

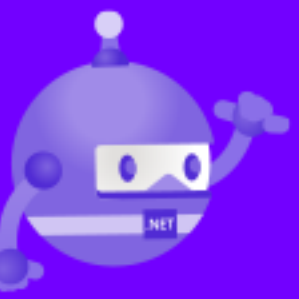
**2021.NET Conf China**

开源共建 | 开放创新 | 开发赋能



# dotnet 内存方面的性能分析

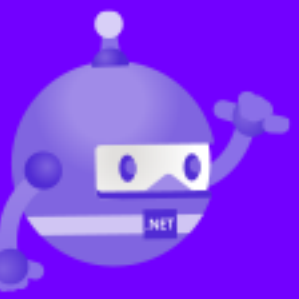
Maoni Stephens  
dotnet GC architect



# 如果您有内存问题，您会怎么解决？

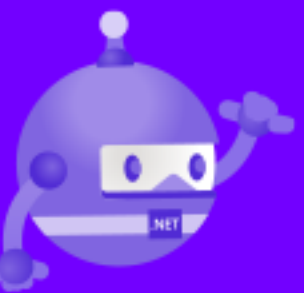
- 在您的性能分析工具里执行您的程序，看看CPU hot paths？
- 捕获一个转储？
- 看看您都分配了哪些对象，能不能减少这些分配？





**当问题很明显的時候**



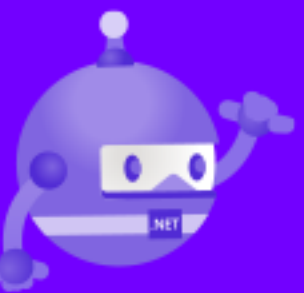


# 最常见的问题 – 内存泄露

- 内存泄露的定义 from [mem-doc](#) -

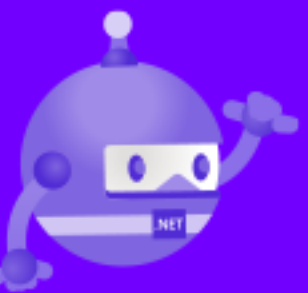
A managed memory leak means you have at least one [user root](#) that refers to, directly or indirectly, more and more objects as the process runs. It's a leak because the GC by definition cannot reclaim memory of these objects so even if the GC tried the hardest (ie, doing a full blocking GC) the heap size still ends up growing.

托管内存泄漏意味着你至少有一个用户根，随着进程的运行，直接或间接地引用了越来越多的对象。这是一个泄漏，因为根据定义，GC不能回收这些对象的内存，所以即使GC尽了最大努力(即做一个全堆阻塞的GC)，堆的大小最终还是会增长。



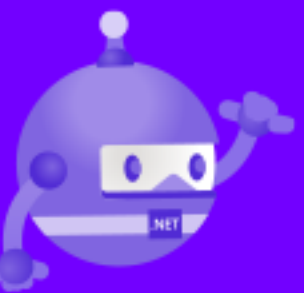
# 有关内存的fundamentals

- 内存泄露就意味着是托管堆的问题吗?
- 每一个dotnet进程都会有不归GC管的内存
- GC 如何从操作系统里获取内存?
  - reserve vs commit



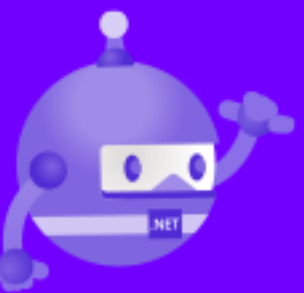
# GC 会调用哪些OS APIs来获取内存?

- 术语是Windows上的, 在Linux上有同样的概念
- Reserve
  - 本段虚拟内存地址只供GC用
  - 没有物理存储空间, 还不可以存写内容
  - Windows – VirtualAlloc (MEM\_RESERVE); Linux – mmap (PROT\_NONE)
- Commit
  - 有物理存储空间, 可以存写内容
  - Windows – VirtualAlloc (MEM\_COMMIT); Linux – mprotect (PROT\_READ | PROT\_WRITE)



# dotnet 运行时是一个用户态的component

- 它可以调用VirtualAlloc,别的用户态components当然也可以
  - 例子: native memory allocator
- 运行时中的其它部分也可以调用VirtualAlloc
  - 例子: type info所占用的内存
- 可以用非托管内存的性能分析工具
  - native memory diagnostics tools for VirtualAlloc/mmap
- 如果它们有共性, 也可以在debugger里设一个断点

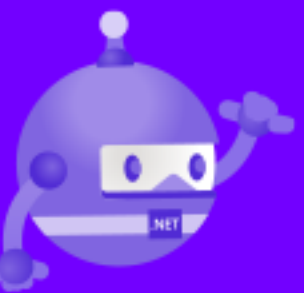


# 一个断点的例子

- 我们观察到这些allocations的共性: 都大于4mb
  - 可以从显示VirtualAlloc的工具里获取 (vmmap, !vadump in windbg, etc)
- 在windbg里设一个条件断点

```
bp KERNELBASE!VirtualAlloc "j (@rdx>400000) 'kb';'g'"
```





# 如果终结器(finalizable objects)引用native memory呢?

- 当然native memory不归GC管,但是可以用别的方法来辨识
- !sos.finalizequeue

```
0:000> !finalizequeue
```

```
[omitted]
```

```
generation 0 has 4 finalizable objects (0015bc90->0015bca0)
```

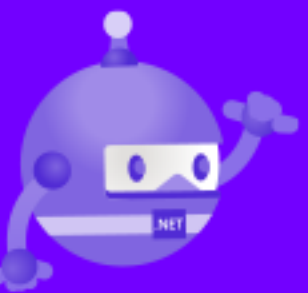
```
generation 1 has 0 finalizable objects (0015bc90->0015bc90)
```

```
generation 2 has 0 finalizable objects (0015bc90->0015bc90)
```

```
Ready for finalization 0 objects (0015bca0->0015bca0)
```

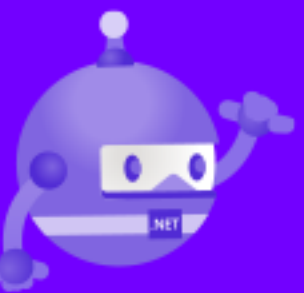
```
[omitted]
```

- Finalizable 怎么样成为Ready for finalization?



# Finalizable objects

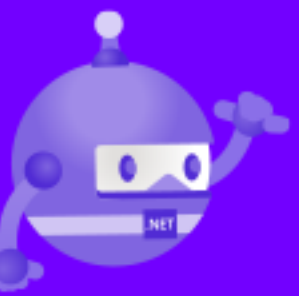
- 分配
  - GC会把它标记到finalize queue上
- 回收
  - GC检测到一个finalizable object 不存活，会将它移到finalize queue上ready for finalization的部分
- 托管线程被唤醒
  - 执行ready for finalization的部分的finalizers，并把这些标记清除
- 可能的性能问题
  - 正常情况下，每个finalizer都被很快执行
  - 如果一个finalizer被阻塞，finalizer thread会被阻塞



# 如何确定是托管堆的泄露?

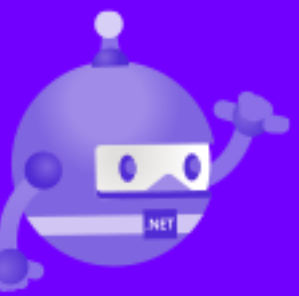
- .net framework
  - 性能计数器: # of total committed bytes
- .net 5.0
  - GC.GetGCMemoryInfo
- .net 6.0
  - dotnet counter: committed bytes
  - !sos.eeheap -gc: committed





# 用转储来分析内存泄露？

- 转储的定义是某一时刻的快照
- 泄露的定义是随着时间的推移，内存会越来越多
- 所以对于同一个进程，需要多个转储
- 早期时候，这的确是我们诊断内存泄漏的主要方法
- 现在可以用[perfview](#)的heap snapshot (Memory\Take Heap Snapshot)+diff



# 一个简单的例子

```
static List<byte[]> listToLeakTo = new List<byte[]>();
object[] objArr = new object[1000];
// populate the array.
int len = objArr.Length;
for (int i = 0; i < len; i++)
{
    objArr[i] = new byte[rand.Next(len) + 1000];
}

ulong counter = 0;
while (true)
{
    byte[] obj = new byte[rand.Next(len) + 1000];
    objArr[rand.Next(len)] = obj;
    counter++;
    if ((counter % 500_000) == 0)
    {
        listToLeakTo.Add(obj);
    }
}
```



Heap Stacks(2,365,348 metric) TestMemoryLeak.gcdump in demo (C:\temp\demo\TestMemoryLeak.gcdump)

FileViewDiffRegressionPresetHelpStack View Help (F1)Understanding Perf DataStarting an AnalysisTroubleshootingTips

UpdateBackForwardTotals Metric: 2,365,348.0 Count: 4,696.0 First: 0.000 Last: 2,130,706,455,608.000 Last-First: 2,130,706,455,608.000 Metric/Interval: 0.00 TimeBucket: 66,584,576,866.5

Start: 0End: 2,130,706,45Priority: v4.0.30319\%!\->-1;v2.0.507Find:GroupPats: [group Framework] mscorlib!=>LIB;System%!=>LIB;Fold%: 1FoldPats: [];mscorlib!StringIncPats:ExcPats: [not reachable from roots]

By Name ?RefFrom-RefTo ?RefTree ?Referred-From ?Refs-To ?Flame Graph ?Notes ?

Name ?	Exc % ?	Exc ?	Exc Ct ?	Inc % ?	Inc ?	Inc Ct ?	Fold ?	Fold Ct ?
[local vars]	65.2	1,541,880	1,003	65.2	1,541,880.0	1,003	1,541,880	1,003
[static vars]	17.7	418,846	706	32.3	764,124.0	3,061	418,846	706
LIB <<System.Private.CoreLib!Diagnostics.Tracing.NativeRuntimeEventSource>>	13.1	309,200	2,334	13.1	309,200.0	2,334	308,476	2,324
[strong Handles]	2.5	59,288	625	2.5	59,288.0	625	59,288	625
LIB <<System.Private.CoreLib!Dictionary<String,Object>>>	1.5	36,078	19	1.5	36,078.0	19	35,998	19
[other roots]	0.0	56	5	2.5	59,344.0	630	56	4
ROOT	0.0	0	0	100.0	2,365,348.0	4,696	0	0
[.NET Roots]	0.0	0	2	100.0	2,365,348.0	4,696	0	1
[static var System.Diagnostics.Tracing.NativeRuntimeEventSource.Log]	0.0	0	1	13.1	309,200.0	2,335	0	0
[static var System.Collections.Generic.Dictionary<System.String,System.Object>.s_dataStore]	0.0	0	1	1.5	36,078.0	20	0	0

Heap Stacks(2,674,242 metric) TestMemoryLeak.1.gcdump in demo (C:\temp\demo\TestMemoryLeak.1.gcdump)

FileViewDiffRegressionPresetHelpStack View Help (F1)Understanding Perf DataStarting an AnalysisTroubleshootingTips

UpdateBackForwardTotals Metric: 2,674,242.0 Count: 4,903.0 First: 0.000 Last: 2,130,706,455,608.000 Last-First: 2,130,706,455,608.000 Metric/Interval: 0.00 TimeBucket: 66,584,576,866.5

Start: 0End: 2,130,706,45Priority: v4.0.30319\%!\->-1;v2.0.507Find:GroupPats: [group Framework] mscorlib!=>LIB;System%!=>LIB;Fold%: 1FoldPats: [];mscorlib!StringIncPats:ExcPats: [not reachable from roots]

By Name ?RefFrom-RefTo ?RefTree ?Referred-From ?Refs-To ?Flame Graph ?Notes ?

Name ?	Exc % ?	Exc ?	Exc Ct ?	Inc % ?	Inc ?	Inc Ct ?	Fold ?	Fold Ct ?
[local vars]	57.6	1,540,225	1,003	57.6	1,540,225.0	1,003	1,540,225	1,003
[static vars]	27.3	728,979	904	40.2	1,074,257.0	3,259	728,979	904
LIB <<System.Private.CoreLib!Diagnostics.Tracing.NativeRuntimeEventSource>>	11.6	309,200	2,334	11.6	309,200.0	2,334	308,476	2,324
[strong Handles]	2.2	59,704	634	2.2	59,704.0	634	59,704	634
LIB <<System.Private.CoreLib!Dictionary<String,Object>>>	1.3	36,078	19	1.3	36,078.0	19	35,998	19
[other roots]	0.0	56	5	2.2	59,760.0	639	56	4
ROOT	0.0	0	0	100.0	2,674,242.0	4,903	0	0
[.NET Roots]	0.0	0	2	100.0	2,674,242.0	4,903	0	1
[static var System.Diagnostics.Tracing.NativeRuntimeEventSource.Log]	0.0	0	1	11.6	309,200.0	2,335	0	0
[static var System.Collections.Generic.Dictionary<System.String,System.Object>.s_dataStore]	0.0	0	1	1.3	36,078.0	20	0	0



Heap Stacks(2,674,242 metric) TestMemoryLeak.1.gcdump in demo (C:\temp\demo\TestMemoryLeak.1.gcdump)

File View **Diff** Regression Preset Help [Stack View Help \(F1\)](#) [Understanding Perf Data](#) [Starting an Analysis](#) [Troubleshooting](#) [Tips](#)

Update Back With Baseline: Heap Stacks(2,365,348 metric) TestMemoryLeak.gcdump in demo (C:\temp\demo\TestMemoryLeak.gcdump) 0.00 TimeBucket: 66,584,576,866.5

Start: 0 Help for Diff

GroupPats: [group Framework] mscorlib!=>LIB;System%!=>LIB; Fold%: 1 FoldPats: [];mscorlib!String IncPats: ExcPats: [not reachable from roots]

By Name ? RefFrom-RefTo ? RefTree ? Referred-From ? Refs-To ? Flame Graph ? Notes ?

Name ?	Exc % ?	Exc ?	Exc Ct ?	Inc % ?	Inc ?	Inc Ct ?	Fold ?	Fold Ct ?
[local vars]	57.6	1,540,225	1,003	57.6	1,540,225.0	1,003	1,540,225	1,002
[static vars]	27.3	728,979	904	40.2	1,074,257.0	3,259	728,979	903
LIB <<System.Private.CoreLib!Diagnostics.Tracing.NativeRuntimeEventSource>>	11.6	309,200	2,334	11.6	309,200.0	2,334	308,476	2,324
[strong Handles]	2.2	59,704	634	2.2	59,704.0	634	59,704	633
LIB <<System.Private.CoreLib!Dictionary<String,Object>>>	1.3	36,078	19	1.3	36,078.0	19	35,998	18
[other roots]	0.0	56	5	2.2	59,760.0	639	56	4
ROOT	0.0	0	0	100.0	2,674,242.0	4,903	0	0
[.NET Roots]	0.0	0	2	100.0	2,674,242.0	4,903	0	1
[static var System.Diagnostics.Tracing.NativeRuntimeEventSource.Log]	0.0	0	1	11.6	309,200.0	2,335	0	0

Diff Heap Stacks(308,894 metric) baseline Heap Stacks

File View Diff Regression Preset Help [Stack View Help \(F1\)](#) [Understanding Perf Data](#) [Starting an Analysis](#) [Troubleshooting](#) [Tips](#)

Update Back Forward Totals Metric: 308,894.0 Count: 207.0 First: 0.000 Last: 2,130,706,455,608.000 Last-First: 2,130,706,455,608.000 Metric/Interval: 0.00 TimeBucket: 66,584,576,866.5

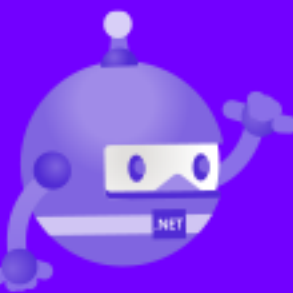
Start: 0.000 End: 2,130,706,45 Priority: v4.0.30319\%!->-1;v2.0.507 Pri1Only: Find:

GroupPats: Fold%: 1 FoldPats: IncPats: ExcPats: [not reachable from roots]

By Name ? RefFrom-RefTo ? RefTree ? Referred-From ? Refs-To ? Flame Graph ? Notes ?

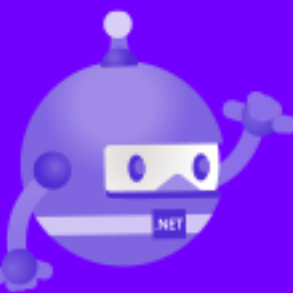
### Objects that refer to System.Private.CoreLib!Byte[] (Bytes > 1K)

Name ?	Inc % ?	Inc ?	Inc Ct ?	Exc % ?	Exc ?	Exc Ct ?	Fold ?	Fold Ct ?
<input checked="" type="checkbox"/> System.Private.CoreLib!Byte[] (Bytes > 1K) ?	100.4	310,133.0	198	100.4	310,133	198	0	0
+ <input checked="" type="checkbox"/> System.Private.CoreLib!Byte[][] (Bytes > 1K) ?	100.4	310,133.0	198	0.0	0	0	0	0
+ <input checked="" type="checkbox"/> System.Private.CoreLib!List<Byte[]> ?	100.4	310,133.0	198	0.0	0	0	0	0
+ <input checked="" type="checkbox"/> [static var System.Collections.Generic.List<System.Byte[]>.listToLeakTo] ?	100.4	310,133.0	198	0.0	0	0	0	0
+ <input checked="" type="checkbox"/> [static vars] ?	100.4	310,133.0	198	0.0	0	0	0	0
+ <input checked="" type="checkbox"/> [.NET Roots] ?	100.4	310,133.0	198	0.0	0	0	0	0
+ <input checked="" type="checkbox"/> ROOT ?	100.4	310,133.0	198	0.0	0	0	0	0




# 但目前为止，我们其实还没有讲到GC

Second	Action	Heap size at the end of that second
1	allocated 1 GB	1 GB
2	allocated 2 GB	3 GB
3	allocated 0 GB	3 GB
4	GC happens(500 MB survives), then allocated 1 GB	1.5 GB
5	allocated 3 GB	4.5 GB

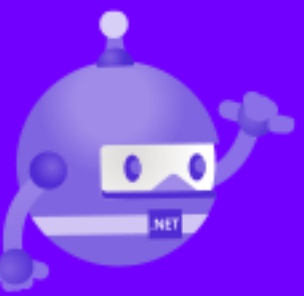


# 但目前为止，我们其实还没有讲到GC

Second	Action	Heap size at the end of that second
1	allocated 1 GB	1 GB
2	allocated 2 GB	3 GB
3	allocated 0 GB	3 GB
4	GC happens(500 MB survives), then allocated 1 GB	1.5 GB
5	allocated 3 GB	4.5 GB

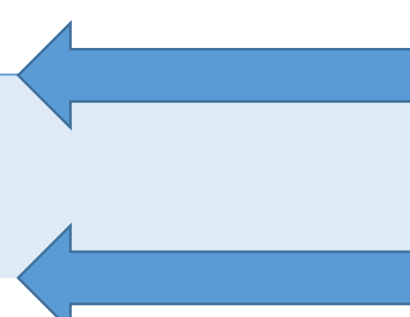


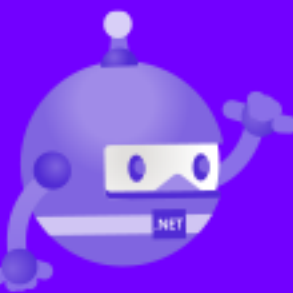




# 但目前为止，我们其实还没有讲到GC

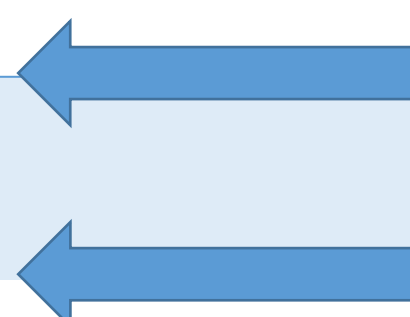
Second	Action	Heap size at the end of that second
1	allocated 1 GB	1 GB
2	allocated 2 GB	3 GB
3	allocated 0 GB	3 GB
4	GC happens(500 MB survives), then allocated 1 GB	1.5 GB
5	allocated 3 GB	4.5 GB

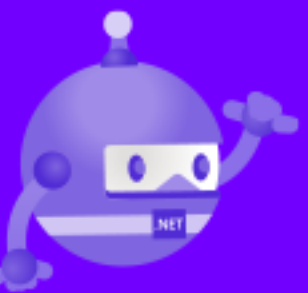




# 但目前为止，我们其实还没有讲到GC

Second	Action	Heap size at the end of that second
1	allocated 1 GB	1 GB
2	allocated 2 GB	3 GB
3	allocated 0 GB	3 GB
4	GC happens(500 MB survives), then allocated 1 GB	1.5 GB
5	allocated 3 GB	4.5 GB





# 什么时候检测跟GC什么时候运行很有关系

- 分代的GC更是如此
- 如果只进行了年轻代的回收, committed size很可能没有大变化
- 老年代(二代)回收也可能没有很大变化, 二代回收可以是
  - 二代阻塞回收(Full blocking GC) – 压缩或清除(compact or sweep)
  - 后台回收(Background GC) – 只清除



Sn – 幸存对象(Survived object); Pn – 固定对象(Pinned object); D – 不存活对象(Dead object); F – 自由对象(Free object)

压缩回收 – 开销比较大，但是可以很大程度的缩小内存

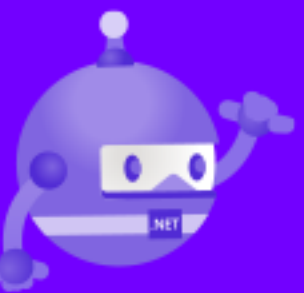


清除回收 – 开销比较小，但是一般不能很大程度的缩小内存



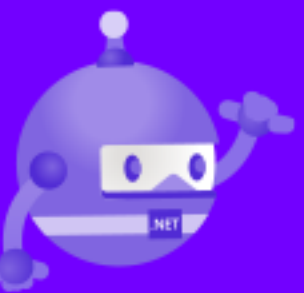
碎片(Fragmentation) – 自由对象的字节总数

- 零代碎片可用来做用户的分配，一代碎片可用来做零代幸存对象的分配，以此类推



# 二代压缩回收会何时发生

- 最可能的原因是高内存负荷
  - 总内存量是50GB, 有40GB被使用, 内存负荷是80%
- 如果内存负荷太高, 机器可能会进入分页的状态
  - 所以GC在这个时候会更加激进
  - 如果GC觉得有必要的话, 它会触发二代压缩回收
- GC的“高内存负荷情况”
  - 默认值是90%(<80GB);97%(>=80GB)
  - 但是可以[通过以下方式](#)来改变
    - COMPlus\_GCHighMemPercent 环境变量
    - System.GC.HighMemoryPercent runtimeconfig.json



# 所以知道什么时候那代的回收发生非常重要

- 强烈推荐跟踪顶层的GC的指标

- 在Windows上用perfview

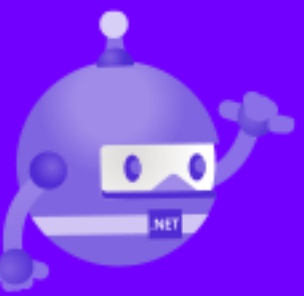
Perfview /GCCollectOnly /nogui /accepteula collect

- 在Linux上用dotnet trace

dotnet trace collect -p <pid> -o <outputpath with .nettrace extension> --profile gc-collect

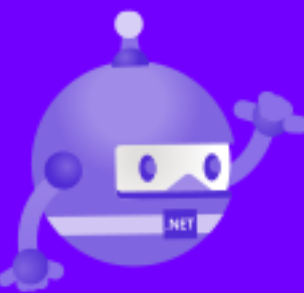
- 轻量命令
- 更多信息请参考[这里](#)





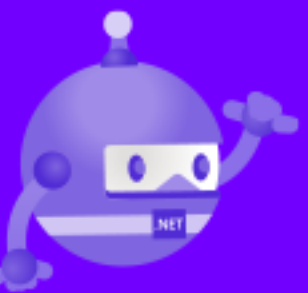
# 几种不同的情况

- 1) 堆越来越大，没有二代的回收发生
- 2) 堆越来越大，有二代压缩回收发生，因为内存负荷足够高
  - GC压缩以后，堆就小了下来
- 3) 情况2重复发生若干次，但是二代回收以后的堆越来越大



# 几种不同的情况

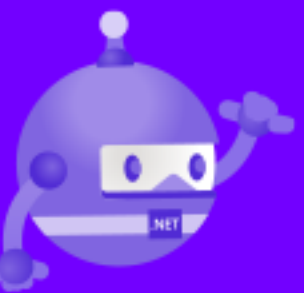
- 1) 堆越来越大，没有二代的回收发生
- 2) 堆越来越大，有二代压缩回收发生，因为内存负荷足够高
  - GC压缩以后，堆就小了下来
- 3) 情况2重复发生若干次，但是二代回收以后的堆越来越大 ← 可确定内存泄露



# 几种不同的情况

- 1) 堆越来越大，没有二代的回收发生
  - 2) 堆越来越大，有二代压缩回收发生，因为内存负荷足够高
    - GC压缩以后，堆就小了下来
  - 3) 情况2重复发生若干次，但是二代回收以后的堆越来越大
- 二代回收的关键!
- 二代回收的时候GC完全不参与决定对象的存活
  - 一个对象存活与否要看有没有[用户根](#)
    - 堆栈变量(stack variables)
    - GC句柄(GC handles)
    - 终结器(finalize queue)



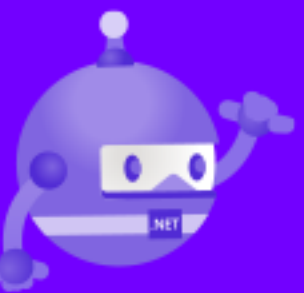


# 一个简单的例子

```
public static int Main()
{
    MaoniType o = new MaoniType(128, 256);
    GCHandle h = GCHandle.Alloc(o, GCHandleType.Weak);
    GC.Collect();
    Console.WriteLine("Collect called, h.Target is {0}",
        (h.Target == null) ? "collected" : "not collected");

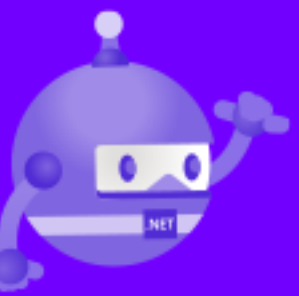
    return 0;
}
```

Output - Collect called, h.Target is not collected



# “Why is GC not collecting my object??”

- JIT会告知GC这个对象需要存活
- JIT可以选择把一个堆栈变量的lifetime延申到end of method
- 这是从dotnet 1.0就开始的behavior!
- 我们可以验证

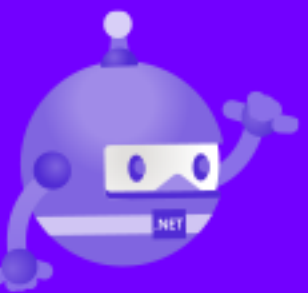


```
[MethodImpl(MethodImplOptions.NoInlining)]
public static void TestLifeTime()
{
    MaoniType o = new MaoniType(128, 256);
    h = GCHandle.Alloc(o, GCHandleType.Weak);
}
public static int Main()
{
    TestLifeTime();
    GC.Collect();
    Console.WriteLine("Collect called, h.Target is {0}",
        (h.Target == null) ? "collected" : "not collected");

    return 0;
}
```

Output: Collect called, h.Target is collected





# .net 5 的一个新诊断功能 – generation aware 分析

- 进行时可以在你设定的这些条件成立的时候捕获一个trace

- 存活的最低字节
- 回收代
- 最低的GC index

- 例子

```
set COMPlus_GCGenAnalysisGen=2
```

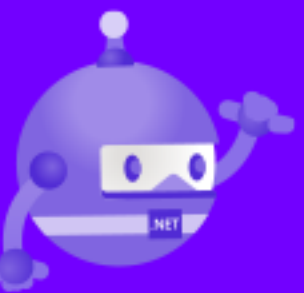
```
set COMPlus_GCGenAnalysisBytes=40000000
```

```
set COMPlus_GCGenAnalysisIndex=3E8
```

当运行时观察到一个二代阻塞回收, index至少是1000, 至少存活了1GB, 会捕获trace

- .net 6 中可以捕获转储

```
set COMPlus_GCGenAnalysisDump=1
```



# 如何获取更多帮助

- [mem-doc](https://github.com/Maoni0/mem-doc) - <https://github.com/Maoni0/mem-doc>
  - 中文版刚出！李时的博客  
<https://www.cnblogs.com/Incerry/p/maoni-mem-doc.html>
- 在twitter上问我 - <https://twitter.com/maoni0>
- 在我们github repo上面发一个issue, 请告诉我们有助于帮助您的[信息](#) - <https://github.com/dotnet/runtime/>
  - 运行时的版本
  - 您已经做了哪些诊断
  - 性能数据

# 2021.NET Conf China

开源共建 | 开放创新 | 开发赋能



## THANKS!

2021 中国 .NET 开发者大会