Luajit+unity性能优化

2016.12

GAME SCIENCE

lua与c#的交互

- c#与c（即lua环境）交互需要耗时
  - c#没有很好的适用于userdata的管理方法，一般做法是通过int映射，这产生额外的消耗
  - C#本身与c的交互性能也（意外地）不高
- 应对方法
  - 核心1：优化调用效率，《lua与c#交互篇》
  - 核心2：减少调用次数
  - lua与c的交互性能相对好一些，部分需要极高性能的库，可以直接用c实现

MAKE IT SIMPLE

为什么要用lua

- lua+unity方案的性能已经达到全面应用于部分类型游戏的级别
  - luajit可以完美集成到unity中并应用到所有主流平台
  - 卡牌、回合rpg等复杂度适中的游戏可以大规模用lua实现
  - moba、mmo、rts等复杂度较高的游戏可以部分应用lua，性能热点则仍然基于c#/c实现
- 强大的热更新能力
- lua有高性能的脚本语言实现：luajit
- il2cpp并不成熟

MAKE IT SIMPLE

GAME SCIENCE

从用好lua开始

- luajit对比原生lua
  - 如果你的项目关注性能，又要保证热更新覆盖率，那么luajit几乎是你唯一的选择
  - 相对更难破解的编码（当然，只是相对。。）
  - 但是luajit，又不是一个很容易掌控其性能的怪物

MAKE IT SIMPLE

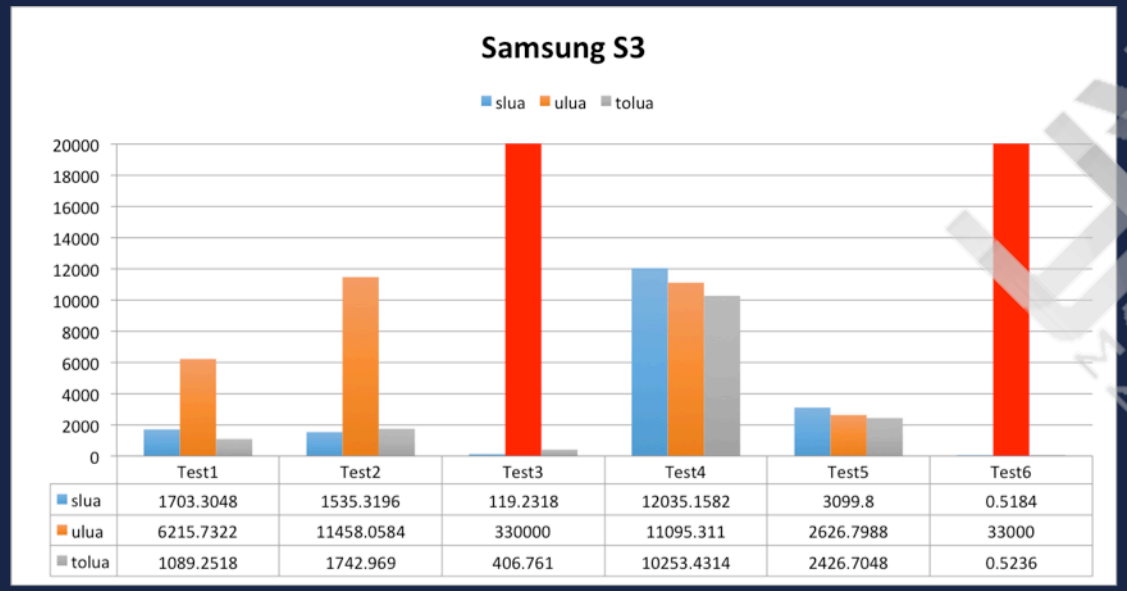
LUAJIT

- 你真的是在使用jit?
  - luajit有两个运行模式：interpreter，jit
  - ios并不支持jit，只能运行在interpreter下
- 安卓支持jit，但是所有代码都jit吗?
  - NO！！
  - jit的启动有一定条件
  - 大量的因素会破坏jit化，在游戏这种高频夸语言交互的案例上尤其明显

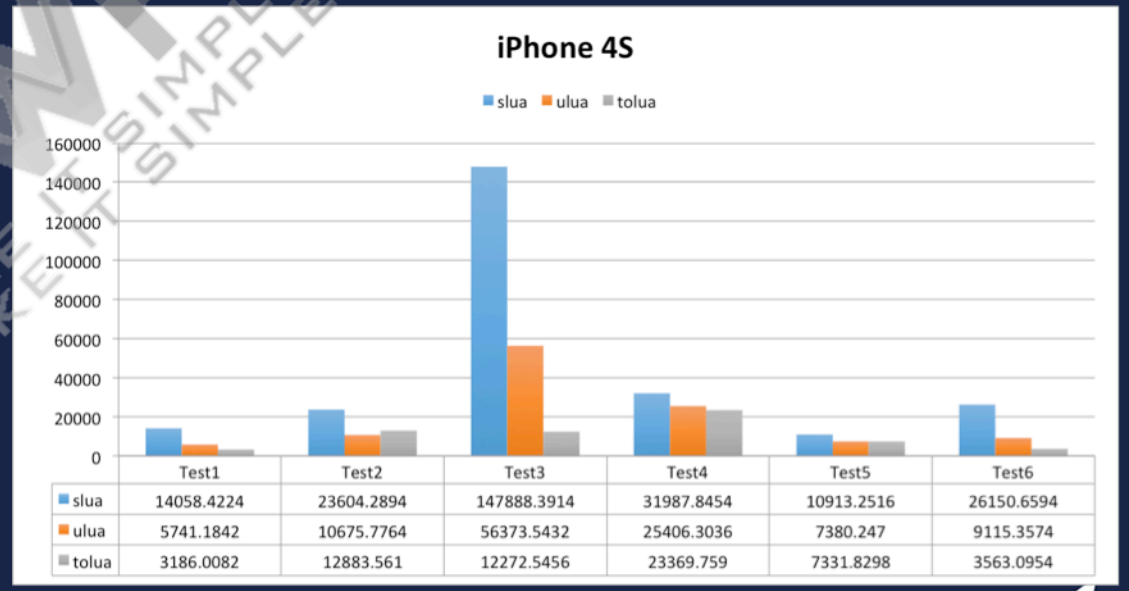
UWA MAKE IT SIMPLE

LUAJIT

- Jit真的更快?



	Test1	Test2	Test3	Test4	Test5	Test6
slua	1703.3048	1535.3196	119.2318	12035.1582	3099.8	0.5184
ulua	6215.7322	11458.0584	330000	11095.311	2626.7988	33000
tolua	1089.2518	1742.969	406.761	10253.4314	2426.7048	0.5236



	Test1	Test2	Test3	Test4	Test5	Test6
slua	14058.4224	23604.2894	147888.3914	31987.8454	10913.2516	26150.6594
ulua	5741.1842	10675.7764	56373.5432	25406.3036	7380.247	9115.3574
tolua	3186.0082	12883.561	12272.5456	23369.759	7331.8298	3563.0954

你并不认识的JIT

- 什么是Trace compiler?
  - 不会立即编译所有代码
  - 在跑了一段时间后，发现了hot code，才将其编译
  - 什么会是hot code？循环、hot call、hot exit

MAKE IT SMILE

你并不认识的JIT

- 运行时深度优化：a+b
  - a/b可以是number/table（__add）
  - +这个操作事前并不知道a和b是什么类型
  - 所以可以生成一个add cpu指令搞定这个加法吗？？

MAKE IT SIMPLE

```lua
local count = 1000000


local function testsum(a, b)
    for i = 1, count do
        a = a + b
    end

    return a
end


function testa()
    local a = 1
    local b = 2
    testsum(a,b)

    print (a)
end

```

```lua

local v3 = {
    __add = function(a, b)
        local ret = {v = a.v + b.v}
        return setmetatable(ret, v3)
    end
}

function testb()
    local a = {v=1}
    local b = {v=2}
    setmetatable(a, v3)
    setmetatable(b, v3)

    testsum(a,b)

    print (a.v)
end

testa()
testb()
```

UWA MAKE IT SIMPLE

GAME SCIENCE

```
1   ---- TRACE 1 start opttest.lua:5
2   0005  ADDVV    0   0   1
3   0006  FORL     2 => 0005
4   ---- TRACE 1 IR
5   0001 >  int SLOAD  #4    CRI
6   0002 >  int LE     0001  +2147483646
7   0003    int SLOAD  #3    CI
8   0004 >  num SLOAD  #1    T
9   0005 >  num SLOAD  #2    T
10  0006  + num ADD    0005  0004
11  0007  + int ADD    0003  +1
12  0008 >  int LE     0007  0001
13  0009 ------ LOOP ------------
14  0010  + num ADD    0006  0005
15  0011  + int ADD    0007  +1
16  0012 >  int LE     0011  0001
17  0013    int PHI    0007  0011
18  0014    num PHI    0006  0010
```

```
50  ---- TRACE 2 start 1/0 opttest.lua:6
51  0005  ADDVV    0   0   1
52  0000  . . FUNCF    6          ; opttest.lua:23
53  0001  . . TDUP     2   1
54  0002  . . TGETS    3   0   0  ; "v"
55  0003  . . TGETS    4   1   0  ; "v"
56  0004  . . ADDVV    3   3   4
57  0005  . . TSETS    3   2   0  ; "v"
58  0006  . . GGET     3   2      ; "setmetatable"
59  0007  . . MOV      4   2
60  0008  . . GGET     5   3      ; "v3"
61  0009  . . CALLT    3   3
62  0000  . . FUNCC               ; setmetatable
63  0006  JFORL    2   1
64  ---- TRACE 2 IR
65  0001 >  int SLOAD  #4    CRI
66  0002 >  int LE     0001  +2147483646
67  0003    int SLOAD  #3    CI
68  0004 >  tab SLOAD  #1    T
69  0005 >  tab SLOAD  #2    T
70  0006    tab FLOAD  0004  tab.meta
71  0007 >  tab EQ     0006  [NULL]
72  0008    tab FLOAD  0005  tab.meta
73  0009 >  tab NE     0008  [NULL]
74  0010    int FLOAD  0008  tab.hmask
75  0011 >  int EQ     0010  +1
76  0012    p32 FLOAD  0008  tab.node
77  0013 >  p32 HREFK  0012  "__add" @0
78  0014 >  fun HLOAD  0013
79  0015 >  fun EQ     0014  opttest.lua:23
80  0016 }+ tab TDUP   {0x1c9bcce0}
81  0017    int FLOAD  0004  tab.hmask
82  0018 >  int EQ     0017  +1
83  0019    p32 FLOAD  0004  tab.node
84  0020 >  p32 HREFK  0019  "v"  @1
85  0021 >  num HLOAD  0020
86  0022    int FLOAD  0005  tab.hmask
```

Luajit –jdump xxx.lua

你并不认识的JIT

- Jit复杂的行为
  - 运行时tracing
  - 根据tracing的结论进行优化
  - guard
  - Side tracing

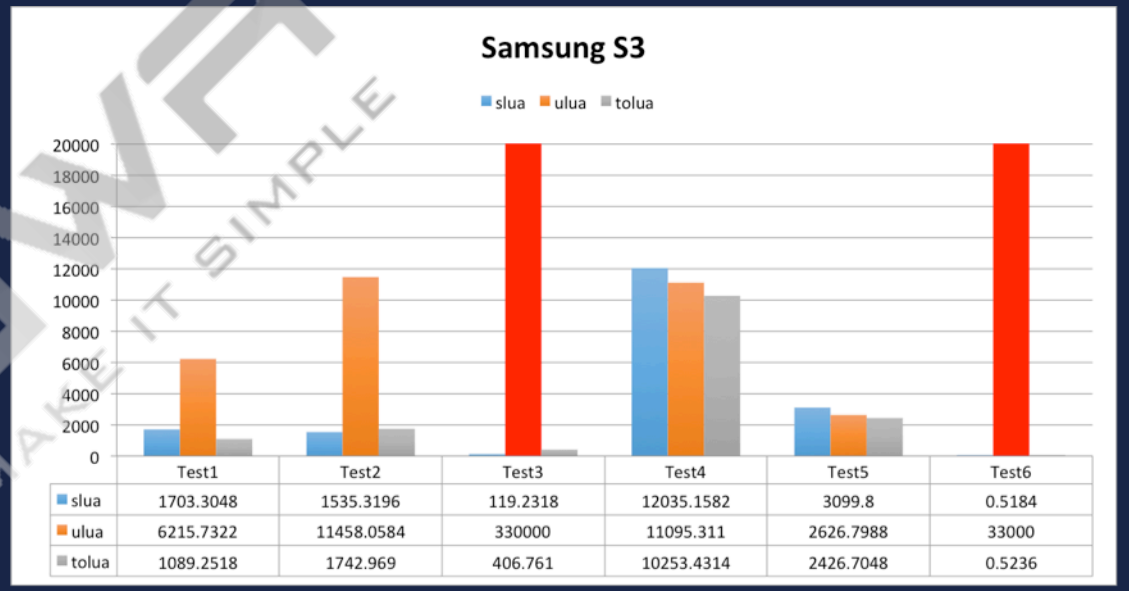
```lua
local count = 1000000


local function testsum(a, b)
    for i = 1, count do
        a = a + b
    end

    return a
end


function testa()
    local a = 1
    local b = 2
    testsum(a,b)

    print (a)
end

```

```
---- TRACE 1 start opttest.lua:5
0005  ADDVV    0   0   1
0006  FORL     2 => 0005
---- TRACE 1 IR
0001 >  int SLOAD  #4    CRI
0002 >  int LE     0001  +2147483646
0003    int SLOAD  #3    CI
0004 >  num SLOAD  #1    T
0005 >  num SLOAD  #2    T
0006  + num ADD    0005  0004
0007  + int ADD    0003  +1
0008 >  int LE     0007  0001
0009 ------ LOOP ------------
0010  + num ADD    0006  0005
0011  + int ADD    0007  +1
0012 >  int LE     0011  0001
0013    int PHI    0007  0011
0014    num PHI    0006  0010
```

MAKE IT SIMPLE

JIT

- 你并不认识的JIT
  - Jit失败
    - 内存分配失败
    - 调用c的代码
    - 不支持jit的指令
  - 对jit不友好的代码
    - 分支代码
    - C functions
    - 大量local变量



	Test1	Test2	Test3	Test4	Test5	Test6
slua	1703.3048	1535.3196	119.2318	12035.1582	3099.8	0.5184
ulua	6215.7322	11458.0584	330000	11095.311	2626.7988	33000
tolua	1089.2518	1742.969	406.761	10253.4314	2426.7048	0.5236

Luajit自身优化

- JIT和interpreter下的优化方法有巨大差异，有时甚至是矛盾的
  - ffi
  - Local缓存
- Interpreter
  - 优化方式与原生lua一致
- JIT
  - http://wiki.luajit.org/Numerical-Computing-Performance-Guide
  - 避免不可预测的分支代码（大量side trace跳转）
  - 充分利用ffi
  - 尽量使用for i=1,n do的循环形式
  - …

UWA MAKE IT SIMPLE

GAME SCIENCE

Luajit profiler

- p.lua
- require(“jit.p”).start("vl12m1i3")
- 可以统计cpu用时高的代码
- 可以发现jit效率低下的情况
- 支持editor/player/手机内启用（需要植入luajit的p.lua）
- 缺点：采样准确性稍差，无gui，需要修改luajit代码才能比较易用
  - 从标准输出导向到自定义log

MAKE IT SIMPLE

Luajit profiler

- C code：非lua时间（c/c#/unity）
- Jit compiler：如果这个值较高（高于5%）必须警惕，因为可能出现了较多jit失败的情况
  - 包含了编译、recording的时间
- Interpreted：未jit化的代码的耗时
- Compiled：已经jit化的代码的耗时

```
80%  C code
 -- 67%  [string]:69 (call lua frame update logic from c#)
 -- 16%  OnUpdate < OnUpdate < OnUpdate < NextFrame
 -- 5%  [string]:163 (not sure what's this)
 -- 3%  [string]:164
15%  JIT Compiler
 -- 51%  OnUpdate < OnUpdate < OnUpdate < OnUpdate
 -- 10%  OnUpdate < OnUpdate < [string]:69
 -- 7%  OnUpdate < [string]:69
 -- 5%  OnUpdate < OnUpdate < OnUpdate < NextFrame
 -- 3%  OnUpdate < OnUpdate < NextFrame < OnUpdate
5%  Interpreted
 -- 37%  OnUpdate < OnUpdate < OnUpdate < OnUpdate
 -- 13%  OnUpdate < OnUpdate < OnUpdate < NextFrame
 -- 5%  OnUpdate < OnUpdate < NextFrame < OnUpdate
 -- 5%  OnUpdate < OnUpdate < [string]:69
 -- 4%  OnUpdate < OnUpdate < OnUpdate < [string]:69
 -- 3%  GetIntAttr < OnUpdate < OnUpdate < OnUpdate
 -- 3%  OnUpdate < [string]:69
 -- 3%  __add < OnUpdate < OnUpdate < OnUpdate
 -- 3%  UpdateUnit < OnUpdate < OnUpdate < OnUpdate
```

Lua+IL2CPP

- IL2CPP会生成惊人大小的binary
- 自动生成的lua导出可能会产生大量代码，导致binary巨大，必须注意精简
- 一些可以考虑的方案
  - 修改自动导出，将相同的实现合并共享

MAKE IT SIMPLE

内存

- Lua table是最主要的内存占用体
  - 如果可以，优先使用连续数组，无论在访问速度还是内存占用都有绝对优势
  - 连续数组：8字节每item（TValue的大小）
  - Hash table：40字节左右每item，根据插入的内容和key的组织可能有很大出入
- 内存泄露追溯

MAKE IT SIMPLE

一些更高级的思路-多线程

- Skynet，服务器lua多线程的应用典型
- 我们使用的方案
  - 单个lua state无法在两个线程同时运行，但可以在不同时刻在其中一个线程运行
  - 将耗时但实时要求较低的逻辑隔离到一个独立update，在副线程执行，通过锁保证同一时刻主副线程只有一个在使用lua state
- lua的全量gc可以在副线程执行
- 大部分unity api不支持在副线程执行

GAME SCIENCE

一些更高级的思路-ffi

- Luajit独有的性能利器
  - 仅在jit开启时有速度优势，ios最好不要使用ffi
  - 使用ffi数据结构代替table，内存占用有绝对优势（可达1:10）
- 通过pinvoke的方式，可以将c#函数注册到c中，然后lua通过ffi调用c，来最终实现lua通过ffi调用c#
- 十分适用于优化核心性能代码
  - 基础数学运算结构，如vector3
  - 高频使用的C#、C导出

MAKE IT SIMPLE

GAME SCIENCE

一些更高级的思路-自行挖掘luajit

- 理解基本的luajit代码结构，掌握阅读其代码的能力
- 小testcase是测试语言特性的好工具
- 但是，luajit的特性之复杂，往往又会在testcase中欺骗你

MAKE IT SIMPLE

怎么看luajit代码

- 基于vs调试
  - 可以帮助理解string/table/tracing/recording/jit编译
- Interpreter：dasc文件
  - 基于汇编实现
  - 包含部分标准库的实现
  - 可对照官方wiki的指令列表
- Jit：lj_asm.c、lj_emit_*.h
  - 包含各个不同平台的指令生成
  - 借助luajit –jdump查看编译结果
- 借助luajit的工具
  - -bl
  - -jdump

UWP MAKE IT SIMPLE

GAME SCIENCE

最后的建议

- luajit是很伟大的产物，但对手游而言却不够完美
- luajit的jit模式不确定因素非常多，建议先以interpreter模式做到性能合格后再考虑jit
- 8020原则：luajit的优化虽然复杂但是只要在最核心的热点代码做好，就能获得很好的效果
- 当然，Lua/luajit的性能优化并不是一个容易做的工作，如果没有足够的把握，立项之初最好要有合理的代码结构，以便发现难以解决的性能瓶颈也可以将代码迁移到c#甚至c++

LWT MAKE IT SIMPLE

GAME SCIENCE

Thanks !

游戏科学

http://www.gamesci.com.cn

http://www.cnblogs.com/zwywilliam