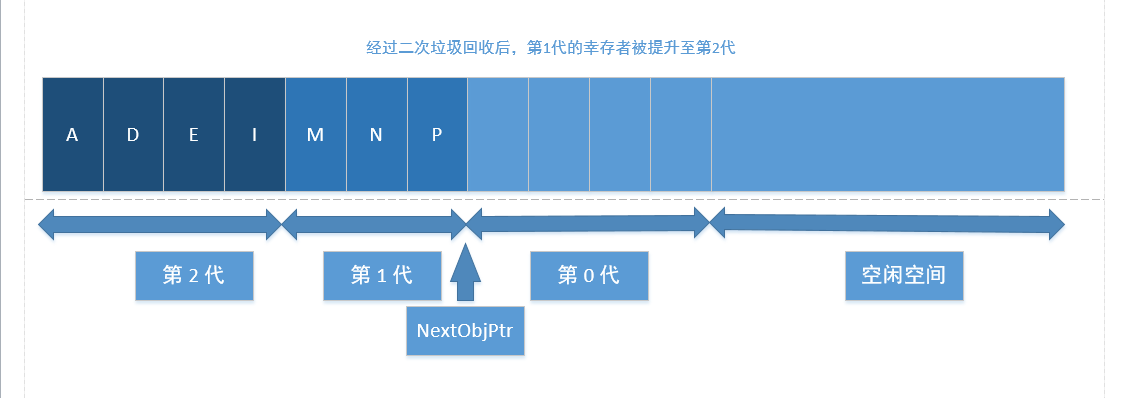
让我们用一些图示具体看看代的工作原理吧

1. 托管堆在程序初始化时不包含对象，这时候添加到堆的对象就是第 0 代对象，这些对象并未经历过GC检查。一段时间后，C，F，H对象被标记为不可达。  
   
2. CLR初始化时为第0代对象选择一个预算容量，假如这时分配一个新对象造成第0代超过预算，此时CLR就会触发一次GC操作。比如说A-H对象正好用完了第 0 代的空间，此时再操作时就会引发一次GC操作。GC后第 0 代对象不包括任何对象，并且第一代对象也已经被压缩整理到连续的地址空间中。  
   

*Tips*：垃圾回收发生于第 0 代满的时候

1. 每次新对象仍然会被分配到第 0 代中，如下图所示，CLR又重新分配了I-N对象，一段时间后，第 0 代和第 1 代都产生了新的垃圾对象  
   

*Tips*：CLR不仅为第 0 代对象选择了预算，也为第 1 代，第 2 代对象选择了预算。  
不过由于GC是自调节的，这意味着GC可能会根据应用程序构造对象的实际情况调整每代的预算（每次GC后，发现对象多存活增加预算，发现少存活减少预算），这样进程工作集大小也会实时不同，进一步优化了GC性能。

1. 疾射此时CLR再为第 0 代对象加入新对象时造成超过第 0 代预算的情况，GC将重新开启。GC将检查第 1 代预算使用情况，**假如第 1 代占用内存远少于预算，GC将只检查第 0 代对象，即便此时原来的第 1 代对象中也出现了垃圾对象**。这符合假设中的第一点，同时GC也不用再遍历整个托管堆，从而优化了GC操作性能。  
   
2. 此后，CLR仍然是按照规则对第 0 代分配对象，知道第 0 代预算被塞满才会发生垃圾回收，把对象补充到第 1 代中，此时分两种情况，假如第 1 代对象空间仍然小于预算，此时第 1 代中的垃圾对象仍然不会进行回收（如4图中所示）。假如第 1 代对象在某个时间段增长到超过预算的阶段，那么CLR将在下一次进行GC回收时，检查第 1 代对象，然后统一回收第 0 代和第 1 代中的垃圾对象。回收以后，第 0 代的幸存对象提升到第 1 代，第 1 代的幸存对象提升到了第 2 代。此时第 0 代回归空余状态  
   

6.至此，CLR已经进行了数次GC操作才最终将对象分配到了第 2 代中

**2.6 使用System.GC类控制垃圾回收**

MSDN上对System.GC类的定义是

控制系统垃圾回收器（一种自动回收未使用内存的服务）。  
上文也提到垃圾回收触发条件之一就是代码显示调用此类下的Collect方法，我们具体用代码结合下**代**的知识演示下

public class Person

{

public string Name { get; set; }

public int Age { get; set; }

}

class Program

{

private static void Main(string[] args)

{

Console.WriteLine("托管堆上分配字节数: {0}", GC.GetTotalMemory(false));

Console.WriteLine("当前系统支持的最大代数", GC.MaxGeneration);

Person person = new Person { Name = "Jeffrey", Age = 100 };

Console.WriteLine(person.ToString());

Console.WriteLine("Generation of s is: {0}", GC.GetGeneration(person));

GC.Collect();

GC.WaitForPendingFinalizers();//等待对象被终结，推荐每次调用Collect方法使用该方法

Console.WriteLine("Generation of s is: {0}", GC.GetGeneration(person));

GC.Collect();

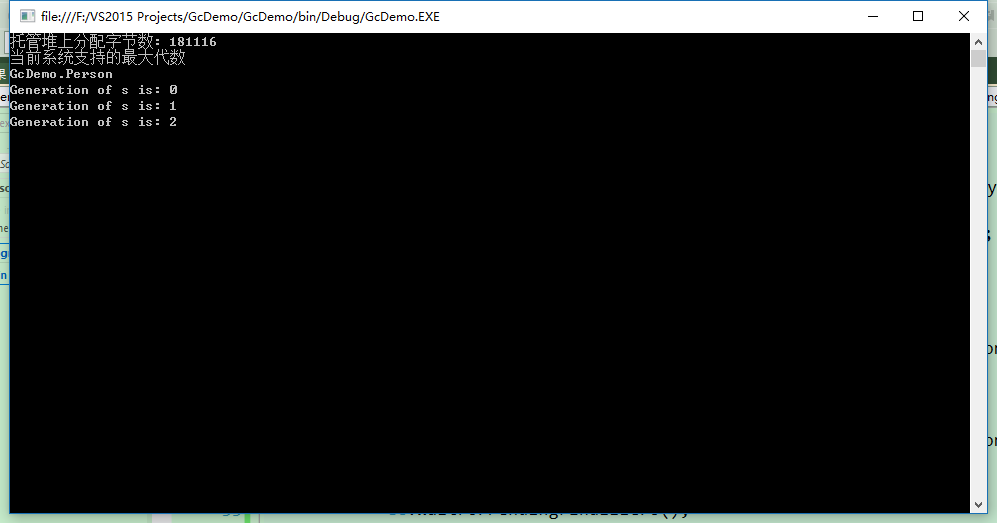
GC.WaitForPendingFinalizers();

Console.WriteLine("Generation of s is: {0}", GC.GetGeneration(person));

Console.ReadKey();

}

}

运行结果如下，可以发现每次回收后，未被回收对象的代都增加了1  


**2.7 非托管对象资源回收**

至此我们大概了解了GC的工作原理和常见垃圾回收的条件和调用方法，对于CLR而言，大多数类型只要分配了内存就能够正常工作，但有的类型除了内存还需要本机资源，比如说常用的FileStream，便需要打开一个文件（本机资源）并保存文件句柄，或者是数据库连接信息，那么我们就需要显式释放非托管对象，因为GC仅能跟踪托管堆上的内存资源。这就引伸出了可终结对象（Finalize）和可处置对象（IDisposable）这两种处理方式

**2.7.1 可终结对象（Finalize）**

当包含本机资源的类型被GC时，GC会回收对象在托管堆上使用的内存，同时提供了一种称为**终结器(Finalization)**的机制，允许对象在判定为垃圾之后，在对象内存在回收之前执行一些代码。当一个对象被判定不可达后，对象将终结它自己，并释放包装着的本机资源，之后，GC再从托管堆中回收对象。

*Tips*：这里的类型都还指的是托管堆上的引用类型

在.NET基类System.Object中, 定义了名为Finalize()的虚方法。开发人员可以重写Object基类的Finalize方法，GC判定对象不可达后，会调用重写的该方法，重写方式如下，类似于C++的析构器写法。

class Finalization{

~Finalization()

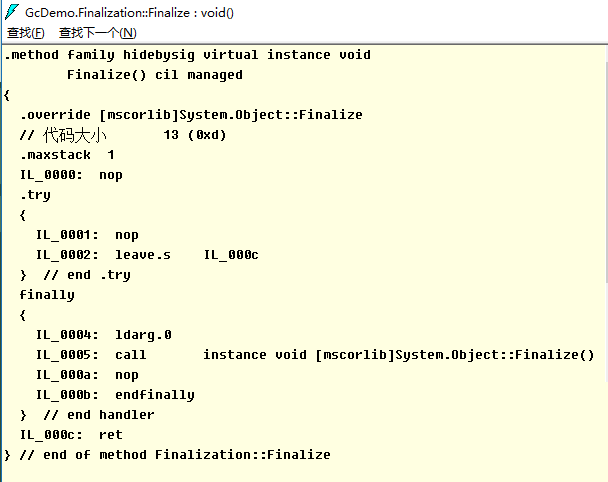
{

//这里的代码会进入Finalize方法

Console.WriteLine("Enter Finalize()");

}

}

以下是Finalize的IL代码，通过查看Finalize的IL代码，可以看到主体的代码放到了一个try 块中，而基类方法则在finally 块中被调用。  


*Tips1*：这些不可达的对象都是在GC完成以后才调用Finalize方法，所以这些对象的内存不是被马上回收的，并且会被提升到下一**代**，这增大了内存损耗，并且Finalize方法的执行时间无法控制，所以原则上并不提倡使用终结器机制，GC调用Finalize方法的[内部实现](https://msdn.microsoft.com/zh-cn/library/system.object.finalize(v=vs.110).aspx)不在这里赘述了。其实重写Finalize方法的必要原因就是C#类通过平台调用或复杂的COM组件任务使用了非托管资源。

*Tips2*：本机资源的清理最终总会发生

如果你必须要使用Finalize的话，Jeffrey给出的建议是“确保Finalize方法尽可能快的执行，要避免所有可能引起阻塞的操作，包括任何线程同步操作，同时也要确保Finalize方法不会引起任何异常，如果有异常垃圾回收器会继续执行其他对象的Finalize方法直接忽略掉异常”。

**2.7.2 可处置对象（IDisposable）**

上文提到Finalize的一些不可避免的缺点，特别是Finalize方法的执行时间是无法控制的，所以假如开发人员想要尽可能快地手动清除本机资源时，可以实现IDisposable接口, 它定义了一个名为Dispose()的方法。这也是我们熟悉的开发模式，比如FileStream类型便实现了IDisposable接口，所以具体的使用这里便不再赘述。只是需要额外说明的是，并不一定要显式调用Dispose方法，才能保证非托管资源得到清理，调用Dispose方法只是**控制这个清理动作的发生时间**而已。同样的，Dispose方法也不会将托管对象从托管堆中删除，我们要记住在正常情况下，只有在GC之后，托管堆中的内存才能得以释放。我们的习惯用法是将Dispose方法放入try finally的finally块中，以确保代码的顺利执行

class Program

{

static void Main(string[] args)

{

FileStream fs = new FileStream("temp.txt",FileMode.Create);

try

{

var charData = new char[] {'1', '2', '3'};

var bytes = new byte[charData.Length];

Encoder enc = Encoding.UTF8.GetEncoder();

enc.GetBytes(charData, 0, charData.Length, bytes, 0, true);

fs.Seek(0, SeekOrigin.Begin);

fs.Write(bytes, 0, bytes.Length);

}

finally

{

fs.Dispose();

}

Console.WriteLine("Success!");

}

}

C#语言也为我们提供了一个using语句，它允许我们使用简单的语法来获得和上述代码相同的效果，查看IL代码也发现具有相同的try finally块，具体就不演示了。

*Tips*：using语句只适用于那些实现了IDisposable接口的类型

**3. 总结**

至此，我们把CLR，托管堆，GC操作触发条件，基于代的GC的内部实现机制，显式释放资源操作都走马观花地整理了一遍。考虑到实际使用中，我们并不会太过于关注一些不常见的用法，所以诸如Finalize的实现细节，以及垃圾回收模式等知识文中也就没有提及，有兴趣的博友可以去[MSDN](https://msdn.microsoft.com/zh-cn/library/system.object.finalize(v=vs.110).aspx)或者翻阅相关书籍扩展下。

对GC实际的理解上，我更喜欢把CLR比作是房东，将托管堆比作是一间大公寓，每次有对象（根）在CLR登记后，CLR就会给它提供一个身份证明（引用地址），记录到房客租赁登记表上（线程栈）。因为这件大公寓空间仍然是有限的，房客的重要性也不一样，所以大公寓将不同的房间划分为天字号，地字号，人字号三种房间（选择预算），房东比较重感情，所以刚来的房客嘛，管你有钱没钱，先给我去人字号带着。每次人字号房间不够住的时候，房东就会安排清理工（GC）来安排房间归属了。对人字号房间的房客，清理工会一个个检查过去，看看有没有房客和房东关系疏远了（不可达），这些没心没肺的（也可能是房东主动提出绝交）全都滚出去，那些剩下来的再安排到地房间去。假如地字号房间没满，清理工就不检查了（代的性能优化），满了再依旧安排。假如你是地字号的，就算你和房东绝交了，也会考虑再让你住些日子。那如果有时候发现一些房客就是暂住下，人数又多，离开又早，那清理工就会调整下房间，把各层级的房间数目再分配下。

匆忙之作，欢迎勘误，不胜感激。

**4. 参考资料**

* [什么是.NET？](http://www.cnblogs.com/shuicao/archive/2012/03/19/2406945.html)
* [.NET 对象生命周期](http://www.cnblogs.com/liusuqi/p/3155942.html)
* [MSDN Magazine Issues and Downloads](https://msdn.microsoft.com/magazine/msdn-magazine-issues)
* [改善C#程序的建议4：C#中标准Dispose模式的实现](http://www.cnblogs.com/luminji/archive/2011/03/29/1997812.html)
* [Fundamentals of Garbage Collection](https://msdn.microsoft.com/zh-cn/library/ee787088(v=vs.110).aspx#the_managed_heap)
* [C# Finalize和Dispose的区别](http://www.cnblogs.com/Jessy/articles/2552839.html)