# .NET性能优化的几点建议

赵劼 @ 2017.10


极客时间
重拾极客精神·提升技术认知
每天10分钟，邀请顶级技术专家，为你传道授业解惑。
扫一扫，试读专栏


主办方 Geekbang› 极客邦科技 InfoQ
ArchSummit
全 球 架 构 师 峰 会 2017
12月8-9日 北京 · 国际会议中心

APSEC 2017

南京大學 NANJING UNIVERSITY × Geekbang 极客邦科技 | InfoQ


AiCon
全球人工智能技术大会 2018
助力人工智能落地
2018.1.13 - 1.14　北京国际会议中心
AiCon
扫描关注大会官网

# 自我介绍

- 赵劼 / 赵姐夫 / Jeffrey Zhao
- 2011年前：互联网
- 2011年起：IBM / JPMorgan Chase & Co.
- 编程语言，代码质量，性能优化……
- 云计算，机器学习，大数据，AI……一窍不通

# 说在前面

- 先评测，再优化
- 专注优化瓶颈
- 重视性能，保持性能导向思维
- 随时优化，持续优化

We should forget about small efficiencies, say about 97% of the time: premature optimization is the root of all evil. *Yet we should not pass up our opportunities in that critical 3%.*

- Donald Knuth

# 零：兵来将挡，水来土掩

# 字符串拼接与StringBuilder

```
string Concat(string a, string b, string c, string d) {
  return a + b + c + d;
}
```

```
string Concat(string a, string b, string c, string d) {
  return new StringBuilder()
    .Append(a).Append(b)
    .Append(c).Append(d)
    .ToString();
}
```

# 字符串拼接与StringBuilder

```
string Concat(int n, string a, string b, string c, string
  var s = "";
  for (var i = 0; i < n; i++) {
    s = s + a + b + c + d;
  }
  return s;
}
```

```
string Concat(int n, string a, string b, string c, string
  var sb = new StringBuilder();
  for (var i = 0; i < n; i++) {
    sb.Append(a).Append(b)
      .Append(c).Append(d)
      .ToString();
  }
  return sb.ToString();
}
```

# 一：了解内存布局

# 老生常谈

- 引用类型
  - 分配在托管堆
  - 受GC管理，影响GC性能
  - 自带头数据（如类型信息）
- 值类型
  - 分配在栈上，或为引用类型对象的一部分
  - 分配在栈时不用显示回收
  - 没有头数据（体积紧凑）
- 注意：分配位置（堆/栈）为实现细节

# 获取对象尺寸

```
> !dumpheap -stat
              MT    Count    TotalSize Class Name
...
000007fef5c1fca0        5          120 System.Object
...

> !dumpheap -mt 000007fef5c1fca0
         Address               MT     Size
0000000002641408 000007fef5c1fca0       24
00000000026428a8 000007fef5c1fca0       24
0000000002642f48 000007fef5c1fca0       24
0000000002642f80 000007fef5c1fca0       24
0000000002645038 000007fef5c1fca0       24
```

- 优点：细节丰富，不含额外对象。
- 缺点：使用麻烦，不含对齐信息。

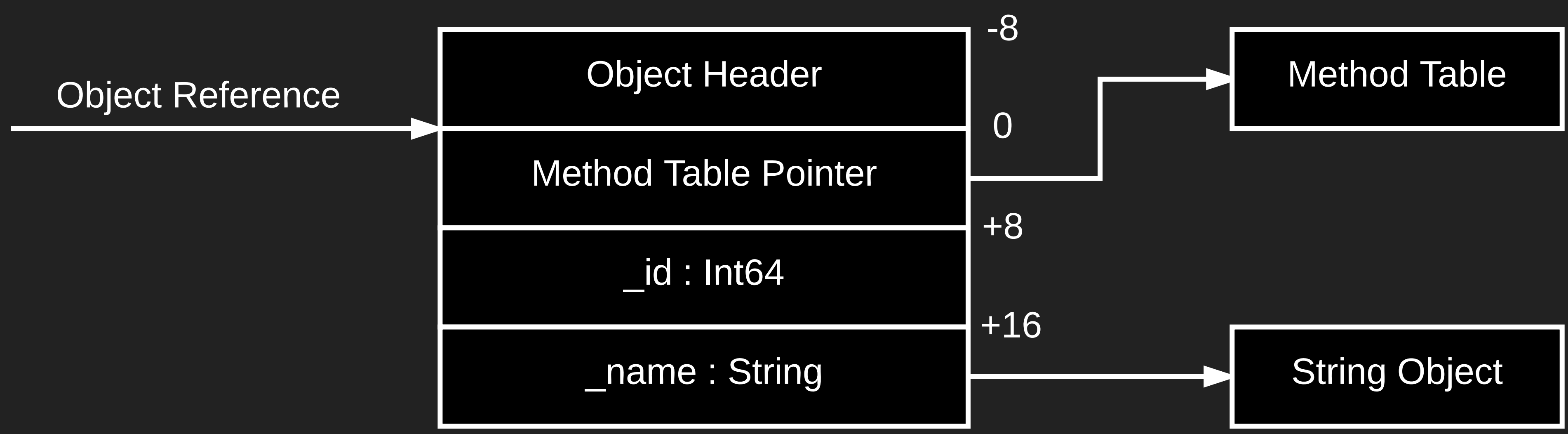
# 获取对象尺寸

```
var currentBytes = GC.GetTotalMemory(true);
var obj = new object(); // Or other types
var objSize = GC.GetTotalMemory(true) - currentBytes;

Console.WriteLine(objSize);

// Output:
// 12 in x86
// 24 in x64

GC.KeepAlive(obj);
```

- 优点：使用简单，包含对齐信息。
- 缺点：丢失细节，包含额外对象。

# 引用类型对象布局

```
class Person {
  private readonly long _id;
  private readonly string _name;
}
```



# 引用类型对象尺寸

```
class MyType1 {
  int Field1; // addr+8, 4 bytes
} // 24 bytes

class MyType2 {
  int Field1; // addr+8, 4 bytes
  int Field2; // addr+12, 4 bytes
} // 24 bytes

class MyType3 {
  int Field1;    // addr+8, 4 bytes
  string Field2; // addr+16, 8 bytes (alignment)
} // 32 bytes
```

# 基础类型数组尺寸

```
new int[0]; // 24 bytes (8 header + 8 MT + 4 length + 4)
new int[9]; // 64 bytes (24 + 4 * 9 + 4)
new int[10]; // 64 bytes (24 + 4 * 10)
new int[11]; // 72 bytes (24 + 4 * 11 + 4)

new byte[8]; // 32 bytes (24 + 1 * 8)
new byte[9]; // 32 bytes (24 + 1 * 9 + 7)

new bool[1]; // 32 bytes (24 + 1 * 1 + 7)
new bool[2]; // 32 bytes (24 + 1 * 2 + 6)
...
new bool[8]; // 32 bytes (24 + 1 * 8)

new string[0]; new string[5]; // ???
```

# 自定义值类型数组尺寸

```
struct MyStruct {
  bool Field1; // 1 bit
  int Field2;  // 4 bytes
  bool Field3; // 1 bit
}

new MyStruct[3]; // 64 bytes (24 + X * 3) => X = 12?
```

```
> !do 0000000002682e38
...
Fields:
 MT     Field   Offset                 Type ... Name
...   400004a        8       System.Boolean ... Field1
...   400004b        c         System.Int32 ... Field2
...   400004c       10       System.Boolean ... Field3
```

# 自定义值类型数组尺寸

```
[StructLayout(LayoutKind.Auto)]
struct MyStruct {
  bool Field1; // 1 bit
  int Field2;  // 4 bytes
  bool Field3; // 1 bit
}

new MyStruct[3]; // 48 bytes (24 + 8 * 3)
```

```
> !do 0000000002932e38
...
Fields:
 MT     Field   Offset                 Type ... Name
...   400004a        c       System.Boolean ... Field1
...   400004b        8         System.Int32 ... Field2
...   400004c        d       System.Boolean ... Field3
```

# 如何改进？

```
class MyItem { }

static IEnumerable<MyItem> GetItems() {
  // ...
}

// Initialization
var allItems = GetItems().ToArray();

// Iteration
foreach (var item in allItems) {
  // do something with item
}
```

```
class MyItem { MyItem Next; }

// Initialization
MyItem head = null;
foreach (var item in GetItems()) {
  head = new Item { Next = head };
}

Reverse(head); // optional

// Iteration
for (var item = head; item != null; item = item.Next) {
  // do something with item
}
```

# 改进后

- 节省内存（可忽略）、无大对象（重要）
- 指令少：少一层间接访问，无数组越界检查
- 布局紧凑：CPU缓存利用得当

# 延伸：双向链表

```
abstract class InplaceLinkedListNode<T>
  where T : InplaceLinkedListNode<T>
{
  T Prev;
  T Next;
} // or interface

class InplaceLinkedList<T>
  where T : InplaceLinkedListNode<T> { }

class MyItem : InplaceLinkedListNode<MyItem> { }
```

# 延伸：二叉树

```
abstract class InplaceBinaryTreeNode<T>
  where T : InplaceBinaryTreeNode<T>
{
  T Left;
  T Right;
  int Size;
} // or interface

class InplaceAvlTree<T>
  where T : InplaceBinaryTreeNode<T> { }

class MyItem : InplaceBinaryTreeNode<MyItem> { }
```

# 二：迎合GC编程

*You might think that building a responsive .NET Framework app is all about algorithms, such as using quick sort instead of bubble sort, but that's not the case. The biggest factor in building a responsive app is allocating memory, especially when your app is very large or processes large amounts of data.*

*- Roslyn Team*

# GC in CLR vs. OpenJDK

- 缺点：配置选项少。
  - Workstation GC vs. Server GC
  - Concurrent GC (Why disable?)
- 优点：配置选项少，但对内存控制程度高。
  - 与其调整参数，不如迎合GC编程
  - 拥有更多避免内存分配的特性

# 老生常谈：托管堆

- Gen 0：新对象，短期对象
- Gen 1：Gen 0和Gen 2之间的过渡
- Gen 2：老对象，长期对象
- 回收：扫描，清理，压缩

# 老生常谈：大对象堆（LOH）

- 保存大于85K的对象
- 随着Gen 2回收而回收
- 回收：不压缩，容易产生碎片

```
// Force compact once
GCSettings.LargeObjectHeapCompactionMode =
  GCLargeObjectHeapCompactionMode.CompactOnce;
```

# 特点

- Gen 0与Gen 1：频率高，速度快
- 大对象堆与Gen 2：频率低，速度慢
- 回收耗时只与存活对象数量有关
    - 与已分配对象数量无关
    - 避免少部分有用对象引用大量无用对象
    - 减少老代对象指向新代对象的引用

# 如何避免垃圾回收

- 避免内存分配
  - 绝大部分垃圾回收由内存分配引起
  - 其他原因：系统资源紧张、主动触发
- 停止垃圾回收

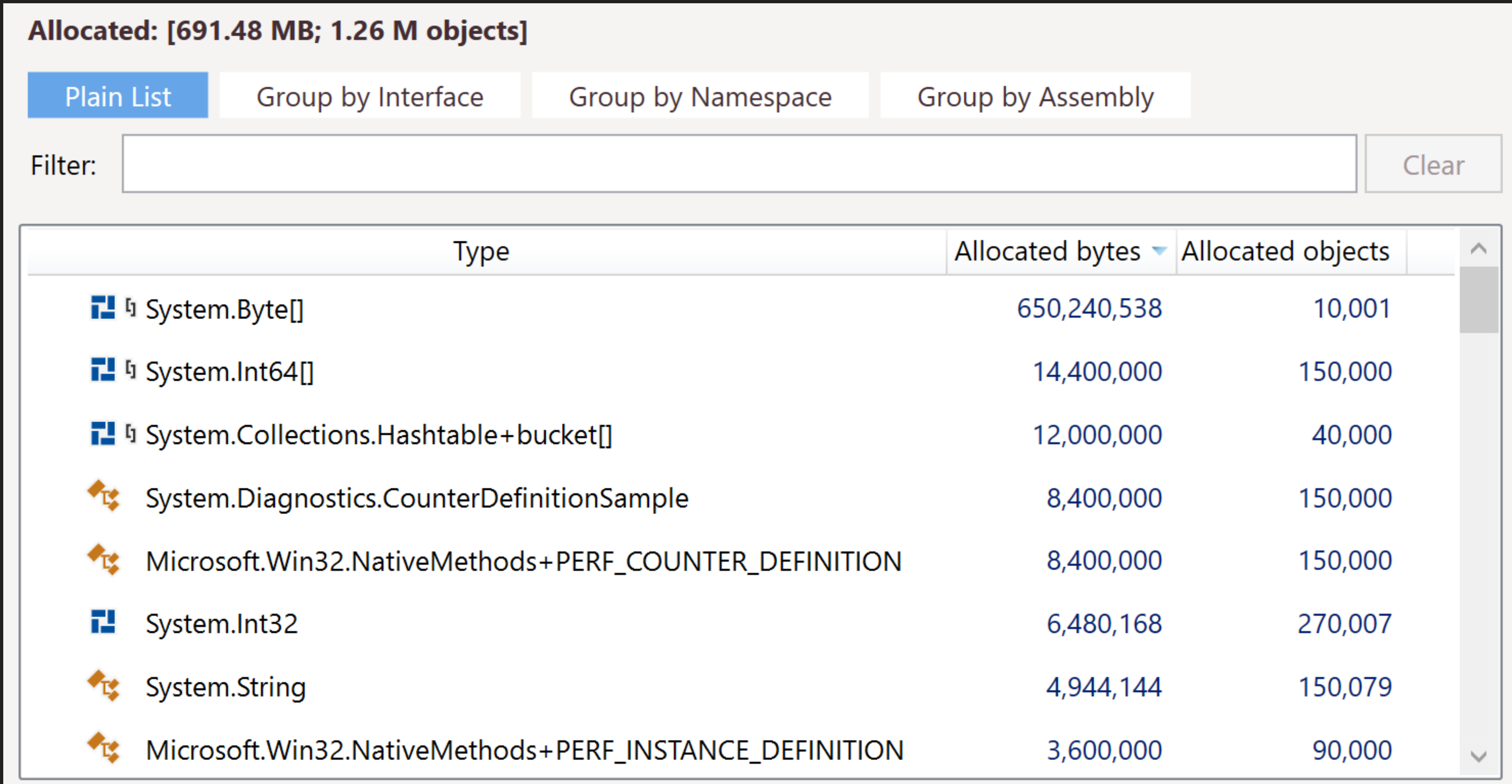
```
// Suppress Gen 2 collection (Workstation only)
GCSettings.LatencyMode = GCLatencyMode.LowLatency;
```

```
// Suppress full (non-concurrent) Gen 2.
GCSettings.LatencyMode = GCLatencyMode.SustainedLowLatency
```

```
GC.TryStartNoGCRegion(...);
// ... No GC will happen here
GC.EndNoGCRegion();
```

# 例：获取CPU使用率

```
// .NET 4.5.2-
var cpuCounter = new PerformanceCounter(
  "Processor",
  "% Processor Time",
  "_Total");

for (var i = 0; i < 10000; i++) {
  cpuCounter.NextValue();
}
```




| Type | Allocated bytes | Allocated objects |
|---|---|---|
| System.Byte[] | 650,240,538 | 10,001 |
| System.Int64[] | 14,400,000 | 150,000 |
| System.Collections.Hashtable+bucket[] | 12,000,000 | 40,000 |
| System.Diagnostics.CounterDefinitionSample | 8,400,000 | 150,000 |
| Microsoft.Win32.NativeMethods+PERF_COUNTER_DEFINITION | 8,400,000 | 150,000 |
| System.Int32 | 6,480,168 | 270,007 |
| System.String | 4,944,144 | 150,079 |
| Microsoft.Win32.NativeMethods+PERF_INSTANCE_DEFINITION | 3,600,000 | 90,000 |

# 关注分配细节

```
void Print(int i) {
  Console.WriteLine("i = " + i); // allocations?
}
```

```
void Print(int i) {
  // String.Concat(object o1, object o2)
  Console.WriteLine(String.Concat("i = ", i));
}
```

```
void Print(int i) {
  // String.Concat(string s1, string s2)
  Console.WriteLine("i = " + i.ToString()); // avoid boxi
}
```

# 关注分配细节

```
enum Color { Red, Green, Yellow }

class Light {
  public readonly Color Color;

  public override int GetHashCode() {
    return Color.GetHashCode(); // allocations?
  }
}
```

```
class Light {
  public override int GetHashCode() {
    return ((int)Color).GetHashCode(); // avoid boxing
  }
}
```

# 其他常见内存分配

- 委托（Delegate）
- 匿名函数（Lambda表达式）
- 字符串操作

# 如何优化？

```
string Concat(int n, string a, string b, string c, string
  var sb = new StringBuilder();

  for (var i = 0; i < n; i++) {
    sb.Append(a).Append(b)
      .Append(c).Append(d)
      .ToString();
  }

  return sb.ToString();
}
```

```
string Concat(int n, string a, string b, string c, string
  var length = CalculateLength(n, a, b, c, d);
  var sb = new StringBuilder(length);

  for (var i = 0; i < n; i++) {
    sb.Append(a).Append(b)
      .Append(c).Append(d)
      .ToString();
  }

  return sb.ToString();
}
```

# Roslyn：重用StringBuilder

```
static StringBuilder AcquireBuilder() {
  var result = cachedStringBuilder; // [ThreadStatic]
  if (result == null)
    return new StringBuilder();

  result.Clear();
  cachedStringBuilder = null;
  return result;
}
```

```
static string GetStringAndReleaseBuilder(StringBuilder sb)
    var result = sb.ToString();
    cachedStringBuilder = sb;
    return result;
}
```

```
string Concat(int n, string a, string b, string c, string
  var sb = AcquireBuilder();

  for (var i = 0; i < n; i++) {
    sb.Append(a).Append(b)
      .Append(c).Append(d)
      .ToString();
  }

  return GetStringAndReleaseBuilder(sb);
}
```

# 复用（大）对象

- 例：StringBuilder，字节数组，对象数组…
- 线程专用，或线程安全栈
- 一次分配到位，避免LOH碎片

# 如何优化？

```
void DoSomethingNeedsTempArrayOfLength<T>(int n) {
  var tempArray = new T[n];
  // ...
}
```

```
abstract class Buffer<T> : IDisposable {
  // array like operations
  public abstract T this[int i] { get; set; }
}

void DoSomethingNeedsTempArrayOfLength<T>(int n) {
  using (var tempBuffer = RequestBuffer<T>(n)) {
    // ...
  }
}
```

```
class StructBuffer<T> : Buffer {
  private readonly T[] _array;

  public override T this[int i] {
    get { return _array[i]; }
    set { _array[i] = value; }
  }
}
```

```
class ClassBuffer<T> : Buffer {
  private readonly object[] _array;

  public override T this[int i] {
    get { return (T)_array[i]; }
    set { _array[i] = value; }
  }
}
```

# 避免LOH碎片

- 预分配，避免自适应分配，例如
  - StringBuilder
  - MemoryStream
  - List<T>
- 分配统一尺寸的对象，或
- 统一尺寸的整数倍

# 理想中的对象生命周期

- 极短：分配后立即丢弃
  - 不会被GC扫描到
  - 不会被提升至Gen 2
- 极长：分配后永不丢弃
  - 快速提升至Gen 2后永久保留，避免压缩
  - 分配至LOH后永久保留，避免碎片

# 优化案例

```
class Order {
  public int OrderId { get; set; }
  public string Side { get; set; }
  public double Price { get; set; }
  // 200+ more fields ...
} // ~1.2KB

// 50000 instances max, ~60M
```

# 优化案例（结果）

```
struct OrderData {
  public int OrderId
  public string Side
  public double Price
  // 200+ more fields ...
} // ~1.2KB

// ~60M
static OrderData[] AllOrderData = new OrderData[50000];
```

# 优化案例（结果）

```
struct Order {
  private readonly int _index;

  public int OrderId {
    get { return AllOrderData[_index].OrderId; }
    set { AllOrderData[_index].OrderId = value; }
  }

  public string Side {
    get { return AllOrderData[_index].Side; }
    set { AllOrderData[_index].Side = value; }
  }
  ...
} // zero in heap, 4 bytes in stack
```

# 优化后

- 优点
  - 更少的对象
  - 更短的GC扫描时间
  - 永不涉及回收的对象
- 缺点
  - 驻留内存较多
  - 重用存储空间带来额外复杂度

# 延伸：双向链表

```
var list = new LinkedList<int>();
for (var i = 0; i < 10000; i++) {
  list.AddLast(i);
}
```

```
// class LinkedListNode<int> {
//    ... 16 bytes head ...
//    Prev -> 8 bytes
//    Next -> 8 bytes
//    List -> 8 bytes
//    Value -> 4 bytes
//    ... 4 bytes alignment ...
// }
// 48 bytes node to save 4 bytes data

// nodes are everywhere in heap
```

# 延伸：双向链表

```
class ArrayLinkedList<T> {
  private Node[] _nodes;

  public struct Node {
    public T Value;
    public int Next;
    public int Prev;
  }

  // "index" is node
  public int AddLast(T value) { }
}
```

```
// 12 bytes to save 4 bytes data
// nodes are kept closely (probably same cache line) in he
```

# 延伸：二叉树

```
class ArrayAvlTree<T> {
  private Node[] _nodes;

  public struct Node {
    public T Value;
    public int Left;
    public int Right;
    public int Size;
  }

  // "index" is node
  public int Add(T value) { }
}
```

# 三、编写不通用的代码

# 代码调用

```
public class MyClass {
    [MethodImpl(MethodImplOptions.NoInlining)]
    public void NormalMethod() { }

    public virtual void VirtualMethod() { }
}
```

```
void CallNormal(MyClass c) {
    c.NormalMethod();
}

void CallVirtual(MyClass c) {
    c.VirtualMethod();
}
```

# 代码调用（非虚方法）

```
void CallNormal(MyClass c) {
    c.NormalMethod();
}
```

```
488bca      mov rcx, rdx
3909        cmp [rcx], ecx ; null check
e8a2a4ffff  call 00007ffe`80c47b40 (MyClass.NormalMethod()
```

# 代码调用（虚方法）

```
void CallVirtual(MyClass c) {
    c.VirtualMethod();
}
```

```
488bca     mov rcx, rdx
488b02     mov rax, [rdx]          ; address of method table
488b4040   mov rax, [rax+0x40]     ; address of methods
ff5020     call qword [rax+0x20] ; address of target methoc
```

# 集合

```
static ICollection<T> CreateCollection(IEnumerable<T> iter
  return name.ToList();
}
```

*In Visual Studio and the new compilers, analysis shows that many of the dictionaries contained a single element or were empty. An empty Dictionary has ten fields and occupies 48 bytes on the heap on an x86 machine. Dictionaries are great when you need a mapping or associative data structure with constant time lookup. However, when you have only a few elements, you waste a lot of space by using a Dictionary.*

*- Roslyn Team*

# 集合创建（Roslyn）

```
static ICollection<string> CreateReadOnlyMemberNames(
  HashSet<string> names) {
  switch (names.Count) {
    case 0:
      return SpecializedCollections.EmptySet<string>();
    case 1:
      return SpecializedCollections.Singleton(names.First
    case 2~6:
      return ImmutableArray.CreateRange(names);
    default:
      return SpecializedCollections.ReadOnlySet(names);
  }
  return names.ToList();
}
```

# FrugalList/Map in WPF

```
class FrugalListBase<T> { }
class SingleItemList<T> : FrugalListBase<T> { }
class ThreeItemList<T> : FrugalListBase<T> { }
class SixItemList<T> : FrugalListBase<T> { }
class ArrayItemList<T> : FrugalListBase<T> { }
```

```
class FrugalMapBase { }
class SingleObjectMap : FrugalMapBase { }
class ThreeObjectMap : FrugalMapBase { }
class SixObjectMap : FrugalMapBase { }
class ArrayObjectMap : FrugalMapBase { }
class SortedObjectMap : FrugalMapBase { }
class HashObjectMap : FrugalMapBase { }
class LargeSortedObjectMap : FrugalMapBase { }
```

# 不通用的迭代器

```
class MyCollection<T> : IEnumerable<T> {
  public struct MyEnumerator : IEnumerator<T> {
    // ...
  }

  public MyEnumerator GetEnumerator() {
    // ...
  }
```

```
  IEnumerator<T> IEnumerable<T>.GetEnumerator() {
    return GetEnumerator();
  }

  IEnumerator IEnumerable.GetEnumerator() {
    return GetEnumerator();
  }
}
```

# 不通用的迭代器

```
var coll = new MyCollection<int>();

// has allocations:
// 1. ((IEnumerable<int>)coll).Where(...)
// 2. anonymous method
foreach (var i in coll.Where(i => i > 0)) {
  // Do something
}
```

```
// no allocation
foreach (var i in coll) {
  if (i > 0) {
    // Do something
  }
}
```

# 简易IDisposable实现

```
class MyClass {}
  private void Done(int i) {
    // ...
  }

  public IDisposable DoSomething(int i) {
    // ...
    return Disposables.Create(() => Done(i));
  }
}
```

```
// Has allocations
using (myClass.DoSomething(10)) {
  // Do something
}
```

# 不通用的IDisposable实现

```
class MyClass {
  public struct DoneToken : IDisposable {
    private readonly MyClass _parent;
    private readonly int _i;

    public void Dispose() {
      _parent.Done(_i);
    }
  }
```

```
  // no allocation
  public DoneToken DoSomething(int i) {
    // ...
    return new DoneToken(this, i);
  }
}
```

# 使用最新版本的C#/.NET

- ValueTask：轻量级Task
- ValueTuple：轻量级Tuple
- Span<T>：泛化版的String/ArraySegment
- ref return：安全地返回一个地址
- readonly ref：按引用传递只读变量

# 无ref return

```
interface IMap<TKey, TValue> {
  TValue Get(TKey key);
  void Set(TKey key, TValue value);
}
```

```
void Increase(IMap<TKey, int> map, TKey key, int n) {
  var value = map.Get(key);
  map.Set(key, value + n);
}
```

# ref return

```
interface IMap<TKey, TValue> {
  ref TValue GetRef(TKey key);
}
```

```
void Increase(IMap<TKey, int> map, TKey key, int n) {
  map.GetRef(key) += n;
}
```

# 五、善用工具

# 常备工具

- WinDBG
- PerfView
- 某商业化CPU Profiler
- 某商业化Memory Profiler
- CLRMD

# CLRMD

```
CLRRuntime runtime = ...;
GCHeap heap = runtime.GetHeap();

foreach (ulong obj in heap.EnumerateObjects()) {
    GCHeapType type = heap.GetObjectType(obj);

    if (type == null) // heap corruption
      continue;

    ulong size = type.GetSize(obj);
    Console.WriteLine(
      "{0,12:X} {1,8:n0} {2,1:n0} {3}",
      obj, size, heap.GetObjectGeneration(obj), type.Name)
}
```

# 建议回顾

- 兵来将挡，水来土掩
- 了解内存布局
- 迎合GC编程
- 编写不通用代码
- 善用工具

Q & A