国际游戏开发者大会

International Game Developers Conference

《天涯明月刀》内存的优化和调试

何 欢

个人简介

时间 & 单位	参与项目	工作内容
2004 – 2010 **育碧电脑**	幽灵战士（2,3） 细胞分裂（4） 雷曼疯狂兔子 终结战争 I Am Alive	菜单、动画、 逻辑、AI、算法
2010 – 2012 **扬讯科技**	手机相关	技术和管理
2012 – 至今 **腾讯科技**	天涯明月刀 ……	引擎、优化等

《天涯明月刀》 wuxia.qq.com

“《天涯明月刀》代表腾讯目前自主研发的最高水准。”

——StevenMa

“所见过的最棒的室外场景渲染。”

——传奇程序员，Epic Games创始人，Tim Sweeney

“《天涯明月刀》的QuickSilver引擎在同等画面水平的横向对比中，拥有最高的运行效率，可以给玩家提供最流畅的游戏体验。”

——NVIDIA技术总监，林楠

天刀引擎？

- C++引擎
  1. 内存管理、指针的使用
  2. 性能优化方便做深做细
- 自研引擎
  1. 基本框架、对象管理、渲染系统等基础功能
  2. 选取、整合、定制第三方库
  3. 自由的实现各种想实现的功能

关于内存的性能、空间、调试……

分享内容

1. 内存性能的提升
2. 内存占用的优化
3. 冲内存问题的检测

分享内容

1. 内存性能的提升
2. 内存占用的优化
3. 冲内存问题的检测

分享内容

1. 内存性能的提升
   1. 慢的原因
   2. 提升手段
2. 内存占用的优化
3. 冲内存问题的检测

内存性能

慢的原因

malloc

new

HeapAlloc

VirtualAlloc

管理算法

管理开销

内核态切换

优化手段

通用内存管理

特殊内存池

400+倍提升

内存性能

通用
内存池

特殊
内存池

第三方内存库 → nedmalloc ptmalloc

http://www.nedprod.com/programs/portable/nedmalloc/

It is more than 125 times faster than the standard Windows XP memory allocator, 4-10 times faster than the standard FreeBSD 6 memory allocator and up to twice as fast as ptmalloc2, the standard Linux memory allocator. It can sustain a **minimum** of between 7.3m and 8.2m malloc & free pair operations per second on a 3400 (2.20Ghz) AMD Athlon64 machine.

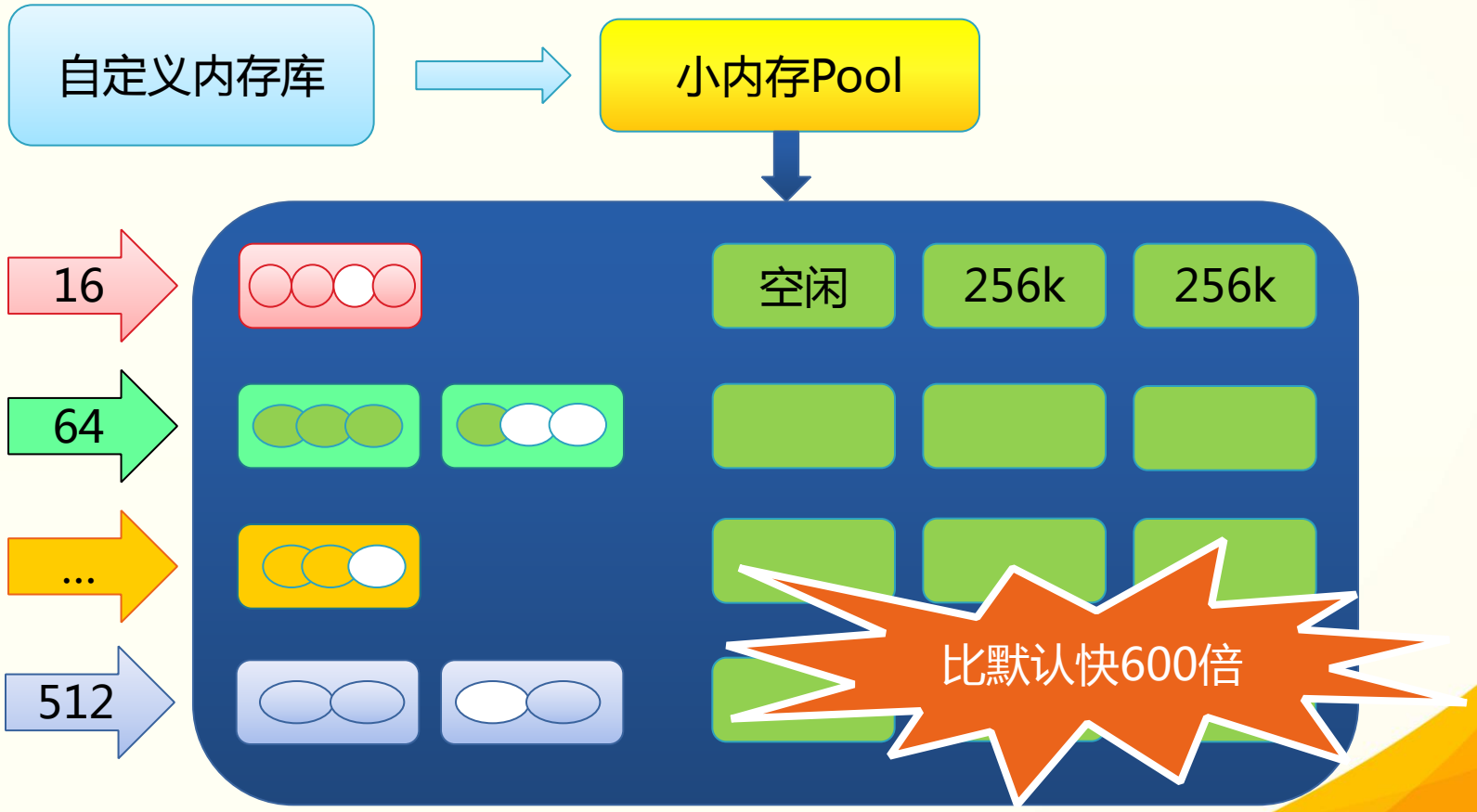
比XP默认快125倍

Open Source

内存性能

通用
内存池

特殊
内存池



内存性能

通用
内存池

特殊
内存池

固定尺寸

申请释放频繁

最大占用已知

固定尺寸块分配

被占用块　最近空闲块　空闲块　下一个空闲块

内存性能

通用
内存池

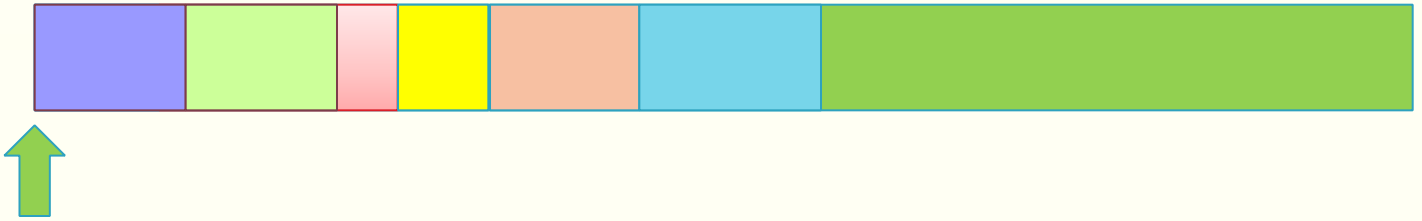
特殊
内存池

临时申请使用

不固定尺寸

最大占用已知

栈式内存分配



后面的空间？

更高层面的“临时”？

双向栈内存

分享内容

1. 内存性能的提升
   1. 慢的原因
   2. 提升手段
2. 内存占用的优化
3. 冲内存问题的检测

分享内容

1. 内存性能的提升
2. 内存占用的优化
3. 冲内存问题的检测

分享内容

1. 内存性能的提升
2. 内存占用的优化
    1. 内存占用的统计
        1. 全局统计
        2. 即时统计
        3. 详细统计
    2. 内存分析和优化
        1. 内容分析
        2. 结构分析
        3. 堆栈分析
3. 冲内存问题的检测

分享内容

1. 内存性能的提升
2. 内存占用的优化
   1. 内存占用的统计
      1. 全局统计
      2. 即时统计
      3. 详细统计
   2. 内存分析和优化
      1. 内容分析
      2. 结构分析
      3. 堆栈分析
3. 冲内存问题的检测

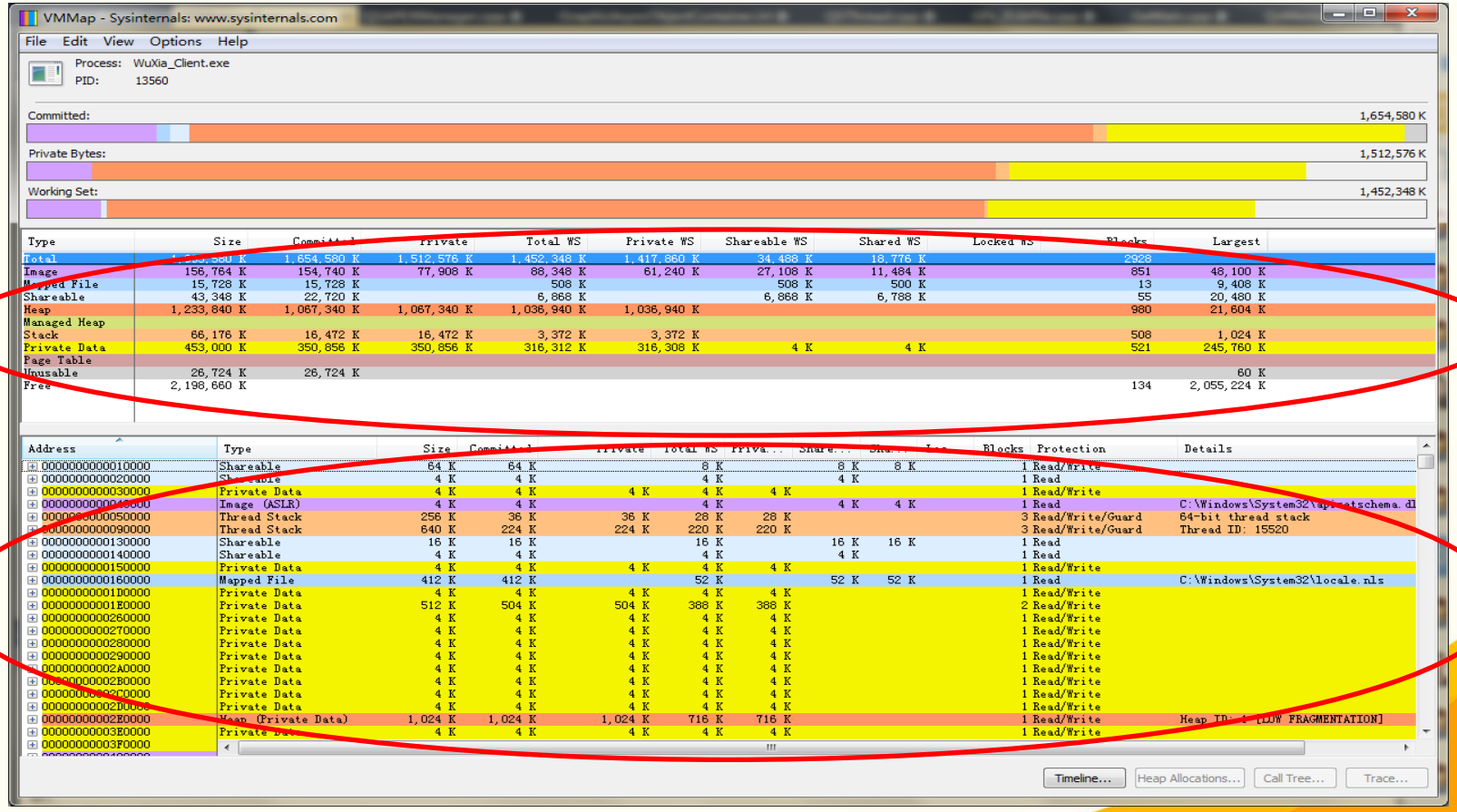
内存统计

全局统计

即时统计

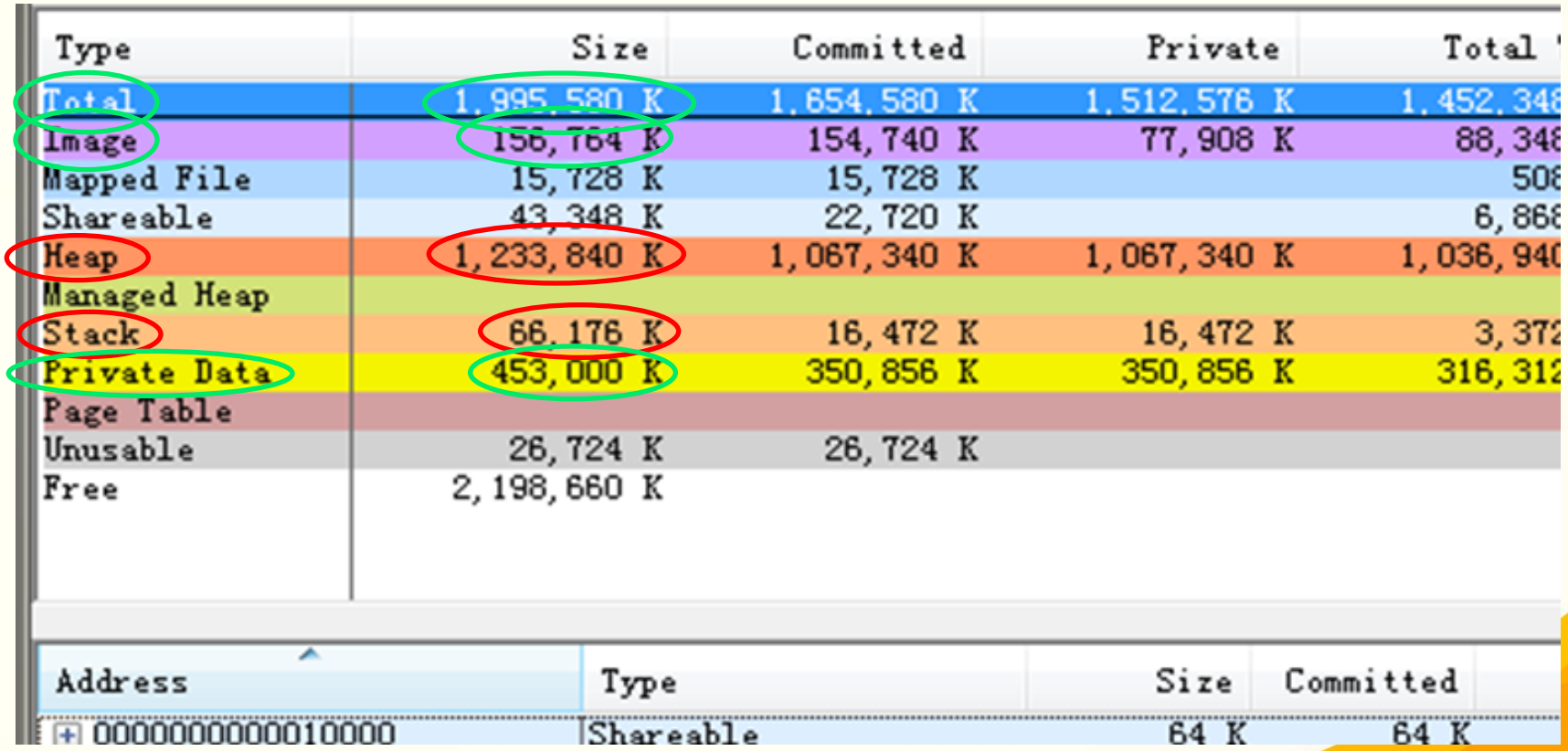
详细统计

VMMap



内存统计 全局统计 即时统计 详细统计

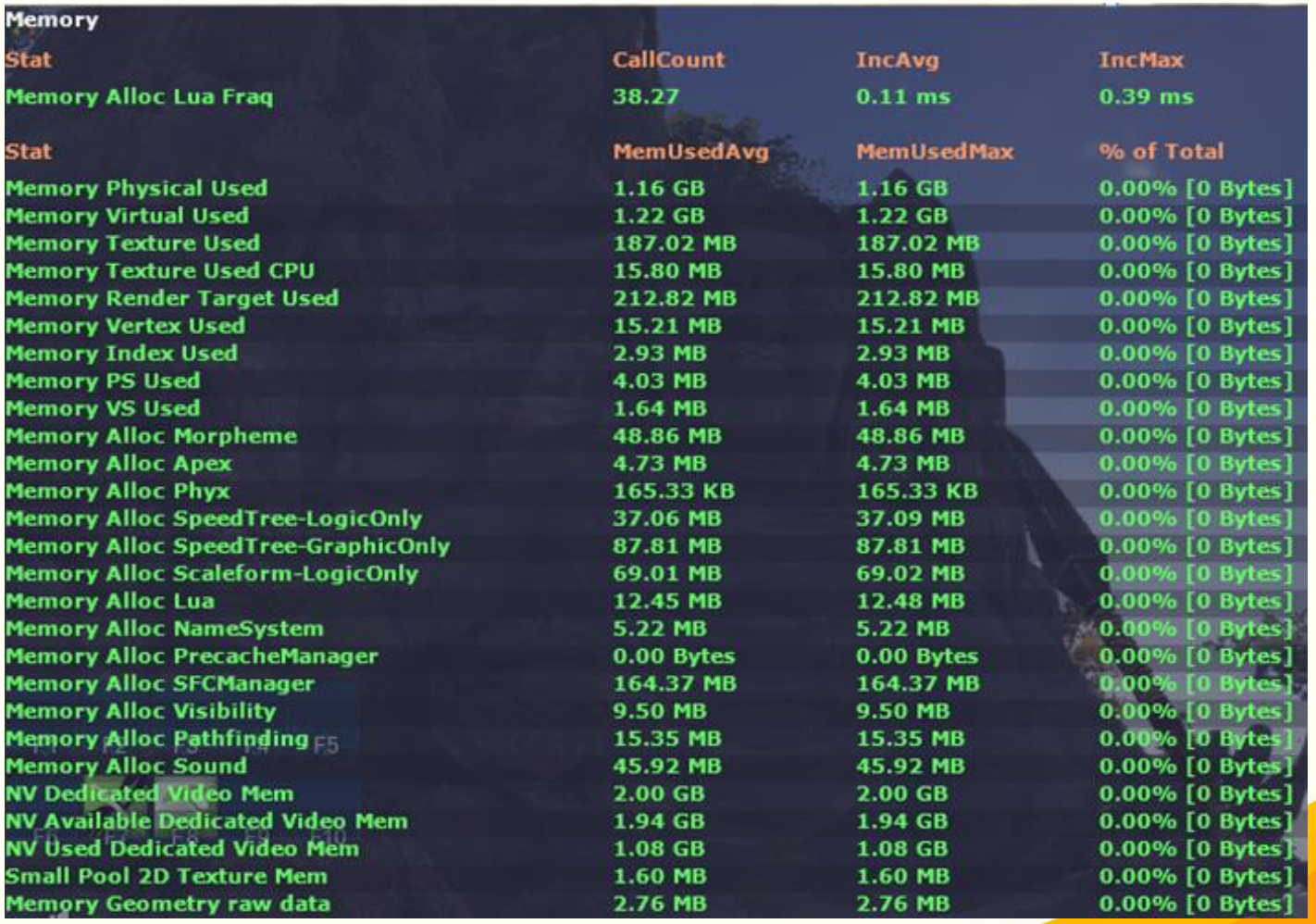
VMMap



Type	Size	Committed	Private	Total
Total	1,995,580 K	1,654,580 K	1,512,576 K	1,452,348
Image	156,764 K	154,740 K	77,908 K	88,348
Mapped File	15,728 K	15,728 K		508
Shareable	43,348 K	22,720 K		6,868
Heap	1,233,840 K	1,067,340 K	1,067,340 K	1,036,940
Managed Heap				
Stack	66,176 K	16,472 K	16,472 K	3,372
Private Data	453,000 K	350,856 K	350,856 K	316,312
Page Table				
Unusable	26,724 K	26,724 K		
Free	2,198,660 K			

Address	Type	Size	Committed
0000000000010000	Shareable	64 K	64 K

内存统计 | 全局统计 | 即时统计 | 详细统计



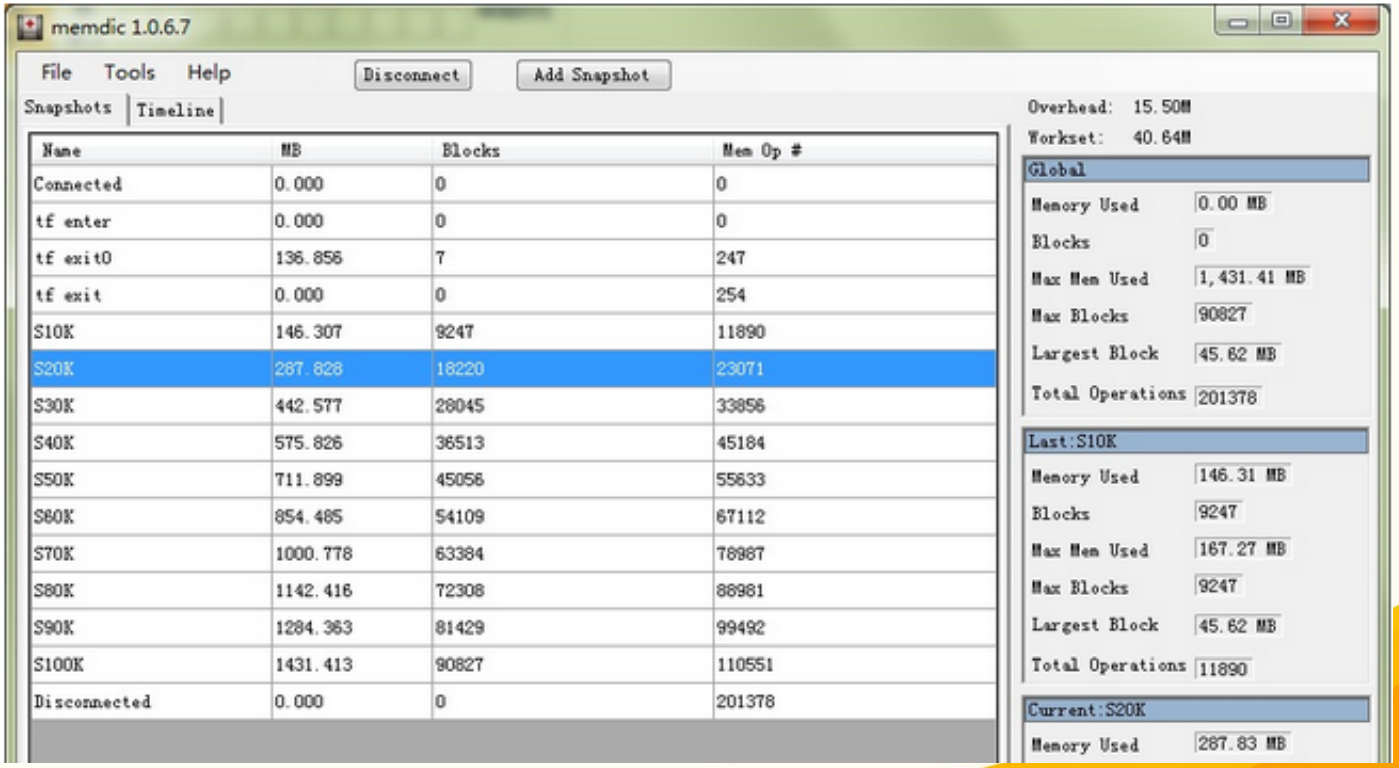
Memory

Stat	CallCount	IncAvg	IncMax
Memory Alloc Lua Fraq	38.27	0.11 ms	0.39 ms
Stat	**MemUsedAvg**	**MemUsedMax**	**% of Total**
Memory Physical Used	1.16 GB	1.16 GB	0.00% [0 Bytes]
Memory Virtual Used	1.22 GB	1.22 GB	0.00% [0 Bytes]
Memory Texture Used	187.02 MB	187.02 MB	0.00% [0 Bytes]
Memory Texture Used CPU	15.80 MB	15.80 MB	0.00% [0 Bytes]
Memory Render Target Used	212.82 MB	212.82 MB	0.00% [0 Bytes]
Memory Vertex Used	15.21 MB	15.21 MB	0.00% [0 Bytes]
Memory Index Used	2.93 MB	2.93 MB	0.00% [0 Bytes]
Memory PS Used	4.03 MB	4.03 MB	0.00% [0 Bytes]
Memory VS Used	1.64 MB	1.64 MB	0.00% [0 Bytes]
Memory Alloc Morpheme	48.86 MB	48.86 MB	0.00% [0 Bytes]
Memory Alloc Apex	4.73 MB	4.73 MB	0.00% [0 Bytes]
Memory Alloc Phyx	165.33 KB	165.33 KB	0.00% [0 Bytes]
Memory Alloc SpeedTree-LogicOnly	37.06 MB	37.09 MB	0.00% [0 Bytes]
Memory Alloc SpeedTree-GraphicOnly	87.81 MB	87.81 MB	0.00% [0 Bytes]
Memory Alloc Scaleform-LogicOnly	69.01 MB	69.02 MB	0.00% [0 Bytes]
Memory Alloc Lua	12.45 MB	12.48 MB	0.00% [0 Bytes]
Memory Alloc NameSystem	5.22 MB	5.22 MB	0.00% [0 Bytes]
Memory Alloc PrecacheManager	0.00 Bytes	0.00 Bytes	0.00% [0 Bytes]
Memory Alloc SFCManager	164.37 MB	164.37 MB	0.00% [0 Bytes]
Memory Alloc Visibility	9.50 MB	9.50 MB	0.00% [0 Bytes]
Memory Alloc Pathfinding	15.35 MB	15.35 MB	0.00% [0 Bytes]
Memory Alloc Sound	45.92 MB	45.92 MB	0.00% [0 Bytes]
NV Dedicated Video Mem	2.00 GB	2.00 GB	0.00% [0 Bytes]
NV Available Dedicated Video Mem	1.94 GB	1.94 GB	0.00% [0 Bytes]
NV Used Dedicated Video Mem	1.08 GB	1.08 GB	0.00% [0 Bytes]
Small Pool 2D Texture Mem	1.60 MB	1.60 MB	0.00% [0 Bytes]
Memory Geometry raw data	2.76 MB	2.76 MB	0.00% [0 Bytes]

内存统计　全局统计　即时统计　详细统计

外挂监控

Memdic是"Memory Medic"的缩写，即内存医疗兵，是一个内存使用统计和诊断的库和工具，尤其对内存泄漏的诊断有帮助。其中，Memdic库是采用C++实现的API，Memdic工具是采用C#实现的GUI。



Name	MB	Blocks	Mem Op #
Connected	0.000	0	0
tf enter	0.000	0	0
tf exit0	136.856	7	247
tf exit	0.000	0	254
S10K	146.307	9247	11890
S20K	287.828	18220	23071
S30K	442.577	28045	33856
S40K	575.826	36513	45184
S50K	711.899	45056	55833
S60K	854.485	54109	67112
S70K	1000.778	63384	78987
S80K	1142.416	72308	88981
S90K	1284.363	81429	99492
S100K	1431.413	90827	110551
Disconnected	0.000	0	201378

内存统计

全局统计

即时统计

详细统计

内部统计

统计代码添加

进程Exe → 自定义内存池中统计

进程Dll → Hooked HeapAlloc

统计内容

调用堆栈

申请尺寸

申请次数

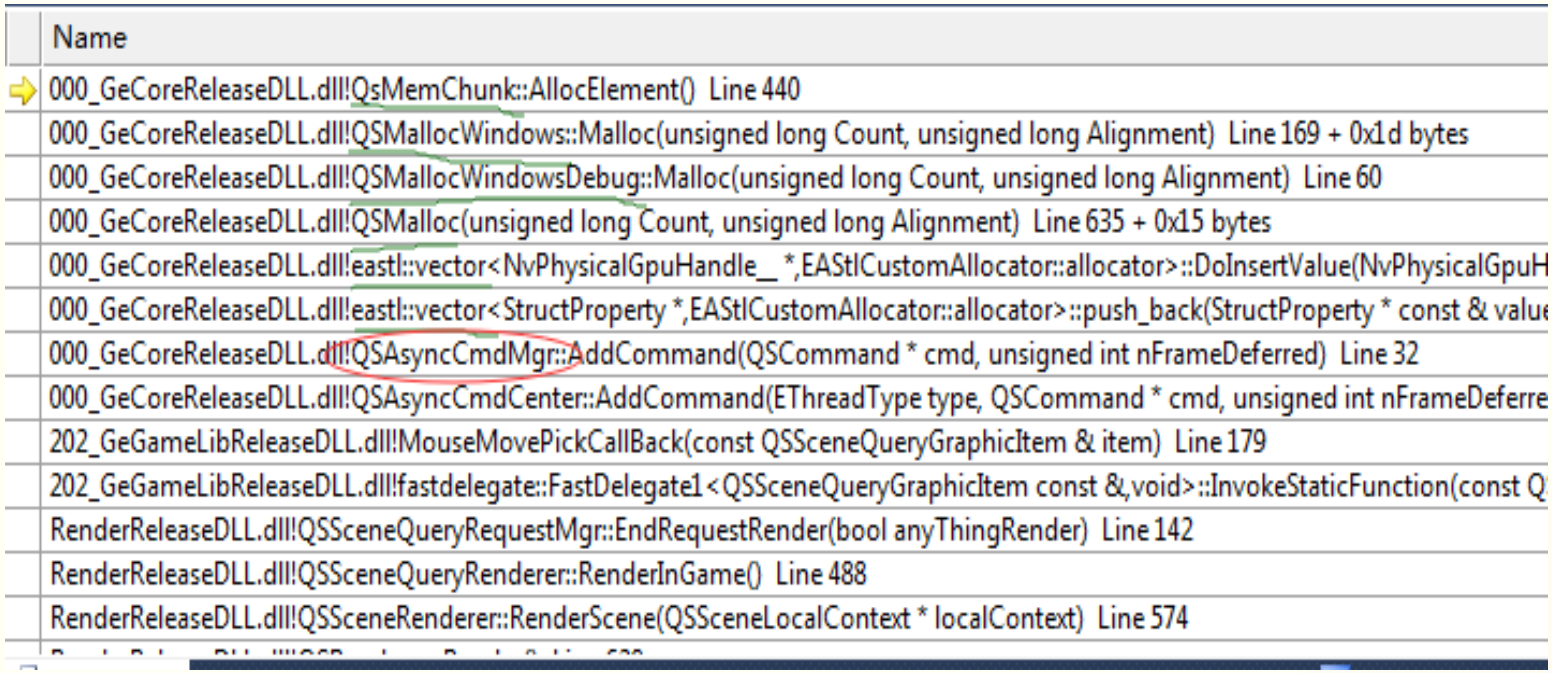
统计内容利用

模块归类

差异报告

内存统计 全局统计 即时统计 详细统计

统计内容利用 模块归类 差异报告



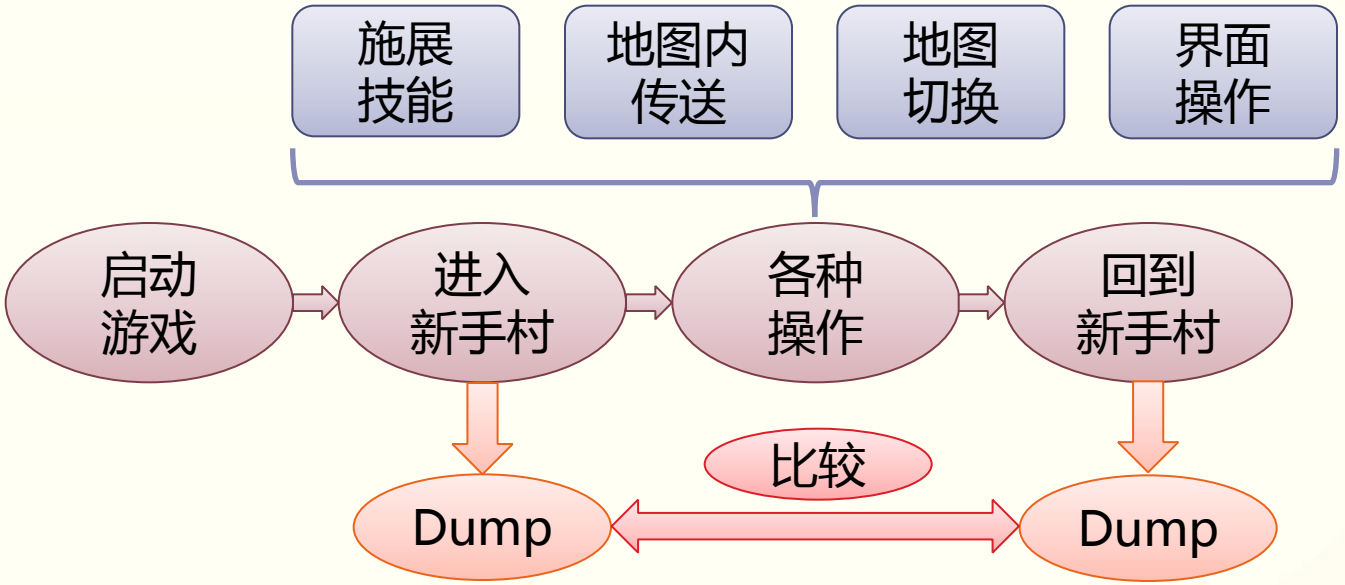
内存统计 全局统计 即时统计 详细统计

统计内容利用 模块归类 差异报告

1	CachePackage	169.265	169.265	1.467196	0.87%
2	AnimSetData	124.0002	124.0002	91.81416	74.04%
3	Scaleform	74.12056	74.11665	45.05735	60.79%
4	[Other]	71.83027	70.80683	46.71133	65.97%
5	SpeedTree	37.52307	37.51916	23.31224	62.13%
6	SpeedTreeAlloc	37.16535	37.16535	30.36953	81.71%
7	EAStlCustomAllocator	33.43606	33.43606	16.6295	49.74%
8	MainProcedure	25.36793	25.36793	13.67171	53.89%
9	QSD3D9Texture	20.27483	20.2225	7.387256	36.53%
10	Tenio	17.00594	17.00594	4.628887	27.22%
11	HierarchicalMap	16.77736	3.652364	2.247699	61.54%
12	QSLuaState	12.56268	12.56268	3.461513	27.55%
13	QSSubGeometry	10.04627	10.04627	3.080373	30.66%
14	QSMegaCollisionFilePolicy	9.77027	9.77027	7.704853	78.86%
15	ShaderMgr	9.738213	9.021525	6.766376	75.00%
16	GFxFramework	9.399768	9.399768	3.100842	32.99%
17	QSD3D9Drv	8.149108	8.149108	4.669148	57.30%
18	QSMapMgr	8.015768	8.015768	0.333161	4.16%
19	NameSystem	8.000091	8.000091	5.276385	65.95%

内存统计　全局统计　即时统计　详细统计

统计内容利用　模块归类　差异报告



分享内容

1. 内存性能的提升
2. 内存占用的优化
    1. 内存占用的统计
        1. 全局统计
        2. 即时统计
        3. 详细统计
    2. 内存分析和优化
        1. 内容分析
        2. 结构分析
        3. 堆栈分析
3. 冲内存问题的检测

分享内容

1. 内存性能的提升
2. 内存占用的优化
    1. 内存占用的统计
        1. 全局统计
        2. 即时统计
        3. 详细统计
    2. 内存分析和优化
        1. 内容分析
        2. 结构分析
        3. 堆栈分析
3. 冲内存问题的检测

分析优化

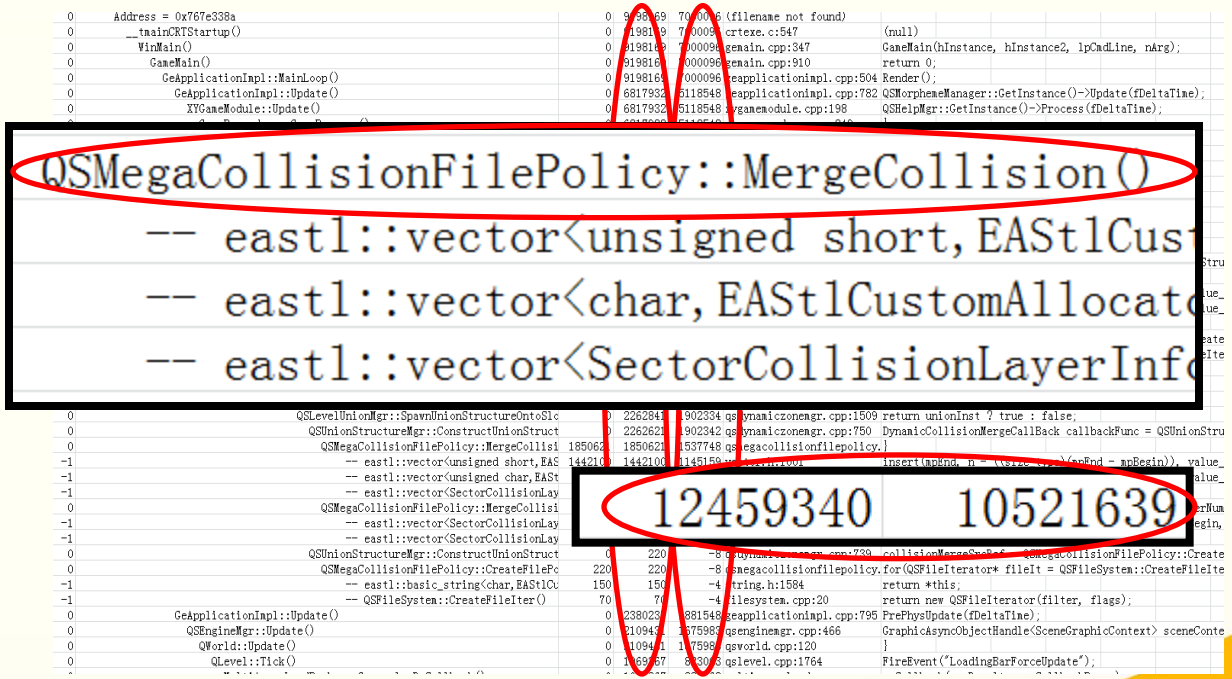
内容分析

结构分析

堆栈分析

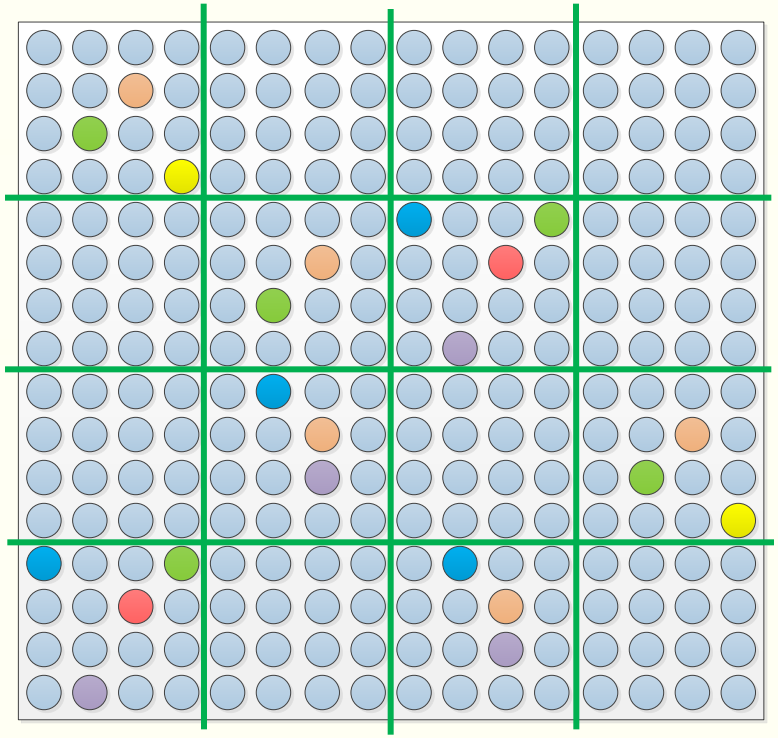
找出可能有“油水”的内存

内存数据压缩 → 优化价值

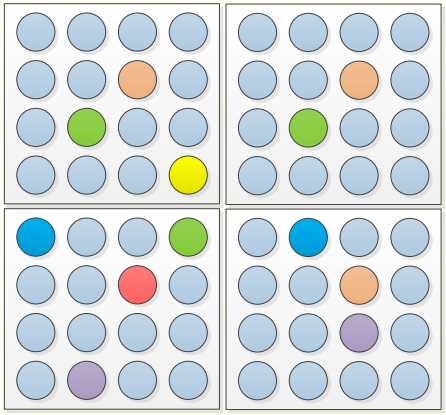


分析优化 内容分析 结构分析 堆栈分析

稀疏数据优化举例



类似Tile Map



1, 0, 0, 0,
0, 2, 3, 0,
0, 4, 0, 1,
3, 0, 4, 0,

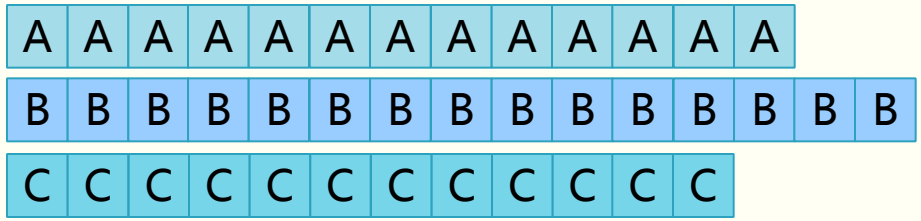
12M->3M

分析优化 内容分析 结构分析 堆栈分析

```
struct A
{
    char cGender;
    int iValue;
    short sValue;
};
```

```
struct B
{
    char cType;
    int iValue;
    char cName[10];
    short sDate;
};
```

```
struct C
{
    char cName[10];
    int iHealth;
    char bAlive;
    int iFlag;
    int iExtra;
};
```



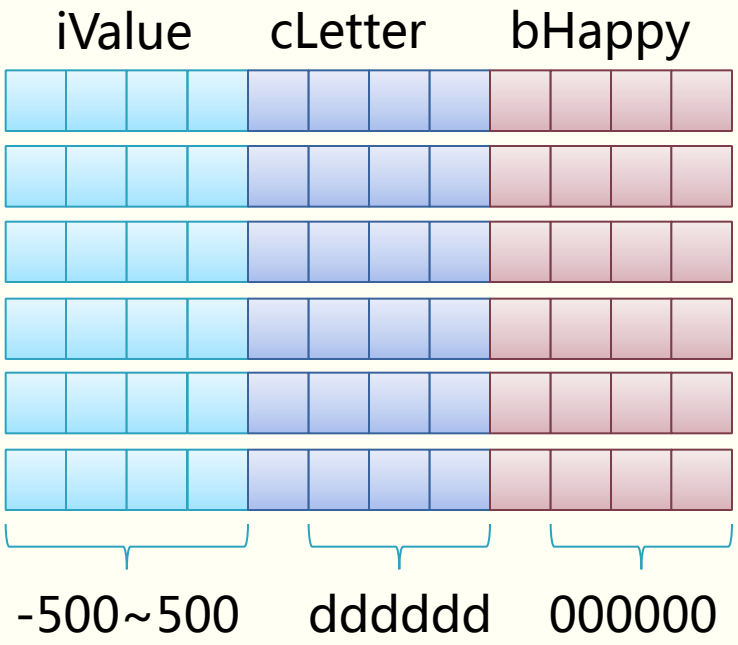
如何优化近200M的静态数据？

上层需要时加载

底层结构分析辅助？

想法：结构定义不合理导致浪费

```
struct TempStruct
{
    int iValue;
    char cLetter;
    int bHappy;
};
struct NewStruct
{
    short iValue;
    char cLetter;
    char bHappy;
};
```



分析优化　内容分析　结构分析　堆栈分析

统计和输出　修改方案　节省空间

	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P
1	TypeName	TypeSize	NbVectors	TotalSize	0	FF	DD	TotalWast	Waste%	MaxSizeIc	MaxSizeSi	MaxWasteI	MaxWaste	WasteOffset	ggestion	Op
2	struct QSParameter	24	39320	4080240	587908	7076	689805	1284789	31.49%	40607	240	36919	133	5, 6, 7, 20, 21	mValue: String -> NameHandle	2040120
3	struct stItemInfo	736	3	4121600	1791082	162160	616882	2570124	6							
4	struct stSkillElement	616	1	4883648	2357048	161295	629405	3147748	6							
5	struct EntityConfig	380	9	7755800	2410240	246894	1369925	4027059	51.						mSkelFilePath: String -> NameHandle m_nSignature: int -> uchar m_nKeyEvent: int -> short	
6	struct stEffectBehavior	200	10926	3525200	1781248	67007	406696	2254951	63.97%	35211	7000					804
	struct stSoundBehavior	124	10926	2037692	676024	63937	329186	1069147	52.47%	42070	1736	42070	891	4, 5, 6, 7, 9,	m_nKeyEvent: int -> short m_kSoundName: String -> NameHandle	870949

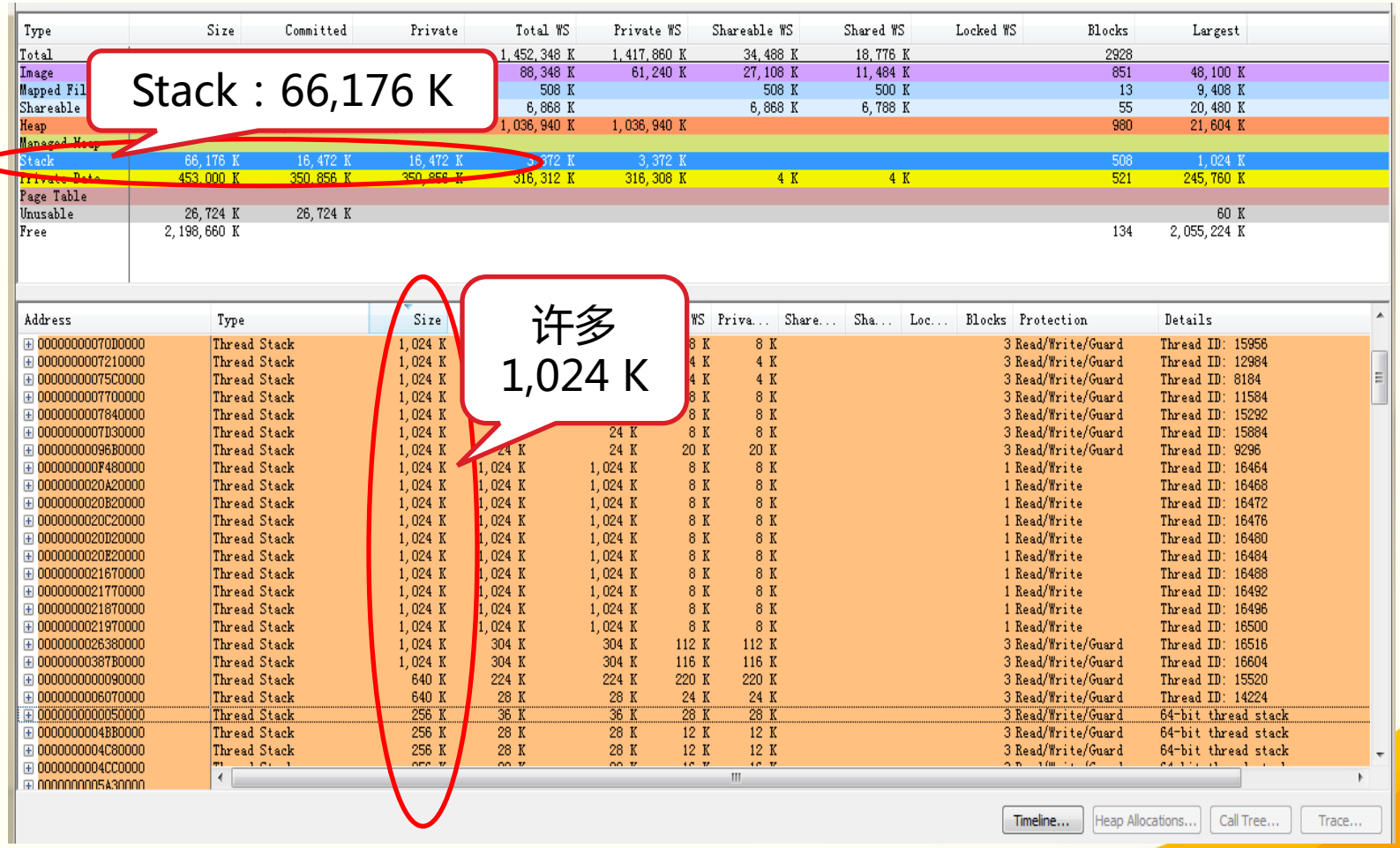
Struct XXXXX

变量XXX：原类型 -> 新类型
变量XXX：移除（永远常数）

修改完可节省空间：XXXXX

针对几个“大头”
做的处理：10+M

分析优化 内容分析 结构分析 堆栈分析



Type	Size	Committed	Private	Total WS	Private WS	Shareable WS	Shared WS	Locked WS	Blocks	Largest
Total				1,452,348 K	1,417,860 K	34,488 K	18,776 K		2928	
Image				88,348 K	61,240 K	27,108 K	11,484 K		851	48,100 K
Mapped File				508 K		508 K	500 K		13	9,408 K
Shareable				6,868 K		6,868 K	6,788 K		55	20,480 K
Heap				1,036,940 K	1,036,940 K				980	21,604 K
Managed Heap										
Stack	66,176 K	16,472 K	16,472 K	3,372 K	3,372 K				508	1,024 K
Private Data	453,000 K	350,856 K	350,856 K	316,312 K	316,308 K	4 K	4 K		521	245,760 K
Page Table										
Unusable	26,724 K	26,724 K								60 K
Free	2,198,660 K								134	2,055,224 K

Address	Type	Size	WS	Priva...	Share...	Sha...	Loc...	Blocks	Protection	Details
00000000070D0000	Thread Stack	1,024 K	8 K	8 K				3	Read/Write/Guard	Thread ID: 15956
0000000007210000	Thread Stack	1,024 K	4 K	4 K				3	Read/Write/Guard	Thread ID: 12984
00000000075C0000	Thread Stack	1,024 K	4 K	4 K				3	Read/Write/Guard	Thread ID: 8184
0000000007700000	Thread Stack	1,024 K	8 K	8 K				3	Read/Write/Guard	Thread ID: 11584
0000000007840000	Thread Stack	1,024 K	8 K	8 K				3	Read/Write/Guard	Thread ID: 15292
00000000007D30000	Thread Stack	1,024 K	24 K	8 K				3	Read/Write/Guard	Thread ID: 15884
00000000096B0000	Thread Stack	1,024 K	24 K	20 K				3	Read/Write/Guard	Thread ID: 9296
000000000F480000	Thread Stack	1,024 K	1,024 K	8 K				1	Read/Write	Thread ID: 16464
0000000020A20000	Thread Stack	1,024 K	1,024 K	8 K				1	Read/Write	Thread ID: 16468
0000000020B20000	Thread Stack	1,024 K	1,024 K	8 K				1	Read/Write	Thread ID: 16472
0000000020C20000	Thread Stack	1,024 K	1,024 K	8 K				1	Read/Write	Thread ID: 16476
0000000020D20000	Thread Stack	1,024 K	1,024 K	8 K				1	Read/Write	Thread ID: 16480
0000000020E20000	Thread Stack	1,024 K	1,024 K	8 K				1	Read/Write	Thread ID: 16484
0000000021670000	Thread Stack	1,024 K	1,024 K	8 K				1	Read/Write	Thread ID: 16488
0000000021770000	Thread Stack	1,024 K	1,024 K	8 K				1	Read/Write	Thread ID: 16492
0000000021870000	Thread Stack	1,024 K	1,024 K	8 K				1	Read/Write	Thread ID: 16496
0000000021970000	Thread Stack	1,024 K	1,024 K	8 K				1	Read/Write	Thread ID: 16500
0000000026380000	Thread Stack	1,024 K	304 K	112 K				3	Read/Write/Guard	Thread ID: 16516
000000003B70000	Thread Stack	1,024 K	304 K	116 K				3	Read/Write/Guard	Thread ID: 16604
0000000000090000	Thread Stack	640 K	224 K	220 K				3	Read/Write/Guard	Thread ID: 15520
0000000006070000	Thread Stack	640 K	28 K	24 K				3	Read/Write/Guard	Thread ID: 14224
0000000000050000	Thread Stack	256 K	36 K	28 K				3	Read/Write/Guard	64-bit thread stack
00000000004BB0000	Thread Stack	256 K	28 K	12 K				3	Read/Write/Guard	64-bit thread stack
00000000004C80000	Thread Stack	256 K	28 K	12 K				3	Read/Write/Guard	64-bit thread stack
00000000004CC0000										
00000000005A30000										

Timeline... Heap Allocations... Call Tree... Trace...

分析优化 | 内容分析 | 结构分析 | 堆栈分析

减少不必要的线程

为什么很多线程1,024K？

进程主线程1,024K

CreateThread() - dwStackSize使用0

Linker配置降低Stack尺寸？

加壳系统 | 第三方DLL | 全局初始化

如何处理？

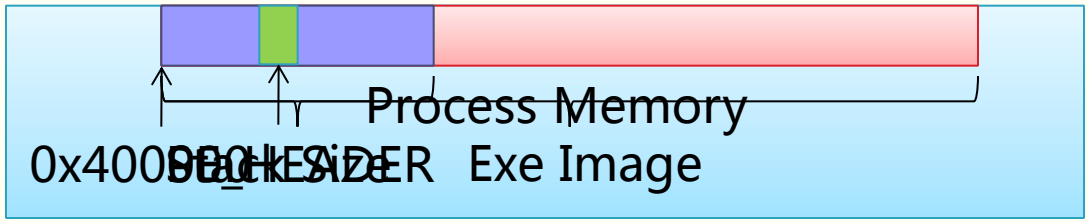
SubSystem	Windows (/SUBSYSTEM:WINDO
Minimum Required Version	
Heap Reserve Size	
Heap Commit Size	
Stack Reserve Size	1048576
Stack Commit Size	
Enable Large Addresses	Yes (/LARGEADDRESSAWARE)
Terminal Server	
Swap Run From CD	No
Swap Run From Network	No
Driver	Not Set

```
HANDLE WINAPI CreateThread(
  _In_opt_  LPSECURITY_ATTRIBUTES  lpThreadAttributes,
  _In_      SIZE_T                 dwStackSize,
  _In_      LPTHREAD_START_ROUTINE lpStartAddress,
  _In_opt_  LPVOID                 lpParameter,
  _In_      DWORD                  dwCreationFlags,
  _Out_opt_ LPDWORD                lpThreadId
);
```

分析优化 | 内容分析 | 结构分析 | 堆栈分析

运行时修改进程堆栈大小

固定BaseAddr | 修改堆栈尺寸




```
void HackModifyStackReserve()
{
    INT FixedBaseAddress = 0x400000;
    PIMAGE_DOS_HEADER dosHeader = (PIMAGE_DOS_HEADER)FixedBaseAddress;
    //check PE file valid
    QSCheck(dosHeader->e_magic == IMAGE_DOS_SIGNATURE);
    IMAGE_NT_HEADERS32* pNTHeader = (IMAGE_NT_HEADERS32*)( FixedBaseAddress + dosHeader->e_lfanew );
    //backup original flag modify stackReserve and write it back
    DWORD originalFlag;
    ::VirtualProtect( &pNTHeader->OptionalHeader.SizeOfStackReserve,8,PAGE_EXECUTE_READWRITE,&originalFlag);
    pNTHeader->OptionalHeader.SizeOfStackReserve = 256*1024;
    ::VirtualProtect( &pNTHeader->OptionalHeader.SizeOfStackReserve,8,originalFlag,&originalFlag);
}
```

分析优化

内容分析

结构分析

堆栈分析

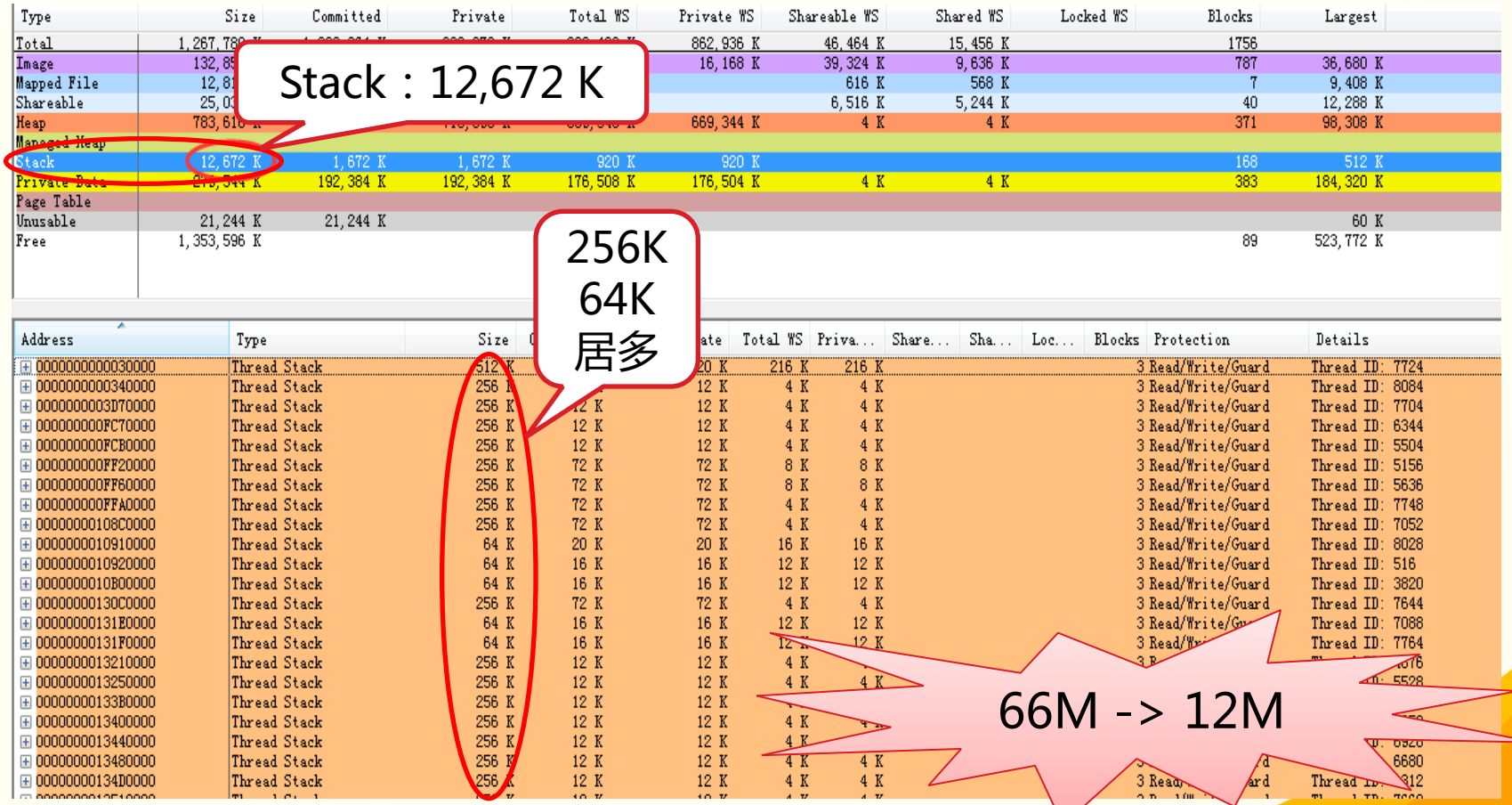
猜想
dwStackSize
不科学

全面掌控线程堆栈大小

Hook线程创建

```
HANDLE  WINAPI Hook_CreateThread(   LPSECURITY_ATTRIBUTES lpThreadAttributes, SIZE_T dwStackSiz
{
    //hack: if stack size requested is larger than 64k, use default stack size (256k, as is set
    SIZE_T dwDesiredStackSize = dwStackSize;

    if (!QSGlobalFlag::mSafeStackSize )
    {
        if( dwStackSize == 2*1024*1024 )
        {
            // hack: render thread will be allocated as 2MB
            dwDesiredStackSize = dwStackSize;
        }
        else if( dwStackSize == 64*1024 )
        {
            // hack: nvumdrv.dll use 64KB as stacksize, it may not big enough to compile some s
            dwDesiredStackSize = 256*1024;
        }
        else if(dwStackSize > 65536)
        {
            dwDesiredStackSize = 0;
        }
    }

    HANDLE ret = Real_CreateThread( lpThreadAttributes, dwDesiredStackSize, lpStartAddress, lpPara
    return ret;
}
```

分析优化 内容分析 结构分析 堆栈分析



Type	Size	Committed	Private	Total WS	Private WS	Shareable WS	Shared WS	Locked WS	Blocks	Largest
Total	1,267,780 K				862,936 K	46,464 K	15,456 K		1756	
Image	132,8				16,168 K	39,324 K	9,636 K		787	36,680 K
Mapped File	12,8					616 K	568 K		7	9,408 K
Shareable	25,0					6,516 K	5,244 K		40	12,288 K
Heap	783,616 K				669,344 K	4 K	4 K		371	98,308 K
Managed Heap										
Stack	12,672 K	1,672 K	1,672 K	920 K	920 K				168	512 K
Private Data	248,344 K	192,384 K	192,384 K	176,508 K	176,504 K	4 K	4 K		383	184,320 K
Page Table										
Unusable	21,244 K	21,244 K								60 K
Free	1,353,596 K								89	523,772 K

Address	Type	Size	ate	Total WS	Priva...	Share...	Sha...	Loc...	Blocks	Protection	Details
0000000000030000	Thread Stack	512 K	20 K	216 K	216 K				3	Read/Write/Guard	Thread ID: 7724
0000000000340000	Thread Stack	256 K	12 K	4 K	4 K				3	Read/Write/Guard	Thread ID: 8084
000000000 3D70000	Thread Stack	256 K	12 K	4 K	4 K				3	Read/Write/Guard	Thread ID: 7704
000000000FC70000	Thread Stack	256 K	12 K	4 K	4 K				3	Read/Write/Guard	Thread ID: 6344
000000000FCB0000	Thread Stack	256 K	12 K	4 K	4 K				3	Read/Write/Guard	Thread ID: 5504
000000000FF20000	Thread Stack	256 K	72 K	8 K	8 K				3	Read/Write/Guard	Thread ID: 5156
000000000FF60000	Thread Stack	256 K	72 K	8 K	8 K				3	Read/Write/Guard	Thread ID: 5636
000000000FFA0000	Thread Stack	256 K	72 K	4 K	4 K				3	Read/Write/Guard	Thread ID: 7748
00000000108C0000	Thread Stack	256 K	72 K	4 K	4 K				3	Read/Write/Guard	Thread ID: 7052
0000000010910000	Thread Stack	64 K	20 K	16 K	16 K				3	Read/Write/Guard	Thread ID: 8028
0000000010920000	Thread Stack	64 K	16 K	12 K	12 K				3	Read/Write/Guard	Thread ID: 516
000000001 0B00000	Thread Stack	64 K	16 K	12 K	12 K				3	Read/Write/Guard	Thread ID: 3820
00000000130C0000	Thread Stack	256 K	72 K	4 K	4 K				3	Read/Write/Guard	Thread ID: 7644
00000000131E0000	Thread Stack	64 K	16 K	12 K	12 K				3	Read/Write/Gu	Thread ID: 7088
00000000131F0000	Thread Stack	64 K	16 K	12 K	12 K				3	Read/Wri	Thread ID: 7764
0000000013210000	Thread Stack	256 K	12 K	4 K					3		6076
0000000013250000	Thread Stack	256 K	12 K	4 K	4 K						5528
00000000133B0000	Thread Stack	256 K	12 K	12 K							
0000000013400000	Thread Stack	256 K	12 K	4 K	4 K						
0000000013440000	Thread Stack	256 K	12 K	4 K							6320
0000000013480000	Thread Stack	256 K	12 K	4 K	4 K					rd	6680
00000000134D0000	Thread Stack	256 K	12 K	4 K	4 K				3	Read	Thread ID: 312

分享内容

1. 内存性能的提升
2. 内存占用的优化
    1. 内存占用的统计
        1. 全局统计
        2. 即时统计
        3. 详细统计
    2. 内存分析和优化
        1. 内容分析
        2. 结构分析
        3. 堆栈分析
3. 冲内存问题的检测

分享内容

1. 内存性能的提升
2. 内存占用的优化
3. 冲内存问题的检测

分享内容

1. 内存性能的提升
2. 内存占用的优化
3. 冲内存问题的检测
    1. 常见冲内存方式
        1. 静态数组越界
        2. 动态数组越界
        3. 访问已释放内存
    2. 检测和处理方法
        1. 简易标记防护
        2. 使用调试库和工具
        3. 额外检查和处理

分享内容

1. 内存性能的提升
2. 内存占用的优化
3. 冲内存问题的检测
    1. 常见冲内存方式
        1. 静态数组越界
        2. 动态数组越界
        3. 访问已释放内存
    2. 检测和处理方法
        1. 简易标记防护
        2. 使用调试库和工具
        3. 额外检查和处理

冲内存方式

静态数组

动态数组

释放内存

堆栈

```
void BadFunction()
{
    int a[10];
    a[-1] = 0x12345678;
    a[10] = 0x87654321;
}
```

Security Cookie

成员变量

```
class Dog
{
    int m_iType;
    char m_szName[16];
    int m_iBehavior;

    void SetName(const char *szName)
    {
        strcpy(m_szName, szName);
    }
};
```

能定位到模块的一般都不难

冲内存方式

静态数组

动态数组

释放内存

全局堆

越界访问检查

```
class Cat
{
    int m_iType;
    vector<float> m_vParams;
    int m_iHealth;

    void Init()
    {
        m_vParams.resize(NB_PARAMS);
    }

    void SetParams(float fRatio, float fValue)
    {
        m_vParams[(int)(fRatio * NB_PARAMS)] = fValue;
    }
};
```

冲内存方式

静态数组

动态数组

释放内存

野指针

```
class DataBase
{
    vector<GameData> m_vData;

};
```

```
class Boss_ZhangSan
{
    GameData *m_pMyData;

    void Init()
    {
        //save pointer for fast access
        m_pMyData = &g_pDB->m_vData[BOSS_ZS_INDEX];
    }

    void Work()
    {
        if (m_pMyData->m_iHealth == 0)
            m_pMyData->m_bDead = true;
    }
};
```

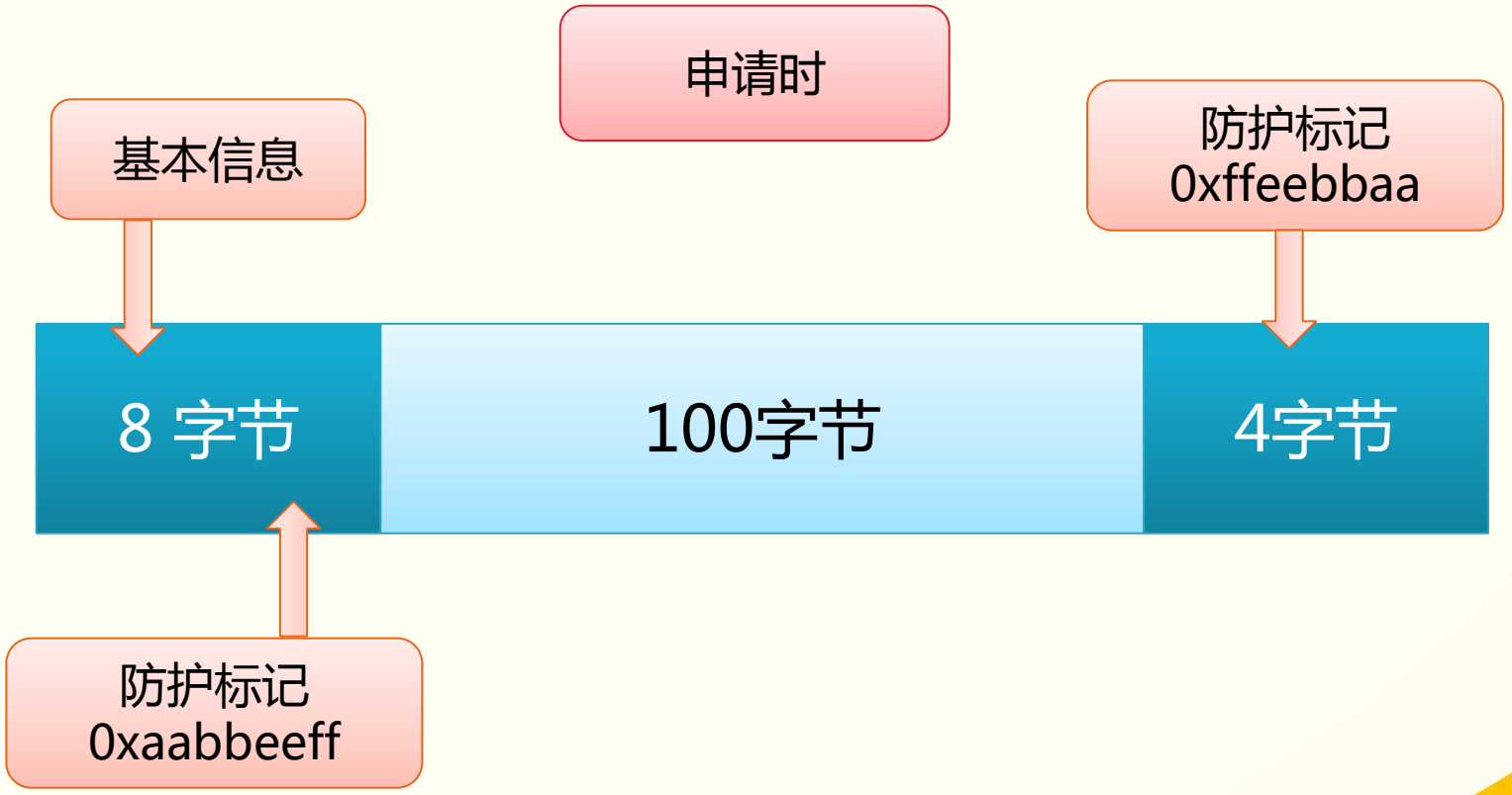
分享内容

1. 内存性能的提升
2. 内存占用的优化
3. 冲内存问题的检测
    1. 常见冲内存方式
        1. 静态数组越界
        2. 动态数组越界
        3. 访问已释放内存
    2. 检测和处理方法
        1. 简易标记防护
        2. 使用调试库和工具
        3. 额外检查和处理

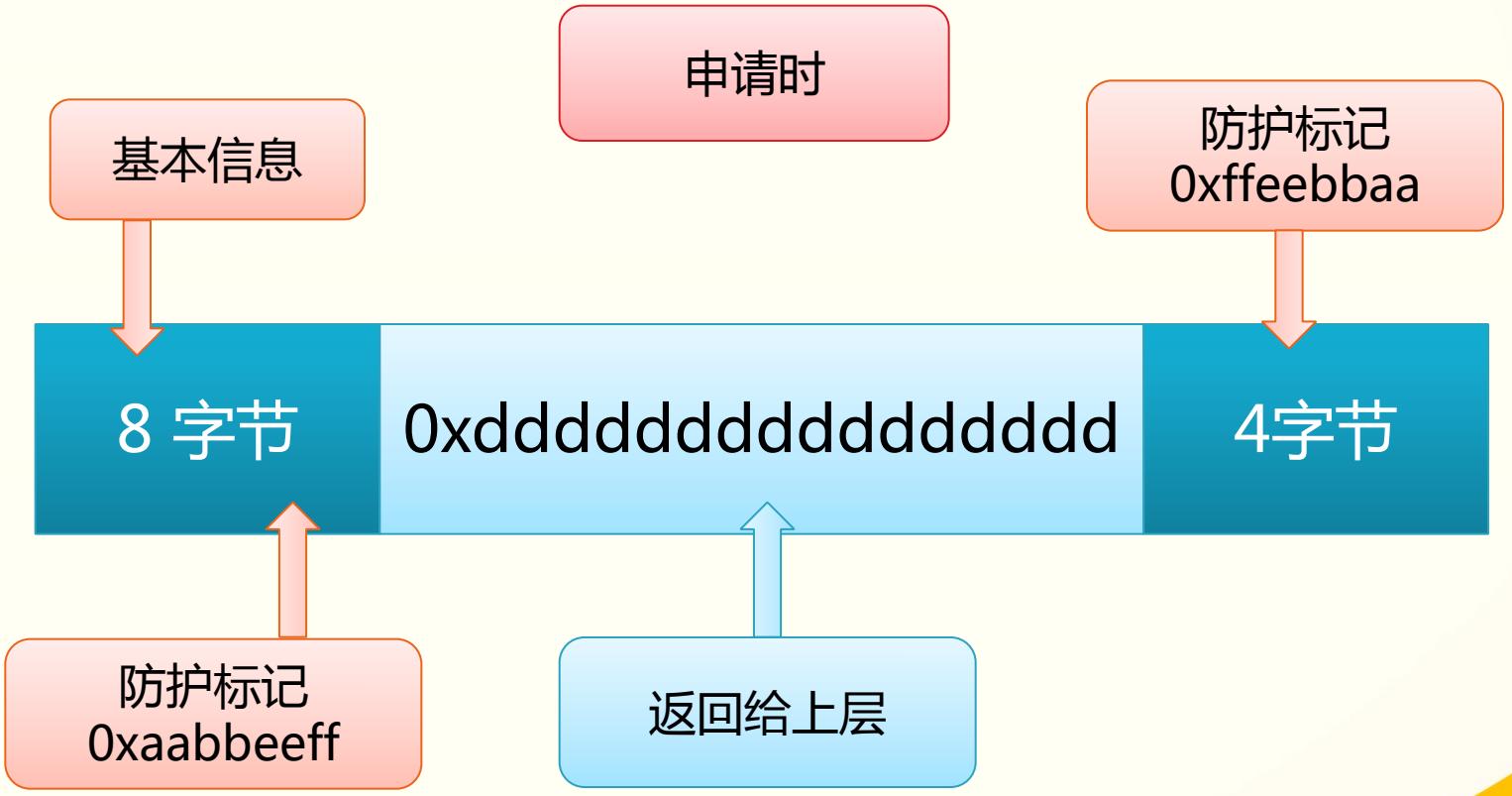
分享内容

1. 内存性能的提升
2. 内存占用的优化
3. 冲内存问题的检测
    1. 常见冲内存方式
        1. 静态数组越界
        2. 动态数组越界
        3. 访问已释放内存
    2. 检测和处理方法
        1. 简易标记防护
        2. 使用调试库和工具
        3. 额外检查和处理

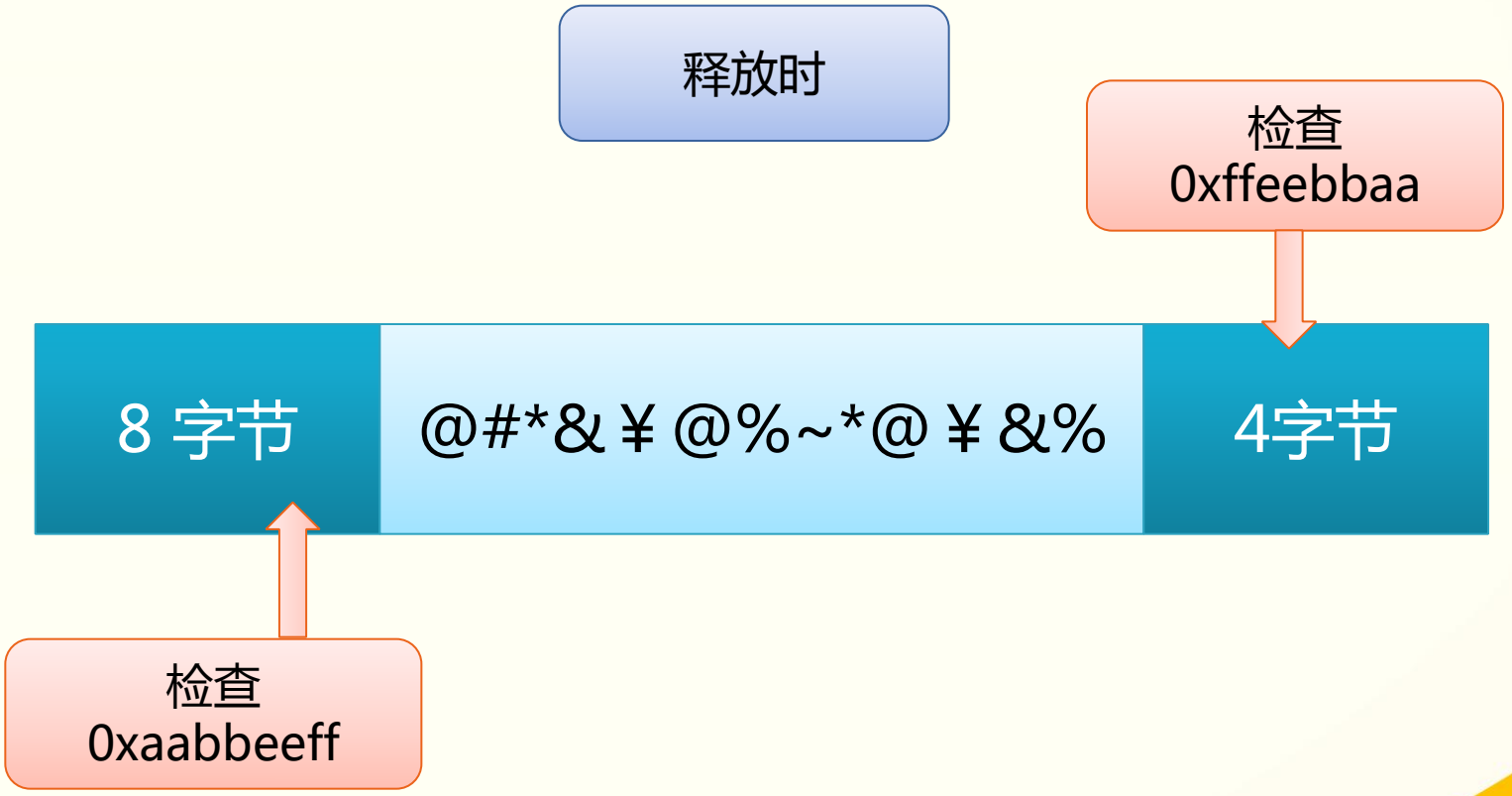
检测方法 | 标记防护 | 库和工具 | 额外处理



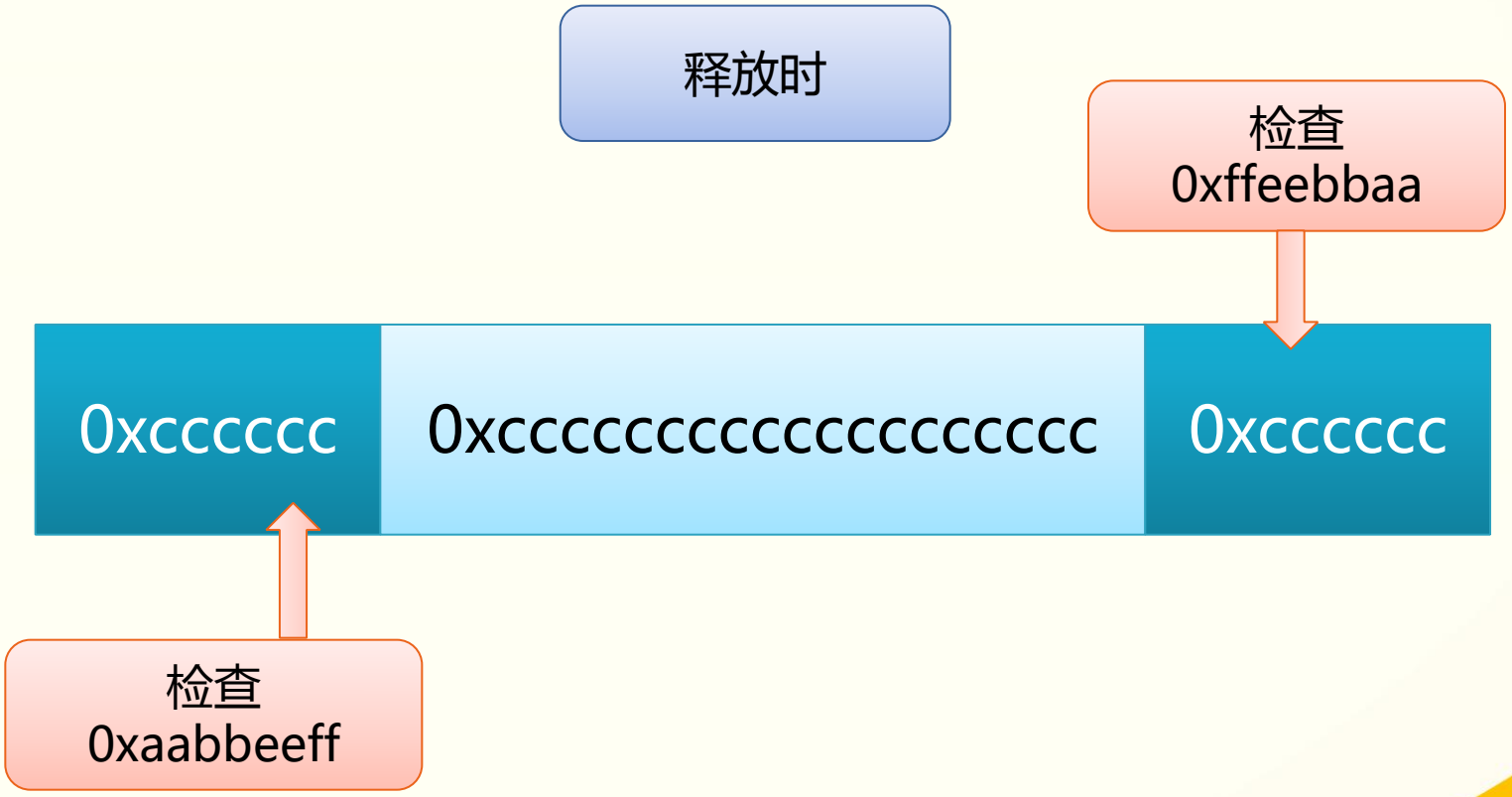
检测方法 | 标记防护 | 库和工具 | 额外处理



检测方法 | 标记防护 | 库和工具 | 额外处理



检测方法 | 标记防护 | 库和工具 | 额外处理



检测方法 | 标记防护 | 库和工具 | 额外处理

```
天涯明月刀 0.21.9785  17745C5266A143FD3664F5D5A3B0BC48
----------------------------------------
Type: EXCEPTION_ACCESS_VIOLATION
Error: Write address 0x0000000C
Address: 59C39CA1

CallStack:
0x59BE0000[59CA1] 000_GeCoreReleaseDLL.dll: (002CF9BC,03A2
```

```
天涯明月刀 0.22.6933  A03DD31C17C72C4FFEF8F6F4C3958A29
----------------------------------------
Type: EXCEPTION_ACCESS_VIOLATION
Error: Read address 0xCCCCCCD4
Address: 77436AC7

CallStack:
0x77410000[26AC7] QSMorphemeReleaseDLL.dll: (794CC260,79
```

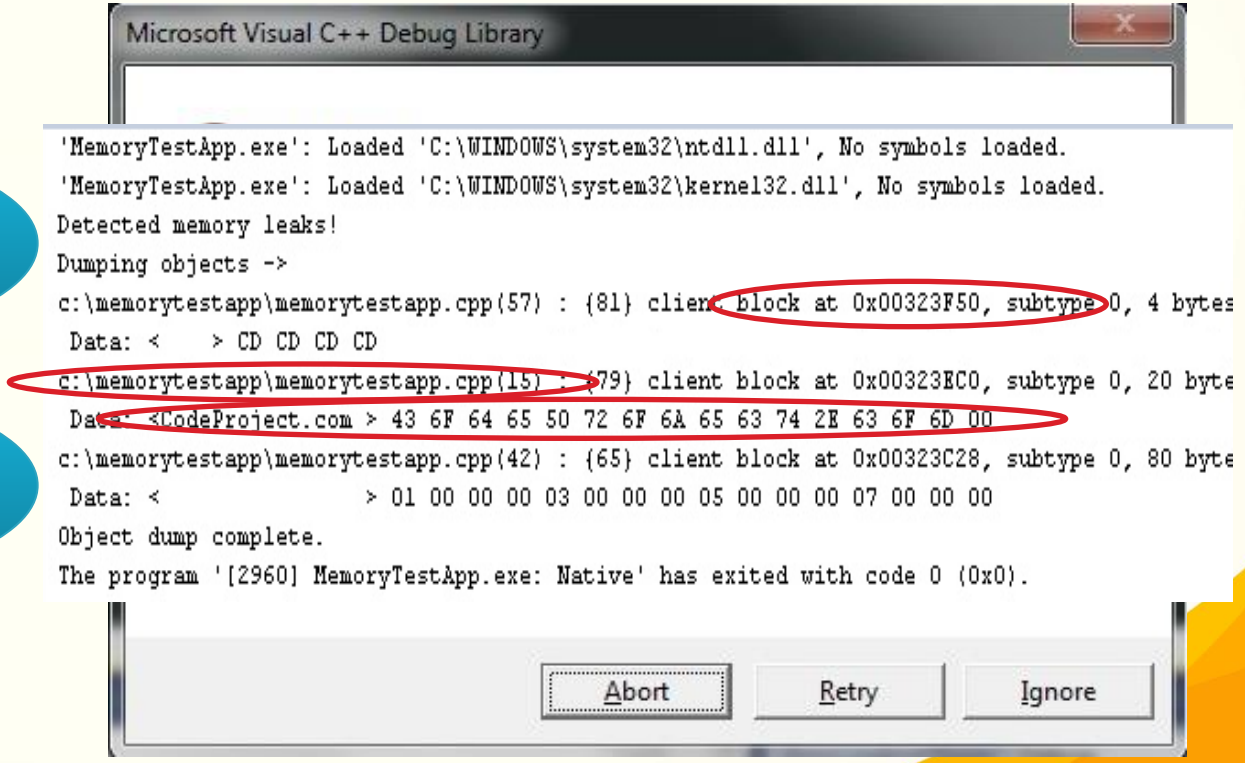
```
天涯明月刀 0.23.5569  15581684BBD695BEA140ACEBB0E50431
----------------------------------------
Type: EXCEPTION_ACCESS_VIOLATION
Error: Write address 0xDDDDDDE1
Address: 0FE29A88

CallStack:
0x0FDA0000[89A88] QSMorphemeReleaseDLL.dll: (3F14C952,793
```

检测方法 | 标记防护 | 库和工具 | 额外处理

CrtDbg

堆完整性检查

内存泄漏监测



检测方法 | 标记防护 | 库和工具 | 额外处理

Gflags / Stomp Allocator

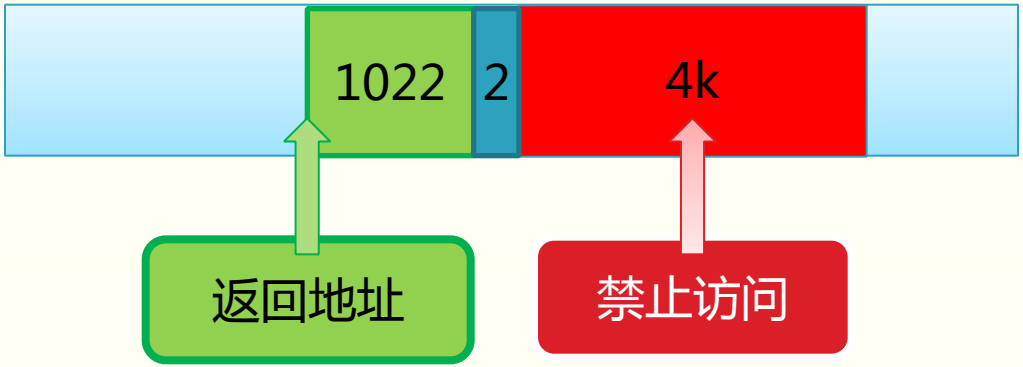
大致思路

1. 在申请的内存前或后放置“无法访问内存页”
2. 一旦“陷阱”被访问，直接抓住凶手现场

检测方法 | 标记防护 | 库和工具 | 额外处理

Gflags / Stomp Allocator

```
char *p = new char[1022];
```



浪费4k

p[-1] p[1022] p[1024] p[6000]

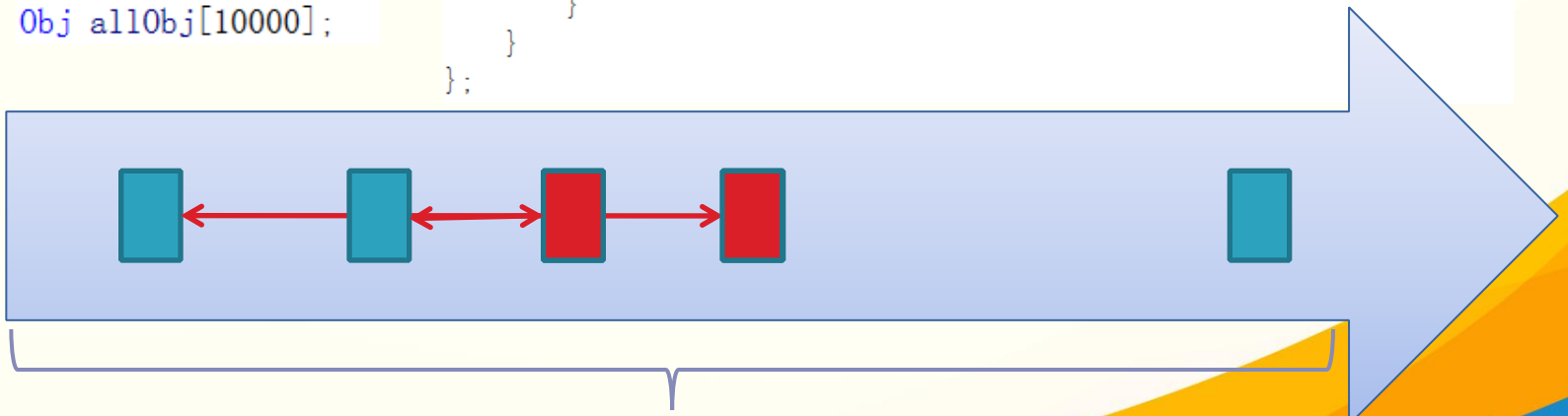
检测方法 | 标记防护 | 库和工具 | 额外处理

二分法

- ✓ 有固定特征的情形
- ✓ 容易重现

```
class Obj
{
    char *m_pData;
    int m_iDataLen;
};
Obj allObj[10000];
```

```
void CheckObj()
{
    for (int i=0;i<10000;i++)
    {
        if (allObj[i].m_pData == NULL && allObj[i].m_iDataLen > 0
            || allObj[i].m_pData != NULL && allObj[i].m_iDataLen == 0)
        {
            SHOW_ERROR;
        }
    }
};
```



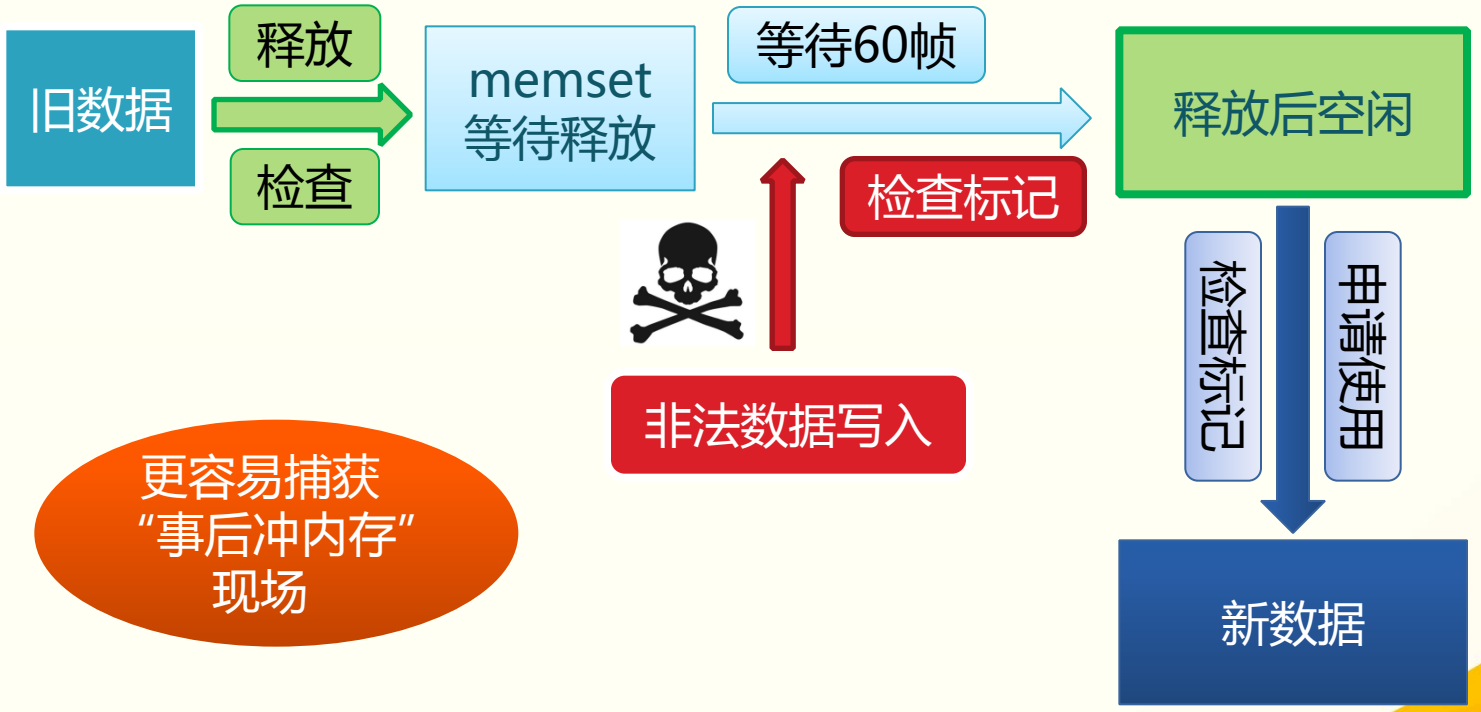
1帧

检测方法 | 标记防护 | 库和工具 | 额外处理

- ✓ 为什么大型游戏难以检查出？
  - ✓ 内存用量大，申请频繁

检测方法　标记防护　库和工具　额外处理

延时释放



检测方法 | 标记防护 | 库和工具 | 额外处理

“事后现场”的处理

1. 保存Full Dump & 分析

分析举例

0x3160	0xA12F		0x27A0	0xA258	0xB718
0xC328	0x3FC8		0x6648	0x9F20	0x5480
0xA438	0xA220	0xC758	0x879F		0x3A38
0xBB58	0xC360	0x7530	0xA3E8		0xF5D8
0x1240	0x61A8	0x86F8	0x5478	0xA750	0xA747
0xE530	0xAA78	0xD0B8	0x7850	0xB640	0xCDC0

940字节宽

940字节宽

检测方法 | 标记防护 | 库和工具 | 额外处理

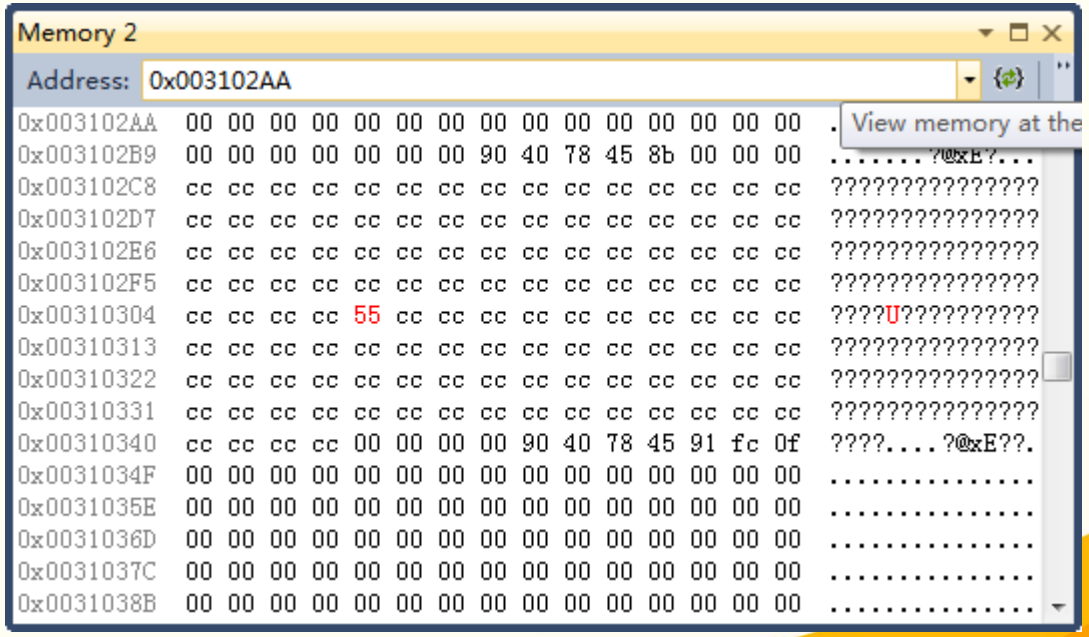
“事后现场”的处理

2. 设置数据断点

因为“凶手”有时候会“重返现场”

① 设置数据断点

② 等“凶手”再次犯案



检测方法　标记防护　库和工具　额外处理

延时释放

数据断点

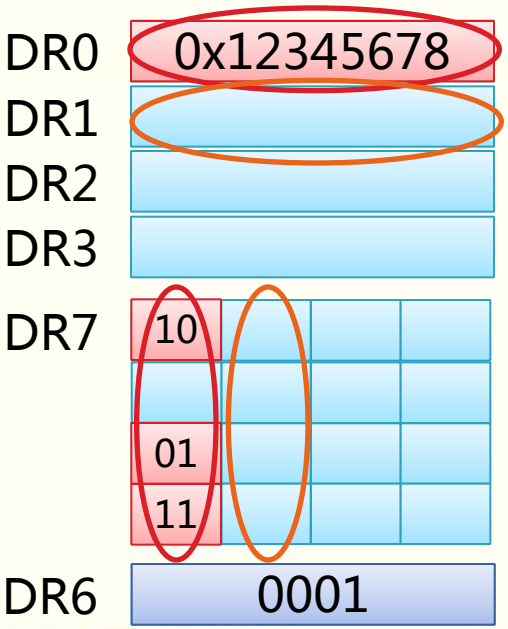
断线问题

现场保护问题

去除弹框

禁止释放使用

程序数据断点



SetThreadContext()

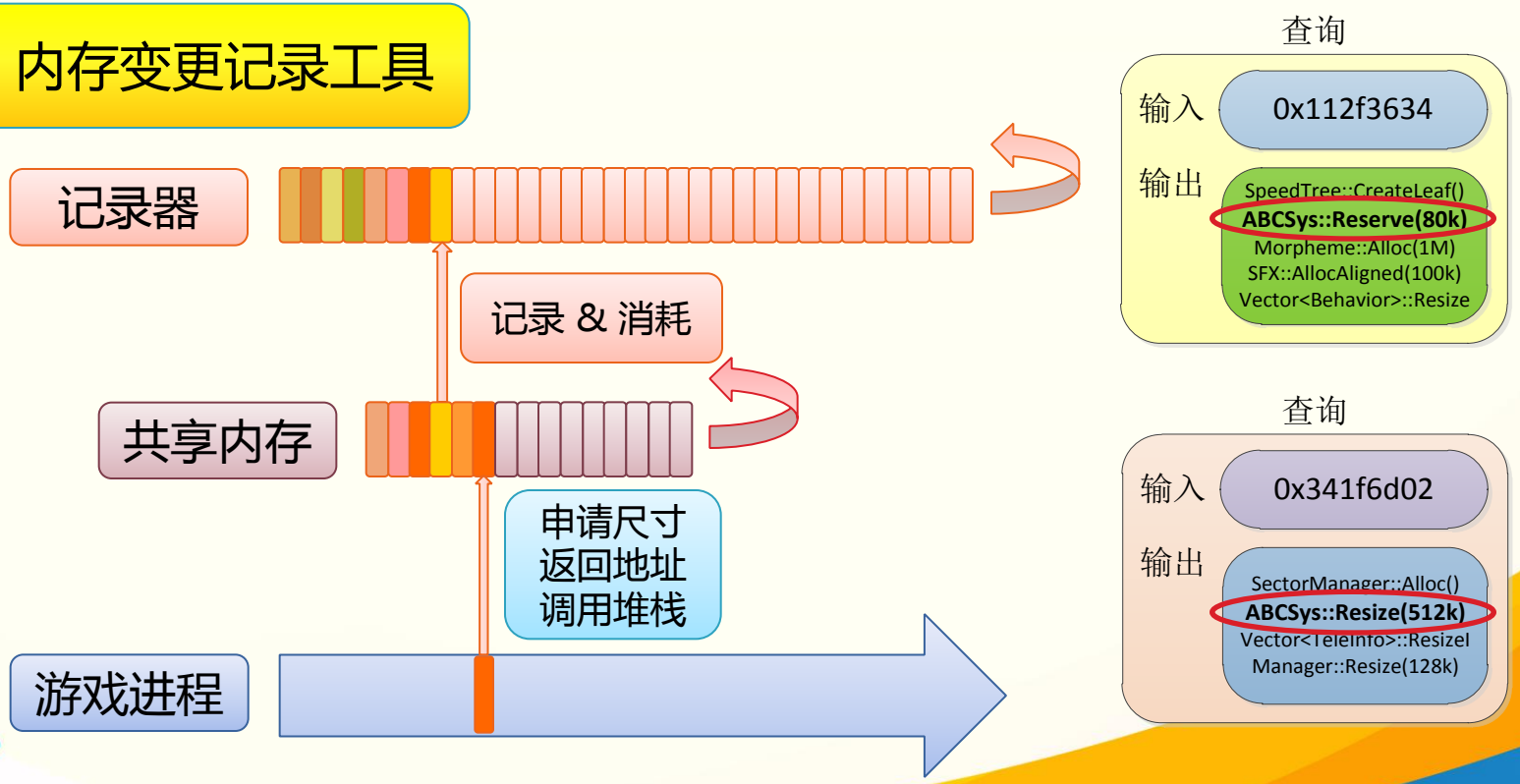
检测方法 | 标记防护 | 库和工具 | 额外处理

其他问题

很久很久以后才“犯案”（超过60帧）

只破坏一次，不“重返现场”

内存变更记录工具



分享内容

1. 内存性能的提升
2. 内存占用的优化
3. 冲内存问题的检测
    1. 常见冲内存方式
        1. 静态数组越界
        2. 动态数组越界
        3. 访问已释放内存
    2. 检测和处理方法
        1. 简易标记防护
        2. 使用调试库和工具
        3. 额外检查和处理

总结

1. 使用自定义内存管理
   - 提升性能、降低占用、跟踪调试
2. 充分利用各种工具
   - 内存统计、性能分析、问题调试
3. 维护好数据结构、挑选合适算法
   - 有益长期开发、节省内存、平衡空间和时间
4. 多添加Assert等断言检测
   - 预防难查问题、抓住问题源头
5. 建立完善的自动测试和报告流程
   - 保障版本稳定、自动定位问题
6. 做优化时在各个层次考虑方法
   - 上层需求、中层调度、下层算法、底层指令

谢 谢

Email: huanhe@tencent.com
QQ: 1815630