对象的生命周期

在 C#中，程序员无法直接在 C#中删除一个托管对象，因为 C#不提供这个功能 那么类的实例就需要通过CLR 调用垃圾回收机制进行清除 ,回收内存。.NET 垃圾回收器会压缩空的内存块来实现优化，为了辅助这一功能，托管堆会保存一个指针，它指向下一个对象将被分配的位置。那么 CLR 是如何使用垃圾回收机制呢？首先，类实例化之后具体的对象会被分配到一块叫托管堆的内存区域上，将托管堆中的对象的引用地址返回给函数中的引用变量，引用变量保存在栈内，要使用对象中的方法只需要使用点操作就可以。特别需要说一下的是，结构是数值类型，它直接分配在栈上（所以的数值类都是这样的，只有引用类型才会保存在托管堆上）。实例化结束之后，垃圾回收器会在一个对象从代码库的任何部分都不可访问的时候，将它从堆中删除，例：

```
static void MakeCar()
{
    Car mycar=new Car();
}
```

在例子中，Car 的引用 mycar 直接在 MakeCar 中创建，并没有被传到该方法以外的作用域的 ,因此 ,在这个方法调用结束之后 ,这个对象就不会再被访问，此时它就是垃圾回收器的回收目标，但是，要知道，一个对象在失去意义之后并不会立即被删除，CLR 调用垃圾回收器的标准是：在创建对象时，先判断对象所需要的内存的大小，在判断目前的托管堆是否有足够的内存保存它，当托管堆没有足够的内存时，CLR 就会调用垃圾回收器进行垃圾回收，因此，在对象失去意义之后还需要等待 CLR 调用垃圾回收器时才能被删除。那么 CLR 是如何判断对象所需的内存，以及堆的内存是否够用？当 C#编译器在遇到 new 关键字时，它

会自动在方法的实现中加上一条 CIL newobj 的指令，它会计算分配对象所需的总内存，检查堆的内存空间，如果内存足够，则调用类型的构造函数，最终将内存中的新变量的引用返回给调用者，它的地址是下一个对象指针的最后位置，若是内存不足，则 CLR 调用垃圾回收器释放内存。在回调地址之后，就将对象的指针指向下一个可用的地址。那么垃圾回收器如何判断一个对象是否还在使用，这就要介绍一个应用程序根，所谓的根，说白了就是存储堆上的对象的引用地址的存储位置，在回收过程中运行库会对对象进行判断，判断程序是否还可以访问它们，也就是说它们的根是否还存在，若是不存在，则标记为垃圾，进而被清除对象，CLR 压缩内存，指针指向正确的位置。但是若是每一次进行垃圾回收的时候都要对托管堆上的所以数据进行一次判定，这种方法就太过于耗时耗力了，于是就有了代，代的设计思路是：对象在堆上存在的时间越长就越可能被保留。所以就将堆上的对象共分为 0-2 的 3 代，垃圾回收器在运行的时候，首先会检测 0 代的对象，并且将那些不需要的对象释放，空出内存，若是空出的内存不够，则会往上面一级的 1 级代进行检测，以此类推，直到获取所需要的内存大小为止，而在这之后，0 代上没被删除的对象就会被标记为 1 代，一代中未被删除的对象就标记为 2 代，但是 2 代还是二代，因为这是上限。在这里还得说一下，其实垃圾回收器使用的两个堆，我们所说的是小对象堆，还有一个大对象堆，他存储的是大于 85k 的对象，因为它的内容太大，对它进行修改的话，花费代价太大，所以在垃圾回收器极少会对它进行修改。

以上是垃圾回收器自动对对上面的数据进行回收，这并不需要人为的进行操控，但是这是对于托管在托管堆上面的对象，若是有些数据不是托管的资源呢？.NET 提供了一个 System.GC 的类类型，它可以通过编程使用一些静态成员

与垃圾回收器进行交互，这种行为也叫作强制垃圾回收，它可以由我们自己决定什么时候释放某个对象的资源，而不用被动等待垃圾回收器运行，一般来说在不希望接下来的代码被垃圾回收器打断的运行时候(垃圾回收器的运行时间是不确定的），或者我需要一次性分配很多的对象的时候都会用到强制回收，强制回收使用 GC.Collect()方法，在它的后面必须要调用 GC.WaitForPendingFinalize()，另外，使用 Finalize()构建可终结对象，当应用程序的应用程序域从内存中卸载的话 ,CLR 就会自动调用它的生命周期中所创建的每一个可终结对象的终结器进行强制回收 ,重写 Finalize()无法像普通的类一样 ,它需要类似 C++的析构语法，此外终结器还需在名称之前加~，他不接受访问修饰符，不接受参数，不支持重载，但是使用这种方式的话，需要两次的来及回收才能真正的释放该资源，并且由于是额外的处理，所以速度回变得非常慢。所以就可以考虑构建一个可处置对象，构建可处置对象需要实现 IDisposable，这个方法不止可以释放一个对象的非托管资源，而且还可以对任何它包含的可处置对象调用 Dipose(),使用这种方法可以有自己调用释放内存，若是忘记调用，也会有垃圾回收器进行释放，所以这种方式在我看来会更安全好用一些。在 C#类中还为实现了 IDisposable 的接口提供了一类语法：using，使用这种语法的好处在于，可以由 Dispose()调用扩展 try/catch 结构，例如：using(class c=new class()){}在编译之后，与 class c=new class();try{}catch(){};是等同的。

在这一个章节内，我觉得还有一个非常使用的泛型类 Lazy<>，但凡被这个类所定义的数据在代码库实际使用它之前是不会被创建的 ,这样就可以使一些不常使用的大数据在实例化对象的时候不同时被创建，进而占用内存空间。例：

```
class song{
    public string Artist{get;set;}
```

```
    public string TrackName{get;set;}
    public double TrackLength{get;set;}
}
class AllTracks
{
    private Song[] allSongs=new Song[10000];
    public AllTracks()
    {
        Console.WriteLine();
    }
    class MediaPlayer()
   {
      public void Play(){};
      private AllTracks allsongs=new AllTracks();
      public AllTracks GetAllTracks()
      {
      return allSongs;
      }
    }
}
main(){
MediaPlayer m=new MediaPlayer();//在这个时候，已经间接的创建 10000
```

个歌曲对象了

```
}
```

若是将 MediaPlayer 修改为：

```
class MediaPlayer()
{
    public void Play(){};
    private Lazy<AllTracks> allsongs=new Lazy<AllTracks>();
    public AllTracks GetAllTracks()
    {
```

```
            return allSongs.Value;
        }
    }
```

调用方式就要改为：

```
main(){
MediaPlayer m=new MediaPlayer();//在这个时候，未创建 10000 个歌曲对象了
AllTracks songs=m.GetAllTracks();//此时，歌曲对象才创建
}
```