https://www.cnblogs.com/chenyishi/p/11718589.html

写个demo来玩一玩linux平台下使用lldb加载sos来调试netcore应用。  
当然，在真实的产线环境中需要分析的数据和难度远远高于demo所示，所以demo的作用也仅仅只能起到介绍工具的作用。  
通常正常情况下，分析个几天才能得出一个结论的的结果都还是比较令人开心的！，很多时候分析来分析去也搞不出个所以然，也是很正常的（当然，也是自己学艺不精(^\_^)）  
在linux平台下的sos调试远没有在windows下面用windbg来得舒服，该有的命令很多都没有。  
微软爸爸还要加油努力啊！如果能做到linux下的dmp能在windows下面用windbg之类的工具那就爽翻了,哈哈，当然不可能，臆想一下下拉。

lldb工具的安装，linux下netcore如何生成dump文件，查看下文  
[centos7使用lldb调试netcore应用转储dump文件](https://www.cnblogs.com/calvinK/p/9263696.html)

### **图片有点多，文章有点长,来一个大纲先**

* 准备DEMO程序的代码
* 生成待调试分析的dump文件
* 目前linux下sos支持的命令
* 模拟分析内存泄漏
* 内存泄漏调试分析结论
* 内存泄漏分析疑问一
* 内存泄漏分析疑问二
* 死循环调试分析
* 内存泄漏调试分析结论

### **准备DEMO程序的代码**

废话不多说，先上demo程序代码。代码超级简单，模拟内存泄漏就简单的往一个静态list里面每次插入1M的byte[]；死循环则就是一个while(true)；  
PS：话说markdown插入代码能不能有收起，展开功能呢。那就爽歪歪拉 @dudu

[IMG_256](https://www.cnblogs.com/chenyishi/p/javascript:void(0);)

namespace linxu\_dump\_lldb.Controllers

{

class env

{

public static bool cpu\_flag;

public static bool setcpu\_flag(bool flag) => cpu\_flag = flag;

public static bool getcpu\_flag() => cpu\_flag;

public static List<byte[]> memory = new List<byte[]>();

}

[Route("api/[controller]/[action]")]

[ApiController]

public class ValuesController : ControllerBase

{

public string index() =>(GC.GetTotalMemory(false) / 1024.0 / 1024).ToString("0.00M");

[HttpGet]

public void begin\_cpu()

{

env.setcpu\_flag(true);

Task.Run(() => {while (env.getcpu\_flag()){}});

}

[HttpGet]

public void begin\_memory()

{

var size\_1m = 1 \* 1024 \* 1024;

for (int i = 0; i < 100; i++) env.memory.Add(new byte[size\_1m]);

}

[HttpGet]

public void end\_cpu() => env.setcpu\_flag(false);

[HttpGet]

public void end\_memory()

{

env.memory.Clear();

GC.Collect();

}}}

[IMG_257](https://www.cnblogs.com/chenyishi/p/javascript:void(0);)

### **生成待调试分析的dump文件**

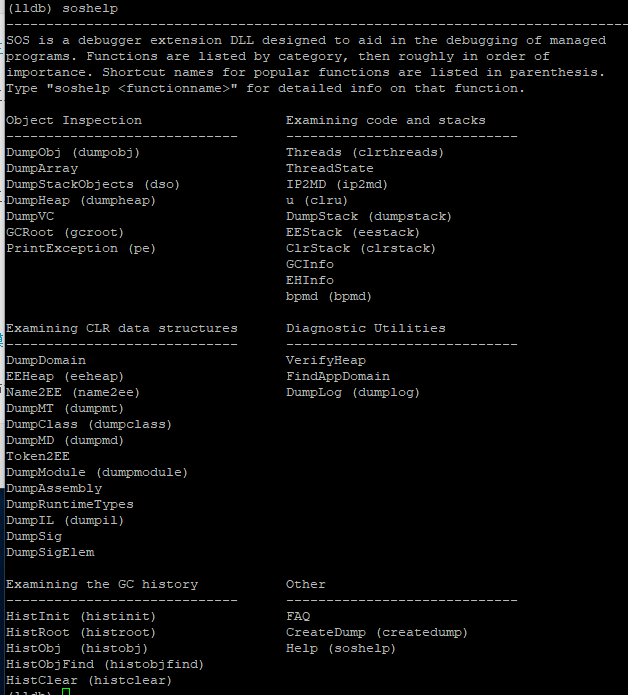
生成模拟内存泄漏的dump

请求接口begin\_memory来个几次后，然后通过createdump工具生成dump包，执行了4-5次begin\_memory，也就是加了大约400-500M的byte[]放到静态变量中

生成死循环的dump包

请求接口begin\_cpu开始异步任务进入死循环，然后通过createdump工具生成dump包

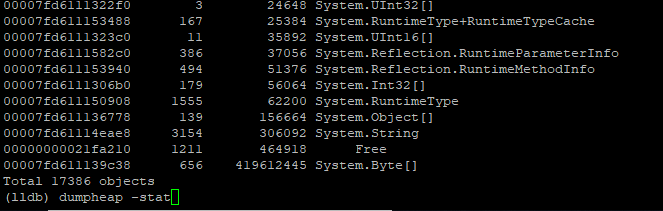
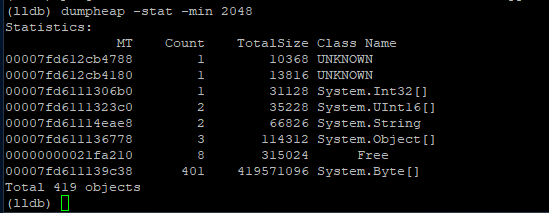
### **目前linux下sos支持的命令**

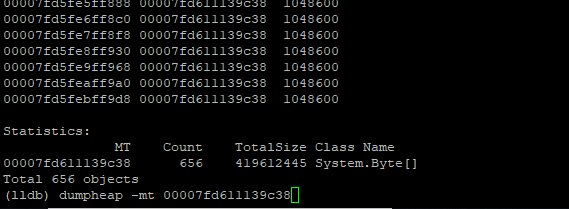
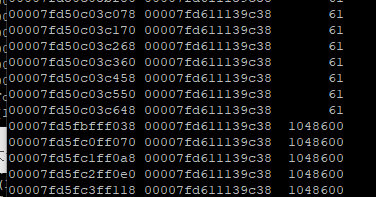
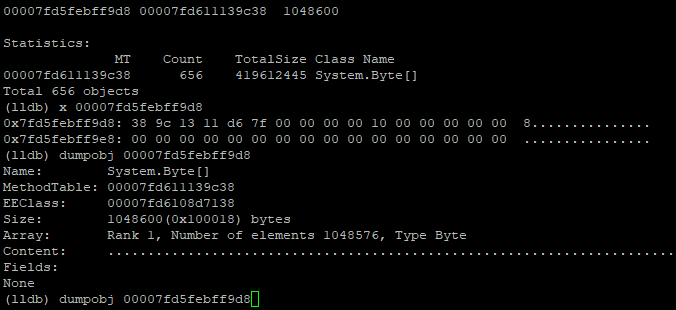
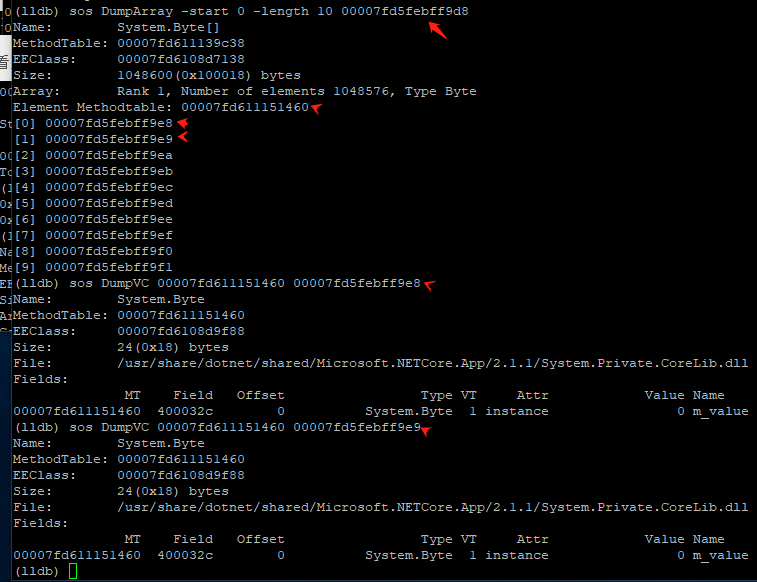
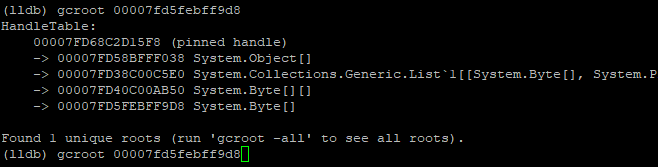
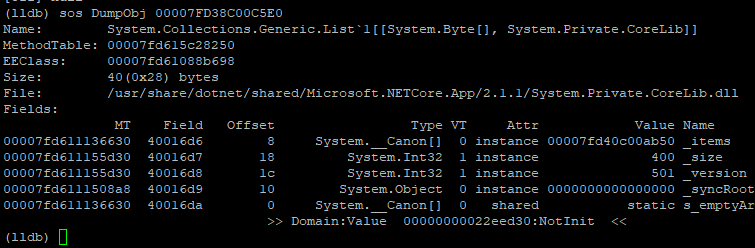
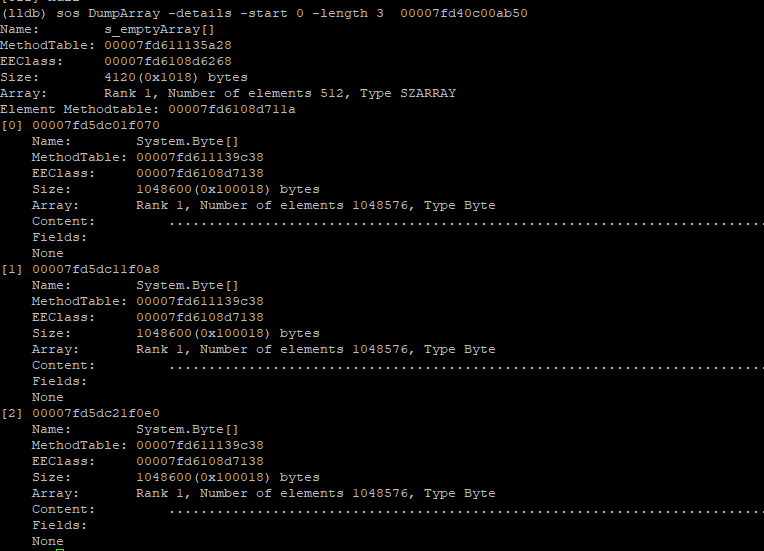
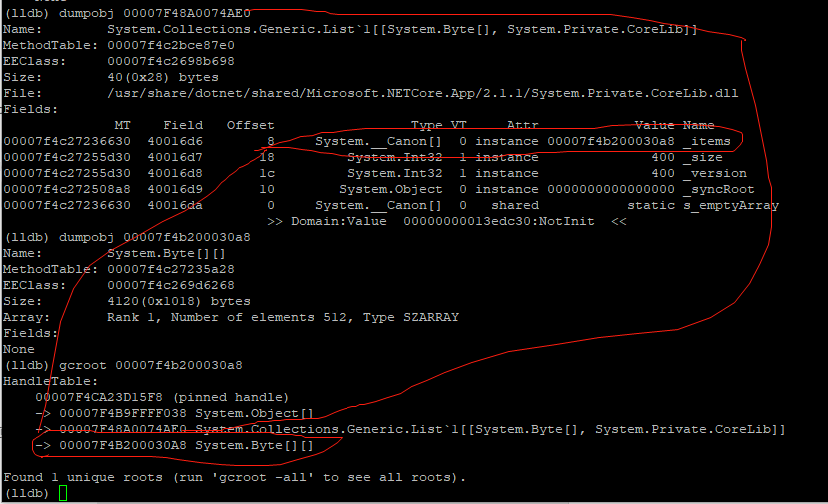
当前dotnet版本2.1.1。如下图所示支持，sos支持的命令，缺少几个比较有用的命令：ProcInfo ，ObjSize ，SyncBlk，其他缺少的赶脚也用不太上。最最重要的是gdb，lldb的调试命令不熟悉，或者说找不到windbg所对应命令还是蛮难受的，需要进一步认真学习才行...  


### **模拟分析内存泄漏**

命令走一个，进入lldb。

/usr/local/llvm-3.9.0/bin/lldb dotnet -c /opt/dump\_file/memory\_dump -o "plugin load /usr/share/dotnet/shared/Microsoft.NETCore.App/2.1.1/libsosplugin.so"

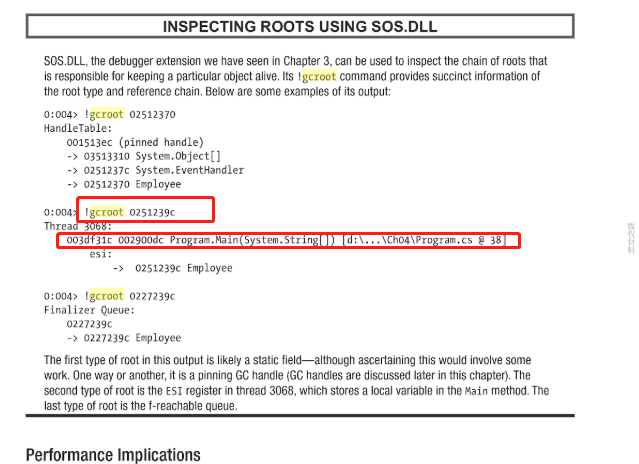
dumpheap -stat 分析先走一波。对堆上面的对象进行统计  
  
大于2kb的对象看一看  


图上反馈byte[]数组对象占的内存最大，而且是远超其他类型的，因此可以判定应该是byte[]在代码的某个地方没有释放。进去跟进去即可。  
真实情况项目情况很可能是占用内存最大，对象最多的string对象。分析起来真的有时候看运气，凭经验！...(^\_^)  
dumpheap -mt addr(byte[]数组的MT地址) 过滤看看类型是byte[]的都有那些对象。  
  
  
看上去特征特别明显，全是大小为1048600的bte[]对象。接下来随便找一个看看具体对象的数据是什么  
dumpobj addr(对象地址)；查看对象的基本结构  
  
内存数据看上去全是 00 00 00。可以说是一个默认的byte[]对象。可以在进入查看一下  
sos DumpArray -start 0 -length 10 00007fd5febff9d8（对象地址）  
查看数据对象，上一张图上我们能看到数组的lenght有1048576个，所以加上-start，-length参数，只查看最前面10个对象。不然刷屏得刷死咯。  
在接着使用  
sos DumpVC（查看值类型命令） 00007fd611151460（数组元素类型的mt地址） 00007fd5febff9e9（数组元素对象的地址）  
a 如下图所示，每个数组元素的类型都是byte，他们的value都是0；  
  
接下来，我们在看看这些个对象的gcroot对象是谁，也就是说这些个对象到底由谁持有  
gcroot addr（对象地址）  
  
在挨个看一看，能发现我们的这个list对象lenth有400个，\_version=501;这是因为我clear过一次，所以。clear+1，add([100])个数组，所以400+100+1=501；  
如果这是时候有一个objsize命令可以使用，我们就能计算出来这个list是一个400M的丑陋大对象。可惜linux下面木有。  
  
那就只能用查看数据的方法看看这个数组的具体详情拉。  
sos DumpArray -details(可以把每个对象的基本结构都打印出来)，能看到他的每一个元素都有1M（size:1048600(0x100018) bytes）大小  
  


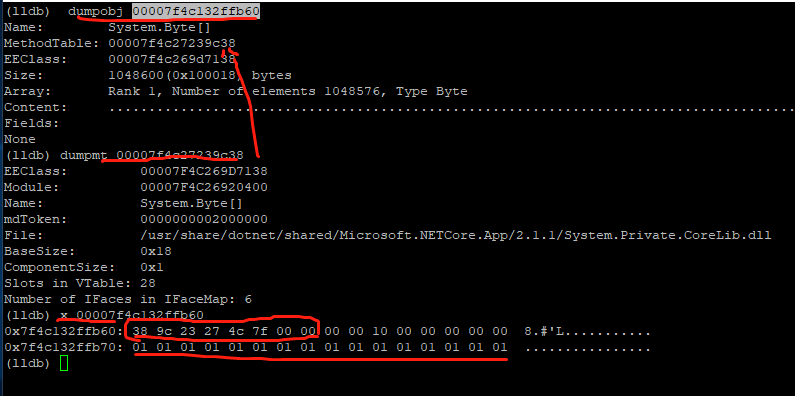
### **内存泄漏调试分析结论**

上图种gcroot有3个结果。  
第一个，用DumpArray查看后发现，应该是一个系统的静态对象，里面存储都是context之类的东西。  
第二个，就是我们的问题list对象。即List<byte[]>  
第三个，是第二个list对象的items。  
所以问题就出在我们这个静态的 list对象上了，那从代码上搜索一下就比较容易发现我们的List<byte[]>在哪里了。

### **疑问一**

  
上图种是书籍Pro .Net Performance: Optimize Your C# Applications第98页的一个列子，可惜没有搞懂他的这个地址怎么出来的，能直接拉出来堆栈信息...

### **疑问二**

按理来说1M应该等于1048576，那为什么这里显示是1048600呢，多余的24byte是啥玩意呢？  
dumpobj查看byte[]对象信息  
dumpmt查看byte[]类型的mt信息  
x addr（对象地址，x命令是lldb的命令，用户查看地址处的内存数据。可以使用 -c 24指定需要查看多少位数据）  
  
x addr 前16位数据小红框标记，最后8位小红框标记。中间的则是1M的01。01:byte数据，代码直接赋值。

[IMG_271](https://www.cnblogs.com/chenyishi/p/javascript:void(0);)

for (int i = 0; i < 100; i++)

{

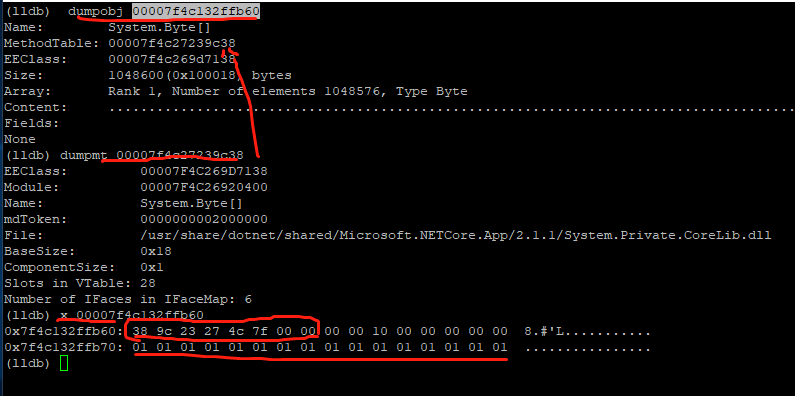
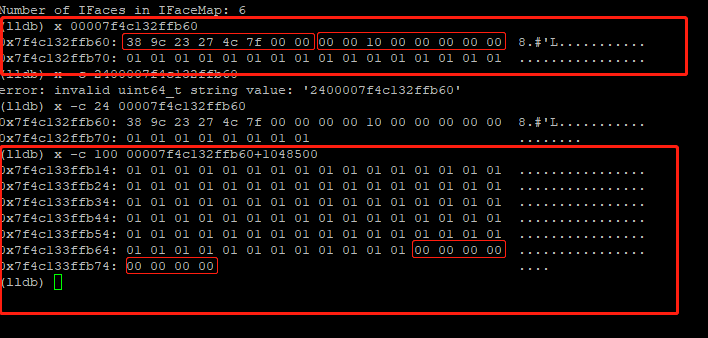
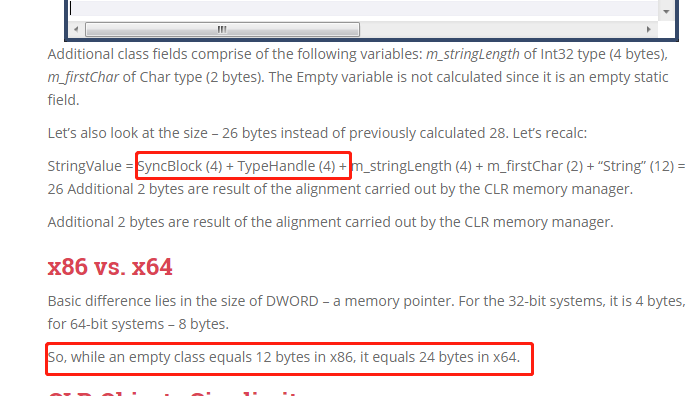
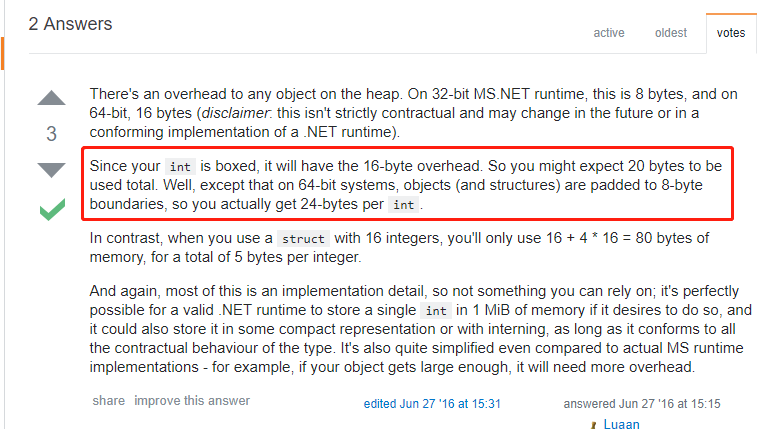
var x = new byte[size\_1m];

for (int j = 0; j < x.Length; j++) x[j] = 1;

env.memory.Add(x);

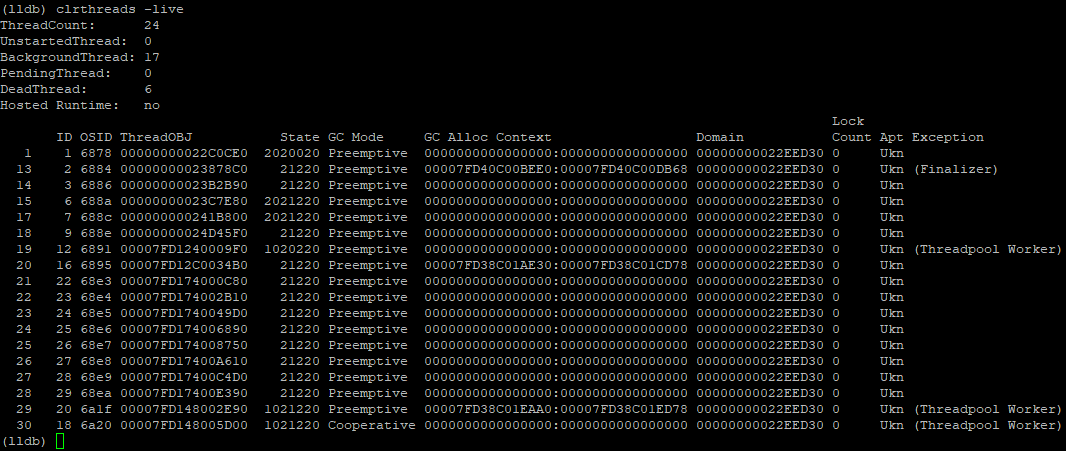
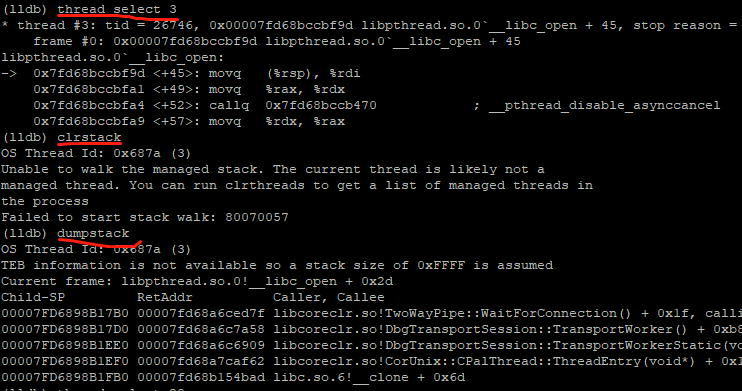
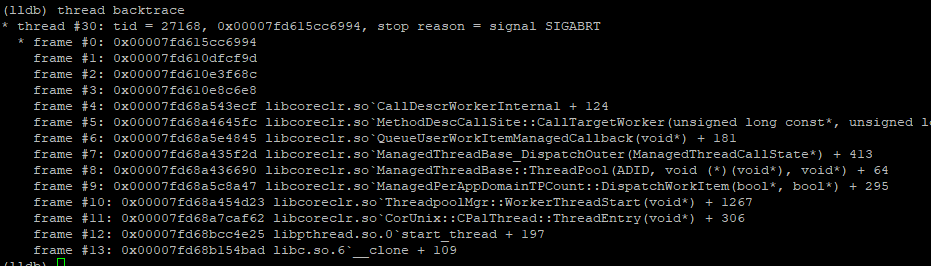
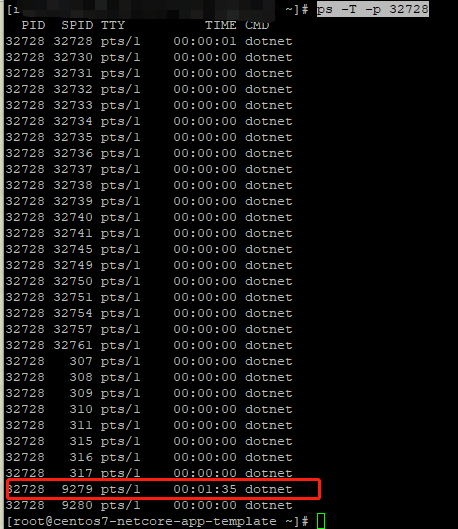
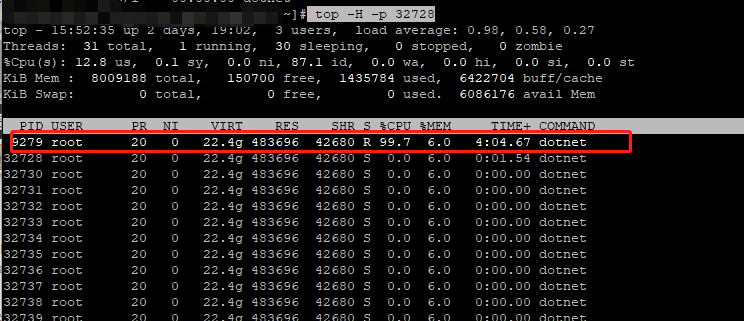
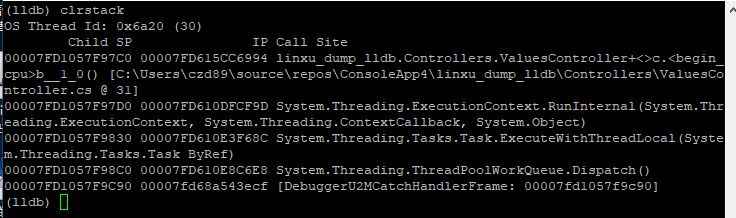
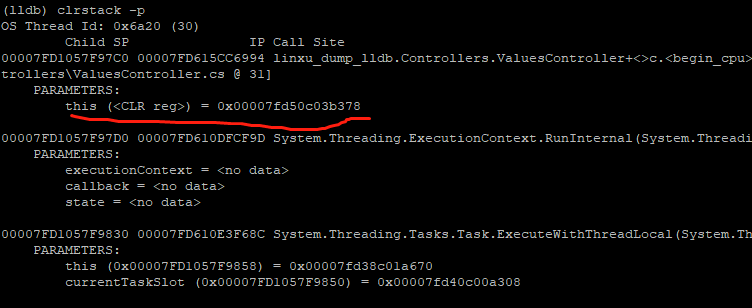
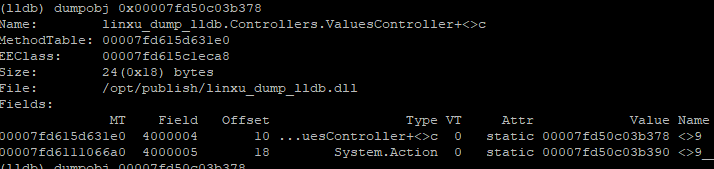
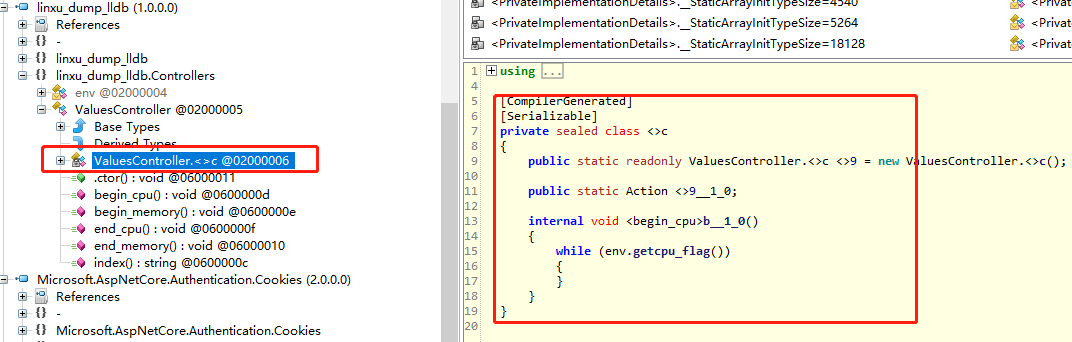
}

[IMG_272](https://www.cnblogs.com/chenyishi/p/javascript:void(0);)

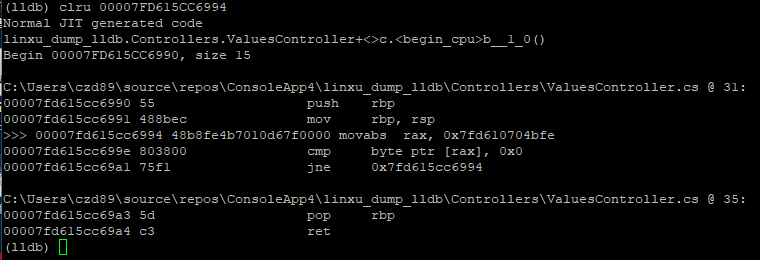
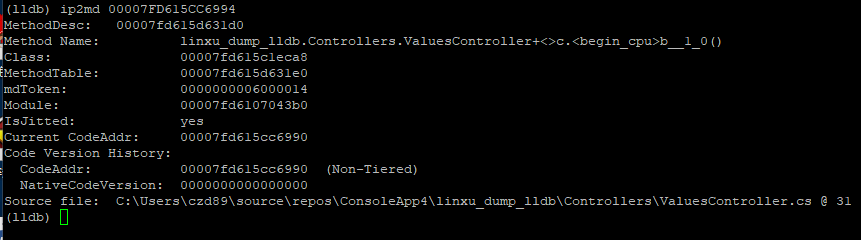
  
  
但是这24位数据内存结构为何这么组织，以及具体的含义就不是特别清楚了，有待考证！！！  
学艺不精！，准备回家看看C#本质论有没有说到这部分内容...或者哪位大哥可以说清楚一下，不胜感激！！！  
google搜索的时候发现 [Pro .Net Performance: Optimize Your C# Applications](https://item.jd.com/1135750606.html)，这本书很屌啊！！！，绝壁值得一看，就是英文不行，求中文版啊!!!，好想吐槽一下国内的垃圾编辑或作者，好的书一本都不翻译，垃圾玩意全翻译过来。  
http://codingsight.com/precise-computation-of-clr-object-size/  
  
https://stackoverflow.com/questions/38056513/why-does-windbg-show-system-int32-variables-as-24-bytes  


### **死循环调试分析**

clrthreads -live 先看看还在运行的线程有那些。然后通过thread select 线程编号（lldb命令）。来切换到当前线程。线程编号不是列表种的id字段，而是最前面一行的id。lldb 可以通过thread list命令来列举所有线程。

  
剩下的工作就是体力活动拉，一个一个看，一个一个分析。  
比如，我们切换到线程3看一看他当前的堆栈信息  
clrstack命令可以查看当前线程在托管代码种的堆栈信息。  
dumstack则可以看到非托管代码种的堆栈信息  
thread backtrace lldb查看堆栈信息的命令。  
  
  
线程3，能看到当前栈在非托管代码中（libcoreclr.so!TwoWayPipe::WaitForConnection），看方法名字也能猜到干嘛的，不太像我们的目标。  
另外，linux下面  
ps -T -p 32728 命令可以查看到进行下线程的基本情况  
top -H -p 32728 更happy。  
所以在排查高cpu问题的时候能提供许多便利性，反而比内存问题要来得方便很多。（图中的pid等数据不是一致性的。因为在写blog的时候图片是多次截取的。）  
  
  
所以在dump包的时候可以记录下来高cpu的线程id，然后通过thread select 找到对应的线程编号。在然后直接切换过去看一看就完事拉。  
所以 thread select 30  
clrstack看一看，嗯！当前线程在 linxu\_dump\_lldb.Controllers.ValuesController+<>c.<begin\_cpu>b\_\_1\_0() [C:\Users\czd89\source\repos\ConsoleApp4\linxu\_dump\_lldb\Controllers\ValuesController.cs @ 31]。  
  
看一看当前栈上面都有一些上面参数  
CLRStack [-a] [-l] [-p]；-p:看参数，-l：看局部变量，-a：=-l+-p；  
  
  
当然，我们的代码是异步的，也没有捕获任何action里面的变量，所以这里的这个参数，以及参数里面的属性啥都没有。  
从dll反编译代码也能和我们lldb看到的东西一一对以上。  


### **内存泄漏调试分析结论**

到这里，问题就很明显能看出来了，当然主要还是我们的DEMO是最简单的。还是开篇说过的那句话：通常正常情况下，分析个几天才能得出一个结论的的结果都还是比较令人开心的！，很多时候分析来分析去也搞不出个所以然，也是很正常的（当然，也是自己学艺不精(^\_^)，当自勉！）  
还能看一看具体方法的汇编代码等信息。  
  


参考资料：  
https://docs.microsoft.com/en-us/dotnet/framework/tools/sos-dll-sos-debugging-extension  
https://github.com/dotnet/coreclr/blob/master/Documentation/building/debugging-instructions.md  
https://lldb.llvm.org/tutorial.html  
https://stackoverflow.com/questions/38056513/why-does-windbg-show-system-int32-variables-as-24-bytes  
http://codingsight.com/precise-computation-of-clr-object-size/  
https://zhuanlan.zhihu.com/p/20838172  
https://blog.csdn.net/inuyashaw/article/details/55095545