.NET 之 垃圾回收机制GC

一、GC的必要性

1、应用程序对资源操作，通常简单分为以下几个步骤：为对应的资源分配内存 → 初始化内存 → 使用资源 → 清理资源 → 释放内存。

2、应用程序对资源（内存使用）管理的方式，常见的一般有如下几种：

[1] 手动管理：C,C++

[2] 计数管理：COM

[3] 自动管理：.NET,Java,PHP,GO...

3、但是，手动管理和计数管理的复杂性很容易产生以下典型问题：

[1] 程序员忘记去释放内存

[2] 应用程序访问已经释放的内存

产生的后果很严重，常见的如内存泄露、数据内容乱码，而且大部分时候，程序的行为会变得怪异而不可预测，还有Access Violation等。

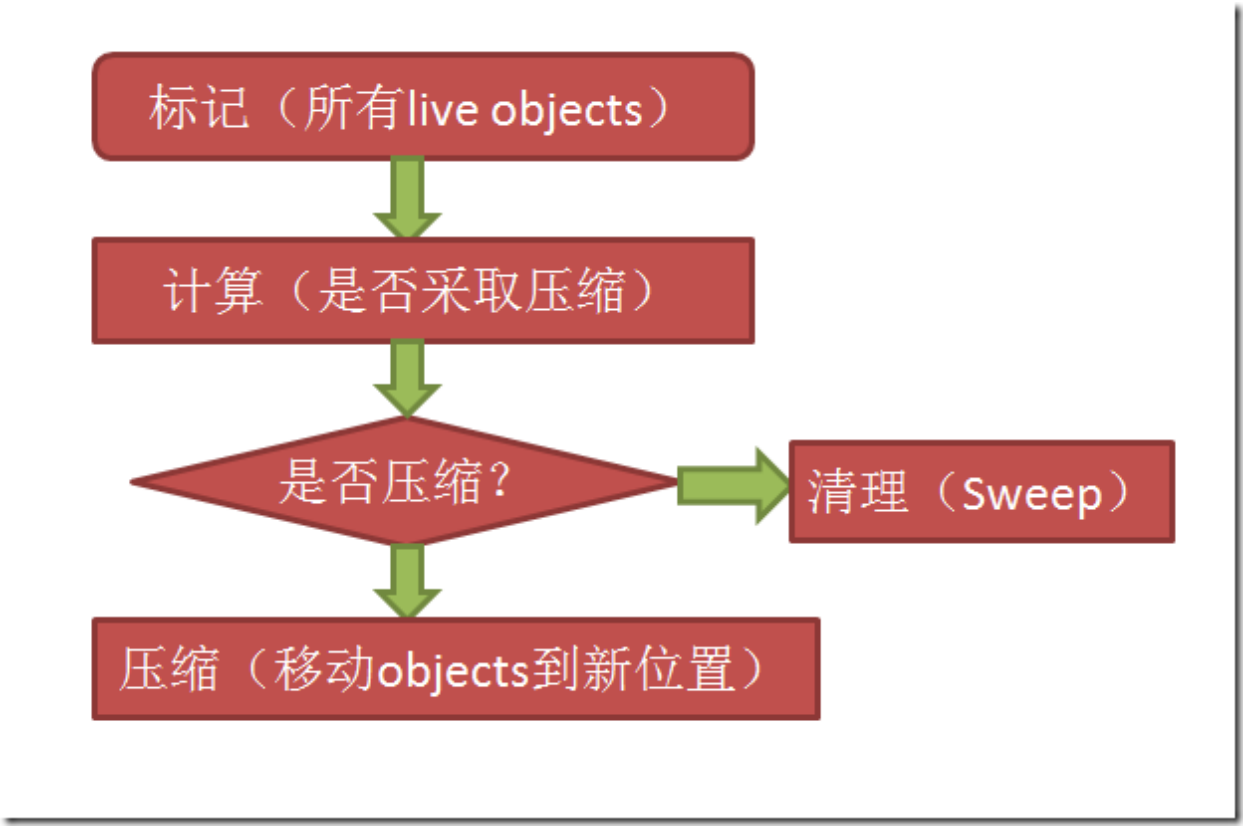
.NET、Java等给出的解决方案，就是通过自动垃圾回收机制GC进行内存管理。这样，问题1自然得到解决，问题2也没有存在的基础。

总结：无法自动化的内存管理方式极容易产生bug，影响系统稳定性，尤其是线上多服务器的集群环境，程序出现执行时bug必须定位到某台服务器然后dump内存再分析bug所在，极其打击开发人员编程积极性，而且源源不断的类似bug让人厌恶。

二、GC是如何工作的

GC的工作流程主要分为如下几个步骤：

标记(Mark) → 计划(Plan) → 清理(Sweep) → 引用更新(Relocate) → 压缩(Compact)



1、标记

目标：找出所有引用不为0(live)的实例

方法：找到所有的GC的根结点(GC Root)，将他们放到队列里，然后依次递归地遍历所有的根结点以及引用的所有子节点和子子节点，将所有被遍历到的结点标记成live。弱引用不会被考虑在内

2、计划和清理

[1] 计划

目标：判断是否需要压缩

方法：遍历当前所有的generation上所有的标记(Live),根据特定算法作出决策

[2] 清理

目标：回收所有的free空间

方法：遍历当前所有的generation上所有的标记(Live or Dead),把所有处在Live实例中间的内存块加入到可用内存链表中去

3、引用更新和压缩

[1] 引用更新

目标： 将所有引用的地址进行更新

方法：计算出压缩后每个实例对应的新地址，找到所有的GC的根结点(GC Root)，将他们放到队列里，然后依次递归地遍历所有的根结点以及引用的所有子节点和子子节点，将所有被遍历到的结点中引用的地址进行更新，包括弱引用。

[2] 压缩

目标：减少内存碎片

方法：根据计算出来的新地址，把实例移动到相应的位置。

三、GC的根节点

本文反复出现的GC的根节点也即GC Root是个什么东西呢?

每个应用程序都包含一组根（root）。每个根都是一个存储位置，其中包含指向引用类型对象的一个指针。该指针要么引用托管堆中的一个对象，要么为null。

在应用程序中，只要某对象变得不可达，也就是没有根（root）引用该对象，这个对象就会成为垃圾回收器的目标。

用一句简洁的英文描述就是:GC roots are not objects in themselves but are instead references to objects.而且，Any object referenced by a GC root will automatically survive the next garbage collection.

.NET中可以当作GC Root的对象有如下几种：

1、全局变量

2、静态变量

3、栈上的所有局部变量(JIT)

4、栈上传入的参数变量

5、寄存器中的变量

注意，只有引用类型的变量才被认为是根，值类型的变量永远不被认为是根。因为值类型存储在堆栈中，而引用类型存储在托管堆上。

四、什么时候发生GC

1、当应用程序分配新的对象，GC的代的预算大小已经达到阈值，比如GC的第0代已满；

2、代码主动显式调用System.GC.Collect()；

3、其他特殊情况，比如，windows报告内存不足、CLR卸载AppDomain、CLR关闭，甚至某些极端情况下系统参数设置改变也可能导致GC回收。

五、GC中的代

代（Generation)引入的原因主要是为了提高性能（Performance），以避免收集整个堆（Heap）。一个基于代的垃圾回收器做出了如下几点假设：

1、对象越新，生存期越短；

2、对象越老，生存期越长；

3、回收堆的一部分，速度快于回收整个堆。

.NET的垃圾收集器将对象分为三代（Generation0,Generation1,Generation2）。不同的代里面的内容如下：

1、G0 小对象(Size<85000Byte)：新分配的小于85000字节的对象。

2、G1:在GC中幸存下来的G0对象

3、G2:大对象(Size>=85000Byte);在GC中幸存下来的G1对象

```
object o = new Byte[85000]; //large object Console.WriteLine(GC.GetGeneration(o)); //output is 2,not 0
```

这里必须知道，CLR要求所有的资源都从托管堆（managed heap）分配，CLR会管理两种类型的堆，小对象堆（small object heap，SOH）和大对象堆（large object heap，LOH），其中所有大于85000byte的内存分配都会在LOH上进行。

代收集规则：当一个代N被收集以后，在这个代里的幸存下来的对象会被标记为N+1代的对象。GC对不同代的对象执行不同的检查策略以优化性能。每个GC周期都会检查第0代对象。大约1/10的GC周期检查第0代和第1代对象。大约1/100的GC周期检查所有的对象。

六、谨慎显式调用GC

GC的开销通常很大，而且它的运行具有不确定性，微软的编程规范里是强烈建议你不要显式调用GC。但你的代码中还是可以使用framework中GC的某些方法进行手动回收，前提是你必须要深刻理解GC的回收原理，否则手动调用GC在特定场景下很容易干扰到GC的正常回收甚至引入不可预知的错误。
比如如下代码：

```
void SomeMethod() { object o1 = new Object(); object o2 = new Object(); o1.ToString();
GC.Collect(); // this forces o2 into Gen1, because it's still referenced o2.ToString(); }
```

如果没有GC.Collect()，o1和o2都将在下一次垃圾自动回收中进入Gen0，但是加上GC.Collect()，o2将被标记为Gen1，也就是0代回收没有释放o2占据的内存

还有的情况是编程不规范可能导致死锁，比如流传很广的一段代码：

```
public class MyClass { private bool isDisposed = false; ~MyClass() { Console.WriteLine("Enter
destructor..."); lock (this) //some situation lead to deadlock { if (!isDisposed) {
Console.WriteLine("Do Stuff..."); } } } }
```

通过如下代码进行调用：

```
            var instance = new MyClass(); Monitor.Enter(instance); instance = null; GC.Collect();
GC.WaitForPendingFinalizers(); Console.WriteLine("instance is gabage collected");
```

上述代码将会导致死锁。原因分析如下：

1、客户端主线程调用代码Monitor.Enter(instance)代码段lock住了instance实例

2、接着手动执行GC回收，主（Finalizer)线程会执行MyClass析构函数

3、在MyClass析构函数内部，使用了lock (this)代码，而主（Finalizer)线程还没有释放instance（也即这里的this），此时主线程只能等待

虽然严格来说，上述代码并不是GC的错，和多线程操作似乎也无关，而是Lock使用不正确造成的。

同时请注意，GC的某些行为在Debug和Release模式下完全不同（Jeffrey Richter在<<CLR Via C#>>举过一个Timer的例子说明这个问题）。比如上述代码，在Debug模式下你可能发现它是正常运行的，而Release模式下则会死锁。

七、当GC遇到多线程

前面讨论的垃圾回收算法有一个很大的前提就是：只在一个线程运行。而在现实开发中，经常会出现多个线程同时访问托管堆的情况，或至少会有多个线程同时操作堆中的对象。一个线程引发垃圾回收时，其它线程绝对不能访问任何线程，因为垃圾回收器可能移动这些对象，更改它们的内存位置。CLR想要进行垃圾回收时，会立即挂起执行托管代码中的所有线程，正在执行非托管代码的线程不会挂起。然后，CLR检查每个线程的指令指针，判断线程指向到哪里。接着，指令指针与JIT生成的表进行比较，判断线程正在执行什么代码。

如果线程的指令指针恰好在一个表中标记好的偏移位置，就说明该线程抵达了一个**安全点。**线程可在安全点安全地挂起，直至垃圾回收结束。如果线程指令指针不在表中标记的偏移位置，则表明该线程不在安全点，CLR也就不会开始垃圾回收。在这种情况下，CLR就会劫持该线程。也就是说，CLR会修改该线程栈，使该线程指向一个CLR内部的一个特殊函数。然后，线程恢复执行。当前的方法执行完后，他就会执行这个特殊函数，这个特殊函数会将该线程安全地挂起。然而，线程有时长时间执行当前所在方法。所以，当线程恢复执行后，大约有250毫秒的时间尝试劫持线程。过了这个时间，CLR会再次挂起线程，并检查该线程的指令指针。如果线程已抵达一个安全点，垃圾回收就可以开始了。但是，如果线程还没有抵达一个安全点，CLR就检查是否调用了另一个方法。如果是，CLR再一次修改线程栈，以便从最近执行的一个方法返回之后劫持线程。然后，CLR恢复线程，进行下一次劫持尝试。所有线程都抵达安全点或被劫持之后，垃圾回收才能使用。垃圾回收完之后，所有线程都会恢复，应用程序继续运行，被劫持的线程返回最初调用它们的方法。

实际应用中，CLR大多数时候都是通过劫持线程来挂起线程，而不是根据JIT生成的表来判断线程是否到达了一个安全点。之所以如此，原因是JIT生成表需要大量内存，会增大工作集，进而严重影响性能。

这里再说一个真实案例。某web应用程序中大量使用Task，后在生产环境发生莫名其妙的现象，程序时灵时不灵，根据数据库日志（其实还可以根据Windows事件跟踪（ETW）、IIS日志以及dump文件），发现了Task执行过程中有不规律的未处理的异常，分析后怀疑是CLR垃圾回收导致，当然这种情况也只有在高并发条件下才会暴露出来。

八、开发中的一些建议和意见

由于GC的代价很大，平时开发中注意一些良好的编程习惯有可能对GC有积极正面的影响，否则有可能产生不良效果。

1、尽量不要new很大的object，大对象（>=85000Byte）直接归为G2代，GC回收算法从来不对大对象堆（LOH）进行内存压缩整理，因为在堆中下移85000字节或更大的内存块会浪费太多CPU时间；

2、不要频繁的new生命周期很短object，这样频繁垃圾回收频繁压缩有可能会导致很多内存碎片，可以使用设计良好稳定运行的对象池（ObjectPool）技术来规避这种问题

3、使用更好的编程技巧，比如更好的算法、更优的数据结构、更佳的解决策略等等

update:.NET4.5.1及其以上版本已经支持压缩大对象堆，可通过System.Runtime.GCSettings.LargeObjectHeapCompactionMode进行控制实现需要压缩LOH。

九、GC线程和Finalizer线程

GC在一个独立的线程中运行来删除不再被引用的内存。

Finalizer则由另一个独立（高优先级CLR)线程来执行Finalizer的对象的内存回收。

对象的Finalizer被执行的时间是在对象不再被引用后的某个不确定的时间，并非和C++中一样在对象超出生命周期时立即执行析构函数。

GC把每一个需要执行Finalizer的对象放到一个队列(从终结列表移至freachable队列）中去，然后**启动另一个线程而不是在GC执行的线程**来执行所有这些Finalizer，GC线程继续去删除其他待回收的对象。

在下一个GC周期，这些执行完Finalizer的对象的内存才会被回收。也就是说一个实现了Finalize方法的对象必需等两次GC才能被完全释放。这也表明有Finalize的方法（Object默认的不算）的对象会在GC中自动"延长"生存周期。

特别注意：负责调用Finalize的线程并不保证各个对象的Finalize的调用顺序，这可能会带来微妙的依赖性问题(见<<CLR Via C#>>一个有趣的依赖性问题）。