

# Contents

<b>1 前言</b>	<b>3</b>
<b>2 UnityProfiler</b>	<b>7</b>
2.1 简介 . . . . .	7
2.2 原理 . . . . .	8
2.3 命令手册 . . . . .	9
2.3.1 alloc . . . . .	9
2.3.2 info . . . . .	9
2.3.3 frame . . . . .	10
2.3.4 next . . . . .	11
2.3.5 prev . . . . .	11
2.3.6 func . . . . .	11
2.3.7 find . . . . .	11
2.3.8 list . . . . .	12
2.3.9 meta . . . . .	13
2.3.10 lock . . . . .	14
2.3.11 stat . . . . .	14
2.3.12 seek . . . . .	14
2.3.13 fps . . . . .	15
2.3.14 help . . . . .	15
2.3.15 quit . . . . .	16
2.4 使用案例 . . . . .	17
2.4.1 追踪渲染丢帧 . . . . .	17
2.4.2 追踪动态内存分配 . . . . .	17
2.5 小结 . . . . .	18
<b>3 MemoryCrawler</b>	<b>18</b>
3.1 简介 . . . . .	18
3.2 原理 . . . . .	19
3.3 命令手册 . . . . .	19
3.3.1 read . . . . .	19
3.3.2 load . . . . .	19
3.3.3 track . . . . .	19
3.3.4 str . . . . .	20

3.3.5	ref . . . . .	21
3.3.6	REF . . . . .	24
3.3.7	uref . . . . .	24
3.3.8	UREF . . . . .	24
3.3.9	kref . . . . .	25
3.3.10	KREF . . . . .	25
3.3.11	ukref . . . . .	26
3.3.12	UKREF . . . . .	26
3.3.13	link . . . . .	26
3.3.14	unlink . . . . .	26
3.3.15	show . . . . .	27
3.3.16	ushow . . . . .	27
3.3.17	find . . . . .	28
3.3.18	ufind . . . . .	28
3.3.19	type . . . . .	28
3.3.20	utype . . . . .	29
3.3.21	stat . . . . .	29
3.3.22	ustat . . . . .	30
3.3.23	list . . . . .	30
3.3.24	ulist . . . . .	30
3.3.25	bar . . . . .	31
3.3.26	ubar . . . . .	31
3.3.27	heap . . . . .	32
3.3.28	save . . . . .	32
3.3.29	uuid . . . . .	33
3.3.30	help . . . . .	33
3.3.31	quit . . . . .	33
3.4	使用案例 . . . . .	34
3.4.1	追踪内存增长 . . . . .	34
3.4.2	追踪内存泄漏 . . . . .	34
3.4.3	优化 Mono 内存 . . . . .	34
3.5	小结 . . . . .	34

# 1 前言

Unity 是个普及度很高拥有大量开发者的游戏开发引擎，其提供的 Unity 编辑器可以快速的开发移动设备游戏，并且通过编辑器扩展可以很容易开发出项目需要的辅助工具，但是 Unity 提供性能调试工具非常简陋，功能简单并且难以使用，如果项目出现性能问题，定位起来相当花时间，并且准确率很低，一定程度上靠运气。

## Profiler

目前 Unity 随包提供的只有 Profiler 工具，里面聚合 CPU、GPU、内存、音频、视频、物理、网络等多个维度的性能数据，但是我们大部分情况下只是用它来定位卡顿问题，也就是主要 CPU 时间消耗（图1）。

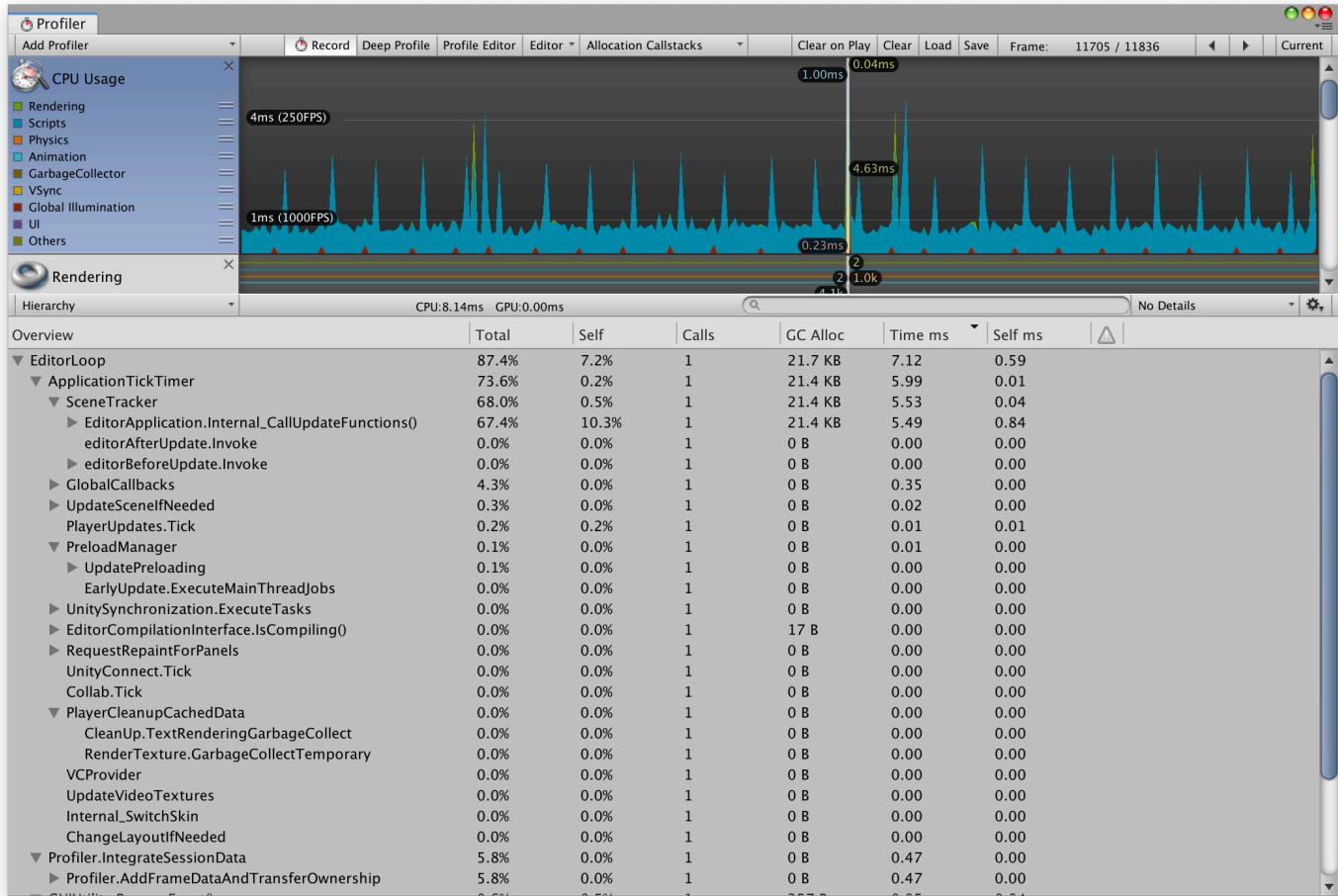


Figure 1: Unity 性能调试工具 Profiler

在 CPU 的维度里面，可以看到当前渲染帧的函数调用层级关系，以及每个函数的时间开销以及调用次数等信息，但是这个工具同一时间只能处理 300 帧左右的数据，如果游戏帧率 30，那么只能看到过去 10 秒的信息，并且需要我们一直盯着图表看才有机会发现意外的丢帧情况，这种设计非常的不友好，违反正常人的操作习惯，因为通常情况下如果我要调试游戏内战斗过程的性能开销，首先我要像普通玩家那样安安静静的玩一把，而不是分散出大部分精力去看一个只有 10 秒历史的滚动图表。这种交互带来两个明显的问题，

- 由于分心去看 Profiler，导致不能全心投入游戏，从而不能收集正常战斗过程的性能数据
- 为了收集数据需要像正常玩家那样打游戏，不能全神关注 Profiler 图表，从而不能发现/查看所有的性能问题

上面两个情形相互排斥，鱼与熊掌不可兼得。从这个角度来看，Profiler 不是一个好的性能调试工具，苛刻的操作条件导致我们很难发现性能问题，想要通过 Profiler 定位所有的性能问题简直是痴人说梦。

## MemoryProfiler

Unity 还提供另外一个内存分析工具 MemoryProfiler(图2)

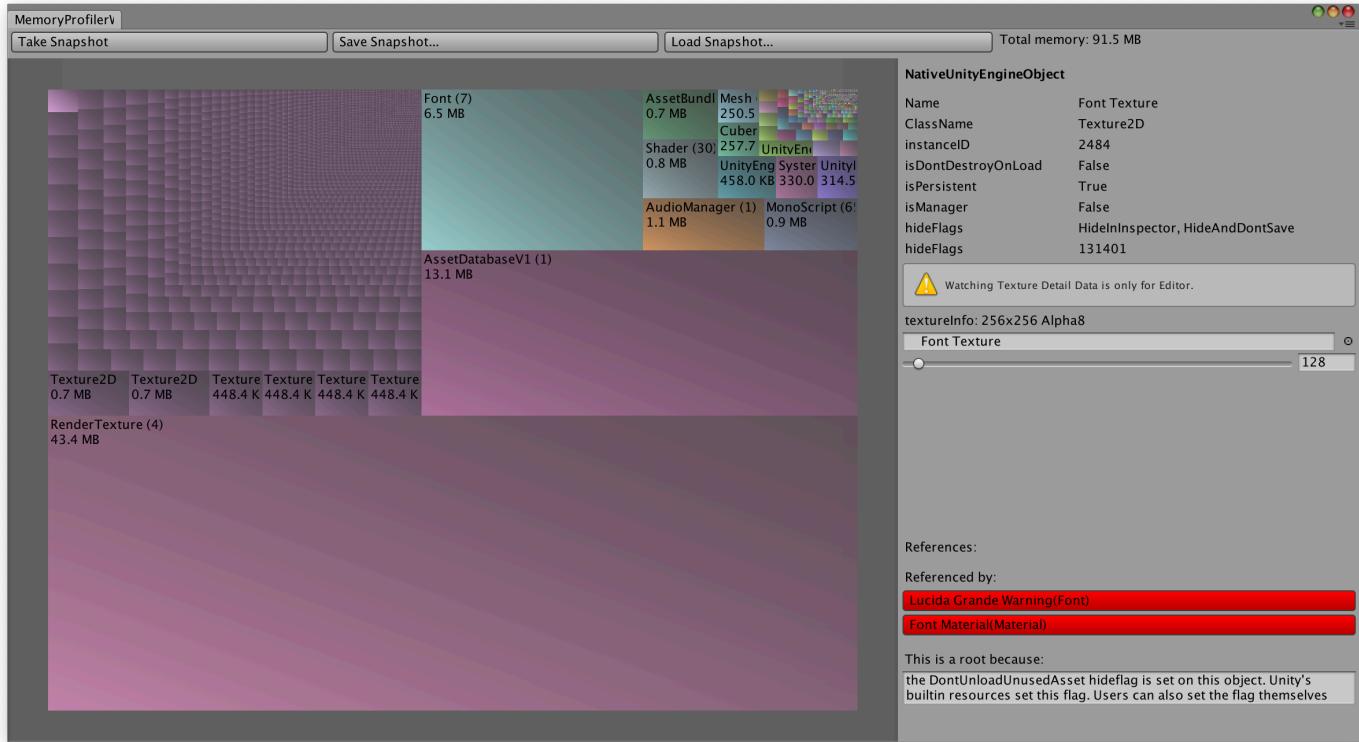


Figure 2: Unity 内存调试工具 MemoryProfiler

在这个界面的左边彩色区域里，*MemoryProfiler* 按类型占用总内存大小绘制对应面积比例的矩阵图，第一次看到还是蛮酷炫的，*Unity* 是想通过这个矩阵图向开发者提供对象内存检索入口，但是实际使用过程中问题多多。

1. 内存分析过程缓慢
2. 在众多无差别的小方格里面找到实际关心的资源很难，虽然可以放大缩小，但感觉并没有提升检索的便利性
3. 每个对象只提供父级引用关系，无法看到完整的对象引用链，容易在跳转过程中搞错上下文关系
4. 引擎对象的引用和 *iL2CPP* 对象的引用混为一谈，让使用者对引用关系的理解模糊不清
5. 没有按引擎对象内存和 *iL2CPP* 对象内存分类区别统计，加深使用者对内存使用的误解

*MemoryProfiler*源码托管在 *Bitbucket*，但是从最后提交记录来看，这个内存工具已经超过 2 年半没有任何更新了，但是这期间 *Unity* 可是发布了好多个版本，想想就有点后怕。

## Commits

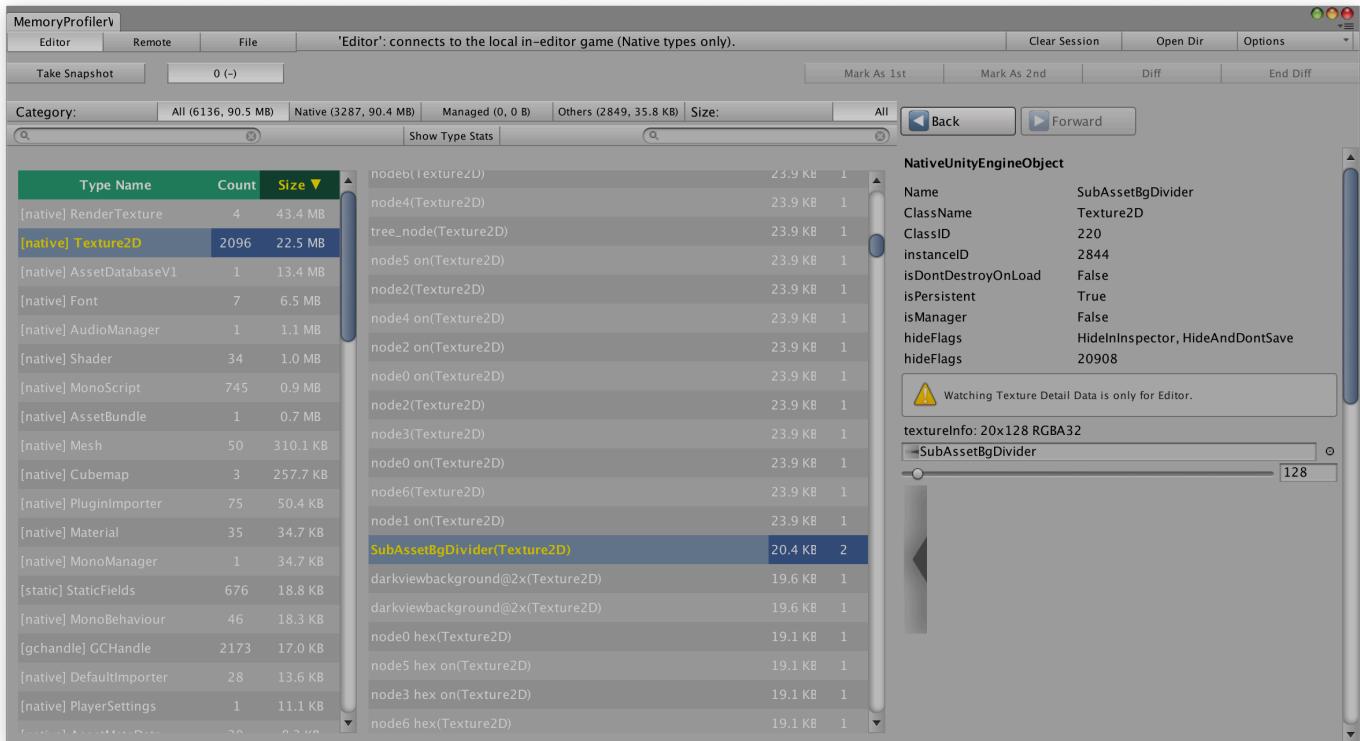


All branches ▾

Find commits

Author	Commit	Message	Date	Builds
Lukasz Paczko...	e8317df M	Merged in lukaszunity/include-mono-support-in-readme (pull request ...)	2017-10-13	
Lukasz Paczko...	abd35a7	Close branch lukaszunity/include-mono-sup...	2017-10-13	<a href="#">lukaszunity/include-mono-sup...</a>
Lukasz Paczko...	8322cc4	Use Mono .NET 3.5 wording instead of 2.0 in ...	2017-09-27	<a href="#">lukaszunity/include-mono-sup...</a>
Lukasz Paczko...	11f359d	Include Mono support in README	2017-09-27	<a href="#">lukaszunity/include-mono-sup...</a>
Tautvydas Žilys	4b2f619 M	Merged in flassari/memoryprofiler/zero-positive-fix (pull request #30) ...	2017-07-29	
Ari Arnbjörnss...	7e8a974	Treat zero as a positive number	2017-07-27	<a href="#">zero-positive-fix</a>
Tautvydas Žilys	a0d5a31 M	Merged in aghogg/memoryprofiler/classid_remove (pull request #31) ...	2017-07-29	
Ashley Hogg	c3d91ad	Removed ClassID field from the object inspector, s...	2017-07-20	<a href="#">classid_remove</a>
Tautvydas Žilys	7126ded M	Merged in aghogg/memoryprofiler/64bit_size_fix (pull request #32) C...	2017-07-29	
Ashley Hogg	6478c5d	Changed size members to 64-bit longs, to fix over...	2017-07-20	<a href="#">64bit_size_fix</a>

有热心开发者也忍受不了 *Unity* 这缓慢的更新节奏，干脆自己动手基于源码在 *github* 上更新优化，并更改了检索的交互方式。



不过这也只是在 *MemoryProfiler* 的基础上有比较大的更新，主要是增加检索的便利性以及内存快照对比，使用起来比 *MemoryProfiler* 初代产品方便了不少，但是由于交互界面的限制，也无法完整展示内存引用关系，内存分析过程依然异常缓慢，甚至会在分析过程中异常崩溃。

**Unity Bug Reporter**

What is the problem related to*	<input type="text" value="Crash bug"/>												
How often does it happen*	<input type="text" value="Please Specify"/>												
Your email address*	<input type="text"/>												
<b>Title*</b>													
<input type="text" value="Describe the problem"/> <div style="border: 1px solid #ccc; padding: 5px; margin-top: 5px;"> <a href="#">How to report bugs</a>      unity3d.com  <a href="#">Public issue tracker</a>      issuetracker.unity3d.com  <a href="#">Unity answers</a>      answers.unity3d.com  <a href="#">Unity forums</a>      forum.unity3d.com  <a href="#">Unity community</a>      unity3d.com  <a href="#">Privacy policy</a>      unity3d.com       </div>													
<b>Details*</b>													
<ol style="list-style-type: none"> <li>1. What happened</li>   <li>2. How we can reproduce it using the example you attached</li> </ol>													
<b>Attached files</b>													
<table border="1" style="width: 100%; border-collapse: collapse;"> <thead> <tr> <th style="padding: 2px;">Path</th> <th style="padding: 2px;">▼</th> <th style="padding: 2px;">Type</th> </tr> </thead> <tbody> <tr> <td style="padding: 2px; vertical-align: top;">/Users/larryhou/Documents/Unity/MemoryProfiler</td> <td style="padding: 2px;"></td> <td style="padding: 2px;">folder</td> </tr> <tr> <td style="padding: 2px; vertical-align: top;">/Users/larryhou/Documents/Unity/MemoryProfiler/Packages/manifest.json</td> <td style="padding: 2px;"></td> <td style="padding: 2px;">.json</td> </tr> <tr> <td colspan="3" style="height: 40px;"></td> </tr> </tbody> </table>		Path	▼	Type	/Users/larryhou/Documents/Unity/MemoryProfiler		folder	/Users/larryhou/Documents/Unity/MemoryProfiler/Packages/manifest.json		.json			
Path	▼	Type											
/Users/larryhou/Documents/Unity/MemoryProfiler		folder											
/Users/larryhou/Documents/Unity/MemoryProfiler/Packages/manifest.json		.json											
<input type="button" value="Add File"/> <input type="button" value="Add Folder"/>	<input type="button" value="Remove"/>												
<b>Report strength:</b> <span style="color: red;">In order to send the report you need to specify how often does the problem happen.</span>													
<div style="background-color: #ccc; height: 10px; width: 100%;"></div>													
<input type="button" value="Preview"/>	<input type="button" value="Cancel"/> <input type="button" value="Send"/>												

鉴于 *Unity* 性能调试工具现实存在问题，我觉得亟待开发面向开发者、提供更多维度、更高效率的性能调试工具，于是我开发了 *UnityProfiler* 和 *MemoryCrawler* 两款工具，分别替代 *Profiler* 以及 *MemoryProfiler* 进行相同领域的性能调试，这两个款均使用纯 C++ 实现，因为经过与 C#、Python 对比后发现 C++ 有绝对的计算优势，可以非常明显提升性能数据分析效率和稳定性。这两款工具的定位是：降低 *Unity* 游戏性能调试的门槛，让拥有不同开发经验的开发者都可以轻松定位各种性能问题，尽管都没有可视化交互界面，不过并不影响分析结果的查看，它们都内置命令行模式的交互方式，并提供了丰富的命令，可以对性能数据做全方位的分析定位。

## 2 UnityProfiler

## 2.1 简介

UnityProfiler 以 *Unity* 引擎自带的 *Profiler* 工具生成的性能数据为基础，提供多种维度的工具来帮助发现性能问题。该工具前期预研阶段使用 *Python* 测试逻辑，最终使用 *C++* 实现。由于是基于 *Unity* 的原生接口获取数据，所以需要保证 *Profiler* 工具打开后能看到性能采集界面，真机调试确保按照官方文档正确配置。

```
MemoryProfiler — UnityProfiler __pfc/20190509152406_PERF.pfc — 109/34
...164824_match.pms ...ofiler/docs — bash ...9152406_PERF.pfc ...oryProfiler — bash ... ...downloads — bash ...7142754_PERF.pfc .../progit/en — bash ...downloads — bash +
```

> alloc

```
[FRAME] index=10000 time=24.850ms fps=40.2 alloc=246 offset=128440337
[FRAME] index=10001 time=24.880ms fps=40.2 alloc=18 offset=128453746
[FRAME] index=10040 time=24.850ms fps=40.2 alloc=246 offset=128972617
[FRAME] index=10041 time=25.140ms fps=39.8 alloc=18 offset=128986026
[FRAME] index=10080 time=24.250ms fps=41.2 alloc=246 offset=129506057
[FRAME] index=10081 time=24.690ms fps=40.5 alloc=18 offset=129519306
[FRAME] index=10092 time=24.740ms fps=40.4 alloc=132 offset=129663381
[FRAME] index=10095 time=24.840ms fps=40.3 alloc=10886 offset=129702904
[FRAME] index=10097 time=24.910ms fps=40.1 alloc=184 offset=129729210
```

> frame

```
[FRAME] index=10000 time=24.850ms fps=40.2 alloc=246 offset=128440337
└ PostLateUpdate.FinishFrameRendering time=39.537%/9.825ms self=0.590%/0.058ms calls=1 *2
  └ Camera.Render time=98.239%/9.652ms self=4.921%/0.475ms calls=5 *3
    └ Culling time=46.726%/4.510ms self=5.344%/0.241ms calls=5 *24
      └ SceneCulling time=88.492%/3.991ms self=23.052%/0.920ms calls=5 *25
        └ CullSendEvents time=73.916%/2.950ms self=2.678%/0.079ms calls=5 *26
          └ WaitForJobGroup time=85.966%/2.536ms self=97.555%/2.474ms calls=8 *27
            └ PrepareSceneNodesCombineJob time=0.946%/0.024ms self=29.167%/0.007ms calls=2 *283
              └ WaitForJobGroup time=70.833%/0.017ms self=23.529%/0.004ms calls=2 *27
                └ PrepareSceneNodesJob time=76.471%/0.013ms self=100.000%/0.013ms calls=2 *284
            └ PrepareSceneNodesSetUp time=0.670%/0.017ms self=100.000%/0.017ms calls=16 *28
            └ CullSceneDynamicObjects time=0.473%/0.012ms self=66.667%/0.008ms calls=1 *277
              └ CullObjectsWithoutUmbra time=33.333%/0.004ms self=100.000%/0.004ms calls=1 *278
            └ SceneNodesInitJob time=0.158%/0.004ms self=100.000%/0.004ms calls=3 *30
            └ WaitForJobSet time=0.118%/0.003ms self=100.000%/0.003ms calls=2 *29
            └ CullSceneDynamicObjectsCombineJob time=0.079%/0.002ms self=50.000%/0.001ms calls=1 *279
              └ CombineJobResults time=50.000%/0.001ms self=100.000%/0.001ms calls=1 *109
            └ ParticleSystem.ScheduleGeometryJobs time=5.322%/0.157ms self=99.363%/0.156ms calls=2 *31
              └ ParticleSystem.GeometryJob time=0.637%/0.001ms self=100.000%/0.001ms calls=1 *32
            └ UpdateRendererBoundingVolumes time=4.237%/0.125ms self=100.000%/0.125ms calls=50 *33
            └ PrepareUpdateRendererBoundingVolumes time=0.983%/0.029ms self=72.414%/0.021ms calls=5 *34
              └ SkinnedMeshPrepareDispatchUpdate time=20.690%/0.006ms self=100.000%/0.006ms calls=5 *35
              └ TransformChangedDispatch time=6.897%/0.002ms self=100.000%/0.002ms calls=5 *36
```

```

MemoryProfiler — UnityProfiler __pfc/20190509152406_PERF.pfc — 109x34
...164824_match.pms ...ofiler/docs -- bash ...9152406_PERF.pfc ...oryProfiler -- bash ...downloads -- bash ...7142754_PERF.pfc .../progit/en -- bash ...downloads -- bash +-
└─PostLateUpdate.SortingGroupsUpdate time=0.012%/0.003ms self=66.667%/0.002ms calls=1 *222
    └─SortingGroupManager.Update time=33.333%/0.001ms self=100.000%/0.001ms calls=1 *223
└─PostLateUpdate.UpdateRectTransform time=0.012%/0.003ms self=100.000%/0.003ms calls=1 *236
    └─TransformChangedDispatch time=0.000%/0.000ms self=nan%/0.000ms calls=1 *36
└─Update.ScriptRunDelayedDynamicFrameRate time=0.012%/0.003ms self=66.667%/0.002ms calls=1 *240
    └─CoroutinesDelayedCalls time=33.333%/0.001ms self=100.000%/0.001ms calls=1 *206
└─FixedUpdate.NewInputBegin FixedUpdate time=0.008%/0.002ms self=50.000%/0.001ms calls=1 *220
    └─NativeInputSystem.NotifyUpdate() time=50.000%/0.001ms self=100.000%/0.001ms calls=1 *156
└─FixedUpdate.ScriptRunDelayedFixedFrameRate time=0.004%/0.001ms self=0.000%/0.000ms calls=1 *243
    └─CoroutinesDelayedCalls time=100.000%/0.001ms self=100.000%/0.001ms calls=1 *206
[      CPU] 'Rendering'=7069000 'Scripts'=1939000 'Physics'=75000 'GarbageCollector'=0 'VSync'=73280
00 'Global Illumination'=60000 'UI'=681000 'Others'=4315000
[      GPU] 'Opaque'=0 'Transparent'=0 'Shadows/Depth'=0 'Deferred PrePass'=0 'Deferred Lighting'=0
'PostProcess'=0 'Other'=0
[      Rendering] 'Batches'=120 'SetPass Calls'=121 'Triangles'=87040 'Vertices'=82944
[      Memory] 'Total Allocated'=205924352 'Texture Memory'=32143360 'Mesh Memory'=20109312 'Material Count'=823 'Object Count'=32852 'Total GC Allocated'=17747968 'GC Allocated'=0
[      Audio] 'Playing Audio Sources'=0 'Audio Voices'=0 'Total Audio CPU'=0 'Total Audio Memory'=0
[      Video] 'Total Video Sources'=0 'Playing Video Sources'=0 'Total Video Memory'=0
[      Physics] 'Active Dynamic'=0 'Active Kinematic'=0 'Static Colliders'=12 'Rigidbody'=0 'Trigger Overlaps'=0 'Active Constraints'=0 'Contacts'=0
[      Physics2D] 'Total Bodies'=0 'Active Bodies'=0 'Sleeping Bodies'=0 'Dynamic Bodies'=0 'Kinematic Bodies'=0 'Static Bodies'=1 'Contacts'=0 'Step Time'=0
[      NetworkMessages] 'Protocol Packets In'=0 'Buffered Msgs In'=0 'Unbuffered Msgs In'=0 'Protocol Packets Out'=0 'Buffered Msgs Out'=0 'Unbuffered Msgs Out'=0 'Pending Buffers'=0
[      NetworkOperations] 'Command'=0 'ClientRPC'=0 'SyncVar'=0 'Sync List'=0 'Sync Event'=0 'User Message'=0 'Object Destroy'=0 'Object Create'=0
[      UI] 'Layout'=347000 'Render'=334000
[      UIDetails] 'UI Batches'=0 'UI Vertices'=0
[GlobalIllumination] 'Total CPU'=437500 'Light Probe'=0 'Setup'=1406 'Environment'=13854 'Input Lighting'=362083 'Systems'=0 'Solve Tasks'=35573 'Dynamic Objects'=14583 'Other Commands'=0 'Blocked Command Write'=0
/> fps
frames=[10000, 10100)=100 fps=40.2±2.1 range=[38.5, 41.9] reasonable=[38.5, 41.9]
/>

```

## 2.2 原理

Unity 编辑器提供的 *Profiler* 调试工具，有多个维度的性能数据，我们比较常用的就是查看 *CPU* 维度的函数调用开销。这个数据可以通过 Unity 未公开的编辑器库 *UnityEditorInternal* 来获取，鉴于未公开也谈不上查阅官方文档来获取性能数据采集细节，所以需要通过反编译查看源码才能知道其实现原理：构造类 *UnityEditorInternal.ProfilerProperty* 对象，调用 *GetColumnAsSingle* 方法来获取函数调用堆栈相关的性能数据。

```

var root = new ProfilerProperty();
root.SetRoot(frameIndex, ProfilerColumn.TotalTime,
    ProfilerViewType.Hierarchy);
root.onlyShowGPUSamples = false;

var drawCalls = root.GetColumnAsSingle(ProfilerColumn.DrawCalls);
samples.Add(sequence, new StackSample
{
    id = sequence,
    name = root.propertyName,
    callsCount = (int)root.GetColumnAsSingle(ProfilerColumn.Calls),
    gcAllocBytes = (int)root.GetColumnAsSingle(ProfilerColumn.GCMemory),
    totalTime = root.GetColumnAsSingle(ProfilerColumn.TotalTime),
}

```

```
    selfTime = root.GetColumnAsSingle(ProfileColumn.SelfTime),  
});
```

除了函数堆栈方面的开销, *Unity* 还有渲染、物理、*UI*、网络等其他维度的数据, 这些数据要通过另外一个接口来获取。

```
for (ProfilerArea area = 0; area < ProfilerArea.AreaCount; area++)  
{  
    var statistics = metadatas[(int)area];  
    stream.Write((byte)area);  
    for (var i = 0; i < statistics.Count; i++)  
    {  
        var maxValue = 0.0f;  
        var identifier = statistics[i];  
        ProfilerDriver.GetStatisticsValues(identifier, frameIndex, 1.0f,  
        provider, out maxValue);  
        stream.Write(provider[0]);  
    }  
}
```

本工具基于以上接口把采集到的数据保存为 *PFC* 格式, 该格式为自定义格式, 使用了多种算法优化数据存储, 比 *Unity* 编辑器录制的原始数据节省 **80%** 的存储空间, 同时用 *C++* 语言编写多种维度的性能分析工具, 可以高效率地定位卡顿问题。

## 2.3 命令手册

### 2.3.1 alloc

**alloc** [*frame\_offset*] [=0] [*frame\_count*] [=0]

参数	可选	描述
<i>frame_offset</i>	是	指定起始帧, 相对于当前帧区间第一帧的整形偏移量
<i>frame_count</i>	是	指定帧数量

alloc 可以在指定的帧区间内搜索所有调用 *GC.Alloc* 分配内存的渲染帧。

```
/> alloc 0 1000  
[FRAME] index=2 time=23.970ms fps=41.7 alloc=10972 offset=12195  
[FRAME] index=124 time=25.770ms fps=38.8 alloc=184 offset=1326925  
[FRAME] index=127 time=24.870ms fps=40.2 alloc=10972 offset=1359192  
[FRAME] index=250 time=25.740ms fps=38.8 alloc=184 offset=2682771  
[FRAME] index=253 time=24.690ms fps=40.5 alloc=10972 offset=2715142
```

如果 *frame\_offset* 和 *frame\_count* 留空, alloc 将所有可用帧作为参数进行条件搜索。

### 2.3.2 info

无参数, 查看当前性能录像的基本信息。

```
/> info
frames=[1, 44611)=44610 elapse=(1557415446.004, 1557416582.579)=1136.574s
↳ fps=39.9±12.8 range=[1.3, 240.6] reasonable=[27.2, 52.5]
```

### 2.3.3 frame

**frame** [frame\_index] [stack\_depth][=0]

参数	可选	描述
frame_index	否	指定帧序号
stack_depth	是	指定函数调用堆栈层级, 默认完整堆栈

frame 可以查看指定渲染帧的详细函数调用堆栈信息, 见下图。

```
/> info
frames=[1, 44611)=44610 elapse=(1557415446.004, 1557416582.579)=1136.574s fps=39.9±12.8 range=[1.3, 240.6] reasonable=[27.2, 52.5]
/> frame 200 1
[FRAME] index=200 time=26.030ms fps=38.4 alloc=0 offset=2144297
└ Initialization.PlayerUpdateTime time=59.516%/15.492ms self=0.123%/0.019ms calls=1 *0
└ PostLateUpdate.FinishFrameRendering time=16.881%/4.394ms self=1.434%/0.063ms calls=1 *2
└ Update.ScriptRunBehaviourUpdate time=6.185%/1.610ms self=0.311%/0.005ms calls=1 *75
└ PreLateUpdate.ScriptRunBehaviourLateUpdate time=2.144%/0.558ms self=0.717%/0.004ms calls=1 *110
└ PostLateUpdate.EnlightenRuntimeUpdate time=1.398%/0.364ms self=1.923%/0.007ms calls=1 *67
└ PostLateUpdate.ProfilerEndFrame time=0.999%/0.260ms self=1.923%/0.005ms calls=1 *126
└ PreUpdate.SendMouseEvents time=0.715%/0.186ms self=2.151%/0.004ms calls=1 *143
└ PostLateUpdate.PlayerUpdateCanvases time=0.699%/0.182ms self=2.747%/0.005ms calls=1 *137
└ PostLateUpdate.PlayerEmitCanvasGeometry time=0.642%/0.167ms self=2.994%/0.005ms calls=1 *133
└ PreLateUpdate.ParticleSystemBeginUpdateAll time=0.611%/0.159ms self=1.887%/0.003ms calls=1 *119
└ PostLateUpdate.UpdateAllRenderers time=0.592%/0.154ms self=20.130%/0.031ms calls=1 *108
└ PostLateUpdate.UpdateCustomRenderTextures time=0.315%/0.082ms self=7.317%/0.006ms calls=1 *147
└ FixedUpdate.ScriptRunBehaviourFixedUpdate time=0.257%/0.067ms self=7.463%/0.005ms calls=1 *150
└ PostLateUpdate.PlayerSendFrameStarted time=0.161%/0.042ms self=23.810%/0.010ms calls=1 *154
└ PreUpdate.PhysicsUpdate time=0.134%/0.035ms self=8.571%/0.003ms calls=1 *165
└ FixedUpdate.ScriptRunDelayedTasks time=0.131%/0.034ms self=20.588%/0.007ms calls=1 *163
└ EarlyUpdate.NewInputBeginFrame time=0.119%/0.031ms self=58.065%/0.018ms calls=1 *168
└ EarlyUpdate.UpdatePreloading time=0.104%/0.027ms self=18.519%/0.005ms calls=1 *171
```

在函数堆栈的底部, 还有当前帧其他性能指标数据, 主要有 CPU、GPU、Rendering、Memory、Audio、Video、Physics、Physics2D、NetworkMessages、NetworkOperations、UI、UIDetails、GlobalIllumination 等 13 类别, 一共 77 个细分性能指标。

```
[PostLateUpdate.UpdateRectTransform time=0.015%/0.004ms self=50.000%/0.002ms calls=1 *236
└ Update.ScriptRunDelayedDynamicFrameRate time=0.015%/0.004ms self=50.000%/0.002ms calls=1 *240
[      CPU] 'Rendering'=3757000 'Scripts'=2073000 'Physics'=83000 'GarbageCollector'=0 'VSync'=15473
000 'Global Illumination'=357000 'UI'=289000 'Others'=1796000
[      GPU] 'Opaque'=0 'Transparent'=0 'Shadows/Depth'=0 'Deferred PrePass'=0 'Deferred Lighting'=0
'PostProcess'=0 'Other'=0
[      Rendering] 'Batches'=21 'SetPass Calls'=19 'Triangles'=0 'Vertices'=0
[      Memory] 'Total Allocated'=58493952 'Texture Memory'=7758848 'Mesh Memory'=122880 'Material Count'
'=28 'Object Count'=4817 'Total GC Allocated'=6246400 'GC Allocated'=0
[      Audio] 'Playing Audio Sources'=0 'Audio Voices'=0 'Total Audio CPU'=0 'Total Audio Memory'=0
[      Video] 'Total Video Sources'=0 'Playing Video Sources'=0 'Total Video Memory'=0
[      Physics] 'Active Dynamic'=0 'Active Kinematic'=0 'Static Colliders'=0 'Rigidbody'=0 'Trigger Overlaps'=0
'Active Constraints'=0 'Contacts'=0
[      Physics2D] 'Total Bodies'=0 'Active Bodies'=0 'Sleeping Bodies'=0 'Dynamic Bodies'=0 'Kinematic Bodies'=0
'Static Bodies'=1 'Contacts'=0 'Step Time'=0
[      NetworkMessages] 'Protocol Packets In'=0 'Buffered Msgs In'=0 'Unbuffered Msgs In'=0 'Protocol Packets Out'=0
'Buffered Msgs Out'=0 'Unbuffered Msgs Out'=0 'Pending Buffers'=0
[      NetworkOperations] 'Command'=0 'ClientRPC'=0 'SyncVar'=0 'Sync List'=0 'Sync Event'=0 'User Message'=0
'Object Destroy'=0 'Object Create'=0
[      UI] 'Layout'=45000 'Render'=244000
[      UIDetails] 'UI Batches'=0 'UI Vertices'=0
[GlobalIllumination] 'Total CPU'=189063 'Light Probe'=0 'Setup'=3334 'Environment'=33646 'Input Lighting'=788
02 'Systems'=0 'Solve Tasks'=27656 'Dynamic Objects'=25834 'Other Commands'=0 'Blocked Command Write'=0
```

每次执行 frame 都会记录当前查询的帧序号，如果所有参数留空，则重复查看当前帧数据。

### 2.3.4 next

**next** [*frame\_offset*] [=1]

参数	可选	描述
<i>frame_offset</i>	是	相对当前帧的偏移

next 命令相当于按照指定帧偏移量修改当前帧序号同时调用 frame 命令生成帧数据。

### 2.3.5 prev

**prev** [*frame\_offset*] [=1]

参数	可选	描述
<i>frame_offset</i>	是	相对当前帧的偏移

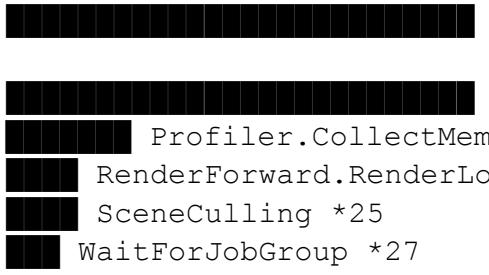
prev 命令相当于按照指定帧偏移量修改当前帧序号同时调用 frame 命令生成帧数据。

### 2.3.6 func

**func** [*rank*] [=0]

参数	可选	描述
<i>rank</i>	是	指定显示排行榜前 <i>rank</i> 个数据

func 在当前可用帧区间内，按照函数名统计每个函数的时间消耗，并按照从大到小的顺序排序，*rank* 参数可以限定列举所有函数的时间统计。

/> func 1 26.27% 1276.54ms #200 /> func 5 26.27% 1276.54ms #200 6.80% 330.49ms #200 4.31% 209.44ms #1818 3.89% 188.95ms #909 3.11% 151.03ms #9071	 WaitForTargetFPS *1  WaitForTargetFPS *1 Profiler.CollectMemoryAllocationStats *128 RenderForward.RenderLoopJob *7 SceneCulling *25 WaitForJobGroup *27
--	---

第一列表示函数时间消耗百分比，第二列表示时间消耗的总毫秒数，第三列表示函数调用的总次数，最后一列以 \* 开头的数字表示函数引用。

### 2.3.7 find

**find** [*function\_ref*]

参数	可选	描述
<i>function_ref</i>	否	指定函数引用

frame 和 func 命令可以生成以 \* 开头的数字函数引用, find 在当前帧区间内查找调用了指定函数的帧。

```
/> func 5
34.99% 849.26ms #100
5.32% 129.07ms #5916
4.92% 119.39ms #100
4.58% 111.14ms #1000
2.20% 53.34ms #500
/> find 128
[FRAME] index=10000 time=24.850ms fps=40.2 offset=128440337
[FRAME] index=10001 time=24.880ms fps=40.2 offset=128453746
[FRAME] index=10002 time=25.070ms fps=39.9 offset=128467283
[FRAME] index=10003 time=25.420ms fps=39.3 offset=128480596
[FRAME] index=10004 time=24.120ms fps=41.4 offset=128494037
[FRAME] index=10005 time=24.930ms fps=40.1 offset=128507158
[FRAME] index=10006 time=25.390ms fps=39.4 offset=128520567
```

### 2.3.8 list

**list** [*frame\_offset*] [=0] [*frame\_count*] [=0]

参数	可选	描述
<i>frame_offset</i>	是	指定始帧, 相对于当前帧区间第一帧的整形偏移量
<i>frame_count</i>	是	指定帧数量

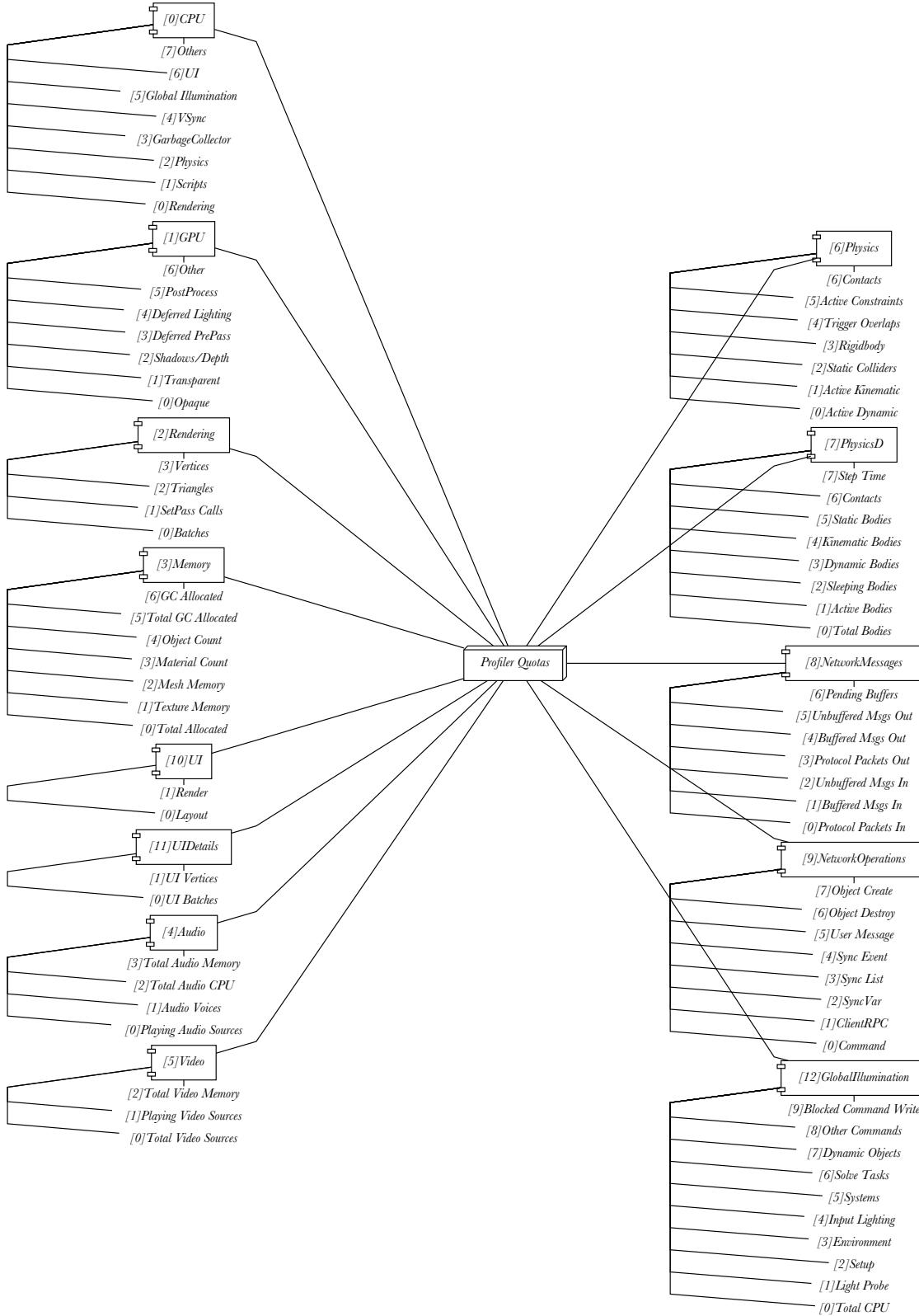
list 列举指定范围的帧基本信息, 如果所有参数留空则列举当前帧区间的所有帧信息。

```
/> list 0 10
[FRAME] index=20000 time=24.900ms fps=40.2 offset=261033153
[FRAME] index=20001 time=25.230ms fps=39.6 offset=261046018
[FRAME] index=20002 time=24.530ms fps=40.8 offset=261059203
[FRAME] index=20003 time=24.840ms fps=40.2 offset=261072356
[FRAME] index=20004 time=24.880ms fps=40.2 offset=261084797
[FRAME] index=20005 time=24.900ms fps=40.2 offset=261097310
[FRAME] index=20006 time=25.270ms fps=39.6 offset=261110143
[FRAME] index=20007 time=24.470ms fps=40.9 offset=261122944
[FRAME] index=20008 time=25.340ms fps=39.5 offset=261135865
[FRAME] index=20009 time=24.400ms fps=41.0 offset=261148698
[SUMMARY] fps=40.2±1.6 range=[39.5, 41.0] reasonable=[39.5, 41.0]
```

该工具同时在所有帧数据底部生成 *fps* 统计数据。

### 2.3.9 meta

meta 查看性能指标索引，包含 *CPU*、*GPU*、*Rendering*、*Memory*、*Audio*、*Video*、*Physics*、*Physics2D*、*NetworkMessages*、*NetworkOperations*、*UI*、*UIDetails*、*GlobalIllumination* 等 13 类别，以及类别属性一共 77 个细分性能指标，每个性能指标由类别和类别属性两个索引确定，比如 *Scripts* 由 0-1 确定，该命令用来为 stat 和 seek 提供输入参数。



### 2.3.10 lock

**lock** [*frame\_index*] [=0] [*frame\_count*] [=0]

参数	可选	描述
<i>frame_index</i>	是	起始帧序号
<i>frame_count</i>	是	锁定相对于起始帧的帧数量

lock 参数留空恢复原始帧区间，一旦锁定帧区间，其他除 info 命令以外的其他命令均在该区间执行相关操作。

```
/> lock 10000 20
frames=[10000, 10020)
/> list
[FRAME] index=10000 time=24.850ms fps=40.2 offset=128440337
[FRAME] index=10001 time=24.880ms fps=40.2 offset=128453746
[FRAME] index=10002 time=25.070ms fps=39.9 offset=128467283
[FRAME] index=10003 time=25.420ms fps=39.3 offset=128480596
[FRAME] index=10004 time=24.120ms fps=41.4 offset=128494037
[FRAME] index=10005 time=24.930ms fps=40.1 offset=128507158
[FRAME] index=10006 time=25.390ms fps=39.4 offset=128520567
[FRAME] index=10007 time=24.590ms fps=40.7 offset=128533880
[FRAME] index=10008 time=24.560ms fps=40.7 offset=128547161
[FRAME] index=10009 time=24.900ms fps=40.2 offset=128560474
[SUMMARY] fps=40.2±1.9 range=[39.3, 41.4] reasonable=[39.3, 41.4]
```

### 2.3.11 stat

**stat** [*profiler\_area*] [*property*]

参数	可选	描述
<i>profiler_area</i>	否	类别索引，meta 命令生成一级索引
<i>property</i>	否	类别属性索引，meta 命令生成的二级索引

stat 在当前帧区间按照参数指标进行数学统计，给出 99.87% 置信区间的边界值，以及均值和标准差信息。

```
/> stat 0 1
[CPU] [Scripts] mean=1874400.000±316545.565 range=[1582000, 2965000]
reasonable=[1582000, 2269000]
```

*range* 表示当前帧区间 *Scripts* 时间消耗的最小值和最大值，单位是纳秒 [1 毫秒 = 1000000 纳秒]，*reasonable* 表示按照 3 倍标准差剔除极大值后的合理取值范围，超出该范围的值应该仔细检查，因为按照统计学在正态分布里面 3 倍标准差可以覆盖 99.87% 的数据。

### 2.3.12 seek

**seek** [*profiler\_area*] [*property*] [*value*] [*predicate*] [= >]

参数	可选	描述
<i>profiler_area</i>	否	类别索引, meta 命令生成一级索引
<i>property</i>	否	类别属性索引, meta 命令生成二级索引
<i>value</i>	否	临界值
<i>predicate</i>	是	> 大于临界值、= 等于临界值、< 小于临界值三种参数

seek 按照参数确定的指标进行所搜比对, 默认列举大于临界值的帧信息, 可以通过 *predicate* 选择大于、等于和小于比对方式进行过滤帧数据。

```
/> stat 0 1
[CPU] [Scripts] mean=1874400.000±316545.565 range=[1582000, 2965000]
↳ reasonable=[1582000, 2269000]
/> seek 0 1 2269000
[FRAME] index=10012 time=23.880ms fps=41.9 offset=128599965
```

调用该命令前建议先用 stat 对性能指标进行简单数学统计, 然后根据最大值或者最小值搜索可能存在性能问题的渲染帧。

### 2.3.13 fps

**fps** [*value*] [*predicate*] [= >]

参数	可选	描述
<i>value</i>	否	临界值
<i>predicate</i>	是	> 大于临界值、= 等于临界值、< 小于临界值三种参数

```
/> fps
frames=[20000, 20100)=100 fps=40.2±1.2 range=[39.2, 41.5] reasonable=[39.2, 41.2]
/> fps 41.2 >
[FRAME] index=20066 time=24.080ms fps=41.5 offset=261880027
```

当参数留空时, fps 统计当前帧区间的帧率信息, 指定临界值后, 则默认过滤大于临界值的帧数据, 可以通过 *predicate* 选择大于、等于和小于比对方式进行过滤帧数据。

### 2.3.14 help

显示帮助信息。

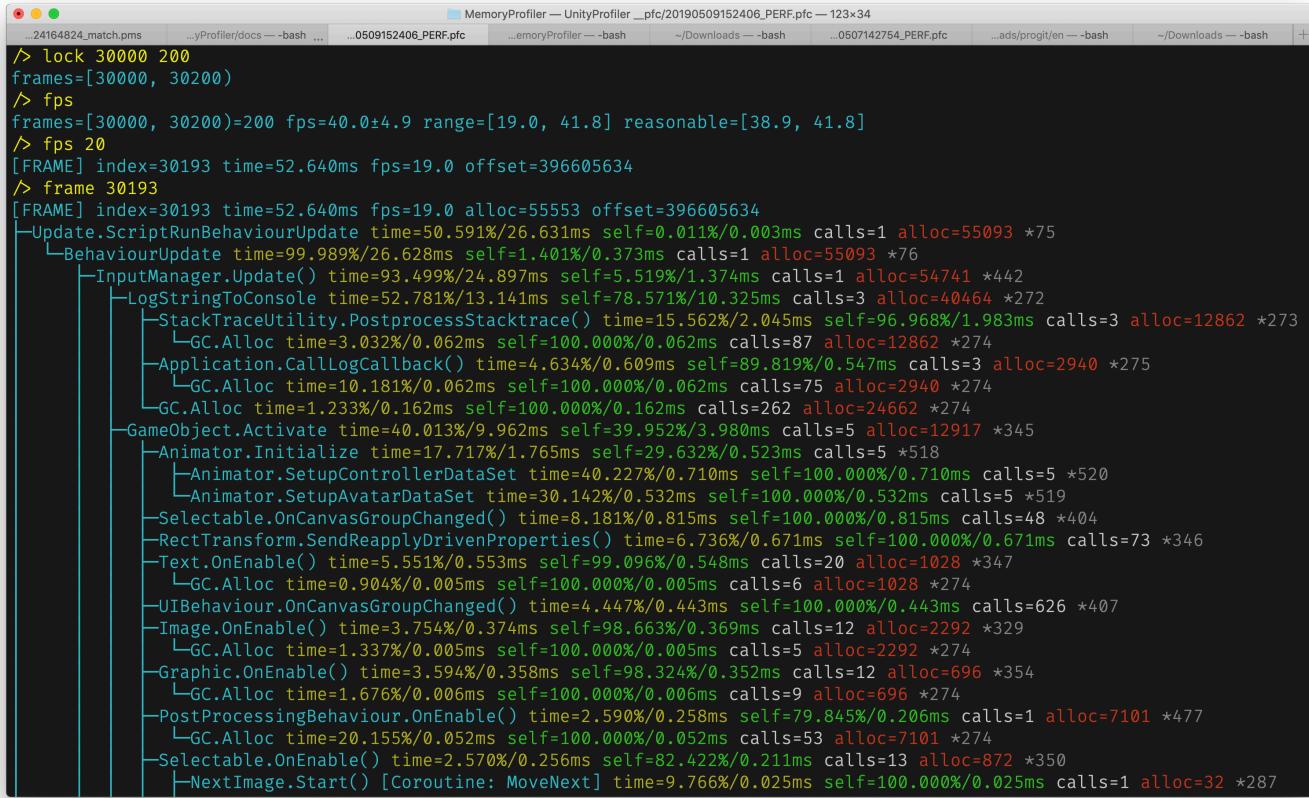
```
> help
alloc [FRAME_OFFSET] [FRAME_COUNT] 搜索申请动态内存的帧
frame [FRAME_INDEX] 查看帧时间消耗详情
func 按照方法名统计时间消耗
find [FUNCTION_NAME_REF]* 按照方法名索引查找调用帧
list [FRAME_OFFSET] [±FRAME_COUNT] [+|-] 列举帧简报 支持排序(+按fps升序 -按fps降序)输出 默认不排序
next [STEP] 查看后STEP[=1]帧时间消耗详情
prev [STEP] 查看前STEP[=1]帧时间消耗详情
meta 查看性能指标参数
lock [FRAME_INDEX] [FRAME_COUNT] 锁定帧范围
stat [PROFILER_AREA] [PROPERTY] 统计性能指标
seek [PROFILER_AREA] [PROPERTY] [VALUE] [>|=|<] 搜索性能指标满足条件(>大于VALUE[默认] =等于VALUE <小于VALUE)的帧
info 性能摘要
  fps [FPS] [>|=|<] 搜索满足条件(>大于FPS =等于FPS <小于FPS[默认])的帧
help 帮助
quit 退出
```

### 2.3.15 quit

退出当前进程。

## 2.4 使用案例

### 2.4.1 追踪渲染丢帧



The screenshot shows the Unity Memory Profiler interface with the following command-line search results:

```
./lock 30000 200
frames=[30000, 30200)
/> fps
frames=[30000, 30200)=200 fps=40.0±4.9 range=[19.0, 41.8] reasonable=[38.9, 41.8]
/> fps 20
[FRAME] index=30193 time=52.640ms fps=19.0 offset=396605634
/> frame 30193
[FRAME] index=30193 time=52.640ms fps=19.0 alloc=55553 offset=396605634
  Update.ScriptRunBehaviourUpdate time=50.591%/26.631ms self=0.011%/0.003ms calls=1 alloc=55093 *75
    BehaviourUpdate time=99.989%/26.628ms self=1.401%/0.373ms calls=1 alloc=55093 *76
      InputManager.Update() time=93.499%/24.897ms self=5.519%/1.374ms calls=1 alloc=54741 *442
        LogStringToConsole time=52.781%/13.141ms self=78.571%/10.325ms calls=3 alloc=40464 *272
          StackTraceUtility.PostprocessStacktrace() time=15.562%/2.045ms self=96.968%/1.983ms calls=3 alloc=12862 *273
            GC.Alloc time=3.032%/0.062ms self=100.000%/0.062ms calls=87 alloc=12862 *274
              Application.CallLogCallback() time=4.634%/0.609ms self=89.819%/0.547ms calls=3 alloc=2940 *275
                GC.Alloc time=10.181%/0.062ms self=100.000%/0.062ms calls=75 alloc=2940 *274
                  GC.Alloc time=1.233%/0.162ms self=100.000%/0.162ms calls=262 alloc=24662 *274
                    GameObject.Activate time=40.013%/9.962ms self=39.952%/3.980ms calls=5 alloc=12917 *345
                      Animator.Initialize time=17.717%/1.765ms self=29.632%/0.523ms calls=5 *518
                        Animator.SetupControllerDataSet time=40.227%/0.710ms self=100.000%/0.710ms calls=5 *520
                          Animator.SetupAvatarDataSet time=30.142%/0.532ms self=100.000%/0.532ms calls=5 *519
                          Selectable.OnCanvasGroupChanged() time=8.181%/0.815ms self=100.000%/0.815ms calls=48 *404
                          RectTransform.SendReapplyDrivenProperties() time=6.736%/0.671ms self=100.000%/0.671ms calls=73 *346
                          Text.OnEnable() time=5.551%/0.553ms self=99.096%/0.548ms calls=20 alloc=1028 *347
                            GC.Alloc time=0.904%/0.005ms self=100.000%/0.005ms calls=6 alloc=1028 *274
                          UIBehaviour.OnCanvasGroupChanged() time=4.447%/0.443ms self=100.000%/0.443ms calls=626 *407
                          Image.OnEnable() time=3.754%/0.374ms self=98.663%/0.369ms calls=12 alloc=2292 *329
                            GC.Alloc time=1.337%/0.005ms self=100.000%/0.005ms calls=5 alloc=2292 *274
                          Graphic.OnEnable() time=3.594%/0.358ms self=98.324%/0.352ms calls=12 alloc=696 *354
                            GC.Alloc time=1.676%/0.006ms self=100.000%/0.006ms calls=9 alloc=696 *274
                          PostProcessingBehaviour.OnEnable() time=2.590%/0.258ms self=79.845%/0.206ms calls=1 alloc=7101 *477
                            GC.Alloc time=20.155%/0.052ms self=100.000%/0.052ms calls=53 alloc=7101 *274
                          Selectable.OnEnable() time=2.570%/0.256ms self=82.422%/0.211ms calls=13 alloc=872 *350
                            NextImage.Start() [Coroutine: MoveNext] time=9.766%/0.025ms self=100.000%/0.025ms calls=1 alloc=32 *287
```

1. 使用 lock 锁定大概帧区间 [30000, 30000+200)
2. 使用无参数 fps 对当前帧区间生成针对 *fps* 统计数据
3. 从数值区间 *range*=[19.0, 41.8] 可以看出，有渲染帧的 *fps* 低于阈值 20，调用 *fps 20* 把 *fps* 值低于 20 的所有帧列出来，从搜索结果可以看出第 30193 帧的 *fps* 等于 19.0
4. 使用 *frame 30193* 查看目标帧的详细数据，可以发现当前帧有内存申请操作，并且 *InputManager.Update()* 函数占用了将近 25 毫秒的时间，其中 *LogStringToConsole*(打印日志) 和 *GameObject.Activate* 占用了大部分时间开销，可以查看更深层级进一步确定时间开销产生细节。

### 2.4.2 追踪动态内存分配

从上面的例子中可以发现动态内存申请会有比较大的时间开销，*alloc* 命令可以快速在指定帧区间进行堆栈搜索，并把所有申请了动态内存的帧数据列出来，方便进一步定位内存产生细节。

```

> alloc 0 100
[FRAME] index=30015 time=24.870ms fps=40.2 alloc=246 offset=394246304
[FRAME] index=30016 time=24.870ms fps=40.2 alloc=18 offset=394259841
[FRAME] index=30049 time=24.860ms fps=40.2 alloc=2444 offset=394702466
[FRAME] index=30052 time=24.860ms fps=40.2 alloc=1720 offset=394742917
[FRAME] index=30055 time=31.030ms fps=32.2 alloc=45581 offset=394783488
[FRAME] index=30056 time=24.960ms fps=40.0 alloc=18 offset=394799969
[FRAME] index=30057 time=24.890ms fps=40.2 alloc=64 offset=394813794
[FRAME] index=30058 time=24.870ms fps=40.2 alloc=10174 offset=394827619
[FRAME] index=30079 time=24.850ms fps=40.2 alloc=184 offset=395107672
[FRAME] index=30082 time=24.920ms fps=40.1 alloc=11128 offset=395147803
[FRAME] index=30095 time=24.890ms fps=40.2 alloc=246 offset=395323224
[FRAME] index=30096 time=24.840ms fps=40.3 alloc=18 offset=395336729
[FRAME] index=30098 time=24.880ms fps=40.2 alloc=2536 offset=395363667
/> frame 30055
[FRAME] index=30055 time=31.030ms fps=32.2 alloc=45581 offset=394783488
  Update.ScriptRunBehaviourUpdate time=50.113%/15.550ms self=0.019%/0.003ms calls=1 alloc=45581 *75
    BehaviourUpdate time=99.981%/15.547ms self=2.322%/0.361ms calls=1 alloc=45581 *76
      ApolloObjectManager.Update() time=79.347%/12.336ms self=55.674%/6.868ms calls=1 alloc=42871 *80
        LogStringToConsole time=17.145%/2.115ms self=68.652%/1.452ms calls=2 alloc=8700 *272
        StackTraceUtility.PostprocessStacktrace() time=16.879%/0.357ms self=97.199%/0.347ms calls=2 alloc=243
  *273
    GC.Alloc time=2.801%/0.010ms self=100.000%/0.010ms calls=20 alloc=2430 *274
    Application.CallLogCallback() time=12.293%/0.260ms self=88.462%/0.230ms calls=2 alloc=1960 *275
      GC.Alloc time=11.538%/0.030ms self=100.000%/0.030ms calls=50 alloc=1960 *274
      GC.Alloc time=2.175%/0.046ms self=100.000%/0.046ms calls=67 alloc=4310 *274
    NextViewPoolBase.SpawnName time=14.770%/1.822ms self=7.190%/0.131ms calls=4 alloc=1024 *599
    Instantiate time=73.161%/1.333ms self=1.050%/0.014ms calls=1 alloc=1024 *321
      Instantiate.Awake time=78.020%/1.040ms self=25.096%/0.261ms calls=1 alloc=724 *327
        Animator.Initialize time=42.115%/0.438ms self=30.822%/0.135ms calls=1 *518
          Animator.SetupControllerDataSet time=40.411%/0.177ms self=100.000%/0.177ms calls=1 *520
          Animator.SetupAvatarDataSet time=28.767%/0.126ms self=100.000%/0.126ms calls=1 *519
        SkinProxy.Awake() time=28.654%/0.298ms self=96.980%/0.289ms calls=1 alloc=616 *517
          GC.Alloc time=3.020%/0.009ms self=100.000%/0.009ms calls=18 alloc=616 *274

```

- 使用 `alloc 0 100` 列出当前帧区间子区间  $[0, 100)$  内有动态内存产生的渲染帧，可以看到 **30055** 帧产生了比较多的内存。
- 使用 `frame 30055` 查看目标帧的详细数据。

## 2.5 小结

UnityProfiler 以巧妙的方式编码性能数据，节省 80% 的存储空间，内置多种命令以超高效率在不同维度帮助大家发现性能问题，并快速定位到产生问题的帧，使用该工具可以有效地帮助大家解决卡顿问题，从而提升游戏质量。

# 3 MemoryCrawler

## 3.1 简介

MemoryCrawler 以 *Unity* 引擎生成的内存快照数据为基础，提供多种维度的工具来帮助发现内存问题。该工具前期预研阶段使用 *Python* 测试逻辑，最终使用 *C++* 实现。由于是基于 *Unity* 的原生接口获取数据，所以需要保证 *Profiler* 工具打开后能看到性能采集界面，真机调试确保按照官方文档正确配置。

内存分析是个高计算密度的过程，使用 *C#* 分析将是个漫长的过程，毕竟是在 *mono* 环境运行，MemoryCrawler 用 *C++* 作为开发语言，同时也获得了可观的分析速度，基本在百毫秒级别，也就是眨眼的功夫就完成数据分析，相比 *Unity* 提供 *MemoryProfiler* 在内存数据的解析速度、问题定位速度方面都是一个质的飞跃。

## 3.2 原理

在命名空间 `UnityEditor.MemoryProfiler` 有个类 `MemorySnapshot` 可以请求生成内存快照，一般情况下内存快照创建需要时间，所以需要侦听 `MemorySnapshot.OnSnapshotReceived` 事件，拿到内存数据就可以做相关的内存分析了。

```
MemorySnapshot.OnSnapshotReceived += OnSnapshotComplete;
MemorySnapshot.RequestNewSnapshot();

private static void OnSnapshotComplete(PackedMemorySnapshot snapshot)
{
    MemorySnapshot.OnSnapshotReceived -= OnSnapshotComplete;
    ExportMemorySnapshot(snapshot, false);
}
```

## 3.3 命令手册

### 3.3.1 read

**read** *uuid*

参数	可选	描述
<i>uuid</i>	是	每个内存快照都有一个唯一 16 字节 36 字符的 <i>uuid</i>

**read** 加载缓存在 *sqlite* 里面的内存快照分析结果，*uuid* 为原始内存快照的标识符，使用 **read** 的前提是存在相应的缓存文件，通过 **uuid** 命令可以查看快照的 *uuid*。如果 *uuid* 留空则，加载当前内存快照的缓存文件。**read** 加载完缓存后，会自动与当前内存快照做差异分析。

### 3.3.2 load

**load** [*pms\_filepath*]

参数	可选	描述
<i>pms_filepath</i>	否	内存快照路径

**load** 从原始内存快照文件加载内存数据并进行内存分析，其他功能与 **read** 相同，会自动与当前内存快照做差异分析。

### 3.3.3 track

**track** [*tracking\_mode*]

参数	可选	描述
<i>tracking_mode</i>	是	内存追踪模式 <i>alloc</i> = 内存增长追踪模式 <i>leak</i> = 内存泄漏追踪模式 ?= 查看当前模式

使用 **read** 或 **load** 加载完另外一个内存快照后，会自动与当前内存快照做内存差异分析，设置追踪模式可以方便地在差异内存里面定位内存问题。*tracking\_mode* 留空时则清除当前追踪模式。

```

/> read e4a5b509-f9cc-a84e-9f90-c502a54fe76e
[0] SnapshotCrawlerCache=19492135
[1] SnapshotCrawlerCache::read=19490006
[2] open=1400984
[3] read_PackedMemorySnapshot=9540012
[4] read_native_types=450704
[5] read_native_objects=2932726
[6] read_managed_types=6111619
[7] read_type_fields=802478
[8] read_vm=42409
[9] read_MemorySnapshotCrawler=8371204
[10] read_managed_objects=8370010

/> track alloc
ENTER TRACKING ALLOC MODE
/> track ?
ENTER TRACKING ALLOC MODE
/> track leak
ENTER TRACKING LEAK MODE
/> track
LEAVE TRACKING MODE

```

### 3.3.4 str

**str** [*address*]

参数	可选	描述
<i>address</i>	否	字符串对象的内存地址

```

/> str 3106572216
0xb92a87b8 130 '很遗憾，我们现在无法向您继续提供服务。我们的数据存储在 EEA 地区之外，为了为您提供服务我们的支
→ 持团队必须从其他司法管辖区访问数据。'

```

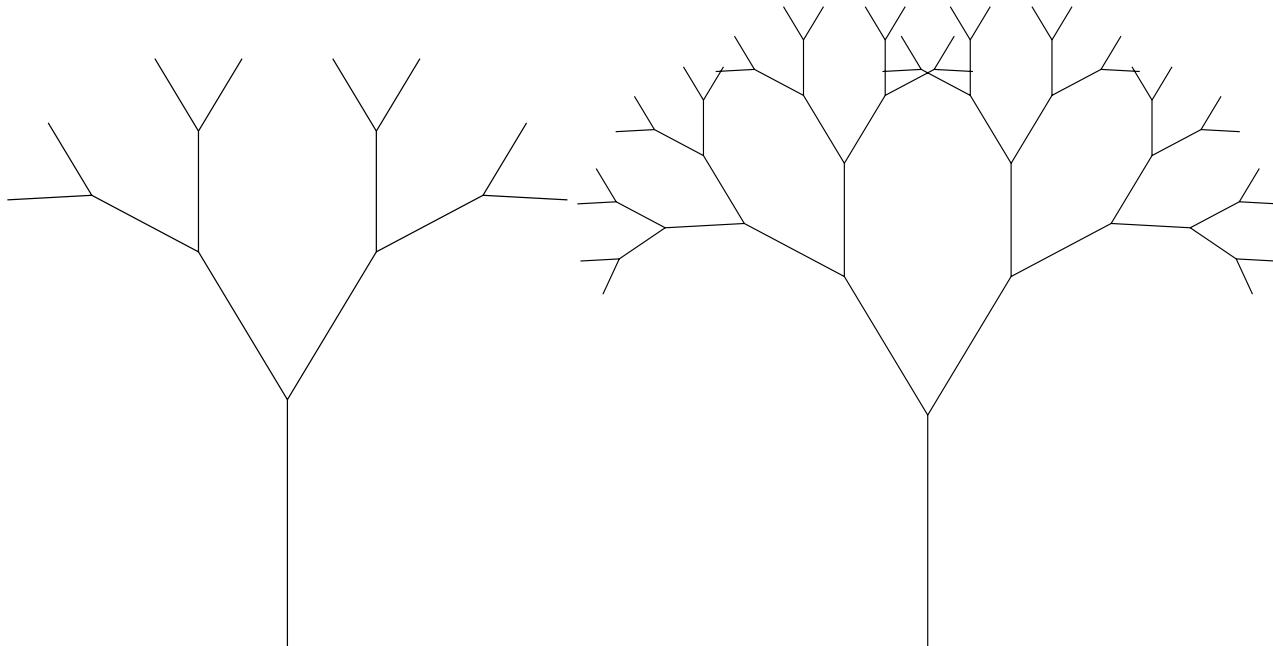
第一列参数为字符串地址的 16 进制形式，第二列为字符串占用内存大小，第三列为字符串内容。

### 3.3.5 ref

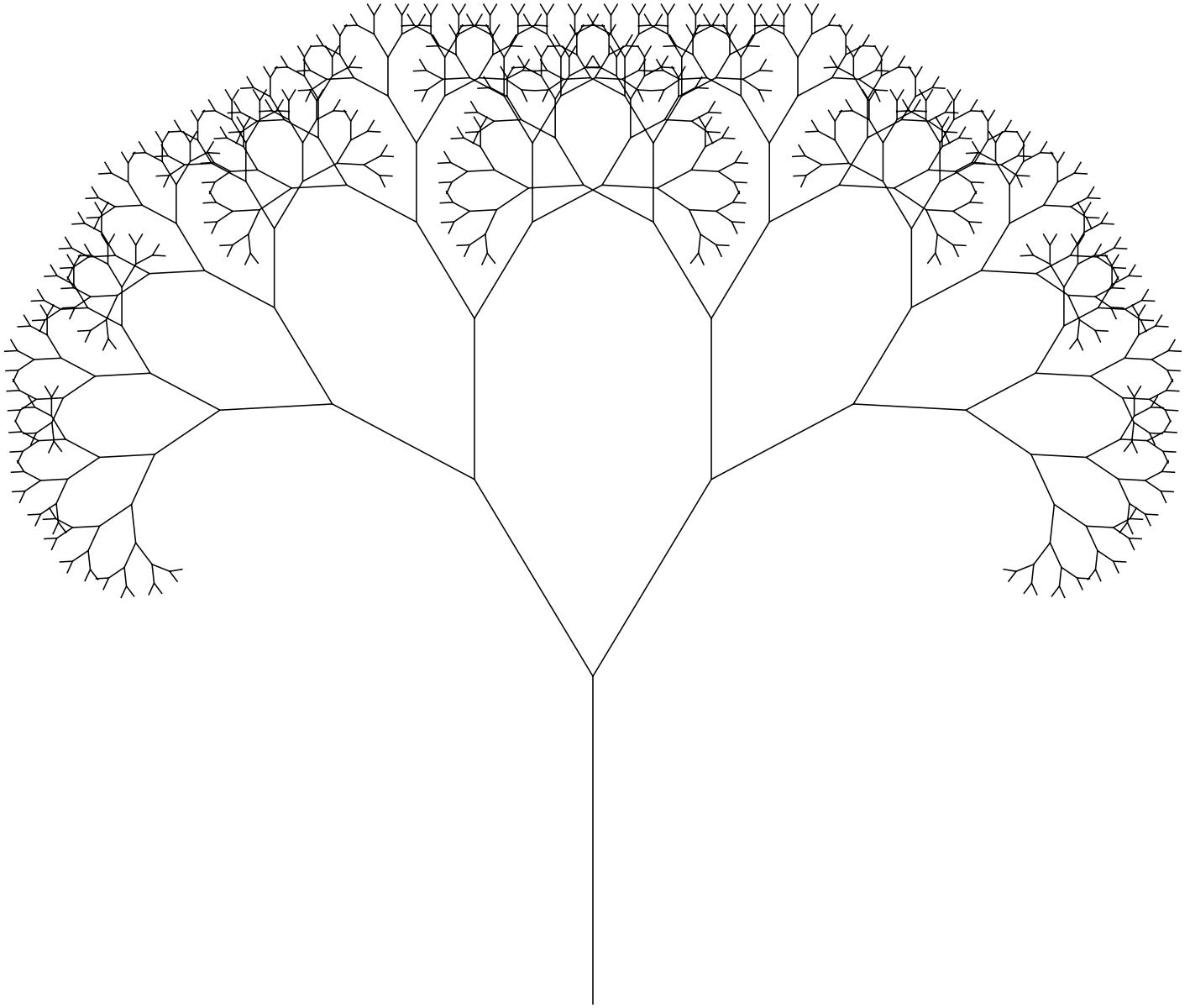
**ref** [*address*]

参数	可选	描述
<i>address</i>	否	对象的内存地址

查找保持当前对象在内存中活跃的关系链，使用递归遍历父级引用关系，但是每层递归最多产生两个分支，因为对象在内存中的引用关系有可能很复杂，导致递归深度过大而耗用过多电脑内存以及过多的关系链。通常我们理想的引用关系是下图这样的，深度有限，引用简单。



但实际上每一层递归节点都有可能产生很多递归分支，导致遍历节点异常庞大，所以 ref 做了对每层递归分支数量做了限制，这样哪怕对象引用关系很复杂的情况下也可以得到部分引用关系，下图只是每层递归 2 个分支一共 9 层递归的模型，实际的引用树状图可能无法用图来描绘出来。



ref 接受一个内存地址参数，可以自动识别八进制、十进制以及十六进制，`3106572216` 为上个例子中的字符串地址。

```
/> ref 3106572216
<GCHandle>::ApplicationTranslator 0xebb97d20
    .{database:translation_protocol.TranslationContainer} 0xc085f460
    ↳ .{_clases:System.Collections.Generic.List<translation_protocol.ClassContainer>}
    ↳ 0xe37d7f00
        .{_items:translation_protocol.ClassContainer}[8] 0xb9153018
    ↳ .{_fields:System.Collections.Generic.List<translation_protocol.LocalizedItem>}
    ↳ 0xb9153000
        .{_items:translation_protocol.LocalizedItem}[0] 0xb91c55e0
```

```

    .{_text:System.String} 0xb92a87b8
<Static>::dataconfig.DataConfigManager::{dataObserver:dataconfig.DataConfigObserver}
    ↳ 0xd361bf18
    .{prev:dataconfig.DataConfigObserver} 0xd361bee0
    .{m_target:ApplicationTranslator} 0xebb97d20
    .{database:translation_protocol.TranslationContainer} 0xc085f460

    ↳ .{_clases:System.Collections.Generic.List<translation_protocol.ClassContainer>}
    ↳ 0xe37d7f00
    .{_items:translation_protocol.ClassContainer}[8] 0xb9153018

    ↳ .{_fields:System.Collections.Generic.List<translation_protocol.LocalizedItem>}
    ↳ 0xb9153000
    .{_items:translation_protocol.LocalizedItem}[0] 0xb91c55e0
    .{_text:System.String} 0xb92a87b8
<Static>::ApplicationTranslator::{_shared:ApplicationTranslator} 0xebb97d20
    .{database:translation_protocol.TranslationContainer} 0xc085f460

    ↳ .{_clases:System.Collections.Generic.List<translation_protocol.ClassContainer>}
    ↳ 0xe37d7f00
    .{_items:translation_protocol.ClassContainer}[8] 0xb9153018

    ↳ .{_fields:System.Collections.Generic.List<translation_protocol.LocalizedItem>}
    ↳ 0xb9153000
    .{_items:translation_protocol.LocalizedItem}[0] 0xb91c55e0
    .{_text:System.String} 0xb92a87b8
<Static>::HardStrings.LaunchStart::{TIPS_REJECT_DATA_POLICY:System.String}
    ↳ 0xb92a87b8

```

在这个例子中，可以看出 3106572216 有四个引用关系，第一个引用被 *Unity* 的 *GCHandle*，其他三个分别被静态对象引用，引用的终点和当前对象之间为引用链经过的对象路径，每一个对象节点都有清晰的变量名以及对象地址。

### 3.3.6 REF

**REF** [*address*]

参数	可选	描述
<i>address</i>	否	对象的内存地址

同 ref 相同，REF 列举保持对象在内存中活跃的关系链，不同的是 REF 尝试遍历所有的引用关系，如果遇到引用关系复杂，递归深度比较大的情况下，会采用递归总量限制保证 REF 在合理的内存开销下列举尽可能多的关系链，对于没有递归到终点的关系链会用星号 \* 标记。对于复杂对象，可能会产生大量的引用链，会花费比较长的时间打印关系链结果。

引用标记	说明
<i>GCHandle</i>	表示对象最终被 <i>Unity</i> 的 <i>GCHandle</i> 管理器引用，如果这是对象仅有的引用， <i>Unity</i> 会在合适的时机自动释放
<i>Static</i>	表示对象被静态类引用
星号	表示递归深度过大，当前引用链被中断
$\infty$	表示对象被环式引用

### 3.3.7 uref

**uref** [*address*]

参数	可选	描述
<i>address</i>	否	对象的内存地址

列举 *Unity* 引擎创建的 *native* 对象引用关系链，与 ref 对应，使用有限递归分支列举部分引用关系链。

```
/> uref 3269982224
<SIS>. {Sprite:0xcd2a8ad0:'mu'} . {Texture2D:0xc2e7f810:'Country_RGB'}
<SIS>. {Sprite:0xcd2a7950:'ml'} . {Texture2D:0xc2e7f810:'Country_RGB'}
```

引用标记	说明
<i>SIS</i>	<i>Store In Scene</i>
<i>DDO</i>	<i>Dont't Destroy Object</i>
<i>UMO</i>	<i>Unity Manager Object</i>
星号	表示递归深度过大，当前引用链被中断
$\infty$	表示对象被环式引用

### 3.3.8 UREF

**UREF** [*address*]

参数	可选	描述
参数	可选	描述
<i>address</i>	否	对象的内存地址

同 REF 类似，但是遍历引擎对象的所有引用关系链。

```
/> UKREF 3269982224
<SIS>. {Sprite:0xcd2a8ad0:'mu'}. {Texture2D:0xc2e7f810:'Country_RGB'}
<SIS>. {Sprite:0xcd2a7950:'ml'}. {Texture2D:0xc2e7f810:'Country_RGB'}
<SIS>. {Sprite:0xcd373b50:'br'}. {Texture2D:0xc2e7f810:'Country_RGB'}
<SIS>. {Sprite:0xcd2c8fd0:'sa'}. {Texture2D:0xc2e7f810:'Country_RGB'}
<SIS>. {Sprite:0xcd3ebad0:'ao'}. {Texture2D:0xc2e7f810:'Country_RGB'}
<SIS>. {Sprite:0xcd2c94d0:'sc'}. {Texture2D:0xc2e7f810:'Country_RGB'}
<SIS>. {Sprite:0xcd2906d0:'gb'}. {Texture2D:0xc2e7f810:'Country_RGB'}
<SIS>. {Sprite:0xcd2c5650:'no'}. {Texture2D:0xc2e7f810:'Country_RGB'}
<SIS>. {Sprite:0xcd375590:'cg'}. {Texture2D:0xc2e7f810:'Country_RGB'}
<SIS>. {Sprite:0xcd2a7e50:'mn'}. {Texture2D:0xc2e7f810:'Country_RGB'}
<SIS>. {Sprite:0xcd29a0d0:'ki'}. {Texture2D:0xc2e7f810:'Country_RGB'}
<SIS>. {Sprite:0xcd2a8d50:'mv'}. {Texture2D:0xc2e7f810:'Country_RGB'}
<SIS>. {Sprite:0xcd299e50:'kh'}. {Texture2D:0xc2e7f810:'Country_RGB'}
<SIS>. {Sprite:0xcd2c6550:'pa'}. {Texture2D:0xc2e7f810:'Country_RGB'}
<SIS>. {Sprite:0xcd2dd2d0:'ws'}. {Texture2D:0xc2e7f810:'Country_RGB'}
<SIS>. {Sprite:0xcd377d90:'dm'}. {Texture2D:0xc2e7f810:'Country_RGB'}
<SIS>. {Sprite:0xcd2a7450:'mh'}. {Texture2D:0xc2e7f810:'Country_RGB'}
<SIS>. {Sprite:0xcd2c6cd0:'pg'}. {Texture2D:0xc2e7f810:'Country_RGB'}
<SIS>. {Sprite:0xcd3eb350:'ag'}. {Texture2D:0xc2e7f810:'Country_RGB'}
<SIS>. {Sprite:0xcd2d7ed0:'uz'}. {Texture2D:0xc2e7f810:'Country_RGB'}
<SIS>. {Sprite:0xcd2d4550:'sv'}. {Texture2D:0xc2e7f810:'Country_RGB'}
```

### 3.3.9 kref

**kref** [*address*]

参数	可选	描述
<i>address</i>	否	对象的内存地址

同 ref 功能相同，但是剔除 \* 和 ∞ 开头的关系链，在数据量比较大的情况下会比较有用。

### 3.3.10 KREF

**KREF** [*address*]

参数	可选	描述
<i>address</i>	否	对象的内存地址

同 REF 功能相同，但是剔除 \* 和 ∞ 开头的关系链，在数据量比较大的情况下会比较有用。

### 3.3.11 ukref

**ukref** [*address*]

参数	可选	描述
<i>address</i>	否	对象的内存地址

同 uref 功能相同，但是剔除 \* 和 ∞ 开头的关系链，在数据量比较大的情况下会比较有用。

### 3.3.12 UKREF

**UKREF** [*address*]

参数	可选	描述
<i>address</i>	否	对象的内存地址

同 UREF 功能相同，但是剔除 \* 和 ∞ 开头的关系链，在数据量比较大的情况下会比较有用。

### 3.3.13 link

**link** [*address*]

参数	可选	描述
<i>address</i>	否	对象的内存地址

对于继承于 *UnityEngine.Object* 类的对象，可以使用该命令查看对应的 native 引擎对象地址，可以方便在不同的内存空间进行审视对象内存。

```
/> link 3816674832
3269982224
/> uref 3269982224
<SIS>. {Sprite:0xcd2a8ad0:'mu'} . {Texture2D:0xc2e7f810:'Country_RGB'}
<SIS>. {Sprite:0xcd2a7950:'ml'} . {Texture2D:0xc2e7f810:'Country_RGB'}
```

### 3.3.14 ulink

**ulink** [*address*]

参数	可选	描述
<i>address</i>	否	对象的内存地址

使用该命令查看 *native* 引擎对象对应的 *il2cpp* 对象，可以方便在不同的内存空间进行审视对象内存。

```
/> ulink 3269982224
3816674832
/> ref 3816674832
<GCHandle>::UnityEngine.Texture2D 0xe37dd610
```

### 3.3.15 show

**show** [*address*]

参数	可选	描述
<i>address</i>	否	对象的内存地址

```
/> show 3495241968
System.String 0xd05528f0
└─length:System.Int32 = 42
└─start_char:System.Char = \u5411
/> show 0xb8fb00e0
translation_protocol.LocalizedItem 0xb8fb00e0
└─_sid:System.String = NULL
└─_uid:System.Nullable<System.UInt32>
  └─value:System.UInt32 = 200160
  └─has_value:System.Boolean = true
└─_text:System.String 0xd05528f0 = '向四周瞬间投出大量的魔刃，对周围半径 3 格的敌人造成
  ↵ 175/225/275 点魔法伤害。'
└─extensionObject:ProtoBuf.IExtension = NULL
```

### 3.3.16 ushow

**ushow** [*address*]

参数	可选	描述
<i>address</i>	否	对象的内存地址

该命令用来查看当前 *native* 引擎对象内部保持的引用关系链。

```
/> ushow 0xbcbff110
'LanguageSelect':GameObject 0xbcbff110
└─'NameTxt2':GameObject 0xb9b8a450=144
  └─'LocalIEFlag':MonoScript 0xc2fa3610=244
```

### 3.3.17 find

**find** [*address*]

参数	可选	描述
<i>address</i>	否	对象的内存地址

find 查看 *il2cpp* 对象并展示相关信息。

```
/> find 0xb8fe1ec0
0xb8fe1ec0 type='translation_protocol.LocalizedItem'1614 size=28
↳ assembly='ProtobufProtocol'
```

对象类型名之后的数字 **1614** 为对象类型引用，可以使用 type 命令查看类型信息。

### 3.3.18 ufind

**ufind** [*address*]

参数	可选	描述
<i>address</i>	否	对象的内存地址

ufind 查看 *native* 引擎对象并展示相关信息。

```
/> ufind 3266512656
0xc2b30710 name='INetworkService' type='MonoScript'158 size=244
```

对象类型名之后的数字 **158** 为对象类型引用，可以使用 utype 命令查看类型信息。

### 3.3.19 type

**type** [*type\_ref*]

参数	可选	描述
<i>type_ref</i>	否	<i>il2cpp</i> 类型引用

type 查看 *il2cpp* 类型信息，通过 find、stat、bar 可以得到类型引用。

```
/> type 1614
0xbba39b40 name='translation_protocol.LocalizedItem'1614 size=28
↳ baseOrElementType='System.Object'0 assembly='ProtobufProtocol'
↳ instanceMemory=329420 instanceCount=11765
  isStatic=false name='_sid' offset=8 typeIndex=23
  isStatic=false name='_uid' offset=12 typeIndex=3520
```

```
isStatic=false name='_text' offset=20 typeIndex=23
isStatic=false name='extensionObject' offset=24 typeIndex=617
```

### 3.3.20 utype

**utype** [*type\_ref*]

参数	可选	描述
<i>type_ref</i>	否	<i>native</i> 引擎类型引用

utype 查看 *native* 引擎类型信息，通过 ufind、ustat、ubar 可以得到类型引用。

```
/> utype 158
name='MonoScript'158 nativeBaseType='TextAsset'156 instanceMemory=812217
↳ instanceCount=3087
```

### 3.3.21 stat

**stat** [*rank*]/[=5]

参数	可选	描述
<i>rank</i>	是	限定 <i>il2cpp</i> 类型实例显示数量

stat 按照 *il2cpp* 类型为分组，列举每个类型内存占用前 *rank* 名的实例对象信息，按照类型总内存从小到大排序。

```
/> stat

[System.Byte] memory=2 type_index=15
0x00000000      1 System.Byte
0x00000010      1 System.Byte

[System.Char] memory=4 type_index=22
0x00000000      2 System.Char
0x00000008      2 System.Char

[DG.Tweening.LogBehaviour] memory=4 type_index=1195
0x00000008      4 DG.Tweening.LogBehaviour

[LogSeverity] memory=4 type_index=2103
0x0000002c      4 LogSeverity

[TheNextMoba.Module.Arena.ArenaPlayMode] memory=4 type_index=2199
0x00000004      4 TheNextMoba.Module.Arena.ArenaPlayMode
```

### 3.3.22 ustat

**ustat** [*rank*] [=5]

参数	可选	描述
<i>rank</i>	是	限定 native 引擎类型实例显示数量

ustat 按照 native 引擎类型为分组，列举每个类型内存占用前 *rank* 名的实例对象信息，按照类型总内存从小到大排序。

```
/> ustat
```

```
[Texture2DArray] memory=4 type_index=170
0xc8384a50      4 'UnityDefault2DArray'

[NavMeshSettings] memory=48 type_index=200
0xc9920c10      48 'NavMeshSettings'

[GUILayer] memory=52 type_index=42
0xcdabd5060      52 'UICamera'

[DelayedCallManager] memory=76 type_index=179
0xcdab4ea0      76 'DelayedCallManager'

[MeshFilter] memory=104 type_index=94
0xc4abe8c0      52 'glow'
0xbb61b9e0      52 'glow'
```

### 3.3.23 list

**list** [*type\_ref*]

参数	可选	描述
<i>type_ref</i>	否	<i>il2cpp</i> 类型引用

list 列举 *type\_ref* 指定 *il2cpp* 类型的所有实例对象信息，该命令的结果受 *track* 设置影响。

```
/> list 2904
[dataconfig.BAG_ITEM_CONF[]][=] memory=288
0xebb91360      16 dataconfig.BAG_ITEM_CONF[]
0xebb8d260      272 dataconfig.BAG_ITEM_CONF[]
[SUMMARY] count=2 memory=288
```

### 3.3.24 ulist

**ulist** [*type\_ref*]

参数	可选	描述
<i>type_ref</i>	否	排行榜数量

ulist 列举 *type\_ref* 指定 native 引擎类型的所有实例对象信息，该命令的结果受 *track* 设置影响。

```
/> ulist 156
[TextAsset] [=] memory=1090701
0xce55af30      113 'dataconfig_mode_sub_type_conf'
0xce559b80      519 'dataconfig_msg_language_conf'
0xce55ac20      103555 'ja_JP_language'
0xce55aa60      125146 'en_US_language'
0xce5598e0      128359 'ru_RU_language'
0xce557ce0      140676 'zh_Hant_TW_language'
0xb1584010      592333 'zh_Hans_CN_language'
[SUMMARY] count=7 memory=1090701
```

### 3.3.25 bar

**bar** [*rank*]

参数	可选	描述
<i>rank</i>	否	排行榜数量

bar 按照 *il2cpp* 类型进行内存统计，并打印前 *rank* 的类型内存分配信息，该命令的结果受 *track* 设置影响。

```
/> bar 5
21.73 21.73 ██████████ System.String 1523764 #17554 *23
16.35 38.08 ██████████ UnityEngine.UIVertex[] 1146656 #473 *2742
16.24 54.32 ██████████ UnityEngine.UIVertex 1139164 #14989 *537
5.95 60.26 ██████ UnityEngine.Vector3 416952 #34746 *442
4.70 64.96 ██████ translation_protocol.LocalizedItem 329420 #11765 *1614
```

第一列为当前类型占用总内存的百分比，第二列为排行榜累积百分比，第三列为类型名，第四列为类型占用内存字节数，以 # 开头的第五列为当前类型的实例数量，以星号 \* 开头的数字为类型引用。

### 3.3.26 ubar

**ubar** [*rank*]

参数	可选	描述
<i>rank</i>	否	排行榜数量

ubar 按照 native 引擎类型进行内存统计，并打印前 *rank* 的类型内存分配信息，该命令的结果受 *track* 设置影响。

```
/> ubar 5
41.18 41.18 [REDACTED] Font 16161642 #9 *132
34.08 75.26 [REDACTED] Texture2D 13373919 #133 *168
11.94 87.20 [REDACTED] ResourceManager 4683935 #1 *191
2.78 89.97 [REDACTED] TextAsset 1090701 #7 *156
2.56 92.54 [REDACTED] MonoBehaviour 1005039 #2102 *63
```

第一列为当前类型占用总内存的百分比，第二列为排行榜累积百分比，第三列为类型名，第四列为类型占用内存字节数，以 # 开头的第五列为当前类型的实例数量，以星号 \* 开头的数字为类型引用。

### 3.3.27 heap

显示当前堆内存信息。

```
/> heap 20
[SUMMARY] blocks=11 memory=13598720
27.80 27.80 3780608 3692K [REDACTED] 3780608 3692K #1
26.33 54.13 3579904 3496K [REDACTED] 3579904 3496K #1
19.76 73.89 2686976 2624K [REDACTED] 2686976 2624K #1
6.90 80.78 937984 916K [REDACTED] 937984 916K #1
5.78 86.57 262144 256K [REDACTED] 786432 768K #3
5.27 91.84 716800 700K [REDACTED] 716800 700K #1
3.49 95.33 475136 464K [REDACTED] 475136 464K #1
2.65 97.98 360448 352K [REDACTED] 360448 352K #1
2.02 100.00 274432 268K [REDACTED] 274432 268K #1
```

### 3.3.28 save

把当前内存快照的分析结果保存为 *sqlite* 格式。

```
/> save
[0] SnapshotCrawlerCache=222266990
[1] SnapshotCrawlerCache::save=222263272
[2] open=6161533
[3] create_native_types=1428120
[4] create_native_objects=1240330
[5] create_managed_types=851252
[6] create_type_fields=1380078
[7] create_objects=832663
[8] insert_native_types=2176337
[9] insert_native_objects=12473052
[10] insert_managed_types=18660698
[11] remove_redundants=35515270
[12] insert_objects=114539829
[13] insert_vm=1615608
[14] insert_strings=24375049
```

### 3.3.29 **uuid**

显示当前内存快照的唯一标识符。

```
/> uuid  
4da88f70-5539-a848-afae-be6c93fd7f4
```

### 3.3.30 **help**

显示帮助。

```
/> help  
read [UUID]* 读取以sqlite3保存的内存快照缓存  
load [PMS_FILE_PATH]* 加载内存快照文件  
track [alloc|leak] 追踪内存增长以及泄露问题  
str [ADDRESS]* 解析地址对应的字符串内容  
ref [ADDRESS]* 列举保持IL2CPP对象内存活跃的引用关系  
uref [ADDRESS]* 列举保持引擎对象内存活跃的引用关系  
REF [ADDRESS]* 列举保持IL2CPP对象内存活跃的全量引用关系  
UREF [ADDRESS]* 列举保持引擎对象内存活跃的全量引用关系  
kref [ADDRESS]* 列举保持IL2CPP对象内存活跃的引用关系并剔除干扰项  
ukref [ADDRESS]* 列举保持引擎对象内存活跃的引用关系并剔除干扰项  
KREF [ADDRESS]* 列举保持IL2CPP对象内存活跃的全量引用关系并剔除干扰项  
UKREF [ADDRESS]* 列举保持引擎对象内存活跃的全量引用关系并剔除干扰项  
link [ADDRESS]* 查看与IL2CPP对象链接的引擎对象  
ulink [ADDRESS]* 查看与引擎对象链接的IL2CPP对象  
show [ADDRESS]* 查看IL2CPP对象内存分布以及变量值  
ushow [ADDRESS]* 查看引擎对象内部的引用关系  
find [ADDRESS]* 查找IL2CPP对象  
ufind [ADDRESS]* 查找引擎对象  
type [TYPE_INDEX]* 查看IL2CPP类型信息  
utype [TYPE_INDEX]* 查看引擎类型信息  
stat [RANK] 按类型输出IL2CPP对象内存占用前RANK名的简报[支持内存追踪过滤]  
ustat [RANK] 按类型输出引擎对象内存占用前RANK名的简报[支持内存追踪过滤]  
bar [RANK] 输出IL2CPP类型内存占用前RANK名图形简报[支持内存追踪过滤]  
ubar [RANK] 输出引擎类型内存占用前RANK名图形简报[支持内存追踪过滤]  
list 列举IL2CPP类型所有活跃对象内存占用简报[支持内存追踪过滤]  
ulist 列举引擎类型所有活跃对象内存占用简报[支持内存追踪过滤]  
heap [RANK] 输出动态内存简报  
save 把当前内存快照分析结果以sqlite3格式保存到本机  
uuid 查看内存快照UUID  
help 帮助  
quit 退出
```

### 3.3.31 **quit**

退出当前内存快照分析。

## 3.4 使用案例

### 3.4.1 追踪内存增长

### 3.4.2 追踪内存泄漏

### 3.4.3 优化 Mono 内存

## 3.5 小结