**[.NET对象生命周期小结](http://www.cnblogs.com/Allen-Li/archive/2013/01/03/2843288.html)**

参考资料

《C#与.NET4高级编程设计（第五版）》

MSDN

主要内容

[new关键字](http://www.cnblogs.com/Allen-Li/archive/2013/01/03/2843288.html#partOne)

[对象的代](http://www.cnblogs.com/Allen-Li/archive/2013/01/03/2843288.html#partTwo)

[垃圾回收过程](http://www.cnblogs.com/Allen-Li/archive/2013/01/03/2843288.html#partThree)

[强制垃圾回收](http://www.cnblogs.com/Allen-Li/archive/2013/01/03/2843288.html#partFour)

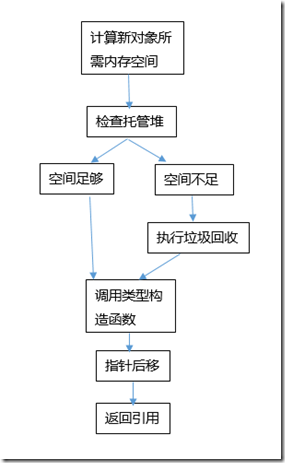
[Finalize与Dispsose方法](http://www.cnblogs.com/Allen-Li/archive/2013/01/03/2843288.html#partFive)

[using语法](http://www.cnblogs.com/Allen-Li/archive/2013/01/03/2843288.html#partSix)

new关键字

new关键字返回的是一个指向堆上对象的引用，并非对象本身，该引用存储在栈内。

new一个对象的过程：

[](http://images.cnitblog.com/blog/343422/201301/03200319-f0a46b8faf7043e6b97914d70d7e5bee.png)

对象的代

垃圾回收时，CLR并不会检测托管堆上的每一个对象，这样会花费大量时间。为了优化检测过程，堆上的每一个对象都属于“某代"：

第0代：从没被标记为回收的新分配的对象

第1代：上一次垃圾回收被标记为可回收但没有被回收的对象

第2代：在一次以上的垃圾回收后没有被回收的对象

代的设计思路是：对象在内存中存在的时间越长，他就更可能应该保留。

垃圾回收首先检测第0代，当第0代回收部分对象后以后足够内存空间为新对象分配，则幸存的标记为可回收未被回收的对象升级至第一代；如果第0代可回收对象全部回收后，内存空间仍不足以保存新对象，则检测第1代对象，若得到了足够内存空间，则第1代幸存对象将升至第2代；如果第1代可回收对象全部回收，内存仍不足，则检测第2代对象，但第2代幸存的对象仍旧是第2代。

垃圾回收过程

CLR通过创建对象图来检测托管堆上的对象是否可被应用程序访问，同一对象不会在对象图上出现两次，未在对象图上出现的对象，说明应用程序已不可访问该对象，它们将被从内存中清除。垃圾回收并不会回收所有不可访问的对象，当有了足够的内存空间提供给新对象，则停止回收。

强制垃圾回收

以下情况可能需要强制垃圾回收：1. 应用程序进入一段代码，而这段代码不希望被可能的垃圾回收中断；2. 应用程序刚分配了很多对象，开发人员想要尽可能多地回收对象以获得内存。

方式：

GC.Collect(); // 强制垃圾回收，同样可传入一个数值来指定要回收的最老的代

GC.WaitForPendingFinalizers(); // 等待垃圾回收结束

Finalize与Dispsose方法

**Finalize方法**

该方法的主要作用为保证.NET对象能在垃圾回收时清除非托管资源。

该方法为一个受保护的虚方法，无法直接调用，重写该方法的唯一原因是，C#类通过PInvoke或者复杂的COM互操作任务使用了非托管资源。

要写自定义的Finalize方法，不能用override进行重写，要通过析构函数来写，析构函数不接受访问修饰符、参数和重载，函数名以 ~ 为前缀。

Class MyFinalizeSample

{

    ~MyFinalizeSample()

    {

        // 清理非托管资源

    }

}

Finalize可以保证对象可以清楚非托管资源，但它的效率非常低，因为在终结这个对象时，至少会进行两次垃圾回收。原因为：当在托管堆上分配对象时，运行库会确定该对象是否包含自定义的Finalize方法，如果包含，这个对象会被标记为“可终结”对象，并在一个由垃圾回收器维护的“终结队列”中保存一个指向该对象的指针。当垃圾回收器确定了释放一个对象的时间时，它会检查“终结队列”中的每一个指针，并将对象从堆上复制到另一个叫做“终结可达表”的托管结构上，下一次垃圾回收时将产生另一个线程来为每一个“终结可达表”上的对象调用Finalize方法。

**Dispose方法**

该方法不同于Finalize，它需要显式调用，如果一个类实现了IDisposable接口，就可以显式地调用对象的Dispose方法来释放资源。垃圾回收器本身不支持IDisposable接口，所以不会调用Dispose方法。如果一个类需要Finalize方法，最好让它实现IDisposable接口，调用对象的Dispose方法来释放资源，这样可以避免Finalize方法带来的性能开销。

当一个类定义了析构函数，如果我们忘了调用Dispose方法，垃圾回收器也会帮我们回收非托管资源，如果我们调用了Dispose方法，可以再调用GC.SuppressFinalize(this)方法来跳过垃圾回收器调用Finalize的过程，所以，这两种释放资源的方式可以结合使用。

using关键字

如果一个类实现了IDisposable接口，我们可以简单的用using语法来创建一个对象：

using( Test test = new Test() )

{

}

当代码块执行结束时或者在执行过程中出现异常时，都会自动调用该对象的Dispose方法以确保资源的释放。简单、方便！