C 编程人员容易犯的10个C#错误 PDF转换可能丢失图片或格式，建议阅读原文 https://www.100test.com/kao_ti2020/464/2021_2022_C___E7_BC_96_E7_A8_8B_E4_c67_464892.htm 我们知道，C#的语法与C 非常相似，实现从C 向C#的转变，其困难不在于语言本身，而在于熟悉.NET的可管理环境和对.NET框架的理解。尽管C#与C 在语法上的变化是很小的，几乎不会对我们有什么影响，但有些变化却足以使一些粗心的C 编程人员时刻铭记在心。在本篇文章中我们将讨论C 编程人员最容易犯的十个错误。 陷阱1： 没有明确的结束方法 几乎可以完全肯定地说，对于大多数C 编程人员而言，C#与C 最大的不同之处就在于碎片收集。这也意味着编程人员再也无需担心内存泄露和确保删除所有没有用的指针。但我们再也无法精确地控制杀死无用的对象这个过程。事实上，在C#中没有明确的destructor。如果使用非可管理性资源，在不使用这些资源后，必须明确地释放它。对资源的隐性控制是由Finalize方法（也被称为finalizer）提供的，当对象被销毁时，它就会被碎片收集程序调用收回对象所占用的资源。finalizer应该只释放被销毁对象占用的非可管理性资源，而不应牵涉到其他对象。如果在程序中只使用了可管理性资源，那就无需也不应当执行Finalize方法，只有在非可管理性资源的处理中才会用到Finalize方法。由于finalizer需要占用一定的资源，因此应当只在需要它的方法中执行finalizer。直接调用一个对象的Finalize方法是绝对不允许的（除非是在子类的Finalize中调用基础类的Finalize。），碎片收集程序会自动地调用Finalize。 从语法上看，C#中的destructor与C 非常相似，但其实它们

是完全不同的。C#中的destructor只是定义Finalize方法的捷径。因此，下面的二段代码是有区别的： ~MyClass() { // 需要完成的任务 } MyClass.Finalize() {// 需要完成的任务 base.Finalize(). } 错误2：Finalize和Dispose使用谁？ 从上面的论述中我们已经很清楚，显性地调用finalizer是不允许的，它只能被碎片收集程序调用。如果希望尽快地释放一些不再使用的数量有限的非可管理性资源（如文件句柄），则应该使用IDisposable界面，这一界面有个Dispose方法，它能够帮你完成这个任务。Dispose是无需等待Finalize被调用而能够释放非可管理性资源的方法。 如果已经使用了Dispose方法，则应当阻止碎片收集程序再对相应的对象执行Finalize方法。为此，需要调用静态方法GC.SuppressFinalize，并将相应对象的指针传递给它作为参数，Finalize方法就能调用Dispose方法了。据此，我们能够得到如下的代码： public void Dispose() { // 完成清理操作 // 通知GC不要再调用Finalize方法 GC.SuppressFinalize(this). } public override void Finalize() { Dispose(). base.Finalize(). } 对于有些对象，可能调用Close方法就更合适（例如，对于文件对象调用Close就比Dispose更合适），可以通过创建一个private属性的Dispose方法和public属性的Close方法，并让Close调用Dispose来实现对某些对象调用Close方法。 由于不能确定一定会调用Dispose，而且finalizer的执行也是不确定的（我们无法控制GC会在何时运行），C#提供了一个Using语句来保证Dispose方法会在尽可能早的时间被调用。一般的方法是定义使用哪个对象，然后用括号为这些对象指定一个活动的范围，当遇到最内层的括号时，Dispose方法就会被自动调用，对该对象进行处理。 using

System.Drawing. class Tester { public static void Main() { using (Font theFont = new Font("Arial", 10.0f)) { //使用theFont对象 } // 编译器将调用Dispose处理theFont对象 Font anotherFont = new Font("Courier",12.0f). using (anotherFont) { // 使用anotherFont对象 } // 编译器将调用Dispose处理anotherFont对象 } } 