1.Unity API

GameObject.GetComponent

Unity是基于组件的开发方式,所以GetComponent是一个高频使用的函数 每次调用GetComponent时,Unity都要去遍历所有的组件来找到目标组件 每次都去查找是不必要的耗费,可以通过缓存的方式来避免这些不必要的开销 其中Transform是用到最多的组件,GameObject内部提供了一个.transform来获取此组件 然而经过测试(2017.2.1p1)发现缓存的效率依然是最高的 所以若要经常访问一个特定组件,将其缓存

```
private Transform m_transform;
void Awake() {
    m_transform = transform;
}

void Start () {
    // 缓存的m_transform,耗时49ms
    for (int i = 0; i < 1000000; i++)
        m_transform.position = Vector3.one;

    // 内部属性transform,耗时77ms
    for (int i = 0; i < 1000000; i++)
        transform.position = Vector3.one;

    // 采用GetComponent,耗时102ms
    for (int i = 0; i < 1000000; i++)
        GetComponent<Transform>().position = Vector3.one;
}
```

GameObject.Find

GameObject.Find会遍历当前所有的GameObject来返回名字相符的对象 所以当游戏内对象很多时,这个函数将很耗时
可以通过缓存的方法,在Start或Awake时缓存一次找到的对象,在后续使用中使用缓存的对象而非继续调用GameObject.Find
或者采用GameObject.FindWithTag来寻找特定标签的对象 如果能在一开始就确定好对象,可以通过Inspector注入的方式,将对象直接拖到Inspector中,从而避免了运行时的查找

Camera.main

Camera.main用来返回场景中的主相机,Unity内部是通过GameObject.FindWithTag来查找tag为MainCamera的相机 当需要频繁访问主相机时,可以将其缓存以获得性能提升

```
private Camera m_mainCamera;
void Awake() {
    m_mainCamera = Camera.main;
}

void Start () {
    // 直接使用Camera.main,耗时164ms
    for (int i = 0; i < 1000000; i++)
        Camera.main.transform.position = Vector3.zero;

    // 采用缓存,耗时74ms
    for (int i = 0; i < 1000000; i++)
        m_mainCamera.transform.position = Vector3.zero;
}
```

GameObject.tag

GameObject.tag常用来比较对象的tag,但是直接采用.tag ==来进行对比的话,每一帧会产生GC Alloc 通过GameObject.CompareTag来进行比较则可以避免掉这些GC,但是前提是比较的tag需在Tag Manager中定义

Transform.SetPositionAndRotation

每次调用Transform.SetPosition或Transform.SetRotation时,Unity都会通知一遍所有的子节点 当位置和角度信息都可以预先知道时,可以通过Transform.SetPositionAndRotation一次调用来同时设置位置和角度,从而避免两次调用导致的性能开销

Animator.Set…

Animator提供了一系列类似于SetTrigger,SetFloat等方法来控制动画状态机 例如:m_animator.SetTrigger("Attack")是用来触发攻击动画 然而在这个函数内部,"Attack"字符串会被hash成一个整数 如果需要频繁触发攻击动画,可以通过Animator.StringToHash来提前进行hash,来避免每次的hash运算

```
// Hash once, use everywhere!
private static readonly int s_Attack = Animator.StringToHash("Attack");
m_animator.SetTrigger(s_Attack);
```

Material.Set…

与Animator类似,Material也提供了一系列的设置方法用于改变Shader 例如:m_mat.SetFloat("Hue", 0.5f)是用来设置材质的名为Hue的浮点数 同样可以通过Shader.PropertyToID来提前进行hash

```
private static readonly int s_Hue = Shader.PropertyToID("Hue");
m_mat.SetFloat(s_Hue, 0.5f);
```

Vector Math

如果需要比较距离,而非计算距离,用SqrMagnitude来替代Magnitude可以避免一次耗时的开方运算 在进行向量乘法时,有一点需要注意的是乘法的顺序,因为向量乘比较耗时,所以应该尽可能的减少向量乘法运算

```
// 耗时:73ms
for (int i = 0; i < 1000000; i++)
    Vector3 c = 3 * Vector3.one * 2;

// 耗时:45ms
for (int i = 0; i < 1000000; i++)
    Vector3 c = 3 * 2 * Vector3.one;
```

可以看出上述的向量乘法的结果完全一致,但是却有显著的耗时差异,因为后者比前者少了一次向量乘法 所以,应该尽可能合并数字乘法,最后再进行向量乘

Coroutine

Coroutine是Unity用来实现异步调用的机制,如果对其不够了解可以参考对Unity中Coroutines的理解 当需要实现一些定时操作时,若在Update中每帧进行一次判断,假设帧率是60帧,需要定时1秒调用一次,则会导致59次无效的Update调用 用Coroutine则可以避免掉这些无效的调用,只需要yield return new WaitForSeconds(1f);即可 当然这里的最佳实践还是用一个变量缓存一下new WaitForSeconds(1f),这样省去了每次都new的开销

SendMessage

SendMessage用来调用MonoBehaviour的方法,然而其内部采用了反射的实现机制,时间开销异常大,需要尽量避免使用 可以用事件机制来取代它

Debug.Log

输出Log是一件异常耗时,而且玩家感知不到的事情 所以应该在正式发布版本时,将其关闭 Unity的Log输出并不会在Release模式下被自动禁用掉,所以需要手动来禁用 可以在运行时用一行代码来禁用Log的输出:Debug.logger.logEnabled = false; 不过最好采用条件编译标签Conditional封装一层自己的Log输出,来直接避免掉Log输出的编译,还可以省去Log函数参数传递和调用的开销 具体参见:Unity3D研究院之在发布版本屏蔽Debug.log输出的Log

2.Csharp

反射

反射是一项异常耗时的操作,因为其需要大量的有效性验证而且无法被编译器优化 而且反射在iOS下还可能存在不能通过AOT的情况,所以应该尽量避免使用反射 可以自己建立一个字符串-类型的字典来代替反射,或者采用delegate的方式来避免反射

内存分配(栈和堆)

在C#中,内存分配有两种策略,一种是分配在栈StackStack上,另一种是分配在堆HeapHeap上 在栈上分配的对象都是拥有固定大小的类型,在栈上分配内存十分高效 在堆上分配的对象都是不能确定其大小的类型,由于其内存大小不固定,所以经常容易产生内存碎片,导致其内存分配相对于栈来说更为低效

值类型和引用类型

在C#中,数据可以分为两种类型:值类型ValueTypeValueType和引用类型ReferenceTypeReferenceType 值类型包括所有数字类型,Bool,Char,Date,所有Struct类型和枚举类型 其类型的大小都是固定,它们都在栈上进行内存分配 引用类型包括字符串,所有类型的数组,所有Class以及Delegate,它们都在堆上进行内存分配

装箱

装箱Boxing指的是将值类型转换为引用类型,而拆箱UnBoxing的是将引用类型转换为值类型装箱和拆箱存在着从栈到堆的互指以及堆内存的开辟,所以它们本质是一项非常耗时的操作,应该尽量避免之

垃圾回收

在堆上分配的内存,其实是由垃圾回收器(Garbage Collector)来负责回收的 垃圾回收算法异常耗时,因为它需要遍历所有的对象,然后找到没有引用的孤岛,将它们标记为「垃圾」,然后将其内存回收掉 频繁的垃圾回收不仅很耗时,还会导致内存碎片的产生,使得下一次的内存分配变得更加困难或者干脆无法分配有效内存,此时堆内存上限会往上翻一倍,而且无法回落,造成内存吃紧 所以应该极力避免GC Alloc,即需要控制堆内存的分配

字符串

字符串连接会导致GC Alloc,例如string gcalloc = "GC" + "Alloc"会导致"GC"变成垃圾,从而产生GC Alloc 又比如:string c = string.Format("one is {0}", 1),也会因为一次装箱操作(数字1被装箱成字符串"1")而产生额外的GC Alloc 所以如果字符串连接是高频操作,应该尽量避免使用+来进行字符串连接 C#提供了StringBuilder类来专门进行字符串的连接

IL2CPP

I2LCPP是Unity提供的将C#的IL码转换为C++代码的服务,由于转成了C++,所以其最后会转换成汇编语言,直接以机器语言的方式执行,而不需要跑在.NET虚拟机上,所以提高了性能 同时由于IL的反编译较为简单,转换成C++后,也会增加一定的反汇编难度 IL2CPP的C++代码虽然是自动生成的,但是其中间的某些过程也可以被人为操纵,从而达到提升性能的目的

Sealed修饰

在C#中,虚函数的调用会比直接调用开销更大,可以用sealed修饰符来修饰掉那些确保不会被继承的类或函数

避免自动判空

在自动转换的C++代码中,IL2CPP默认会对所有Nullable的变量做判空 在某些非常确定参数不为空的场合,这种检测是不必要的 具体步骤是复制Il2CppSetOptionAttribute.cs文件到Assets目录下,然后在类或者函数定义上加一个修饰语句[Il2CppSetOption(Option.NullChecks, false)]即可以禁用整个类或者函数的判空检测

避免数组越界检测

同理,IL2CPP也会默认对所有数组的读写做越界检测,可以通过修饰语句[Il2CppSetOption(Option.ArrayBoundsChecks, false)]来将其禁用

3.数据结构

容器类型

容器应该针对不同的使用场合进行选择,主要看使用场合哪种操作的频率较高 例如:

- 经常需要进行随机下标访问的场合,优先选择数组Array)或列表(List)
- 经常需要进行查找的场合,优先选择字典(Dictionary)
- 经常需要插入或删除的场合,优先选择链表(LinkedList)
- 还有一些特殊的数据结构,适用于特殊的使用场合 例如:
- 不能存在相同元素的,可以选择HashSet
- 需要后进先出的,用来优化递归函数调用的,可以选择Stack
- 需要先进先出的,可以选择Queue

对象池

对象池(Object Pool)可以避免频繁的对象生成和销毁 游戏对象的生成,首先需要开辟内存,其次还可能会引起GC Alloc,最后还可能会引发磁盘I/O 频繁的销毁对象会引发严重的内存碎片,使得堆内存的分配更加

困难

所以在有大量对象需要重复生成和销毁时,一定要采用对象池来缓存好创建的对象,等到它们无需使用时,不需要将其销毁,而是将其放入对象池中,可以免去下次的生成

```
public class ObjectPool<T> where T : new()
{
    private Stack<T> objs;

    public ObjectPool(){
        objs = new Stack<T>();
    }
    // 获取对象池里的对象
    public T GetObject(){
        T obj = objs.Count > 0 ? objs.Pop() : new T();
        return obj;
    }
    // 回收对象池里的对象
    public void ReturnObject(T obj){
        if (obj != null)
            objs.Push(obj);
    }
}
```

空间划分

在计算空间碰撞或者寻找最近邻居时,如果空间很庞大,需要参与计算的对象太多的情况下,用两层循环逐个遍历去计算的复杂度为平方级 可以借助于空间划分的数据结构来使复杂度降低到N*Log(N) 四叉树一般用来划分2D空间,八叉树一般用来划分3D空间,而KD树则是不限空间维度 具体参考:KD树的应用与优化

4.算法

循环

循环的使用非常常见,也非常容易成为性能热点 应该尽量避免在循环内进行耗时或无效操作,尤其是这个循环在每帧的Update调用中时

```
void Update() {
    for (int i = 0; i < count; i++)
        if (condition)
            excuteFunc(i);
```

以上的循环遍历中,无论condition为真或者为假,循环都会执行count次,若condition为假,则相当于白跑了count次

```
void Update() {
    if (condition)
        for (int i = 0; i < count; i++)
                excuteFunc(i);
}
```

将判断条件提出循环外,则可以避免白跑了的问题

另一个需要注意的是小心多重循环的顺序问题,应该尽量把遍历次数较多的循环放在内层

```
void Start() {
    // 耗时:37ms
    for (int i = 0; i < 1000000; i++)
        for (int j = 0; j < 2; j++)
            int k = i * j;
    // 耗时:13ms
    for (int i = 0; i < 2; i++)
        for (int j = 0; j < 1000000; j++)
            int k = i * j;
}
```

当内外层循环数有较多数量级上的差别时,将忙的循环放在内层性能更高,因为其避免了更多次内层循环计数器初始化的调用

数学运算

开方运算,三角函数这些都是耗时的数学运算,应尽量避免之 如果只是单纯比较距离而不是计算距离的话,就可以用距离的平方来表示,可以节约掉一次耗时的开方运算

三角运算可以通过简单的向量运算来规避之,具体参考:向量运算在游戏开发中的应用和思考

又比如如果经常需要除一个常数,比如用万分位整数来表示小数需要经常除10000,可以改成乘0.0001f,可以规避掉较乘法更为耗时的除法运算 实际验算证明,现代的编译器会对此进行优化,所以没有必要为此牺牲可读性 很多时候还是要先测算再去写代码会比较好

缓存

缓存的本质就是用空间换时间 例如之前在Unity API中提到的很多耗时的函数,都可以用缓存来提升性能 包括对象池,也是缓存技术的一种 针对于需要依赖复杂运算而且后续要经常用到值,可将其缓存起来,以避免后续的计算,从而获取性能提升