

# 基础

## 1. C#函数中 new 一个struct 对象 会不会产生垃圾回收?

```
1      public struct CoordsStruct    {
2          public CoordsStruct(double x, double y)
3          { X = x;
4            Y = y;          }
5          public double X { get; set; }
6          public double Y { get; set; }
7          public override string ToString() => $"({X}, {Y})";
8      }
9      public class CoordsClass      {
10         public CoordsClass(double x, double y)
11         {   X = x;
12           Y = y;          }
13         public double X { get; set; }
14         public double Y { get; set; }
15         public override string ToString() => $"({X}, {Y})";
16     }
17     CoordsStruct coordStruct = new CoordsStruct(1, 1);
18     CoordsClass coordsClass = new CoordsClass(1, 1);
```

```
.method private hidebysig static void Main(string[] args) cil managed
{
    .entrypoint
    .custom instance void System.Runtime.CompilerServices.NullableContextAttribute::.ctor(uint8) = ( 01 00 01 00 00 )
    // 代码大小      27 (0x1b)
    .maxstack 3
    .locals init (valuetype Program/CoordsStruct U_0)
    IL_0000: nop
    IL_0001: ldloc.s    U_0
    IL_0003: ldc.r8      1.
    IL_000c: ldc.r8      1.
    IL_0015: call        instance void Program/CoordsStruct::.ctor(float64,
                                                             float64)
    IL_001a: ret
} // end of method Program::Main
```

```
.method private hidebysig static void Main(string[] args) cil managed
{
    .entrypoint
    .custom instance void System.Runtime.CompilerServices.NullableContextAttribute::.ctor(uint8) = ( 01 00 01 00 00 )
    // 代码大小      26 (0x1a)
    .maxstack 2
    .locals init (class Program/CoordsClass U_0)
    IL_0000: nop
    IL_0001: ldc.r8      1.
    IL_000a: ldc.r8      1.
    IL_0013: newobj      instance void Program/CoordsClass::.ctor(float64,
                                                             float64)
    IL_0018: stloc.0
    IL_0019: ret
} // end of method Program::Main
```

1. `new struct` 调用 `ldlocal.s` 向栈压入本地变量地址，之后调用构造函数
2. `new class` 调用 `newobj` 创建一个值类型的新对象，并将对象引用推送到计算堆栈上，对象值放在堆上面
3. 所以new 一个struct对象不产生垃圾回收

```
1  CoordsStruct coordStruct = new CoordsStruct(1, 1);
2  CoordsClass coordsClass = new CoordsClass(1, 1);
```

```

3
4 var StructA = coordStruct;
5 var ClassB = coordsClass;
6
7 StructA.X = 1000;
8 StructA.Y = 1000;
9 ClassB.X = 2000;
10 ClassB.Y = 2000;
11 // 输出
12 //coordStruct      (1, 1)
13 //StructA          (1000, 1000)
14 //coordsClass      (2000, 2000)
15 //ClassB           (2000, 2000)

```

由此可看出 new 出来的 struct 对象是值类型，赋值操作会有拷贝（传参也会有拷贝），new class 对象是引用。

参考博客：[CLR、内存分配和垃圾回收](#)

## 2. C#在函数中 new 一个数组 会不会产生垃圾回收？

```

1 int[] intArray = new int[]{1,2,3};
2 char[] charArray = new char[]{'1','2','3'};
3 string[] stringArray = new string[]{"12","34","56"};

```

```

.method private hidebysig static void Main(string[] args) cil managed
{
    .entrypoint
    .custom instance void System.Runtime.CompilerServices.NullableContextAttribute::ctor(uint8) = ( 01 0
    // 代码大小      252 (0xfc)
    .maxstack 5
    .locals init (int32[] V_0,
        char[] V_1,
        string[] V_2,
        valuetype Program/CoordsStruct[] V_3,
        class Program/CoordsClass[] V_4)

    IL_0000: nop
    IL_0001: ldc.i4.3
    IL_0002: newarr [System.Runtime]System.Int32
    IL_0007: dup
    IL_0008: ldtoken field valuetype '<PrivateImplementationDetails>'/ '__StaticArrayInitTypeSize=12'
    IL_000d: call void [System.Runtime]System.Runtime.CompilerServices.RuntimeHelpers::InitializeArray
        (byref T[], RuntimeFieldHandle* byref, IntPtr, RuntimeFieldHandle*)

    IL_0012: stloc.0
    IL_0013: ldc.i4.3
    IL_0014: newarr [System.Runtime]System.Char
    IL_0019: dup
    IL_001a: ldtoken field valuetype '<PrivateImplementationDetails>'/ '__StaticArrayInitTypeSize=6'
    IL_001f: call void [System.Runtime]System.Runtime.CompilerServices.RuntimeHelpers::InitializeArray
        (byref T[], RuntimeFieldHandle* byref, IntPtr, RuntimeFieldHandle*)

```

```

IL_0024: stloc.1
IL_0025: ldc.i4.3
IL_0026: newarr      [System.Runtime]System.String
IL_002b: dup
IL_002c: ldc.i4.0
IL_002d: ldstr      "12"
IL_0032: stelem.ref
IL_0033: dup
IL_0034: ldc.i4.1
IL_0035: ldstr      "34"
IL_003a: stelem.ref
IL_003b: dup
IL_003c: ldc.i4.2
IL_003d: ldstr      "56"
IL_0042: stelem.ref
IL_0043: stloc.2
IL_0044: ldc.i4.3

```

无论是int char 还是string数组，都调用了 `newarr` 命令：为值申请内存分配在托管堆上，并且添加引用在栈上，会产生垃圾回收

### 3. C#在函数中 new 一个struct对象数组 会不会产生垃圾回收？

```

1      CoordsStruct[] coordStruct = new CoordsStruct[]
2      {
3          new CoordsStruct(1, 1),
4          new CoordsStruct(2, 2),
5          new CoordsStruct(3, 3),
6      };
7      CoordsClass[] coordsClass = new CoordsClass[]
8      {
9          new CoordsClass(1, 1),
10         new CoordsClass(2, 2),
11         new CoordsClass(3, 3),
12     };

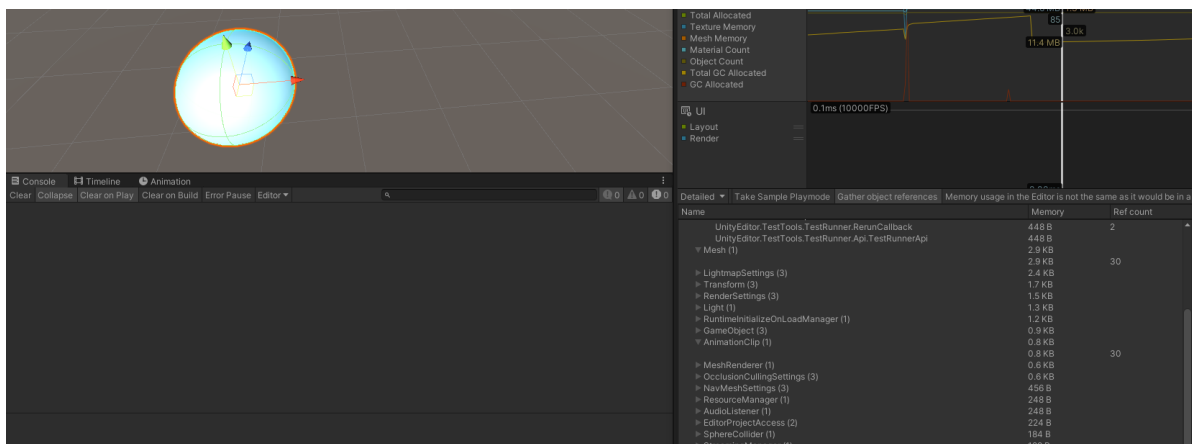
```

|                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                        |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                            |
|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <pre> IL_00a6: newarr      Program/CoordsClass IL_00ab: dup IL_00ac: ldc.i4.0 IL_00ad: ldc.r8      1. IL_00b6: ldc.r8      1. IL_00bf: newobj      instance void Program/CoordsClass::.ctor(float64, float64)  IL_00c4: stelem.ref IL_00c5: dup IL_00c6: ldc.i4.1 IL_00c7: ldc.r8      2. IL_00d0: ldc.r8      2. IL_00d9: newobj      instance void Program/CoordsClass::.ctor(float64, float64)  IL_00de: stelem.ref IL_00df: dup IL_00e0: ldc.i4.2 IL_00e1: ldc.r8      3. IL_00ea: ldc.r8      3. IL_00f3: newobj      instance void Program/CoordsClass::.ctor(float64, float64)  IL_00f8: stelem.ref IL_00f9: stloc.s     0_4 IL_00fb: ret // end of method Program::Main </pre> | <pre> IL_0045: newarr      Program/CoordsStruct IL_004a: dup IL_004b: ldc.i4.0 IL_004c: ldc.r8      1. IL_0055: ldc.r8      1. IL_005e: newobj      instance void Program/CoordsStruct::.ctor(float64, float64)  IL_0063: stelen      Program/CoordsStruct IL_0068: dup IL_0069: ldc.i4.1 IL_006a: ldc.r8      2. IL_0073: ldc.r8      2. IL_007c: newobj      instance void Program/CoordsStruct::.ctor(float64, float64)  IL_0081: stelen      Program/CoordsStruct IL_0086: dup IL_0087: ldc.i4.2 IL_0088: ldc.r8      3. IL_0091: ldc.r8      3. IL_009a: newobj      instance void Program/CoordsStruct::.ctor(float64, float64)  IL_009f: stelen      Program/CoordsStruct IL_00a4: stloc.3 IL_00a5: ldc.i4.3 </pre> |
|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|

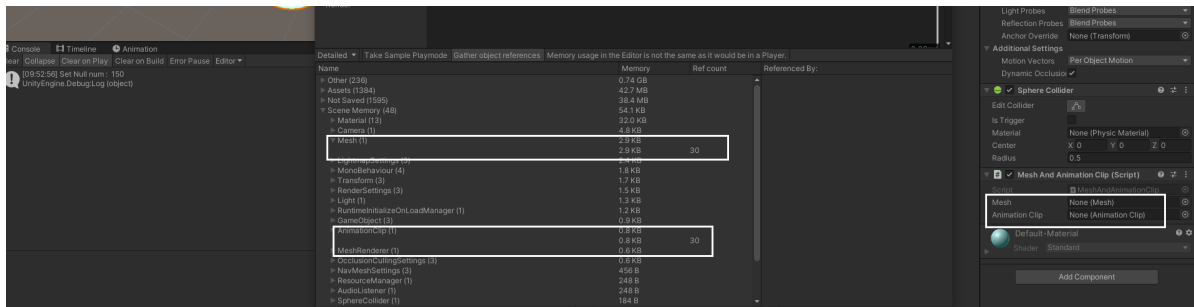
无论是new struct 还是class数组，都调用了 `newarr` 和 `newobj` 命令，会产生垃圾回收。

## 4. unity的部分对象如：Mesh, AnimationClip 等对象 new 出来之后需不需要主动释放？

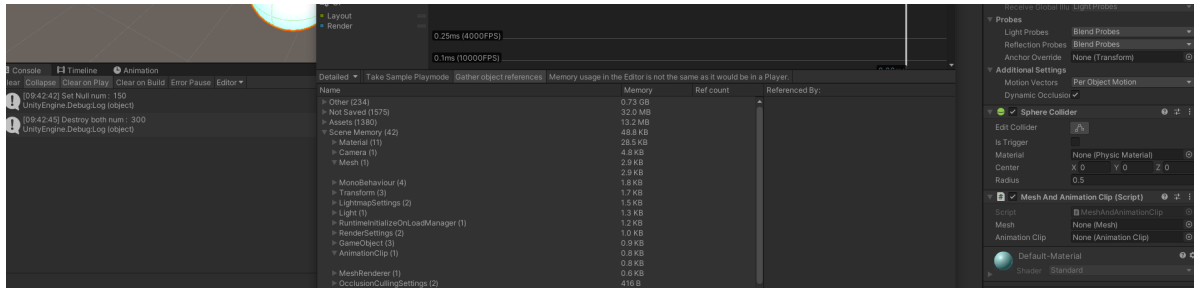
```
1 public Mesh mesh;
2 public AnimationClip animationClip;
3
4 private int num;
5 // Start is called before the first frame update
6 void Start()
7 {
8     mesh = new Mesh();
9     animationClip = new AnimationClip();
10    num = 0;
11 }
12
13 // Update is called once per frame
14 void FixedUpdate()
15 {
16     num = num + 1;
17     if (num == 50 * 5)
18     {
19         mesh = null;
20         animationClip = null;
21         Debug.Log("Set Null num : " + num);
22     }
23     if (num == 50 * 10)
24     {
25         Destroy(mesh);
26         Destroy(animationClip);
27         Debug.Log("Destroy both num : " + num);
28     }
29     if (num == 50 * 15)
30     {
31         GC.Collect();
32         Debug.Log("GC.Collect num : " + num);
33     }
```



将mesh 和 Animation Clip置空引用次数没有归零



调用 Destroy 之后引用次数归零



## 5. 想办法评估下 Lua和C# 在循环里持续产生内存垃圾对性能的影响

C#占用:

```
1 public class CShapeGC : MonoBehaviour
2 {
3     public class obj
4     {
5         public int num;
6         public obj(int num) { this.num = num; }
7     }
8     void FixedUpdate()
9     {
10         // 一秒50次，一秒生成150个对象
11         obj a = new obj(1);
12         obj b = new obj(2);
13         obj c = new obj(3);
14     }
15 }
```

lua占用:

```
1 LuaGC.cs
2 public class LuaGC : MonoBehaviour
3 {
4     LuaState lua = null;
5     [SerializeField] string luaFile = "luaObject";
6     LuaFunction func;
7     void Start()
8     {
9         lua = new LuaState();
10        lua.Start();
11        string filePath = Application.dataPath + "/Test";
12        lua.AddSearchPath(filePath);
13        lua.Require(luaFile);
```

```

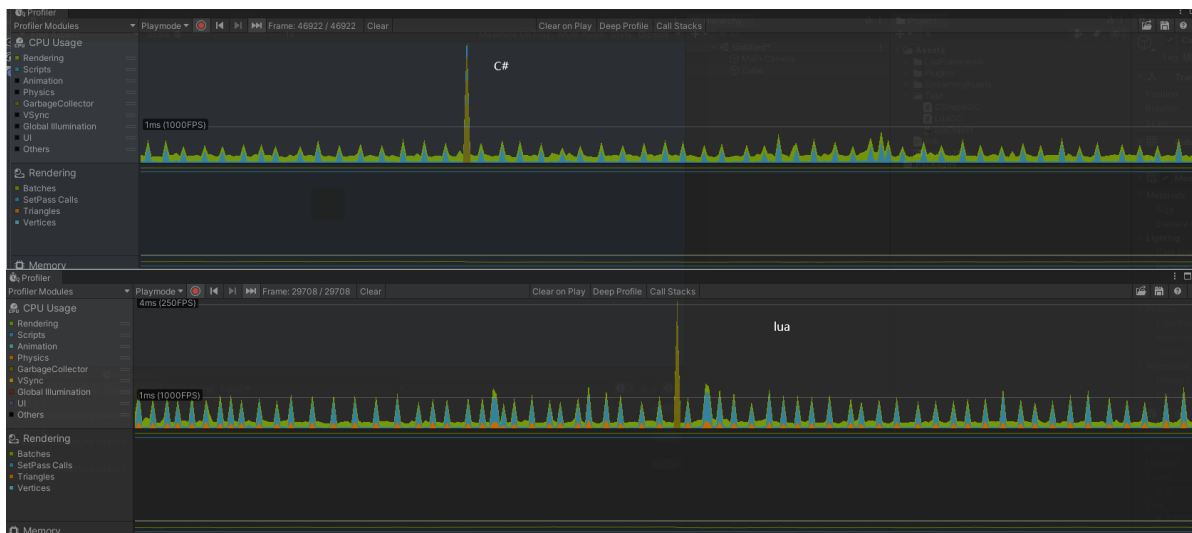
14         func = lua.GetFunction("initLuaObject");
15
16     }
17     void FixedUpdate()
18     {
19         // 每秒生成150个对象
20         func.Call();
21         func.Call();
22         func.Call();
23     }
24 }

```

```

1  -- luaObject.lua
2  LuaObject = { num = 0 }
3  local mt =
4  {
5      __index = Lua_SpawnObj,
6      __call = function(self, num) self.num = num end
7  }
8  function LuaObject:new(num)
9      self = self or {}
10     setmetatable(self, mt)
11     self(num)
12     return self
13 end
14 function initLuaObject()
15     local tab = LuaObject:new(123)
16     print(tab.num)
17 end

```



从表中可以看出，lua的平均消耗在1ms上下，GC到达4ms。C#的平均消耗在1ms一半位置，GC也没有打到4ms。

## 6. 加载unity资源：AnimatorController，材质，贴图，动作文件 需不需要释放，需要的释放使用什么接口？

[Resources和AssetBundle最详细的解析](#)

### 1. Resources

1. 一般用 `Resources.Load` 加载 `Asset\Resources` 目录下的特定资源，一般还会实例化到场景中。

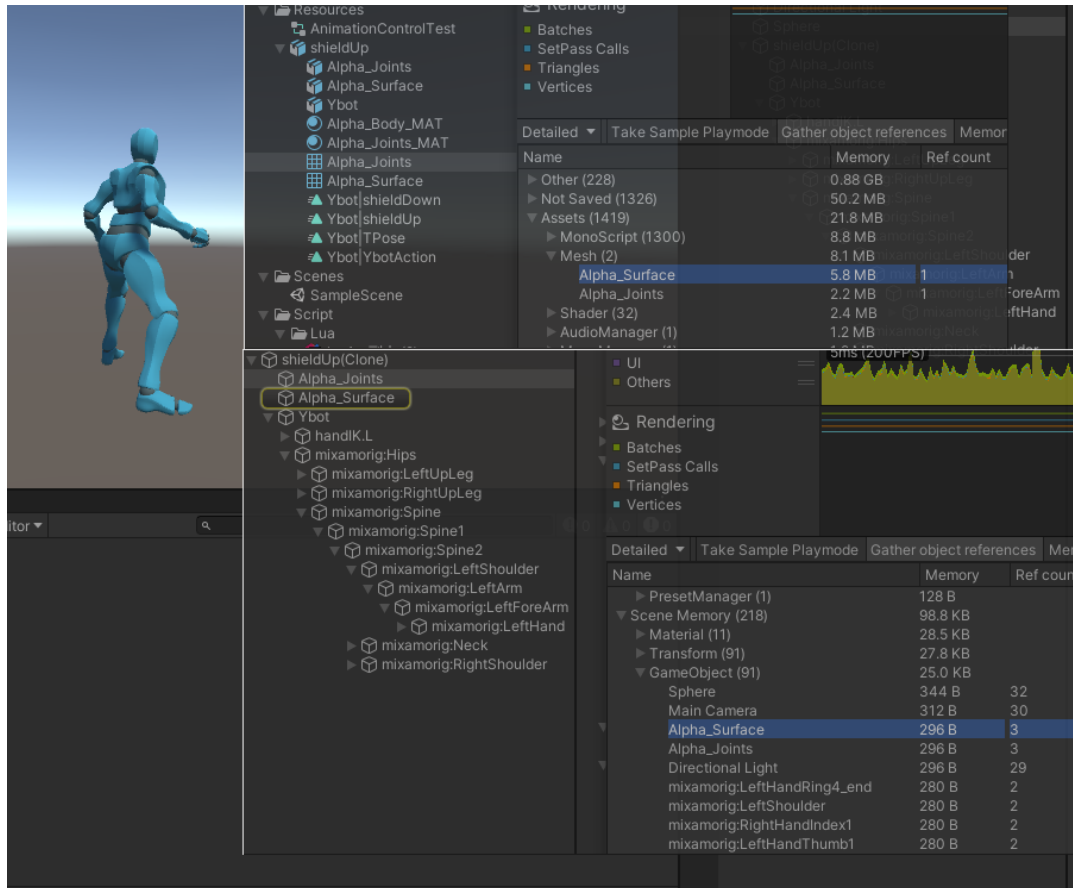
1. 资源被加载到Asset中，还会产生一个Clone到Scene Memory中

2. 关于Resources的方法:

1. `FindObjectsOfTypeAll`: 返回某一种类型的所有资源
2. `Load`: 通过路径加载资源
3. `LoadAll`: 加载该Resources下的所有资源
4. `LoadAsync`: 异步加载资源，通过协程实现
5. `UnloadAsset`: 卸载加载的资源
6. `UnloadUnusedAssets`: 卸载在内存中未使用的资源（整个游戏对象层级视图后未访问到某资源（包括脚本组件），则将其视为未使用的资源）

```
3. 1 public class LoadResourcesTwoWays : MonoBehaviour
    2 {
    3     private GameObject modelInstantiate;
    4     // Start is called before the first frame update
    5     void Start()
    6     {
    7         var obj = Resources.Load("shieldup");
    8
    9         modelInstantiate = Instantiate(obj) as GameObject;
   10         modelInstantiate.transform.position = Vector3.zero;
   11
   12         obj = null;
   13         Resources.UnloadUnusedAssets();
   14         // LoadResourcesToScene();
   15         // LoadABResource();
   16     }
   17     // Update is called once per frame
   18     void Update()
   19     {
   20         if (Input.GetKey(KeyCode.A))
   21         {
   22             Destroy(modelInstantiate);
   23         }
   24     }
   25 }
```

没有执行 Destory 的内存，Scene Memory内存占用



调用 Destory 之后Assets内存里还有，Scene Memory内存从218变成42

| Name                                 | Memory  | Ref count |
|--------------------------------------|---------|-----------|
| AnimatorStateMachine (1)             |         |           |
| ▶ Other (233)                        | 0.88 GB |           |
| ▶ Not Saved (1340)                   | 54.5 MB |           |
| ▼ Assets (1428)                      | 52.0 MB |           |
| ▶ Shader (36)                        | 31.8 MB |           |
| ▼ Mesh (2)                           | 8.8 MB  |           |
| ▶ Alpha_Surface (1)                  | 5.8 MB  |           |
| ▶ Alpha_Joints (1)                   | 3.0 MB  | 26        |
| ▶ MonoScript (1300)                  | 8.8 MB  |           |
| ▶ AudioManager (1)                   | 1.2 MB  |           |
| ▶ MonoManager (1)                    | 1.2 MB  |           |
| ▶ MonoImporter (51)                  | 48.6 KB |           |
| ▶ Material (5)                       | 21.7 KB |           |
| ▶ PlayerSettings (1)                 | 13.9 KB |           |
| ▶ FBXImporter (1)                    | 10.7 KB |           |
| Scene Memory (42)                    | 49.8 KB |           |
| ▶ Material (13)                      | 32.0 KB |           |
| ▶ Camera (1)                         | 4.8 KB  |           |
| ▶ MonoBehaviour (4)                  | 1.8 KB  |           |
| ▶ Transform (3)                      | 1.7 KB  |           |
| ▶ LightmapSettings (2)               | 1.5 KB  |           |
| ▶ ResourceManager (1)                | 1.4 KB  |           |
| ▶ Light (1)                          | 1.3 KB  |           |
| ▶ RuntimeInitializeOnLoadManager (1) | 1.2 KB  |           |
| ▶ RenderSettings (2)                 | 1.0 KB  |           |
| ▼ GameObject (3)                     | 0.9 KB  |           |
| ▶ Sphere                             | 344 B   | 32        |
| ▶ Main Camera                        | 312 B   | 30        |
| ▶ Directional Light                  | 296 B   | 29        |
| ▶ MeshRenderer (1)                   | 0.6 KB  |           |
| ▶ OcclusionCullingSettings (2)       | 416 B   |           |

4. 可以使用 DestroyImmediate(object, true) 立即摧毁Asset中的内存

1. 但是这只会在「只有在Asset，没有在SceneMemory占用」的情况下生效



2. 如果同时存在两者的占用，那 `DestroyImmediate` 只能释放在 `SceneMemory` 中的内存，
3. 且即使释放了 `Scene Memory` 的内存后，**再次使用** `DestroyImmediate` 也不能释放在 `Asset` 裡面的内存占用

5. `Resources.UnloadAsset`：卸载 `Asset` 加载的资源

1. 

```
1 Resources.UnloadAsset(modelInstantiate);
2 //该方法无法释放SceneMemory中的内存（场景中的Clone），只能释放Asset裡面的内存
```

6. `Resources.UnloadUnusedAssets`：卸载在内存中未使用的资源（整个游戏对象层级视图后未访问到某资源（包括脚本组件），则将其视为未使用的资源）

1. 

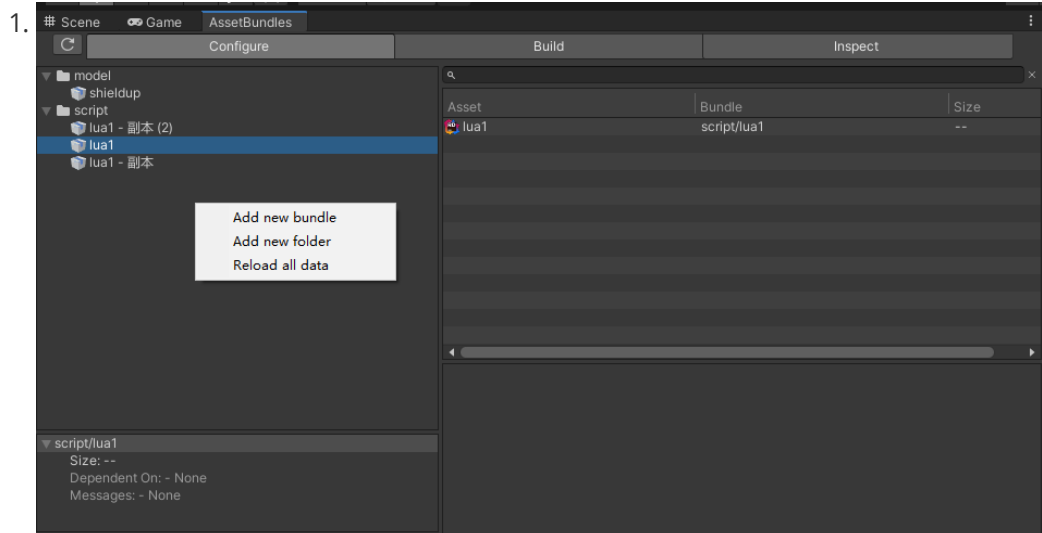
```
1 model = null;
2 AnimationControlTest = null;
3 Resources.UnloadUnusedAssets();
4 // 释放所有Resource加载的所有Asset内存
5 // 以把Asset和SceneMemory裡的内存一并释放
```

7. 总结：**Resources加载的资源是需要释放**，即使调用了 `Destory(obj)`，也要记得 `DestroyImmediate(obj, true)` 或者 `UnloadAsset(obj)` 或者 `UnloadUnusedAssets()` 释放 `Assets` 里面的内存

1. 对于不会重複使用的 `asset`可以加载完之后马上调用 `UnloadAsset`
2. 在场景 `scene` 关闭前，调用 `DestroyImmediate(obj, true)` 清除所有的 **`SceneMemory`** 裡的占用
3. 最后再使用 `UnloadUnusedAssets` 确认释放所有内存

## 2. AssetBundle

1. 先打包AB包

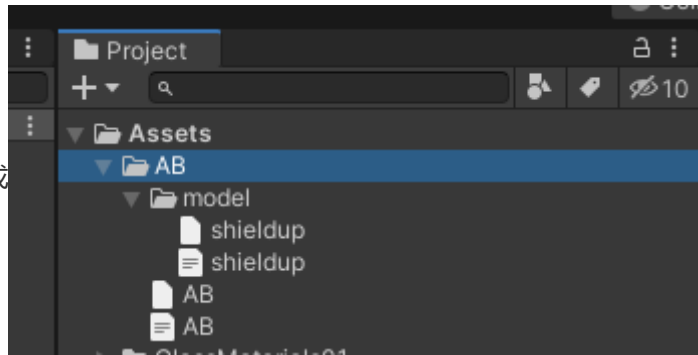


2.



1. No Compression 不压缩，明显会增大AB包的体积，但是在用的是否加载速度会快很多
2. LZMA，即所有资源一次性压缩，AB包的体积最小，但是对于资源调用时的速度会慢很多，因为调用任何一个资源都需要全局解压
3. LZ4，局部压缩，就是对于每一个资源单独压缩，用的时候就是用到哪一个，就解压哪一个

3. 打包完成



2. 卸载AB包加载的资源内存占用有两种：

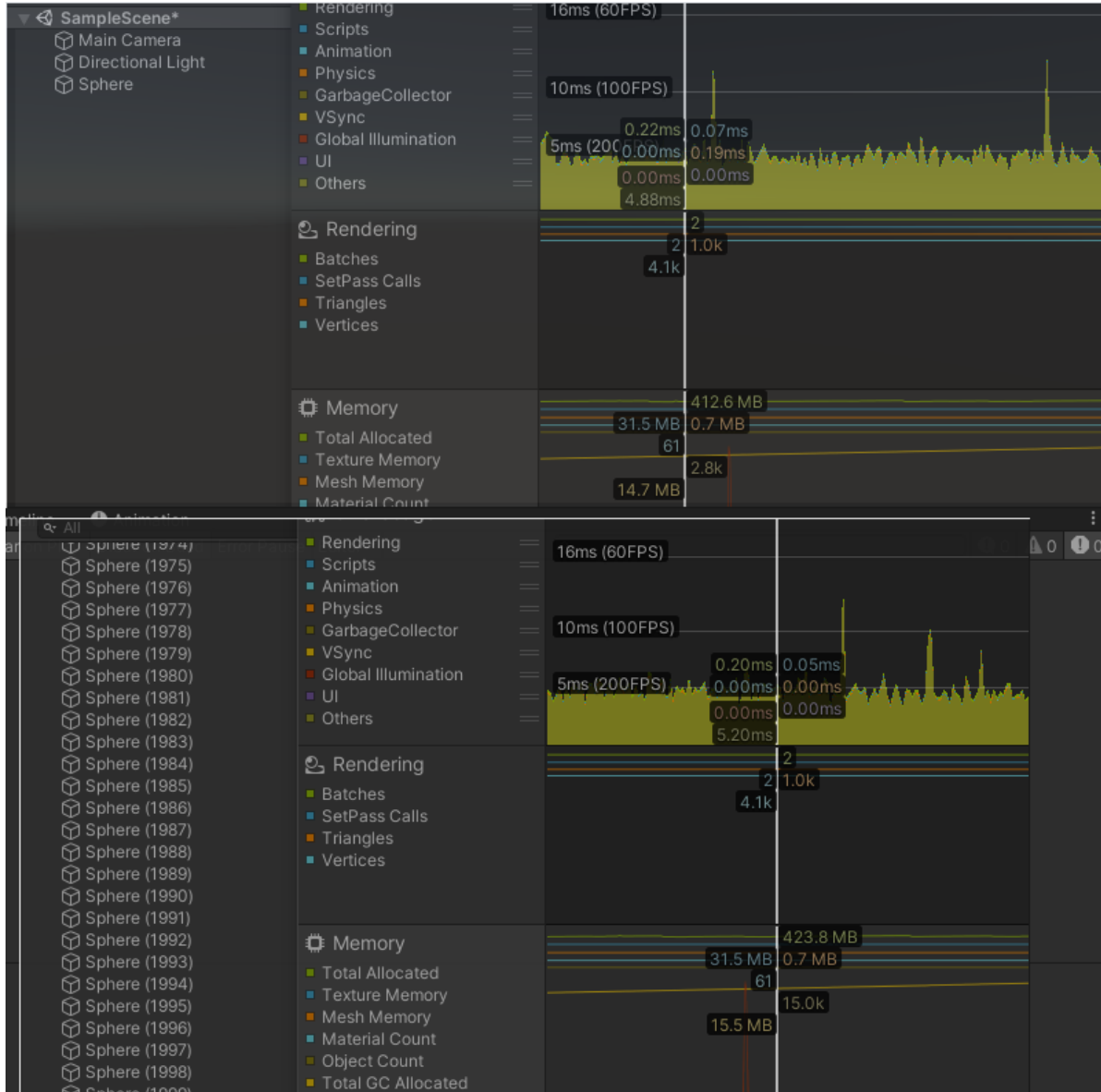
1. `(instance)ab.Unload(bool unloadAllLoadedObject)` 只释放某个AB自身的内存占有
  1. When `unloadAllLoadedObjects` is **false**, compressed file data inside the bundle itself will be freed, but any instances of objects loaded from this bundle will remain intact
  2. (bundle中的压缩文件被释放，实例化成功的保持原样)
  3. When `unloadAllLoadedObjects` is **true**, all objects that were loaded from this bundle will be destroyed as well. If there are GameObjects in your Scene referencing those assets, the references to them **will become missing**.
  4. (所有对象都要被销毁，scene中的asset引用也会被释放)
2. `(static)AssetBundle.UnloadAllAssetBundles(bool unloadAllLoadedObject)` 释放所有AB的占有内存
3. 二者 都需要配合 `Destroy` 销毁场景中的实例Instance，使用 `unload` 销毁Assets中的内存

## 7. MonoBehaviour 数量过多 对性能影响如何，或者一个 MonoBehaviour 自带的消耗有哪些？

1. 每一个 `Monobehaviour` 都是通过反射来调用 生命周期函数的 (`Awake`, `Update`, `LateUpdate` 等)

1. MonoBehaviour会在游戏开始时首先得到所有写有生命周期函数的脚本，保存下来后再调用这些生命周期方法。
2. 只要在 MonoBehaviour 中声明了 update 这样的生命周期函数，无论函数体里是否有东西，该方法都会被执行，从而对CPU造成一定的负荷
  - 场景上带有 MonoBehaviour 且带有各种生命周期（尤其是 update 函数）的脚本越多，CPU 负荷越大，即使所有生命周期的方法体都没有内容

代码中只写一句 `public class MonoBehaviourTest : MonoBehaviour { }` 的情况对比



```

1 public class MonoBehaviourTest : MonoBehaviour
2 {
3     private void Awake(){ }
4     private void OnEnable(){ }
5     void Start(){ }
6     private void FixedUpdate(){ }
7     private void OnTriggerEnter(Collider other){ }
8     private void OnTriggerStay(Collider other){ }
9     private void OnTriggerExit(Collider other){ }
10    private void OnCollisionEnter(Collision collision){ }
11    private void OnCollisionStay(Collision collision){ }
12    private void OnCollisionExit(Collision collision){ }
13    private void OnMouseDown(){ }
14    private void OnMouseUp(){ }
15    void update(){ }

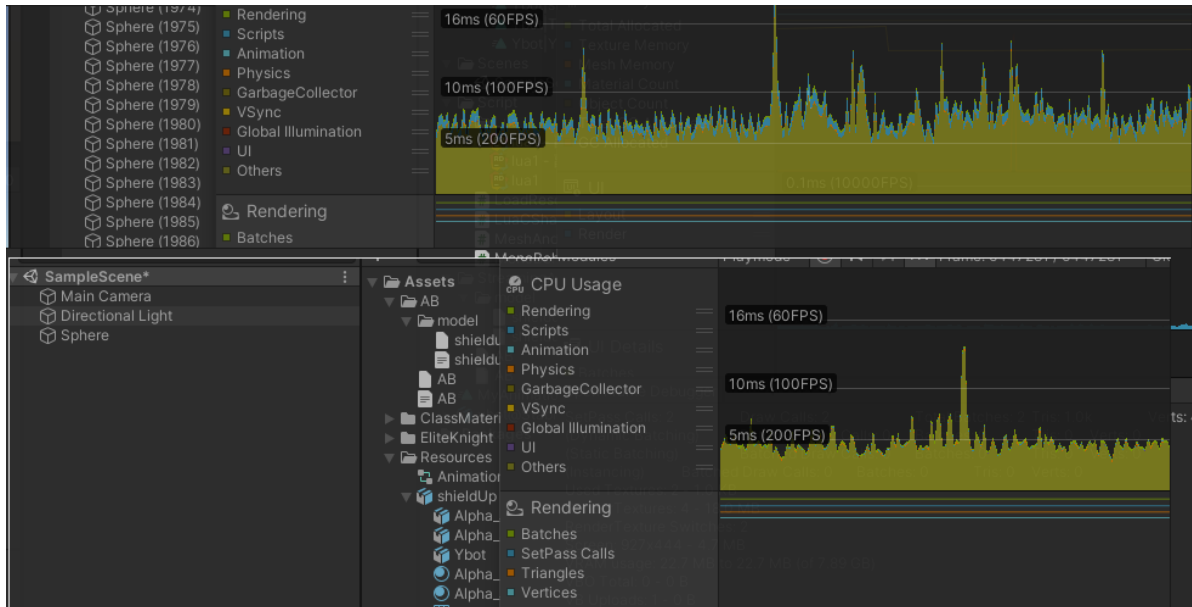
```

```

16     private void LateUpdate(){ }
17     private void OnRenderImage(RenderTexture source, RenderTexture
destination){ }
18     private void OnDisable(){ }
19     private void OnDestroy(){ }
20     private void OnApplicationQuit(){ }
21 }

```

测试代码声明内置生命周期函数后，性能消耗有明显上升



## 8. Lua的底层数据类型有哪些？

```

1  /*
2  ** basic types
3  */
4  #define LUA_TNONE      (-1)
5  #define LUA_TNIL        0
6  #define LUA_TBOOLEAN    1
7  #define LUA_TLIGHTUSERDATA  2
8  #define LUA_TNUMBER     3
9  #define LUA_TSTRING     4
10 #define LUA_TTABLE      5
11 #define LUA_TFUNCTION   6
12 #define LUA_TUSERDATA   7
13 #define LUA_TTHREAD     8
14 #define LUA_NUMTAGS     9

```

| 常用数据类型   | 描述                                                                                                                  |
|----------|---------------------------------------------------------------------------------------------------------------------|
| nil      | 空值                                                                                                                  |
| boolean  | 布尔值：false和true。                                                                                                     |
| number   | 双精度浮点数                                                                                                              |
| string   | 字符串（可用双引号或单引号）                                                                                                      |
| function | 函数类型：由 C 或 Lua 编写的函数                                                                                                |
| userdata | 主要用来表示在C/C++中定义的类型，即用来实现扩展lua，这些扩展代码通常是用C/C++来实现的。对lua 虚拟机来说userdata提供了一块原始的内存区域                                    |
| thread   | 用于执行协同程序 <a href="#">协同程序（线程thread）</a>                                                                             |
| table    | Lua 中的表（table）是一个"关联数组"（associative arrays），数组的索引可以是数字、字符串或表类型。在 Lua 里，table 的创建是通过"构造表达式"来完成，最简单构造表达式是{}，用来创建一个空表。 |

| 不常用数据类型       | 描述                                                             |
|---------------|----------------------------------------------------------------|
| none          | 仅用于 C API <a href="#">Is 'none' one of basic types in Lua?</a> |
| lightUserData | 轻量级用户数据是表示 C 指针的值（即 void * 值） <a href="#">Light Userdata</a>   |
| NUMTYPES      | 双精度浮点数                                                         |

string table function thread 四种在 vm 中以引用方式共享，是需要被 GC 管理回收的对象。其它类型都以值形式存在。——[Lua GC 的源码剖析\(1\)](#)

## 9. 为什么说Lua一切皆Table,Table有哪两种存储形式,Table是如何Resize的，了解Resize的代价

1. Lua的table由 **数组部分（array part）** 和 **哈希部分（hash part）** 组成。
  1. 数组部分索引的key是1~n的整数，
  2. 哈希部分是一个哈希表，哈希表本质是一个数组，它利用哈希算法将键转化为数组下标，若下标有冲突，则会将冲突的下标上创建一个链表，用链地址法解决哈希冲突。
  3. table的 key 值可以是除了 nil 之外的任何类型的值
2. 向table中插入数据时，如果table满了，table会重新设置数据部分 和 哈希表的大小，**容量是成倍增加的（C++ vector）**，哈希部分还要对哈希表中的数据进行整理。
  1. 没有赋初始值的table，数组和部分哈希部分默认容量为0。

```
1 //luaC语言源码：ltable.c
2 void luaH_newKey(lua_State *L, Table *t, const TValue *key, TValue *value){
3     mp = mainpositionTV(t, key);
4     if (!isempty(gval(mp)) || isdummy(t)) { /* main position is taken?
5     */
6         Node *othern;
7         Node *f = getfreepos(t); /* get a free place */
8         if (f == NULL) { /* cannot find a free place? */
9             rehash(L, t, key); /* grow table */ // 重置哈希
10            /* whatever called 'newkey' takes care of TM cache */
```

```

10     luaH_set(L, t, key, value); /* insert key into grown table */
11     return;
12 }
13 }
14 static void rehash (lua_State *L, Table *t, const TValue *ek) {
15     unsigned int asize; /* optimal size for array part 数组最佳大小*/
16     unsigned int na; /* number of keys in the array part 数组部分的 key
(index) 的数量*/
17     unsigned int nums[MAXABITS + 1];
18     int i;
19     int totaluse;
20     for (i = 0; i <= MAXABITS; i++) nums[i] = 0; /* reset counts */
21     setlimittoresize(t);
22     na = numusearray(t, nums); /* count keys in array part */
23     totaluse = na; /* all those keys are integer keys */
24     totaluse += numusehash(t, nums, &na); /* count keys in hash part
*/
25     /* count extra key */
26     if (ttisinteger(ek))
27         na += countint(ivalue(ek), nums);
28     totaluse++;
29     /* compute new size for array part */
30     asize = computesizes(nums, &na);
31     /* resize the table to new computed sizes */
32     luaH_resize(L, t, asize, totaluse - na);
33 }
34 ** Compute the optimal size for the array part of table 't'. 'nums' is a
35 ** "count array" where 'nums[i]' is the number of integers in the table
36 ** between  $2^{(i-1)} + 1$  and  $2^i$ . 'pna' enters with the total number of
37 ** integer keys in the table and leaves with the number of keys that
38 ** will go to the array part; return the optimal size. (The condition
39 ** 'twotoi > 0' in the for loop stops the loop if 'twotoi' overflows.)
40 */
41 static unsigned int computesizes (unsigned int nums[], unsigned int *pna) {
42     int i;
43     unsigned int twotoi; /*  $2^i$  (candidate for optimal size) */
44     unsigned int a = 0; /* number of elements smaller than  $2^i$  */
45     unsigned int na = 0; /* number of elements to go to array part */
46     unsigned int optimal = 0; /* optimal size for array part */
47     /* loop while keys can fill more than half of total size */
48     for (i = 0, twotoi = 1;
49         twotoi > 0 && *pna > twotoi / 2;
50         i++, twotoi *= 2) {
51         a += nums[i];
52         if (a > twotoi/2) { /* more than half elements present? */
53             optimal = twotoi; /* optimal size (till now) */
54             na = a; /* all elements up to 'optimal' will go to array part */
55         }
56     }
57     lua_assert((optimal == 0 || optimal / 2 < na) && na <= optimal);
58     *pna = na;
59     return optimal;
60 }

```

1. 从c语言源码中可以看出，获得的新table长度在  $2^{(i-1)} + 1$  and  $2^i$  之间，
2. Lua Table在非构造阶段，不论是Array还是Hash部分都是以2的幂次增加的（事实上在构造阶段Hash部分也只能按2的幂次增加）。每当扩容以后，原数据会重新再插入新的内存块中。

3. 由 C++中vector满时，申请新内存地址造成的重大开销，建议初始化阶段赋值 或者 开辟预计大小空间初始化。
4. `resize` 代价很高，当我们把一个新键值赋给表时，若数组和哈希表已经满了，则会lua在申请内存基础上还需要 重置哈希(`rehash`)。
5. 重置哈希的代价是高昂的。首先会在内存中分配一个新的长度的数组，然后将所有记录再全部哈希一遍，将原来的记录转移到新数组中。

```
1  -- 测试内存占用
2  collectgarbage("stop")
3  local mem = collectgarbage("count")
4  tab = {}
5  LogInfoFormat("emery table is: %d \t memory usage: %s ", 0,
6  (collectgarbage("count") - mem) * 1024)
7  for i = 1,10 do
8      tab[i] = i
9      LogInfoFormat("table length : %d \t memory usage: %s ", #tab,
10 (collectgarbage("count") - mem) * 1024)
11 end
```

```
emery table is: 0      memory usage: 64
table length : 1      memory usage: 88
table length : 2      memory usage: 88
table length : 3      memory usage: 165
table length : 4      memory usage: 255
table length : 5      memory usage: 349
table length : 6      memory usage: 439
table length : 7      memory usage: 501
table length : 8      memory usage: 591
table length : 9      memory usage: 745
table length : 10     memory usage: 835
```

## 10. table 遍历有几种形式 有什么不同

### 1. 第一种：pairs迭代器

- 对所有 键值key遍历 通过 `next()` 函数判断下一个元素
- 但是会出现随机遍历，可能不会按顺序遍历
- 也可以用 `tabName.next()` 测试是不是空表

```
1  tab = {a = "qwe", b = "asd", 3, 2, 1, c = "zxc"}
2  for k, v in pairs(tab) do
3      print(k, v)
4  end
5  tab.d = "qweasd"
```

```

6  print("添加新索引")
7  for k, v in pairs(tab) do
8      print(k, v)
9  end
10
11  table.remove(tab, "b")
12  print("remove索引")
13  for k, v in pairs(tab) do
14      print(k, v)
15  end
16  输出
17  a   qwe
18  b   asd
19  c   zxc
20  3   1
21  2   2
22  1   3
23  添加新索引
24  a   qwe
25  b   asd
26  c   zxc
27  3   1
28  2   2
29  1   3
30  d   qweasd
31  remove索引
32  a   qwe
33  c   zxc
34  3   1
35  2   2
36  1   3
37  d   qweasd

```

## 2. 第二种: ipairs迭代器

- 遍历数组, 顺序遍历, 如果中间key有空, 则不会遍历后面的

```

1  tab = {}
2  for i= 4,1,-1 do
3      tab[i] = i
4  end
5  tab[6] = "asdqwe"
6  for k, v in ipairs(tab) do
7      print(k, v)
8  end
9  输出
10  1   1
11  2   2
12  3   3
13  4   4

```

# 11. 详细描述下项目使用的 class 机制

- 声明全局变量 `_G["__class"]` 让C#调用



```

1  if _G["__class"] == nil then
2      _G["__class"] = {}  --__class作为index, 赋值为一个空表
3  end
4  local _class = _G["__class"] or {}

```

```

LuaDLL.lua_getglobal(L, "G");          LuaHookSetup.cs 337
LuaDLL.lua_pushstring(L, "G");         LuaHookSetup.cs 339
LuaDLL.lua_getglobal(L, "G");          LuaHookSetup.cs 404
LuaDLL.lua_pushstring(L, "G");         LuaHookSetup.cs 406

```

```

1  local rawset = rawset
2  local setmetatable = setmetatable
3
4
5  -- 输出如果index值是固定结构报错
6  local function __disable_newindex(t, key, value)
7      LogErrorFormat('properties is fix struct, forbid new index (new index:
8  %s value: %s)', key, value)
9  end
10 -- 处理新的index
11 local function __properties_newindex(t, key, value)
12     local properties = t.Properties
13     if properties[key] == nil then
14         rawset(t, key, value)
15     else
16         properties[key] = value
17     end
18 end
19 -- 构造函数, 如果有父类, 则可以再次调用 本构造函数
20 local function __ctor(obj, class_type, ...)
21     if class_type.super then
22         __ctor(obj, class_type.super, ...)
23     end
24     if class_type.ctor then
25         class_type.ctor(obj, ...)
26     end
27 end
28 -- 通过父类来初始化, 实现继承
29 local function __init_super(obj, self_class_type, ...)
30     local typeSuper = self_class_type.super
31     while typeSuper ~= nil do
32         local typeSuperL = typeSuper
33         local objSuper = {}
34         obj[typeSuper] = objSuper  --将父类设置到这个该对象的
35         self_class_type.super 索引里
36         setmetatable(objSuper, {__index=  -- 设置元表, 该被访问时, 会调用下列的
37         函数
38             function(t, k)
39                 local ret=_class[typeSuperL][k] --得到父类的 继承函数
40                 if type(ret) == "function" then
41                     local func = ret
42                     ret = function(self1, ...)
43                         return func(obj, ...)
44                     end
45                     t[k] = ret  -- 如果是函数类型, 保存下父类继承函数, 没有就nil

```

```

43         else
44             ret = nil
45         end
46         return ret
47     end
48 })
49     typeSuper = typeSuper.super
50 end
51 end
52
53 function class(super, enable_properties)
54     ---@class BaseClass
55     local class_type={} -- class_type 类模板
56     class_type.ctor=false
57     class_type.super=super -- 父类赋值
58     class_type.new=function(self_class_type, ...) -- 定义new成员方法
59         local obj= {}
60         setmetatable(obj, _class[class_type].__Metatable)
61         -- 多重继承调用指定父类方法设定
62         __init_super(obj, self_class_type, ...) -- 调用上面的函数 初始
        化父类
63         __ctor(obj, class_type, ...) -- 调用上面的 构造函数
64         return obj
65     end
66
67     if enable_properties then
68         class_type.new_with_properties = function(self_class_type,
        properties, ...)
69             local obj= {Properties = properties}
70             local mt = {__index = properties, __newindex =
71             __properties_newindex }
72             --元表设置索引的时候调用properties, 赋值操作table[key] = value调
73             用 __properties_newindex
74             --设置元表
75             setmetatable(obj, mt)
76             setmetatable(properties,
77             _class[class_type].__PropertiesMetatable)
78
79             -- 多重继承调用指定父类方法设定
80             __init_super(obj, self_class_type, ...)
81             __ctor(obj, class_type, ...)
82             return obj
83         end
84     end
85
86     -- vtbl可以理解为类容器
87     local vtbl=
88     {
89         __is_class = true,
90         is_class_type = function(type)
91             -- 调用函数类型 和 成员表类型 是否相同
92             if class_type == type then
93                 return true
94             end
95             local typeSuper = class_type.super
96             while typeSuper ~= nil do
97                 if typeSuper == type then
98                     return true
99                 end
100                 typeSuper = typeSuper.super
101             end
102             return false
103         end
104     }
105 end

```

```

96         end
97         typeSuper = typeSuper.super
98     end
99     return false
100 end,
101 -- 声明成员变量
102     DeclareVar = function(obj, name, value)
103         if obj[name] ~= nil then
104             error(string.format("成员变量:%s 已存在, 声明失败", name))
105         return
106         end
107         rawset(obj, name, value)
108     end
109 }
110
111 vtbl.__Metatable = {__index = vtbl}
112 if enable_properties then
113     vtbl.__PropertiesMetatable = { __index = vtbl, __newindex =
__disable_newindex }
114 end
115
116 _class[class_type]= vtbl
117 -- 设置 class_type 赋值的时候调用该函数
118 setmetatable(class_type,{__newindex=
119     function(t,k,v)
120         vtbl[k]=v
121     end
122 })
123
124 if super then
125     --如果有父类 先从_class中找到父类 父类给vtbl的super赋值
126     vtbl.__super = super
127     -- 如果没有这个成员, 就去父类里面找
128     setmetatable(vtbl,{__index=
129         function(t,k)
130             local ret=_class[super][k]
131             vtbl[k]=ret
132             return ret
133         end
134     })
135 end
136 -- 返回类模板, 可以添加ctor函数, 可以添加成员函数, 这些都会被加到vtbl中,
137 return class_type
138 end

```

## 12. 大体了解下Lua的GC机制

lua 垃圾收集器Garbage Collection [官网GC相关ppt](#)

[lua GC](#)

### 1. 哪些对象会被回收

#### 1. 所有的lua对象都要被回收 (All objects in Lua are subject to garbage collection)

1. tables, functions, "modules", threads

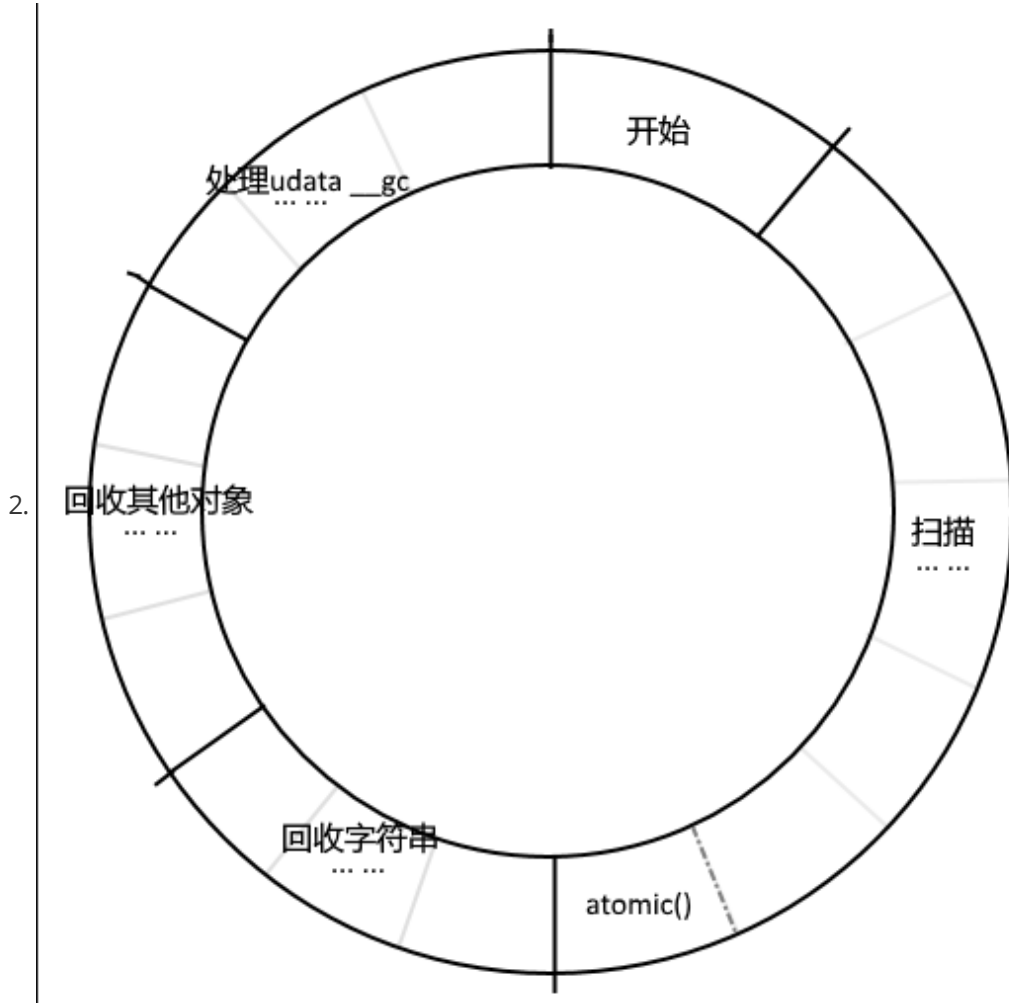
#### 2. 只保留root set 中 可访问的对象

1. root set: the registry注册表 and shared metatable 共享元表

2. 注册表registry 包含 global table (\_G), 主线程the main thread 和 package.loaded
2. 基础的收集器 是 标记 mark 和 扫描 sweep

### 1. GC的流程

1. 开始->扫描标记 -> 字符串回收 -> 其他GC对象回收 -> 终止



3. 开始和atomic外，每一步后可以插入用户代码，每次只完成一个子任务

### 2. 四个阶段

#### 1. 开始阶段

1. root set设置为活跃，root set 由lua可直接访问的对象组成
2. root包含 mainthread、全局表、注册表、基本类型元表 这4个，在开始阶段设置 gray的起点,加入gray链
3. 一个活跃对象可到达的对象也是活跃的
4. 当所有活跃对象被标记后，该阶段结束。

```
5. 1 //c GC初始化，一步完成
2 /* mark root set */
3 static void markroot (lua_State *L) {
4     global_State *g = G(L);
5
6     // 1. 清空链表准备这次GC流程
7     g->gray = NULL;
8     g->grayagain = NULL;
9     g->weak = NULL;
10
11     // 2. 标记mainthread、全局表、注册表为灰色并加入链表gray
12     // 从这几个开始在扫描阶段将它们引用的对象依次遍历
13     markobject(g, g->mainthread);
```

```

14 markvalue(g, gt(g->mainthread));/* make global table be
traversed before main stack */
15 markvalue(g, registry(L));
16 markmt(g); /*基本类型的元方法*/
17
18 //GC 切换到扫描阶段
19 g->gcstate = GCSpropagate;
20 }

```

## 2. 扫描阶段：每个对象都有三种状态之一：白 灰 黑

### 1. 白色：未访问过的（Non-visited objects are marked white）

1. 对象**创建后的初始标记**，表示还没有进入过标记阶段。
2. 如果经过扫描后，在**回收阶段还是白色**，表明该对象没有被其他对象引用，进行回收

### 2. 灰色：（Visited but not-traversed objects）

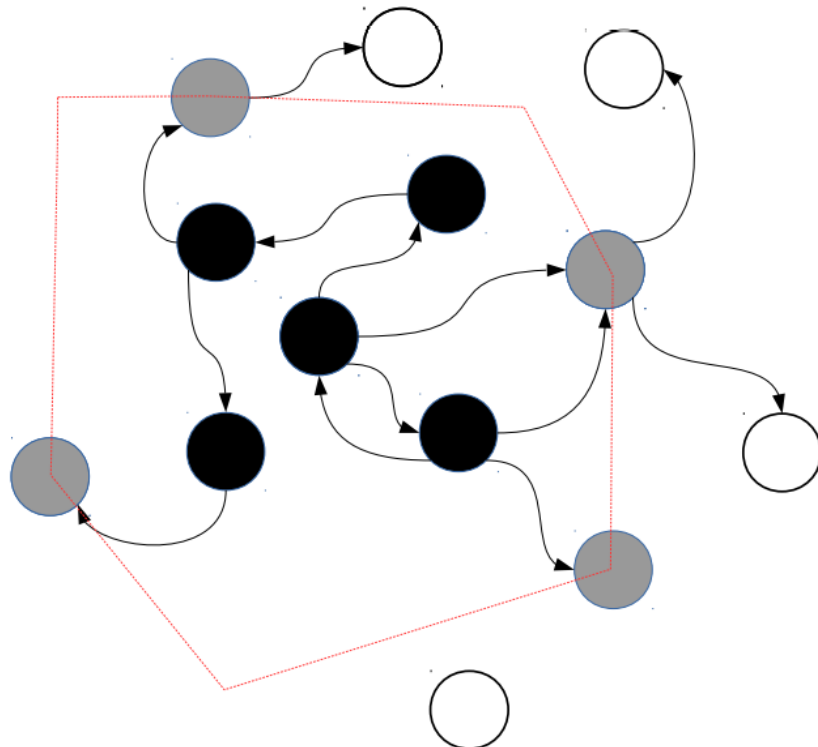
1. 此对象**被扫描过**，且被**其他对象引用了**，
2. 但这个对象引用的其他对象还没有被扫描过，需要加入gray链等待扫描

### 3. 黑色：对象自身以及它引用的对象都已被标记

1. 此对象和它直接引用的对象都已经被扫描过，表明这个对象可以从灰链中删除了

1. root set的对象都是黑色或者灰色
2. 黑色对象不能指向白色，灰色对象是黑色和白色的分界线
3. 黑色变成灰色：for i = 1, N do a[i] = something end
4. 赋值元表可以让白色元表变成灰色：setmetatable(obj, mt)

### 4.



### 5. 新建对象的处理：

1. 如果对象在**GC开始阶段创建**，不需要做特殊处理
2. 如果一个对象在**GC扫描阶段创建**，且引用它的对象还在灰链中，那么不需要做特殊处理
3. 如果对象在**GC扫描阶段**且引用它的对象已经被标记为黑色从灰链中删除，那么要做**向前屏障**，将这个对象从白色标记为灰色加入gray链（引用对象是

table除外)

4. 如果对象在**GC扫描阶段** 且 引用它的**table**已经被标记为黑色从灰链中删除, 则做**向后屏障**, 将table从黑色重新标记为灰色, 加入grayagin链, 在后续原子化扫描阶段处理
5. 如果对象在**GC扫描阶段后创建**, lua采用**双白色机制**, 在不同的GC轮次中交替采用不同的白色, 只有对象标记为当前白才会被回收, 在atomic()函数最后会切换白色, 保证新建对象不会在回收阶段被错误回收
6. 在**结果扫描阶段后**, **所有被引用的对象**都会被标记为黑色, **没有被引用的对象**标记为白色。
7. 回收阶段所有白色的对象会被回收, 且回收不影响用户代码执行, 能够异步进行。
8. 扫描阶段的**任务量 (开销)** 只和活动对象**个数**有关, 和分配的内存大小无关。
  1. 比如在某次GC开始前创建了50,000个对象, 总共占1GB内存, 如果这些对象在扫描开始前大部分都不被引用 (非活动的), 那么扫描阶段的任务量是很小的,
  2. 但是如果它们大部分都能存活到GC的扫描阶段, 那么任务量就会非常大。

### 3. 回收阶段

1. 回收阶段分为回收**字符串**和**其他对象**
2. 因为字符串存储在 global\_State的stringtable 中, 而其他对象存储在 root\_gc链表中
3. 回收方式也差别:
  1. 回收字符串每次回收一个桶, 回收对象每次回收GCSWEEP\_MAX个。
  2. 回收对象时, 遍历global\_State的rootgc链表, 对于一个对象
    1. 如果是黑色, 将它重新标记为当前白, 等待下一轮GC
    2. 如果是白色, 将它从rootgc中删除并释放内存

### 4. 结束阶段

1. GC的最后处理userdata的finalize, 即释放用户内存, 具体方法是调用用户注册的 \_\_gc方法, 每次处理一个userdata

```
2. 1  typedef struct global_State {
    2      //...
    3      lu_byte currentwhite; //当前的白色类型, 用于
    4      luaC_white()
    5      // 当前的GC状态, 有5个, 在lgc.h定义
    6      lu_byte gcstate; /* state of garbage collector */
    7      // strt中字符串散列桶索引, 字符串回收阶段每次回收一个散列桶的字符串, 记录对应的散列桶索引
    8      int sweepstrgc; /* position of sweep in `strt' */
    9      // 所有新建的对象都会暂存在这里, 在GC的清理阶段会增量地遍历整个链表。新建对象会加在最*前面*, 见luaC_link()
    10     GCObject *rootgc; /* list of all collectable objects
    11     */
    12     // 保存rootgc中当前回收到的位置, 下次从这个位置继续回收
    13     GCObject **sweepgc; /* position of sweep in `rootgc'
    14     */
    15     // 灰色链表
    16     GCObject *gray; /* list of gray objects */
    17     // 需要一次性扫描处理的, 不可被打断的对象的灰色链表, 比如
    18     LUA_THREAD
    19     GCObject *grayagain; /* list of objects to be
    20     traversed atomically */
    21     // 弱表
```

```

17     GCObject *weak; /* list of weak tables (to be cleared)
    */
18     // 有__gc方法的userdata, 会在GC阶段调用__gc释放native侧的引用。指向链表最后一个
19     GCObject *tmudata; /* last element of list of userdata
    to be GC */ // taggedmethodudata带__gc的udata
20     //...
21 }

```

## 13. Lua的全局变量跟local变量的区别，Lua是如何查询一个全局变量的，local的变量的作用域规则是怎么样的？

1. 区别：

1. 全局变量，存在全局表：\_G 表中，通过string常量或者string变量作为键来索引全局变量

```

1. 1 //luaolib.h
2   /* global table */
3   #define LUA_GNAME    "_G"
4
5   //lapi.c
6   /*
7   ** Get the global table in the registry. Since all predefined
8   ** indices in the registry were inserted right when the
9   ** registry
10  ** was created and never removed, they must always be in the
11  ** array
12  ** part of the registry.
13  */
14  #define getGtable(L) \
15      (&hvalue(&G(L)->l_registry)->array[LUA_RIDX_GLOBALS - 1])
16
17  LUA_API int lua_getglobal (lua_State *L, const char *name) {
18      const TValue *G;
19      lua_lock(L);
20      G = getGtable(L);
21      return auxgetstr(L, G, name);
22  }
23
24  LUA_API void lua_setglobal (lua_State *L, const char *name) {
25      const TValue *G;
26      lua_lock(L); /* unlock done in 'auxsetstr' */
27      G = getGtable(L);
28      auxsetstr(L, G, name);
29  }
30
31  /*
32  ** get functions (Lua -> stack)
33  */
34  static int auxgetstr (lua_State *L, const TValue *t, const char
35  *k) {
36      const TValue *slot;
37      TString *str = luaS_new(L, k);
38      if (luaV_fastget(L, t, str, slot, luaH_getstr)) {
39          setobj2s(L, L->top, slot);
40      }
41  }

```

```

37     api_incr_top(L);
38 }
39 else {
40     setsvalue2s(L, L->top, str);
41     api_incr_top(L);
42     luaV_finishget(L, t, s2v(L->top - 1), L->top - 1, slot);
43 }
44 lua_unlock(L);
45 return ttype(s2v(L->top - 1));
46 }

```

2. local变量, 在栈上通过 整数索引获得, 最多200个

```

1. 1  /* maximum number of local variables per function (must be
2     smaller
3     than 250, due to the bytecode format) */
4     #define MAXVARS      200
5     /*
6     ** Register a new local variable in the active 'Proto' (for
7     debug
8     ** information).
9     */
10    static int registerlocalvar (LexState *ls, FuncState *fs,
11        TString *varname) {
12        Proto *f = fs->f;
13        int oldsize = f->sizelocvars;
14        luaM_growvector(ls->L, f->locvars, fs->ndebugvars, f-
15            >sizelocvars,
16            LocVar, SHRT_MAX, "local variables");
17        while (oldsize < f->sizelocvars)
18            f->locvars[oldsize++].varname = NULL;
19        f->locvars[fs->ndebugvars].varname = varname;
20        f->locvars[fs->ndebugvars].startpc = fs->pc;
21        luaC_objbarrier(ls->L, f, varname);
22        return fs->ndebugvars++;
23    }
24    /*
25    ** Create a new local variable with the given 'name'. Return
26    its index
27    ** in the function.
28    */
29    static int new_localvar (LexState *ls, TString *name) {
30        lua_State *L = ls->L;
31        FuncState *fs = ls->fs;
32        Dyndata *dyd = ls->dyd;
33        Vardesc *var;
34        checklimit(fs, dyd->actvar.n + 1 - fs->firstlocal,
35            MAXVARS, "local variables");
36        luaM_growvector(L, dyd->actvar.arr, dyd->actvar.n + 1,
37            dyd->actvar.size, Vardesc, USHRT_MAX, "local
38            variables");
39        var = &dyd->actvar.arr[dyd->actvar.n++];
40        var->vd.kind = VDKREG; /* default */
41        var->vd.name = name;
42        return dyd->actvar.n - 1 - fs->firstlocal;
43    }
44 }

```



## 2. 如何查询一个全局变量:

```

1.      a = "***+**+***+**+***+**+***+**+***+**+"
2      b = 987654
3      c = "123"
4      d = pairs
5      for k, v in pairs(_G) do
6          print(k,v)
7      end
8      -- 输出
9      -- assert    function: 0xc1
10     -- collectgarbage function: 0xc4
11     -- dofile     function: 0xc6
12     -- error      function: 0xc8
13     -- getmetatable function: 0xca
14     -- ipairs     function: 0xcc
15     -- load        function: 0xce
16     -- loadfile   function: 0xd0
17     -- next       function: 0xd2
18     -- pairs      function: 0xd4
19     -- pcall      function: 0xd6
20     -- print      function: 0x20
21     -- rawequal   function: 0xd9
22     -- rawget     function: 0xdb
23     -- rawlen     function: 0xdd
24     -- rawset     function: 0xdf
25     -- select     function: 0xe1
26     -- setmetatable function: 0xe3
27     -- tonumber   function: 0xe5
28     -- tostring   function: 0xe7
29     -- type       function: 0xe9
30     -- xpcall     function: 0xeb
31     -- _G         table: 0x3
32     -- _VERSION   Lua 5.3
33     -- require    function: 0xe
34     -- package    table: 0x7
35     -- coroutine  table: 0xf
36     -- table      table: 0x10
37     -- os         table: 0x11
38     -- string     table: 0x12
39     -- math       table: 0x14
40     -- utf8       table: 0x15
41     -- debug      table: 0x16
42     -- fengari    table: 0x17
43     -- js         table: 0x18
44     -- sysLuaStartTimer function: 0xfa
45     -- sysLuaStopTimer function: 0xfc
46     -- json       table: 0x85
47     -- @jsTriggerList table: 0x88
48     -- jsTriggerRegister function: 0x89
49     -- log        table: 0x96
50     -- gpio       table: 0xa5
51     -- adc        table: 0xb0
52     -- pwm        table: 0xba
53     -- JS_FUNCTION_SYS_RUN function: 0x105
54     -- a          ***+**+***+**+***+**+***+**+***+**+
55     -- b          987654
56     -- c          123
```

### 3. 作用域规则：（和C语言类似）

1. 函数内定义局部变量，作用域在函数中。
2. local定义在文件中，作用域就在文件中
3. 所有Lua的标准库都是通过**全局变量**暴露给使用者。
4. 优先权：局部变量覆盖全局变量

## 14. 详细说明元表的机制和它的一些特性

### [元表 LuaOS文档](#)

元表通过对于某个table定制特定的元方法，来让table拥有 面对非预定义行为的指定解决方法（感觉类似C++运算符重载）

```

1  local t2 = { a = 2 }
2  local t1 = { a = 1 }
3  local mt = {
4      __add = function(t1, t2)  --元方法
5          return t1.a + t2.a
6      end
7  }
8  function mt:__call(str)  --元方法
9      print(str)
10 end
11 setmetatable(t1, mt)
12 setmetatable(t2, mt)
13 print("__add:", t1 + t2)
14 t1("11111111qweasdzxc")
15 t2("222222qweasdzxc")
16 --输出
17 -- __add:  3
18 --11111111qweasdzxc
19 --222222qweasdzxc
20 --string

```

- `__add: +` 操作。如果任何不是数字的值（包括不能转换为数字的字符串）做加法，Lua 就会尝试调用元方法。首先、Lua 检查第一个操作数（即使它是合法的），如果这个操作数没有为“`__add`”事件定义元方法，Lua 就会接着检查第二个操作数。一旦 Lua 找到了元方法，它将把两个操作数作为参数传入元方法，元方法的结果（调整为单个值）作为这个操作的结果。如果找不到元方法，将抛出一个错误。
- `__sub: -` 操作。行为和“add”操作类似。
- `__mul: *` 操作。行为和“add”操作类似。
- `__div: /` 操作。行为和“add”操作类似。
- `__mod: %` 操作。行为和“add”操作类似。
- `__pow: ^`（次方）操作。行为和“add”操作类似。
- `__unm: -`（取负）操作。行为和“add”操作类似。
- `__idiv: //`（向下取整除法）操作。行为和“add”操作类似。
- `__band: &`（按位与）操作。行为和“add”操作类似，不同的是 Lua 会在任何一个操作数无法转换为整数时 尝试取元方法。

- `__bor: |` (按位或) 操作。行为和 “band” 操作类似。
- `__bxor: ~` (按位异或) 操作。行为和 “band” 操作类似。
- `__bnot: ~` (按位非) 操作。行为和 “band” 操作类似。
- `__shl: <<` (左移) 操作。行为和 “band” 操作类似。
- `__shr: >>` (右移) 操作。行为和 “band” 操作类似。
- `__concat: ..` (连接) 操作。行为和 “add” 操作类似，不同的是 Lua 在任何操作数即不是一个字符串也不是数字 (数字总能转换为对应的字符串) 的情况下尝试元方法。
- `__len: #` (取长度) 操作。如果对象不是字符串，Lua 会尝试它的元方法。如果有元方法，则调用它并将对象以参数形式传入，而返回值 (被调整为单个) 则作为结果。如果对象是一张表且没有元方法，Lua 使用表的取长度操作 (参见 §3.4.7)。其它情况，均抛出错误。
- `__eq: ==` (等于) 操作。和 “add” 操作行为类似，不同的是 Lua 仅在两个值都是表或都是完全用户数据且它们不是同一个对象时才尝试元方法。调用的结果总会被转换为布尔量。
- `__lt: <` (小于) 操作。和 “add” 操作行为类似，不同的是 Lua 仅在两个值不全为整数也不全为字符串时才尝试元方法。调用的结果总会被转换为布尔量。
- `__le: <=` (小于等于) 操作。和其它操作不同，小于等于操作可能用到两个不同的事件。首先，像 “lt” 操作的行为那样，Lua 在两个操作数中查找 “\_\_le” 元方法。如果一个元方法都找不到，就会再次查找 “\_\_lt” 事件，它会假设 `a <= b` 等价于 `not (b < a)`。而其它比较操作符类似，其结果会被转换为布尔量。
- `__index: 索引 table[key]`。当 `table` 不是表或是表 `table` 中不存在 `key` 这个键时，这个事件被触发。此时，会读出 `table` 相应的元方法。  
 尽管名字取成这样，这个事件的元方法其实可以是一个函数也可以是一张表。如果它是一个函数，则以 `table` 和 `key` 作为参数调用它。如果它是一张表，最终的结果就是以 `key` 取索引这张表的结果。(这个索引过程是走常规的流程，而不是直接索引，所以这次索引有可能引发另一次元方法。)
- `__newindex: 索引赋值 table[key] = value`。和索引事件类似，它发生在 `table` 不是表或是表 `table` 中不存在 `key` 这个键的时候。此时，会读出 `table` 相应的元方法。  
 同索引过程那样，这个事件的元方法即可以是函数，也可以是一张表。如果是一个函数，则以 `table`、`key`、以及 `value` 为参数传入。如果是一张表，Lua 对这张表做索引赋值操作。(这个索引过程是走常规的流程，而不是直接索引赋值，所以这次索引赋值有可能引发另一次元方法。)  
 一旦有了 “newindex” 元方法，Lua 就不再做最初的赋值操作。(如果有必要，在元方法内部可以调用 `rawset` 来做赋值。)
- `__call: 函数调用操作 func(args)`。当 Lua 尝试调用一个非函数的值的时候会触发这个事件 (即 `func` 不是一个函数)。查找 `func` 的元方法，如果找得到，就调用这个元方法，`func` 作为第一个参数传入，原来调用的参数 (`args`) 后依次排在后面。

## 15. lua 那些情况会产生 垃圾回收

### 1. 被动触发:

1. 周期性的自动进行 (从C语言源码看，GCdept大于0时，就会触发自动GC)
2. 堆上的内存不足时触发

### 2. 主动触发:

1. 主动调用 `collectgarbage("collect")`

```

2  ** Does one step of collection when debt becomes positive. 'pre'/'pos'
3  ** allows some adjustments to be done only when needed. macro
4  ** 'condchangemem' is used only for heavy tests (forcing a full
5  ** GC cycle on every opportunity)
6  */
7  #define luaC_condGC(L,pre,pos) \
8      { if (G(L)->GCdebt > 0) { pre; luaC_step(L); pos;}; \
9        condchangemem(L,pre,pos); }
10 /* more often than not, 'pre'/'pos' are empty */
11 #define luaC_checkGC(L)      luaC_condGC(L,(void)0,(void)0)
12
13 /*
14  ** Union of all Lua values
15  */
16 typedef union Value {
17     struct GCObject *gc;    /* collectable objects */
18     void *p;               /* light userdata */
19     lua_CFunction f; /* light C functions */
20     lua_Integer i;        /* integer numbers */
21     lua_Number n;         /* float numbers */
22 } Value;

```

## 16. C# 不想让外部使用new的方式 生成对象，有什么办法？

经典单例模式，private 化构造函数，在类内部函数，new出来对象，调用构造函数

同一时间内只允许一个实例对某些数据进行操作

```

1      public class ObjectClass
2      {
3          private static ObjectClass singleton;
4          private ObjectClass()
5          {
6          }
7
8          public static ObjectClass GetSingleton()
9          {
10             if (singleton == null)
11             {
12                 singleton = new ObjectClass();
13             }
14             return singleton;
15         }
16     }

```

## 17. 非运行时的代码比如OnDrawGizmos 里面 new出来的资源，删除时机的隐患是什么（注意不加编辑器运行的属性，OnDisable，OnDestroy是不会执行的）

OnDrawGizmos 在程序一运行就执行,每帧都运行

```

1 using System.Collections;
2 using System.Collections.Generic;
3 using UnityEngine;
4 public class OnDeawGizmosObject : MonoBehaviour
5 {
6     private void OnDrawGizmos()
7     {
8         Gizmos.DrawWireSphere(Vector3.zero, 10);
9         GameObject go = new GameObject();
10    }
11 }

```

因为不会调用 `OnDisable` `OnDestroy` 所以流程可能出现问题，需要手动 `Destory` 不然可能造成内存泄漏

并且如果有在最后 `OnDisable` `OnDestroy` 流程中，保存数据，传递数据等操作，会失效

## 18. 什么是lua的尾调用，它有什么作用？

### 函数尾调用

1. 尾调用是一种类似在函数结尾的 `goto` 调用，当函数最后一个动作是调用另外一个函数时，我们称这种调用尾调用。

```

1. 1 function f()
2     return g()
3 end

```

2. `f` 调用 `g` 后不会再做任何事情，这种情况下当被调用函数 `g` 结束时程序不需要返回到调用者 `f`；所以尾调用之后程序不需要在栈中保留关于调用者的任何信息。

2. 尾调用**不需要使用栈空间**，那么尾调用递归的层次可以无限制的。

```

1. 1 function foo(n)
2     if n > 0 then return foo(n - 1) end
3 end

```

3. 正确的尾调用：需要做完`f`调用`g`不再做任何事情，错误的尾调用会创建一个栈，多次调用后可能导致**栈溢出**

```

1. 1 function f(x)
2     g(x)
3     return
4 end
5 return g(x) + 1           -- must do the addition
6 return x or g(x)         -- must adjust to 1 result
7 return (g(x))             -- must adjust to 1 result

```

## 19. 评估Lua字符串拼接的消耗，有没效率更高的拼接方式

```

1 --加载sys库
2 sys = require("sys")
3
4 function operatorConcat()
5     local str = "string"
6     for i = 1, 100000 do

```

```

7      str = str .. "a"
8      end
9  end
10 function formatConcat()
11     local result = "string"
12     for i = 1, 100000 do
13         result = string.format("%s%s", result, "a")
14     end
15 end
16 function tableConcat()
17     local t = {}
18     for i = 1, 100000 do
19         table.insert(t,"string")
20     end
21     table.concat(t)
22 end
23 local startTime = os.clock()
24 operatorConcat()
25 print("operatorConcat cost time:", os.clock() - startTime)
26
27 startTime = os.clock()
28 formatConcat()
29 print("formatConcat cost time:", os.clock() - startTime)
30
31 startTime = os.clock()
32 tableConcat()
33 print("tableConcat cost time:", os.clock() - startTime)
34
35 输出:
36 operatorConcat cost time:  3.4745999999577
37 formatConcat cost time: 11.6083000000022
38 tableConcat cost time:  0.42940000002272

```

| 方式           | 时间               |
|--------------|------------------|
| ..           | 3.4745999999577  |
| format       | 11.6083000000022 |
| table.concat | 0.42940000002272 |

## 20. lua 表的2种初始化方式：哪种性能高 差多少？为什么？

1. 

```

1  for i = 1, 1000000 do
2      local a = {}
3      a[1] =1; a[2] =2; a[3] =3
4  end

```

2. 

```

1  for i = 1, 1000000 do
2      local a = {true,true,true}
3      a[1] =1; a[2] =2; a[3] =3
4  end

```

一眼第二种要开辟内存，耗时肯定高

```
1 function first( ... )
2     for i = 1, 1000000 do
3         local a = {}
4         a[1] = 1; a[2] = 2; a[3] = 3
5     end
6 end
7 function second( ... )
8     for i = 1, 1000000 do
9         local a = {true,true,true}
10        a[1] = 1; a[2] = 2; a[3] = 3
11    end
12 end
13 local startTime = os.clock()
14 first()
15 print("first cost time:", os.clock() - startTime)
16
17 startTime = os.clock()
18 second()
19 print("second cost time:", os.clock() - startTime)
20
21 输出:
22 first cost time:      0.7533000000119
23 second cost time:    0.92859999996426
```

| 方式                | 时间               |
|-------------------|------------------|
| first cost time:  | 0.7533000000119  |
| second cost time: | 0.92859999996426 |

## 21. 测试一下以下2种代码的性能差异

```
1. 1 for i = 1, 1000000 do
2     local x = math.sin(i)
3 end
```

```
2. 1 local sin = math.sin
2   for i = 1, 1000000 do
3       local x = sin(i)
4   end
```

```
1 function first( )
2     for i = 1, 1000000 do
3         local x = math.sin(i)
4     end
5 end
6 function second( )
7     local sin = math.sin
8     for i = 1, 1000000 do
9         local x = sin(i)
10    end
11 end
```

```

12 local startTime = os.clock()
13 first()
14 print("first cost time:", os.clock() - startTime)
15 startTime = os.clock()
16 second()
17 print("second cost time:", os.clock() - startTime)
18 -- 输出
19 -- first cost time: 0.84560000002375
20 -- second cost time: 0.6414000000359

```

第一种需要多次取hash, mash.sin,这个点的性能消耗

| 方式                | 时间               |
|-------------------|------------------|
| first cost time:  | 0.84560000002375 |
| second cost time: | 0.6414000000359  |

## 22. local 变量的访问，在函数中定义直接访问和在文件中定义 性能上有何区别？

[高性能 Lua 技巧](#)

直接回答：

1. 函数中直接访问local变量，是在函数的自己的寄存器上（lua虚拟机）一个数组+索引实现。
2. 文件定义中访问local变量，需要取到文件的table中取hash，从中获得文件中的局部变量。
3. 预编译时便能将所有的局部变量存到寄存器中。所以，在 Lua 中访问局部变量是很快的
4. 每个函数都有其自己的寄存器。由于每条指令只有 8 个 bit 用来指定寄存器，每个函数便可以使用多至 250 个寄存器。
5. 如果 a 和 b 是局部变量，语句 a = a + b 只生成一条指令：ADD 0 0 1（假设 a 和 b 分别在寄存器 0 和 1 中）。对比之下，如果 a 和 b 是全局变量，生成上述加法运算的指令便会如下：

```

1. 1 GETGLOBAL    0 0      ; a
   2 GETGLOBAL    1 1      ; b
   3 ADD           0 0 1
   4 SETGLOBAL    0 0      ; a

```

```

1  --函数内局部变量
2  function localFunctionVariable( )
3      local startTime = os.clock()
4      local localVariable = 1
5      for i = 1, 1000000 do
6          localVariable = localVariable + 1
7      end
8      print("函数中local变量访问 cost time:", os.clock() - startTime,
localVariable)
9  end
10 -- 文件内局部变量
11 local localFileVariable = 1
12 function localFile( )
13     local startTime = os.clock()
14     for i = 1, 1000000 do
15         localFileVariable = localFileVariable + 1

```



```

16     end
17     print("文件中local变量访问 cost time:", os.clock() -
startime,localFilevariable)
18 end
19 -- 全局变量
20 globalVaribal = 1
21 function global( )
22     local starttime = os.clock()
23     for i = 1, 1000000 do
24         globalVaribal = globalVaribal + 1
25     end
26     print("全局变量访问 cost time:", os.clock() - starttime, globalVaribal)
27 end
28
29 localFunctionVariable()
30 localFile()
31 global()
32
33 --输出
34 --函数中local变量访问 cost time:    0.19130000001132    1000001
35 --文件中local变量访问 cost time:    0.24780000001192    1000001
36 --全局变量访问 cost time:        0.30239999997502    1000001

```

| 方式         | 时间               |
|------------|------------------|
| 函数中local变量 | 0.19130000001132 |
| 文件中local变量 | 0.24780000001192 |
| 全局变量       | 0.30239999997502 |