C# 的 IDisposable 接口

我在微软的团队快被微软 C# 里面的各种 IDisposable 对象给折腾疯了……

故事比较长，先来科普一下。如果你没有用过 C#，IDisposable 是 C# 针对“资源管理”设计的一个接口，它类似于 Java 的 Closeable 接口。这类接口一般提供一个“方法”（比如叫 Dispose 或者 Close），你的资源（比如文件流）实现这个接口。使用资源的人先“打开资源”，用完之后调用这个方法，表示“关闭资源”。比如，文件打开，读写完了之后，调用 close 关掉。看似很简单？;-)

相比于 Java，C# 大部分时候是更好的语言，然而它并没有全面超越 Java。一个显著的不足之处就是 C# 的 IDisposable 接口引起的头痛，要比 Java 的 Closeable 大很多。经过我分析，这一方面是因为 .NET 库代码里面实现了很多没必要的 IDisposable，以至于你经常需要思考如何处理它们。另一方面是由于微软的编码规范和 Roslyn 静态分析引起的误导，使得用户对于 IDisposable 接口的“正确使用”过度在乎，导致代码无端变得复杂，导致 IDisposable 在用户代码里面传染。

IDisposable 的问题

回来说说我们的代码，本来没那么多问题的，结果把 Roslyn 静态分析一打开，立马给出几百个警告，说“你应该调用 Disposable 成员的 Dispose 方法”（CA2213），或者说“类型含有 disposable 成员，却没有实现 IDisposable 接口”（CA1001）。

奇葩的是，C# 里面有些很小却很常用的对象，包括 ManualResetEvent, Semaphore, ReaderWriterLockSlim 都实现了 IDisposable 接口，所以经常搞得你不知所措。按

官方的“规矩”，你得显式的调用所有这些对象的 Dispose 方法进行“释放”，而不能依赖 GC 进行回收。所以你的代码经常看起来就像这个样子：

```
void foo()
{
  var event = new ManualResetEvent(false);
  // 使用 _event ...
  event.Dispose();
}
```

貌似没什么困难嘛，我们把每个对象的 Dispose 方法都调用一下，不就得了？然而问题远远不是这么简单。很多时候你根本搞不清楚什么时候该释放一个对象，因为它存在于一个复杂，动态变化的数据结构里面。除非你使用引用计数，否则你没有办法确定调用 Dispose 的时机。如果你过早调用了 Dispose 方法，而其实还有人在用它，就会出现严重的错误。

这问题就像 C 语言里面的 free，很多时候你不知道该不该 free 一块内存。如果你过早的 free 了内存，就会出现非常严重而蹊跷的内存错误，比泄漏内存还要严重很多。举一个 C 语言的例子：

```
void main()
{
    int *a = malloc(sizeof(int));
    *a = 1;

    int *b = malloc(sizeof(int));
    *b = 2;

    free(a);

    int *c = malloc(sizeof(int));
    *c = 3;

    printf("%d, %d, %d\n", *a, *b, *c);
}
```

你知道这个程序最后是什么结果吗？自己运行一下看看吧。所以对于复杂的数据结构，比如图节点，你就只好给对象加上引用计数。相信我，使用引用计数很痛苦。或者如果你的内存够用，也不需要分配释放很多中间结果，那你就干脆把这些对象都放进一个“池子”，到算法结束以后再一并释放它们……

是的 C# 有垃圾回收（GC），所以你以为不用再考虑这些低级问题了。不幸的是，IDisposable 接口以及对于它兢兢业业的态度，把这麻烦事给带回来了。以前在 Java 里用此类对象，从来没遇到过这么麻烦的事情，最多就是打开文件的时

候要记得关掉（关于文件，我之后会细讲一下）。

我不记得 Java 的等价物（Closeable 接口）引起过这么多的麻烦，Java 的 Semaphore 根本就没有实现 Closeable 接口，也不需要在用完之后调用什么 Close 或者 Dispose 之类的方法。作为一个眼睛雪亮的旁观者，我开始怀疑 C# 里的那些像 Semaphore 之类的小东西是否真的需要显式的“释放资源”。

.NET 库代码实现不必要的 IDisposable 接口

为了搞明白 C# 库代码里面为什么这么多 IDisposable 对象，我用 JetBrains 出品的反编译器 dotPeek （好东西呀）反编译了 .NET 的库代码。结果发现好些库代码实现了完全没必要的 IDisposable 接口。这说明有些 .NET 库代码的作者其实没有弄明白什么时候该实现 IDisposable ，以及如何有意义地实现它。

这些有问题的类，包括常用的 HashAlgorithm（各种 SHA 算法的父类）和 MemoryStream。其中 HashAlgorithm 的 Dispose 方法完全没必要，这个类的源代码看起来是这个样子：

```
public abstract class HashAlgorithm : IDisposable, ICryp
  ...
  protected internal byte[] HashValue;
  ...
  protected virtual void Dispose(bool disposing)
  {
      if (disposing)
      {
          if (HashValue != null)
              Array.Clear(HashValue, 0, HashValue.Length
          HashValue = null;
          m_bDisposed = true;
      }
  }
}
```

看明白了吗？它不过是在把内部数组 HashValue 的每个元素清零，然后把指针设为 null。这个库代码作者没有搞明白的是，如果你的 Dispose 方法只是在把一些成员设为 null，那么你根本就不需要实现 IDisposable。为什么呢？因为把引用设为 null 并不等于 C 语言里面的 free，它并不

能立即回收那份内存，就算你的对象里面有一个很大的数组也一样。我发现有些 C# 程序员喜欢在使用对象之后把引用赋值为 null，就像这样写代码：

```
void foo()
{
  BigObject x = new BigObject();
  // ...
  // 使用 x 指向的对象 ...
  // ...
  x = null;
}
```

`x = null` 是毫无意义的。写出这样的代码，说明他们不明白 GC 是如何工作的，以为把引用设为 null 就可以释放内存，以为不把引用设为 null，内存就不会被回收！再进一步，如果你仔细看 HashAlgorithm 的源代码，就会发现 HashValue 这个成员数组其实没有必要存在，因为它保存的只是上一次调用 ComputeHash() 的结果而已。这种保存结果的事情，本来应该交给使用者去做，而不是包揽到自己身上。这个数组的存在，还导致你没法重用同一个 HashAlgorithm 对象，因为有共享的成员 HashValue，所以不再是 thread safe 的。

其实在 C# 里面，你没有办法可以手动回收内存，因为内存是由 GC 统一管理的。就算你实现 Dispose，在里面把成员设置为 null，内存也只有等下次 GC 执行的时候才可能被回收。举一个例子：

```
class Foo : IDisposable
{
  private byte[] _data = new byte[1000000000];

  public void Dispose()
  {
   _data = null;    // 没用的
  }
}
```

在这个例子里面，Foo 类型的 Dispose 只是在把 _data 设为 null，这是毫无意义的。如果你想释放掉这块数组，那么你只需要等不再有人使用 Foo 对象。比如：

```
void UseFoo()
{
  Foo foo = new Foo();
  // 使用 f...
  foo.Dispose();  // 没必要
  foo = null;     // 没必要
}
```

这里的 `foo.Dispose()` 是完全没必要的。你甚至没必要写

`foo = null`，因为 foo 是一个局部变量，它一般很快就会离开作用域的。当函数执行完毕，或者编译器推断 foo 不会再次被使用的时候，GC 会回收整个 Foo 对象，包括里面的巨大数组。

所以正确的做法应该是完全不要 Dispose，不实现 IDisposable 接口。有些人问，要是 Foo 对象被放进一个全局哈希表之类的数据结构，GC 没法释放它，就需要 Dispose 了吧？这也是一种常见的误解。如果你真要回收全局哈希表里的 Foo 对象，你只需要把 Foo 对象从哈希表里面删掉就可以了。一旦哈希表对 Foo 对象的引用没有了，GC 运行的时候就会发现它成了垃圾，里面的 _data 数组自然也是垃圾，所以一起就回收掉了。

所以简言之，Dispose 不是用来给你回收内存用的。在 Dispose 方法里把成员设为 null，并不会导致更快的内存释放。有人可能以为 HashAlgorithm 是为了“安全”考虑，所以在 Dispose 方法里对数组清零。然而 IDisposable 是用于释放“资源”的接口，把安全清零这种事情放在这个接口里面，反而会让人误解，造成疏忽。

而且从源代码里的注释看来，HashAlgorithm 的这个方法确实是为了释放资源，而不是为了什么安全考虑。这些库代码实现 IDisposable，意味着这个接口会通过这些库代码不必要的传递到用户代码里面去，导致很多不知情用户的代码被迫实现 IDisposable，造成“传染”。

作为练习，你可以分析一下 MemoryStream 的 Dispose 方法，为什么是没必要的：

```
protected override void Dispose(bool disposing)
{
    try
    {
        if (disposing)
        {
            _isOpen = false;
            _writable = false;
            _expandable = false;
#if FEATURE_ASYNC_IO
        _lastReadTask = null;
#endif
        }
    }
    finally
    {
        // Call base.Close() to cleanup async IO resour
        base.Dispose(disposing);
    }
}
```

另外，我发现 AutoResetEvent，ManualResetEvent，ReaderWriterLockSlim，Semaphore 这些 IDisposable 对象，里面的所谓“资源”，归根结底都是一些很小的 Windows event 对象，而且它们都继承了 SafeHandle。SafeHandle 本身有一个“析构函数”（finalizer），它看起来是这个样子：

```
~SafeHandle()
{
 Dispose(false);
}
```

当 SafeHandle 被 GC 回收的时候，GC 会自动自动调用这个析构函数，进而调用 Dispose。也就是说，你其实并不需要手动调用这些对象（例如 ManualResetEvent, Semaphore 之类）的 Dispose 方法，因为 GC 会调用它们。这些对象占用资源不多，系统里也不会有很多这种对象，所以 GC 完全应该有能力释放它们占用的系统资源。

文件的特殊性质

很多人谈到这个问题，就会举文件的例子来反驳你，说：“你不应该依靠 GC 来释放 IDisposable 对象。你应该及时关闭文件，所以对于其它 IDisposable 资源，也应该及时关闭，不应该等 GC 来释放它。” 这些人没有抓住问题的关键，所以他们把文件和其它 IDisposable 资源一概而论。

文件是一种很特殊的资源，它和其它 IDisposable 对象是很不一样的。你之所以需要在用完一个文件之后立即关掉它，而不能等 GC 来做这事，是因为文件是一种隐性的“全局资源”。这种“全局”，是从程序语言语义的角度来看的。文件很像程序里的全局变量，无论从什么地方都可以访问。

使用文件的时候，你使用文件名来读写它，任何知道这个名字的进程都可以访问这个文件。（我们这里忽略权限之类的问题，那跟语义是不相关的。）这使得文件成为一种“全局资源”，也就是说它不是 thread safe 的。在并发系统里面，在任何一个时刻，只能有一个进程打开文件进行写操作。然后这个文件就被它“锁住”了，其它进程不能打开，否则就会出现混乱。所以如果这个进程不及时关掉文件，其它人就没法

用它。

写文件其实是给它加了锁，当然你必须及时进行解锁，而不能等 GC 这种非实时的方式来帮你解锁。否则即使你不再引用这个文件，其他人仍然没法立即进入锁定的区域，这就造成了不必要的等待。所以文件的所谓“打开”和“关闭”操作，本质上隐含了加锁和解锁操作。

文件是很特殊的资源。系统里的大部分其它资源，都不像文件这样是共享的，而是分配给进程“私人使用”的。系统里面可以有任意多个这样的资源，你用任何一个都可以，它们的使用互不干扰，不需要加锁，所以你并不需要非常及时的关闭它们。这种资源的性质，跟内存的性质几乎完全一样。

像 C# 里的 ManualResetEvent, Semaphore, ReaderWriterLockSlim 就属于这种非共享资源，它们的性质跟内存非常相似。就算它们实现了 IDisposable 接口，关闭它们的重要性也跟关闭文件相差非常大。我通过测试发现，就算你把它们完全交给 GC 处理，也不会有任何问题。无论你是否调用它们的 Dispose 方法，系统性能都一模一样。只不过如果你调用 Dispose，计算花的时间还要稍微多一些。

官方文档和 Roslyn 静态分析不可靠

微软官方文档和 Roslyn 静态分析说一定要调用 Dispose，其实是把不是问题的问题拿出来，让没有深入理解的人心惊胆战。结果把代码给搞复杂了，进而引发更严重的问题。很多人把 Roslyn 静态分析的结果很当回事，而其实看了 Roslyn 静态分析的源代码之后，我发现他们关于 Dispose 的静态分析实现，是相当幼稚的作法。基本的流分析（flow analysis）都没有，靠肤浅的表象猜测，所以结果是非常不准确的，导致很多 false positive。回忆一下我的 PySonar 全局流分析，以及我在 Coverity 是干什么的，你就知道我为什么知道这些 ;-)

另外 Roslyn 分析给出的警告信息，还有严重的误导性质，会导致一知半解的人过度紧张。比如编号为 CA1001 的警告对你说：“Types that own disposable fields should be

disposable。” 如果你严格遵循这一“条款”，让所有含有 IDispoable 的成员的类都去实现 IDisposable，那么 IDisposable 接口就会从一些很小的对象（比如常见的 ManualResetEvent），很快扩散到其它用户对象里去。许多对象都实现 IDisposable 接口，却没有任何对象真正的调用 Dispose 方法。最终结果跟你什么都不做是一样的，只不过代码变复杂了，还浪费了时间和精力。