

昵称: zblade  
园龄: 4年3个月  
粉丝: 73  
关注: 1  
+加关注

## Unity优化之GC——合理优化Unity的GC

转载请标明出处<http://www.cnblogs.com/zblade/>

最近有点繁忙，白天干活晚上抽空写点翻译，还要运动，所以翻译工作进行的有点缓慢。  
=。=

PS: 最近重新回来更新了一遍，文章还是需要反复修改才能写的顺畅，多谢各位的支持：  
D

本文续接前面的unity的渲染优化，进一步翻译Unity中的GC优化，英文链接在下：[英文地址](#)

### 介绍：

在游戏运行的时候，数据主要存储在内存中，当游戏的数据在不需要的时候，存储当前数据的内存就可以被回收以再次使用。内存垃圾是指当前废弃数据所占用的内存，垃圾回收（GC）是指将废弃的内存重新回收再次使用的过程。

Unity中将垃圾回收当作内存管理的一部分，如果游戏中废弃数据占用内存较大，则游戏的性能会受到极大影响，此时垃圾回收会成为游戏性能的一大障碍点。

本文我们主要学习垃圾回收的机制，垃圾回收如何被触发以及如何提GC收效率来提高游戏的性能。

### Unity内存管理机制简介

要了解垃圾回收如何工作以及何时被触发，我们首先需要了解unity的内存管理机制。Unity主要采用自动内存管理的机制，开发时在代码中不需要详细地告诉unity如何进行内存管理，unity内部自身会进行内存管理。这和使用C++开发需要随时管理内存相比，有一定的优势，当然带来的劣势就是需要随时关注内存的增长，不要让游戏在手机上跑“飞”了。

unity的自动内存管理可以理解为以下几个部分：

- 1) unity内部有两个内存管理池：堆内存和堆栈内存。堆栈内存(stack)主要用来存储较小的和短暂的数据，堆内存(heap)主要用来存储较大的和存储时间较长的数据。
- 2) unity中的变量只会在堆栈或者堆内存上进行内存分配，变量要么存储在堆栈内存上，要么处于堆内存上。
- 3) 只要变量处于激活状态，则其占用的内存会被标记为使用状态，则该部分的内存处于被分配的状态。
- 4) 一旦变量不再激活，则其所占用的内存不再需要，该部分内存可以被回收回到内存池中被再次使用，这样的操作就是内存回收。处于堆栈上的内存回收及其快速，处于堆上的内存并不是及时回收的，此时其对应的内存依然会被标记为使用状态。
- 5) 垃圾回收主要是指堆上的内存分配和回收，unity中会定时对堆内存进行GC操作。

在了解了GC的过程后，下面详细了解堆内存和堆栈内存的分配和回收机制的差别。

### 堆栈内存分配和回收机制

堆栈上的内存分配和回收十分快捷简单，因为堆栈上只会存储短暂的或者较小的变量。内存分配和回收都会以一种顺序和大小可控制的形式进行。

2018年11月						
日	一	二	三	四	五	六
28	29	30	31	1	2	3
4	5	6	7	8	9	10
11	12	13	14	15	16	17
18	19	20	21	22	23	24
25	26	27	28	29	30	1
2	3	4	5	6	7	8

## 搜索

<input type="text"/>	<input type="button" value="找找看"/>
<input type="text"/>	<input type="button" value="谷歌搜索"/>

## 常用链接

[我的随笔](#)  
[我的评论](#)  
[我的参与](#)  
[最新评论](#)  
[我的标签](#)

## 随笔分类

[Unity Shader笔记\(4\)](#)  
[Unity3D个人学习笔记\(14\)](#)  
[解忧bug店\(2\)](#)  
[游戏随笔栏\(16\)](#)

## 随笔档案

[2018年11月 \(2\)](#)  
[2018年10月 \(1\)](#)  
[2018年8月 \(2\)](#)  
[2018年7月 \(2\)](#)  
[2018年6月 \(1\)](#)  
[2018年5月 \(5\)](#)  
[2018年4月 \(6\)](#)  
[2018年2月 \(1\)](#)  
[2017年12月 \(1\)](#)  
[2017年10月 \(2\)](#)  
[2017年9月 \(2\)](#)  
[2017年6月 \(1\)](#)  
[2017年5月 \(2\)](#)  
[2017年3月 \(4\)](#)  
[2017年2月 \(4\)](#)

## 文章分类

[Unity3D个人学习笔记](#)

## 最新评论

1. Re: Unity实现c#热更新方案探究(三)

堆栈的运行方式就像[stack](#)：其本质只是一个数据的集合，数据的进出都以一种固定的方式运行。正是这种简洁性和固定性使得堆栈的操作十分快捷。当数据被存储在堆栈上的时候，只需要简单地在其后进行扩展。当数据失效的时候，只需要将其从堆栈上移除。

## 堆内存分配和回收机制

堆内存上的内存分配和存储相对而言更加复杂，主要是堆内存上可以存储短期较小的数据，也可以存储各种类型和大小的数据。其上的内存分配和回收顺序并不可控，可能会要求分配不同大小的内存单元来存储数据。

堆上的变量在存储的时候，主要分为以下几步：

1) 首先，unity检测是否有足够的闲置内存单元用来存储数据，如果有，则分配对应大小的内存单元；

2) 如果没有足够的存储单元，unity会触发垃圾回收来释放不再被使用的堆内存。这步操作是一步缓慢的操作，如果垃圾回收后有足够大小的内存单元，则进行内存分配。

3) 如果垃圾回收后并没有足够的内存单元，则unity会扩展堆内存的大小，这步操作会很缓慢，然后分配对应大小的内存单元给变量。

堆内存的分配有可能会变得十分缓慢，特别是在需要垃圾回收和堆内存需要扩展的情况下，通常需要减少这样的操作次数。

## 垃圾回收时的操作

当堆内存上一个变量不再处于激活状态的时候，其所占用的内存并不会立刻被回收，不再使用的内存只会在GC的时候才会被回收。

每次运行GC的时候，主要进行下面的操作：

- 1) GC会检查堆内存上的每个存储变量；
- 2) 对每个变量会检测其引用是否处于激活状态；
- 3) 如果变量的引用不再处于激活状态，则会被标记为可回收；
- 4) 被标记的变量会被移除，其所占有的内存会被回收到堆内存上。

GC操作是一个极其耗费的，堆内存上的变量或者引用越多则其运行的操作会更多，耗费的时间越长。

## 何时会触发垃圾回收

主要有三个操作会触发垃圾回收：

- 1) 在堆内存上进行内存分配操作而内存不够的时候都会触发垃圾回收来利用闲置的内存；
- 2) GC会自动的触发，不同平台运行频率不一样；
- 3) GC可以被强制执行。

特别是在堆内存上进行内存分配时内存单元不足的时候，GC会被频繁触发，这就意味着频繁在堆内存上进行内存分配和回收会触发频繁的GC操作。

## GC操作带来的问题

在了解GC在unity内存管理中的作用后，我们需要考虑其带来的问题。最明显的问题是GC操作会需要大量的时间来运行，如果堆内存上有大量的变量或者引用需要检查，则检查的操作会十分缓慢，这就会使得游戏运行缓慢。其次GC可能会在关键时候运行，例如在CPU处于游戏的性能运行关键时刻，此时任何一个额外的操作都可能会带来极大的影响，使得游戏帧率下降。

另外一个GC带来的问题是堆内存的碎片化。当一个内存单元从堆内存上分配出来，其大小取决于其存储的变量的大小。当该内存被回收后堆内存上的时候，有可能使得堆内存被分割成碎片化的单元。也就是说堆内存总体可以使用的内存单元较大，但是单独的内存单元较小，在下一次内存分配的时候不能找到合适大小的存储单元，这也会触发GC操作或者堆内存扩展操作。

@Real陈麟看一下系列文章2开头的解释文章...

--zblade

2. Re:Unity实现c#热更新方案探究(三)  
为什么要使用IRuntime而不直接使用创建Appdomain呢？不能跨平台？

--Real陈麟

3. Re:Unity实现c#热更新方案探究(三)  
学习了，干货满满的博客

--Real陈麟

4. Re:MMORPG战斗系统随笔（一）、战斗系统流程简介  
好字儿！

--OptimusGuardian

5. Re:MMORPG战斗系统随笔（三）、AI系统简介

@zblade引用@TesterWei最近希望静下心来再好好琢磨一下整个游戏框架，以前只局限于逻辑开发，最近开始慢慢整理如何学习整个框架，任重道远，共勉同感 w w w.l earun.cn?fuid=.....

--张乾坤

## 阅读排行榜

1. Unity优化之GC——合理优化Unity的GC(14645)
2. MMORPG战斗系统随笔（三）、AI系统简介(6625)
3. 对C#热更新方案ILRuntime的探究(4038)
4. Unity渲染优化中文翻译（三）——GPU的优化策略(3243)
5. unity静态批处理原理解(3013)

## 评论排行榜

1. unity静态批处理原理解(17)
2. lua中 table 重构index/pairs元方法优化table内存占用(11)
3. MMORPG战斗系统随笔（三）、AI系统简介(7)
4. MMORPG战斗系统随笔（一）、战斗系统流程简介(7)
5. Unity中溶解shader的总结(6)

## 推荐排行榜

1. MMORPG战斗系统随笔（三）、AI系统简介(10)
2. Unity使用C++作为游戏逻辑脚本的研究（二）(6)
3. MMORPG战斗系统随笔（一）、战斗系统流程简介(5)
4. Lua和C#调用探秘(4)
5. Unity实现c#热更新方案探究(三)(4)

堆内存碎片会造成两个结果，一个是游戏占用的内存会越来越大，一个是GC会更加频繁地被触发。

## 分析GC带来的问题

GC操作带来的问题主要表现为帧率运行低，性能间歇中断或者降低。如果游戏有这样的表现，则首先需要打开unity中的profiler window来确定是否是GC造成。

了解如何运用profiler window，可以参考[此处](#)，如果游戏确实是由GC造成的，可以继续阅读下面的内容。

## 分析堆内存的分配

如果GC造成游戏的性能问题，我们需要知道游戏中的哪部分代码会造成GC，内存垃圾在变量不再激活的时候产生，所以首先我们需要知道堆内存上分配的是什么变量。

### 堆内存和堆栈内存分配的变量类型

在Unity中，值类型变量都在堆栈上进行内存分配，其他类型的变量都在堆内存上分配。如果你不知道值类型和引用类型的差别，可以查看[此处](#)。

下面的代码可以用来理解值类型的分配和释放,其对应的变量在函数调用完后会立即回收：

```
void ExampleFunction()
{
    int localInt = 5;
}
```

对应的引用类型的参考代码如下，其对应的变量在GC的时候才回收：

```
void ExampleFunction()
{
    List localList = new List();
}
```

### 利用profiler window 来检测堆内存分配：

我们可以在profiler window中检查堆内存的分配操作：在CPU usage分析窗口中，我们可以检测任何一帧cpu的内存分配情况。其中一个选项是GC Alloc，通过分析其来定位是什么函数造成大量的堆内存分配操作。一旦定位该函数，我们就可以分析解决其造成问题的原因从而减少内存垃圾的产生。现在Unity5.5的版本，还提供了deep profiler的方式深度分析GC垃圾的产生。

## 降低GC的影响的方法

大体上来说，我们可以通过三种方法来降低GC的影响：

- 1) 减少GC的运行次数；
- 2) 减少单次GC的运行时间；
- 3) 将GC的运行时间延迟，避免在关键时候触发，比如可以在场景加载的时候调用GC

似乎看起来很简单，基于此，我们可以采用三种策略：

1) 对游戏进行重构，减少堆内存的分配和引用的分配。更少的变量和引用会减少GC操作中的检测个数从而提高GC的运行效率。

2) 降低堆内存分配和回收的频率，尤其是在关键时刻。也就是说更少的事件触发GC操作，同时也降低堆内存的碎片化。

3) 我们可以试着测量GC和堆内存扩展的时间，使其按照可预测的顺序执行。当然这样操作的难度极大，但是这会大大降低GC的影响。

## 减少内存垃圾的数量

减少内存垃圾主要可以通过一些方法来减少：

### 缓存

如果在代码中反复调用某些造成堆内存分配的函数但是其返回结果并没有使用，这就会造成不必要的内存垃圾，我们可以缓存这些变量来重复利用，这就是缓存。

例如下面的代码每次调用的时候就会造成堆内存分配，主要是每次都会分配一个新的数组：

```
1 void OnTriggerEnter(Collider other)
2 {
3     Renderer[] allRenderers = FindObjectsOfType<Renderer>();
4     ExampleFunction(allRenderers);
5 }
```

对比下面的代码，只会生产一个数组用来缓存数据，实现反复利用而不需要造成更多的内存垃圾：

```
1 private Renderer[] allRenderers;
2
3 void Start()
4 {
5     allRenderers = FindObjectsOfType<Renderer>();
6 }
7
8 void OnTriggerEnter(Collider other)
9 {
10     ExampleFunction(allRenderers);
11 }
```

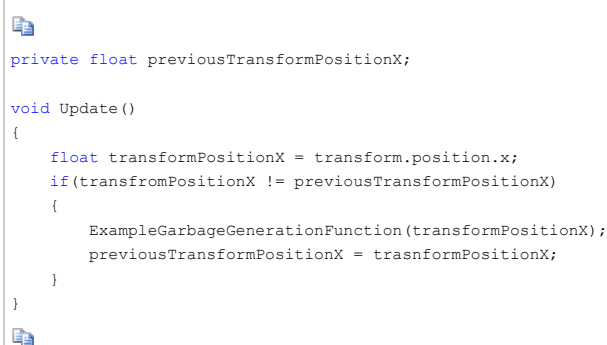
### 不要在频繁调用的函数中反复进行堆内存分配

在MonoBehaviour中，如果我们需要进行堆内存分配，最坏的情况就是在其反复调用的函数中进行堆内存分配，例如Update()和LateUpdate()函数这种每帧都调用的函数，这会造成大量的内存垃圾。我们可以考虑在Start()或者Awake()函数中进行内存分配，这样可以减少内存垃圾。

下面的例子中，update函数会多次触发内存垃圾的产生：

```
1 void Update()
2 {
3     ExampleGarbageGenerationFunction(transform.position.x);
4 }
```

通过一个简单的改变，我们可以确保每次在x改变的时候才触发函数调用，这样避免每帧都进行堆内存分配：



```
private float previousTransformPositionX;

void Update ()
{
    float transformPositionX = transform.position.x;
    if(transformPositionX != previousTransformPositionX)
    {
        ExampleGarbageGenerationFunction(transformPositionX);
        previousTransformPositionX = transformPositionX;
    }
}
```

另外的一种方法是在update中采用计时器，特别是在运行有规律但是不需要每帧都运行的代码中，例如：

```
1 void Update()
2 {
3     ExampleGarbageGeneratingFunction()
4 }
```

通过添加一个计时器，我们可以确保每隔1s才触发该函数一次：

```
1 private float timeSinceLastCalled;
2 private float delay = 1f;
3 void Update()
```

```

4  {
5      timSinceLastCalled += Time.deltaTime;
6      if(timeSinceLastCalled > delay)
7      {
8          ExampleGarbageGenerationFunction();
9          timeSinceLastCalled = 0f;
10     }
11 }
12

```

通过这样细小的改变，我们可以使得代码运行的更快同时减少内存垃圾的产生。

**附：**不要忽略这一个方法，在最近的项目性能优化中，我经常采用这样的方法来优化游戏的性能，很多对于固定时间的事件回调函数中，如果每次都分配新的缓存，但是在操作完后并不释放，这样就会造成大量的内存垃圾，对于这样的缓存，最好的办法就是当前周期回调后执行清除或者标志为废弃。

### 清除链表

在堆内存上进行链表的分配的时候，如果该链表需要多次反复的分配，我们可以采用链表的clear函数来清空链表从而替代反复多次的创建分配链表。

```

1  void Update()
2  {
3      List myList = new List();
4      PopulateList(myList);
5  }

```

通过改进，我们可以将该链表只在第一次创建或者该链表必须重新设置的时候才进行堆内存分配，从而大大减少内存垃圾的产生：

```

1  private List myList = new List();
2  void Update()
3  {
4      myList.Clear();
5      PopulateList(myList);
6  }

```

### 对象池

即便我们在代码中尽可能地减少堆内存的分配行为，但是如果游戏有大量的对象需要产生和销毁依然会造成GC。对象池技术可以通过重复使用对象来降低堆内存的分配和回收频率。对象池在游戏中广泛的使用，特别是在游戏中需要频繁的创建和销毁相同的游戏对象的时候，例如枪的子弹这种会频繁生成和销毁的对象。

要详细的讲解对象池已经超出本文的范围，但是该技术值得我们深入的研究[This tutorial on object pooling on the Unity Learn site](#)对于对象池有详细深入的讲解。

**附：**对象池技术属于游戏中比较通用的技术，如果有闲余时间，大家可以学习一下这方面的知识。

### 造成不必要的堆内存分配的因素

我们已经知道值类型变量在堆栈上分配，其他的变量在堆内存上分配，但是任然有一些情况下的堆内存分配会让我们感到吃惊。下面让我们分析一些常见的不必要的堆内存分配行为并对其进行优化。

#### 字符串

在c#中，字符串是引用类型变量而不是值类型变量，即使看起来它是存储字符串的值的。这就意味着字符串会造成一定的内存垃圾，由于代码中经常使用字符串，所以我们需要对其格外小心。

c#中的字符串是不可变更的，也就是说其内部的值在创建后是不可被变更的。每次在对字符串进行操作的时候（例如运用字符串的“加”操作），unity会新建一个字符串用来存储新的字符串，使得旧的字符串被废弃，这样就会造成内存垃圾。

我们可以采用以下的一些方法来最小化字符串的影响：

1) 减少不必要的字符串的创建，如果一个字符串被多次利用，我们可以创建并缓存该字符串。

2) 减少不必要的字符串操作，例如如果在Text组件中，有一部分字符串需要经常改变，但是其他部分不会，则我们可以将其分为两个部分的组件，对于不变的部分就设置为类似常量字符串即可，见下面的例子。

3) 如果我们需要实时的创建字符串，我们可以采用StringBuilderClass来代替，String Builder专为不需要进行内存分配而设计，从而减少字符串产生的内存垃圾。

4) 移除游戏中的Debug.Log()函数的代码，尽管该函数可能输出为空，对该函数的调用依然会执行，该函数会创建至少一个字符（空字符）的字符串。如果游戏中有大量的该函数的调用，这会造成内存垃圾的增加。

在下面的代码中，在Update函数中会进行一个string的操作，这样的操作就会造成不必要的内存垃圾：

```
1 public Text timerText;
2 private float timer;
3 void Update()
4 {
5     timer += Time.deltaTime;
6     timerText.text = "Time:" + timer.ToString();
7 }
```

通过将字符串进行分隔，我们可以剔除字符串的加操作，从而减少不必要的内存垃圾：

```
1 public Text timerHeaderText;
2 public Text timerValueText;
3 private float timer;
4 void Start()
5 {
6     timerHeaderText.text = "TIME:";
7 }
8
9 void Update()
10 {
11     timerValueText.text = timer.ToString();
12 }
```

## Unity函数调用

在代码编程中，当我们调用不是我们自己编写的代码，无论是Unity自带的还是插件中的，我们都可能会产生内存垃圾。Unity的某些函数调用会产生内存垃圾，我们在使用的时候需要注意它的使用。

这儿没有明确的列表指出哪些函数需要注意，每个函数在不同的情况下有不同的使用，所以最好仔细地分析游戏，定位内存垃圾的产生原因以及如何解决问题。有时候缓存是一种有效的办法，有时候尽量降低函数的调用频率是一种办法，有时候用其他函数来重构代码是一种办法。现在来分析unity中常见的造成堆内存分配的函数调用。

在Unity中如果函数需要返回一个数组，则一个新的数组会被分配出来用作结果返回，这不容易被注意到，特别是如果该函数含有迭代器，下面的代码中对于每个迭代器都会产生一个新的数组：

```
void ExampleFunction()
{
    for(int i=0; i < myMesh.normals.Length;i++)
    {
        Vector3 normal = myMesh.normals[i];
    }
}
```

对于这样的问题，我们可以缓存一个数组的引用，这样只需要分配一个数组就可以实现相同的功能，从而减少内存垃圾的产生：

```
1 void ExampleFunction()
2 {
```



```

3 |     Vector3[] meshNormals = myMesh.normals;
4 |     for(int i=0; i < meshNormals.Length;i++)
5 |     {
6 |         Vector3 normal = meshNormals[i];
7 |     }
8 | }

```

此外另外的一个函数调用GameObject.name 或者 GameObject.tag也会造成预想不到的堆内存分配，这两个函数都会将结果存为新的字符串返回，这就会造成不必要的内存垃圾，对结果进行缓存是一种有效的办法，但是在Unity中都对应的有相关的函数来替代。对于比较gameObject的tag，可以采用GameObject.CompareTag()来替代。

在下面的代码中，调用gameObject.tag就会产生内存垃圾：

```

1 | private string playerTag="Player";
2 | void OnTriggerEnter(Collider other)
3 | {
4 |     bool isPlayer = other.gameObject.tag == playerTag;
5 | }

```

采用GameObject.CompareTag()可以避免内存垃圾的产生：

```

1 | private string playerTag = "Player";
2 | void OnTriggerEnter(Collider other)
3 | {
4 |     bool isPlayer = other.gameObject.CompareTag(playerTag);
5 | }

```

不只是GameObject.CompareTag，unity中许多其他的函数也可以避免内存垃圾的生成。比如我们可以用Input.GetTouch()和Input.touchCount()来代替Input.touches，或者用Physics.SphereCastNonAlloc()来代替Physics.SphereCastAll()。

## 装箱操作

装箱操作是指一个值类型变量被用作引用类型变量时候的内部变换过程，如果我们向带有对象类型参数的函数传入值类型，这就会触发装箱操作。比如String.Format()函数需要传入字符串和对象类型参数，如果传入字符串和int类型数据，就会触发装箱操作。如下面代码所示：

```

1 | void ExampleFunction()
2 | {
3 |     int cost = 5;
4 |     string displayString = String.Format("Price:{0} gold",cost);
5 | }

```

在Unity的装箱操作中，对于值类型会在堆内存上分配一个System.Object类型的引用来封装该值类型变量，其对应的缓存就会产生内存垃圾。装箱操作是非常普遍的一种产生内存垃圾的行为，即使代码中没有直接的对变量进行装箱操作，在插件或者其他的函数中也有可能产生。最好的解决办法是尽可能的避免或者移除造成装箱操作的代码。

## 协程

调用 StartCoroutine()会产生少量的内存垃圾，因为unity会生成实体来管理协程。所以在游戏的关键时刻应该限制该函数的调用。基于此，任何在游戏关键时刻调用的协程都需要特别的注意，特别是包含延迟回调的协程。

yield在协程中不会产生堆内存分配，但是如果yield带有参数返回，则会造成不必要的内存垃圾，例如：

```

1 | yield return 0;

```

由于需要返回0，引发了装箱操作，所以会产生内存垃圾。这种情况下，为了避免内存垃圾，我们可以这样返回：

```

1 | yield return null;

```

另外一种对协程的错误使用是每次返回的时候都new同一个变量，例如：

```

1 | while(!isComplete)
2 | {
3 |     yield return new WaitForSeconds(1f);

```

```
4 } }
```

我们可以采用缓存来避免这样的内存垃圾产生：

```
1 WaitForSeconds delay = new WaitForSeconds(1f);
2 while(!isComplete)
3 {
4     yield return delay;
5 }
```

如果游戏中的协程产生了内存垃圾，我们可以考虑用其他的方式来替代协程。重构代码对于游戏而言十分复杂，但是对于协程而言我们也可以注意一些常见的操作，比如如果用协程来管理时间，最好在update函数中保持对时间的记录。如果用协程来控制游戏中事件的发生顺序，最好对于不同事件之间有一定的信息通信的方式。对于协程而言没有适合各种情况的方法，只有根据具体的代码来选择最好的解决办法。

### foreach 循环

在unity5.5以前的版本中，在foreach的迭代中都会生成内存垃圾，主要来自于其后的装箱操作。每次在foreach迭代的时候，都会在堆内存上生产一个System.Object用来实现迭代循环操作。在unity5.5中解决了这个问题，比如，在unity5.5以前的版本中，用foreach实现循环：

```
1 void ExampleFunction(List listOfInts)
2 {
3     foreach(int currentInt in listOfInts)
4     {
5         DoSomething(currentInt);
6     }
7 }
```

如果游戏工程不能升级到5.5以上，则可以用for或者while循环来解决这个问题，所以可以改为：

```
1 void ExampleFunction(List listOfInts)
2 {
3     for(int i=0; i < listOfInts.Count; i++)
4     {
5         int currentInt = listOfInts[i];
6         DoSomething(currentInt);
7     }
8 }
```

### 函数引用

函数的引用，无论是指向匿名函数还是显式函数，在unity中都是引用类型变量，这都会在堆内存上进行分配。匿名函数的调用完成后都会增加内存的使用和堆内存的分配。具体函数的引用和终止都取决于操作平台和编译器设置，但是如果减少GC最好减少函数的引用。

### LINQ和常量表达式

由于LINQ和常量表达式以装箱的方式实现，所以在使用的时候最好进行性能测试。

### 重构代码来减小GC的影响

即使我们减小了代码在堆内存上的分配操作，代码也会增加GC的工作量。最常见的增加GC工作量的方式是让其检查它不必检查的对象。struct是值类型的变量，但是如果struct中包含有引用类型的变量，那么GC就必须检测整个struct。如果这样的操作很多，那么GC的工作量就大大增加。在下面的例子中struct包含一个string，那么整个struct都必须在GC中被检查：

```
1 public struct ItemData
2 {
3     public string name;
4     public int cost;
5     public Vector3 position;
6 }
7 private ItemData[] itemData;
```



我们可以将该struct拆分为多个数组的形式，从而减小GC的工作量：

```
1 private string[] itemNames;
2 private int[] itemCosts;
3 private Vector3[] itemPositions;
```

另外一种在代码中增加GC工作量的方式是保存不必要的Object引用，在进行GC操作的时候会对堆内存上的object引用进行检查，越少的引用就意味着越少的检查工作量。在下面的例子中，当前的对话框中包含一个对下一个对话框引用，这就使得GC的时候会去检查下一个对象框：

```
1 public class DialogData
2 {
3     private DialogData nextDialog;
4     public DialogData GetNextDialog()
5     {
6         return nextDialog;
7     }
8 }
9 }
```

通过重构代码，我们可以返回下一个对话框实体的标记，而不是对话框实体本身，这样就没有多余的object引用，从而减少GC的工作量：

```
1 public class DialogData
2 {
3     private int nextDialogID;
4     public int GetNextDialogID()
5     {
6         return nextDialogID;
7     }
8 }
```

当然这个例子本身并不重要，但是如果我们的游戏中包含大量的含有对其他Object引用的object，我们可以考虑通过重构代码来减少GC的工作量。

## 定时执行GC操作

### 主动调用GC操作

如果我们知道堆内存存在被分配后并没有被使用，我们希望可以主动地调用GC操作，或者在GC操作并不影响游戏体验的时候（例如场景切换的时候），我们可以主动的调用GC操作：

```
1 System.GC.Collect();
```

通过主动的调用，我们可以主动驱使GC操作来回收堆内存。


## 总结

通过本文对于unity中的GC有了一定的了解，对于GC对于游戏性能的影响以及如何解决都有一定的了解。通过定位造成GC问题的代码以及代码重构我们可以更有效的管理游戏的内存。

接着我会继续写一些Unity相关的文章。翻译的工作，在后面有机会继续进行。

分类: [Unity3D个人学习笔记](#)



 [zblade](#)  
[关注 - 1](#)  
[粉丝 - 73](#)

[+加关注](#)

2

0

« 上一篇: [Unity渲染优化中文翻译 \(三\) ——GPU的优化策略](#)

» 下一篇: [Unity Shader 知识点总结 \(一\)](#)