# C#基础：

## 值类型与引用类型的区别

* 值类型的数据存储在内存的栈中；引用类型的数据存储在内存的堆中，而内存单元中只存放堆中对象的地址。
* 值类型存取速度快，引用类型存取速度慢。
* 值类型表示实际数据，引用类型表示指向存储在内存堆中的数据的指针或引用
* 值类型继承自System.ValueType，引用类型继承自System.Object
* 值类型的变量直接存放实际的数据，而引用类型的变量存放的则是数据的地址，即对象的引用。
* 值类型变量直接把变量的值保存在堆栈中，引用类型的变量把实际数据的地址保存在堆栈中，而实际数据则保存在堆中。注意，堆和堆栈是两个不同的概念，在内存中的存储位置也不相同，堆一般用于存储可变长度的数据，如字符串类型;而堆栈则用于存储固定长度的数据，如整型类型的数据int(每个int变量占用四个字节)。由数据存储的位置可以得知，当把一个值变量赋给另一个值变量时，会在堆栈中保存两个完全相同的值;而把一个引用变量赋给另一个引用变量，则会在堆栈中保存对同一个堆位置的两个引用，即在堆栈中保存的是同一个堆的地址。在进行数据操作时，对于值类型，由于每个变量都有自己的值，因此对一个变量的操作不会影响到其它变量;对于引用类型的变量，对一个变量的数据进行操作就是对这个变量在堆中的数据进行操作，如果两个引用类型的变量引用同一个对象，实际含义就是它们在堆栈中保存的堆的地址相同，因此对一个变量的操作就会影响到引用同一个对象的另一个变量。
* 值类型主要由两类组成：结构、枚举； 结构分为以下几类：Numeric（数值）类型、整型、浮点型、decimal、bool、用户定义的结构。
* 引用类型的变量又称为对象，可存储对实际数据的引用。声明引用类型的关键字：class、interface、delegate、内置引用类型： object、string

## 结构体和类有何区别？

结构体是一种值类型，而类是引用类型。（值类型、引用类型是根据数据存储的角度来分的）就是值类型用于存储数据的值，引用类型用于存储对实际数据的引用。那么结构体就是当成值来使用的，类则通过引用来对实际数据操作

## 如何理解委托

委托类似于 C++ 函数指针，但它是类型安全的。委托允许将方法作为参数进行传递。委托可用于定义回调方法。委托可以链接在一起；例如，可以对一个事件调用多个方法。方法不需要与委托签名精确匹配。有关更多信息，请参见协变和逆变。C# 2.0 版引入了匿名方法的概念，此类方法允许将代码块作为参数传递，以代替单独定义的方法。

## C#中的接口和类有什么异同？

异：接口不能直接实例化。

　　接口不包含方法的实现。

　　接口、类和结构可从多个接口继承。但是C#类只支持单继承：类只能从一个基类继承实现。类定义可在不同的源文件之间进行拆分。

同：接口、类和结构可从多个接口继承。

　　接口类似于抽象基类：继承接口的任何非抽象类型都必须实现接口的所有成员。

接口可以包含事件、索引器、方法和属性

## 请描述Interface与抽象类之间的不同

抽象类表示该类已经有一些具体的实现，而接口只是定义各方法，并没有具提实现，子类继承抽象父类，可以只是继承部分方法或直接引用父类的方法。类实现接口必须实现该接口所有方法

## 如何理解.net中的垃圾回收机制？

.NETFramework 的垃圾回收器管理应用程序的内存分配和释放。每次您使用 new 运算符创建对象时，运行库都从托管堆为该对象分配内存。只要托管堆中有地址空间可用，运行库就会继续为新对象分配空间。但是，内存不是无限大的。最终，垃圾回收器必须执行回收以释放一些内存。垃圾回收器优化引擎根据正在进行的分配情况确定执行回收的最佳时间。当垃圾回收器执行回收时，它检查托管堆中不再被应用程序使用的对象并执行必要的操作来回收它们占用的内存

## 概述.NET中的GC机制。

GC的全称是garbage collection，中文名称垃圾回收，是.NET中对内存管理的一种功能。垃圾回收器跟踪并回收托管内存中分配的对象，定期执行垃圾回收以回收分配给没有有效引用的对象的内存。当使用可用内存不能满足内存请求时，GC会自动进行。

在进行垃圾回收时，垃圾回收器会首先搜索内存中的托管对象，然后从托管代码中搜索被引用的对象并标记为有效，接着释放没有被标记为有效的对象并收回内存，最后整理内存将有效对象挪动到一起

## **GC是什么? 为什么要有GC?** GC是垃圾收集器。程序员不用担心内存管理，因为垃圾收集器会自动进行管理。要请求垃圾圾收集，可以调用下面的方法之一：  System.gc() Runtime.getRuntime().gc()

## 接口是否可以继承接口？抽象类是否可以实现接口？抽象类是否可以继承实体类？

接口是可以继承接口的，抽象类是可以实现接口的，抽象类可以继承实体类，但是有个条件，条件是，实体类必须要有明确的构造函数。

## 构造器Constructor是否可以被继承？是否可以被Override?

Constructor不可以被继承，因此不能被重写（Overriding），但可以被重载(Overloading).

## HashMap和Hashtable区别？

HashMap是Hashtable的轻量级实现，非线程安全的实现他们都实现了map接口，主要区别是HashMap键值可以为空null,效率可以高于Hashtable。

## Collection和Collections的区别？

Collection是集合类的上级接口，Collections是针对集合类的一个帮助类，它提供一系列静态方法来实现对各种集合的搜索，排序，线程安全化操作。

## Override, Overload,的区别？

Override是重写的意思，它表示重写基类的方法，而且方法的名称，返回类型，参数类型，参数个数要与基类相同。

Overload是重载是意思，它也表示重写基类的方法，但是只要方法名相同，别的可以不同。

## UDP和TCP连接有和异同？

TCP是传输控制协议，提供的是面向连接的，是可靠的，字节流服务，当用户和服务器彼此进行数据交互的时候，必须在他们数据交互前要进行TCP连接之后才能传输数据。TCP提供超时重拨，检验数据功能。

UDP是用户数据报协议，是一个简单的面向数据报的传输协议，是不可靠的连接

两者的区别：

第一种：

TCP面向连接，在发送数据之前，需要客户端和服务器之间建立连接，而UDP不需要实时连接，在每次传输时根据IP和端口号将报文发送给指定的用户。

TCP中有数据安全校验，更为安全和复杂，能够保障数据正确的传输，且有重发机制，能提供可靠的传输。

UDP结构较为简单，开销小，传输速度快，但不保证传输数据的正确性，因此UDP的传输不可靠。

但随着网络的不断升级，数据在传输过程中差错率已大大降低，UDP可以一对多的传输，且其程序开销小，传输速度快，因此其应用也越来越广泛。

第二种：

1.TCP提供的是面向连接的、可靠的数据流传输；UDP提供的是非面向连接的、不可靠的数据流传输。

2.TCP提供可靠的服务，通过TCP连接传送的数据，无差错、不丢失，不重复，按序到达；UDP尽最大努力交付，即不保证可靠交付。

3.TCP面向字节流；UDP面向报文。

4.TCP连接只能是点到点的；UDP支持一对一、一对多、多对一和多对多的交互通信。

5.TCP首部开销20字节；UDP的首部开销小，只有8个字节。

6.TCP的逻辑通信信道是全双工的可靠信道；UDP的逻辑通信信道是不可靠信道。

## ArrayList和List<Int>的主要区别

ArrayList 是一个数组，继承实现了List基类，是比较传统使用的程序数组变量  
List<\*>是调用List<T>，用了C#的泛型特性

## Unity3D的协程和C#线程之间的区别是什么？

线程不安全  
同一个时刻只有一个coroutinue运行，但线程可以并行执行，所谓多线程。而coroutinue可以让出当前执行权限，让其他coroutinue插队执行  
golang里面是goroutinue  
lua里面也是协程

Python既提供协程，也提供线程

# LUA基础：

## pairs与ipairs的区别？

* pairs: 迭代 table，可以遍历表中所有的 key 可以返回 nil
* ipairs: 迭代数组，不能返回 nil,如果遇到 nil 则退出

## Table的底层实现机制？

C++底层是结构体，内部含2中存储结构数组，哈希表。

整数key，会用array来存放，而其它数据类型key会存放在哈希表上

## Table底层空间是怎样扩展的？

无论是array还是hash表，都是以2的倍数进行扩展的。

比较有区别的是，array数组sizearray记录的是真实大小，而hash表的lsizenode记录的是2的倍数。当hash表空间满的时候，才会重新分配array和hash表。

比较重要的两个函数是rehash和resize，前一个是重新算出要分配的空间，后一个是创建空间。

## 闭包是什么？

通过调用含有一个内部函数加上该外部函数持有的外部局部变量（upvalue）的外部函数（就是工厂）产生的一个实例函数

## Require、dofile、loadfile的区别？

* require（moduleName）：用来加载模块；其参数只要是文件名即可；利用模式匹配搜索加载文件；require只会被调用执行一次，会判断文件是否被加载避免重复加载；
* dofile(modulename) 参数必须带上文件名后缀 而且需要文件所在的绝对路径
  + 它得主要功能：读入加载的文件进行编译和执行 每调用一次dofile 文件都会被编译和执行一次
* loadfile(modulename) 只编译模块 但是不执行模块 loadfile将模块编译之后当作函数返回

# 数学基础：

## 简述四元数的作用，四元数对欧拉角的优点？

增量旋转

避免万向锁

给定方位的表达式有2种，互为正负（欧拉角有多种）

## 向量的点乘、叉乘以及归一化的意义？

1）点乘描述了2个方向的相似程度

2） 叉乘得到垂直于这2个的向量的一个向量

3）标准化向量，在一些方向，角度求解中应用，只关心相互间的方位，不考虑长度

## 矩阵相乘的意义及注意点

意义在于向量的变换，旋转，投影，平移

左乘和右乘，不满足交换律（这个答法比较高校书本式，缺少经验的说法）

# Unity：

## FixUpdate与Update的区别？

update跟当前平台的帧数有关，而FixedUpdate是真实时间，所以处理物理逻辑的时候要把代码放在FixedUpdate而不是Update。

Update是在每次渲染新的一帧的时候才会调用，也就是说，这个函数的更新频率和设备的性能有关以及被渲染的物体（可以认为是三角形的数量）。在性能好的机器上可能fps 30，差的可能小些。这会导致同一个游戏在不同的机器上效果不一致，有的快有的慢。因为Update的执行间隔不一样了。

而FixedUpdate，是在固定的时间间隔执行，不受游戏帧率的影响。有点像Tick。所以处理Rigidbody的时候最好用FixedUpdate。

PS：FixedUpdate的时间间隔可以在项目设置中更改，Edit->ProjectSetting->time  找到Fixedtimestep。就可以修改了。

## Resources与AssetsBundle的区别，以及优缺点

Resources构建项目的时候，所有的Resources目录下的文件会被合并为一个序列化文件。该文件会有自己的metadata信息和索引信息。内部用红黑树实现资源查找，用于索引相应的File GUID和Local ID，并且它还要记录在序列化文件中的偏移量。

AssetBundle就像传统的压缩包一样，由两个部分组成：包头和数据段。

包头包含有关AssetBundle的信息，比如标识符、压缩类型和内容清单。清单是一个以Objects name为键的查找表。每个条目都提供一个字节索引，用来指示该Objects在AssetBundle数据段的位置。在大多数平台上，这个查找表是用平衡搜索树实现的。具体来说，Windows和OSX派生平台（包括iOS）都采用了红黑树。因此，构建清单所需的时间会随着AssetBundle中Assets的数量增加而线性增加。

Resources仅支持同步加载

AssetsBundle支持同步和异步

Resources的缺点：

Resources内的资源会增加应用程序的启动时间和构建时长。

Resources内的资源无法增量更新，这是现在手机游戏开发的致命点。

AssetBundles的缺点，比如：

* 无法直观地看到包内的资源情况。
* 异步加载，需要写比较繁琐的回调处理。
* 调试的时候，无法通过Hierarchy直接定位到资源。
* 使用之前需要花费时间进行打包，尤其是在开发的时候，调整资源频繁，如果忘记打包可能导致Bug。

## LOD是什么，优缺点是什么？

LOD(Level of detail)多层次细节，是最常用的游戏优化技术。它按照模型的位置和重要程度决定物体渲染的资源分配，降低非重要物体的面数和细节度，从而获得高效率的渲染运算。缺点是增加了内存。

## MipMap是什么，作用？

在三维计算机图形的贴图渲染中有常用的技术，为加快渲染进度和减少图像锯齿，贴图被处理成由一系列被预先计算和优化过的图片组成的文件，这样的贴图被称为MipMap。

## GPU的工作原理

* 顶点处理：这阶段GPU读取描述3D图形外观的顶点数据并根据顶点数据确定3D图形的形状及位置关系，建立起3D图形的骨架。在支持DX8和DX9规格的GPU中，这些工作由硬件实现的Vertex Shader（定点着色器）完成。
* 光栅化计算：显示器实际显示的图像是由像素组成的，我们需要将上面生成的图形上的点和线通过一定的算法转换到相应的像素点。把一个矢量图形转换为一系列像素点的过程就称为光栅化。例如，一条数学表示的斜线段，最终被转化成阶梯状的连续像素点。
* 纹理帖图：顶点单元生成的多边形只构成了3D物体的轮廓，而纹理映射（texture mapping）工作完成对多变形表面的帖图，通俗的说，就是将多边形的表面贴上相应的图片，从而生成“真实”的图形。TMU（Texture mapping unit）即是用来完成此项工作。
* 像素处理：这阶段（在对每个像素进行光栅化处理期间）GPU完成对像素的计算和处理，从而确定每个像素的最终属性。在支持DX8和DX9规格的GPU中，这些工作由硬件实现的Pixel Shader（像素着色器）完成。
* 最终输出：由ROP（光栅化引擎）最终完成像素的输出，1帧渲染完毕后，被送到显存帧缓冲区。

## 什么是渲染管道？

是指在显示器上为了显示出图像而经过的一系列必要操作。 渲染管道中的很多步骤，都要将几何物体从一个坐标系中变换到另一个坐标系中去。主要步骤有：

本地坐标->视图坐标->背面裁剪->光照->裁剪->投影->视图变换->光栅化

# 优化方面：

## CPU优化：

主要可归结为两大类：引擎模块性能开销和自身代码性能开销。

其中，引擎模块中又可细致划分为渲染模块、动画模块、物理模块、UI模块、粒子系统、加载模块和GC调用等等。

### 渲染模块

渲染模块可以说是任何游戏中最为消耗CPU性能的引擎模块，因为几乎所有的游戏都离不开场景、物体和特效的渲染。对于渲染模块的优化，主要从以下两个方面入手：

（1）降低Draw Call Draw Call是渲染模块优化方面的重中之重，一般来说，Draw Call越高，则渲染模块的CPU开销越大。至于原因[这里](http://stackoverflow.com/questions/4853856/why-are-draw-calls-expensive)

降低Draw Call的方法则主要是减少所渲染物体的材质种类，并通过Draw Call Batching来减少其数量。

（2）简化资源

简化资源是非常行之有效的优化手段。在大量的移动游戏中，其渲染资源其实是“过量”的，过量的网格资源、不合规的纹理资源等等。

关于渲染模块在CPU方面的优化方法还有很多，比如LOD、Occlusion Culling和Culling Distance等等。

### UI模块

在NGUI的优化方面，UIPanel.LateUpdate为性能优化的重中之重，它是NGUI中CPU开销最大的函数，没有之一。

对于UIPanel.LateUpdate的优化，主要着眼于UIPanel的布局，其原则如下：

* 尽可能将动态UI元素和静态UI元素分离到不同的UIPanel中（UI的重建以UIPanel为单位），从而尽可能将因为变动的UI元素引起的重构控制在较小的范围内；
* 尽可能让动态UI元素按照同步性进行划分，即运动频率不同的UI元素尽可能分离放在不同的UIPanel中；
* 控制同一个UIPanel中动态UI元素的数量，数量越多，所创建的Mesh越大，从而使得重构的开销显著增加。

### 加载模块

加载模块的性能开销比较集中，主要出现于场景切换处，且CPU占用峰值均较高。

场景切换时的主要性能开销主要体现在两个方面，前一场景的场景卸载和下一场景的场景加载。

* 场景卸载  
  对于Unity引擎而言，场景卸载一般是由引擎自动完成的，即当我们调用类似Application.LoadLevel的API时，引擎即会开始对上一场景进行处理，其性能开销主要被以下几个部分占据：
* Destroy  
  引擎在切换场景时会收集未标识成“DontDestoryOnLoad”的GameObject及其Component，然后进行Destroy。同时，代码中的OnDestory被触发执行，这里的性能开销主要取决于OnDestroy回调函数中的代码逻辑。
* Resources.UnloadUnusedAssets  
  一般情况下，场景切换过程中，该API会被调用两次，一次为引擎在切换场景时自动调用，另一次则为用户手动调用（一般出现在场景加载后，用户调用它来确保上一场景的资源被卸载干净）。在我们测评过的大量项目中，该API的CPU开销主要集中在500ms~3000ms之间。其耗时开销主要取决于场景中Asset和Object的数量，数量越多，则耗时越慢。
* 场景加载  
  场景加载过程的性能开销又可细分成以下几个部分：
* 资源加载。
* Instantiate实例化
  + 大量重复资源实例化
  + 脚本代码的序列化上

### 动画系统和物理系统音音频系统和粒子系统等

### 代码效率

## 内存优化：

内存的开销无外乎以下三大部分：

* 资源内存占用；
* 引擎模块自身内存占用；
* 托管堆内存占用

### 资源内存占用

* 纹理
  + 纹理格式
    - Android平台的ETC、iOS平台的PVRTC、Windows PC上的DXT
  + 纹理尺寸
    - 纹理尺寸越大，则内存占用越大。所以，尽可能降低纹理尺寸，如果512x512的纹理对于显示效果已经够用，那么就不要使用1024x1024的纹理，因为后者的内存占用是前者的四倍
  + Mipmap功能
    - Mipmap旨在有效降低渲染带宽的压力，提升游戏的渲染效率。但是，开启Mipmap会将纹理内存提升1.33倍。对于具有较大纵深感的3D游戏来说，3D场景模型和角色我们一般是建议开启Mipmap功能的
  + UI纹理基本不开启
  + Read & Write
  + 纹理资源的“Read & Write”功能在Unity引擎中是默认关闭的
* 网格
  + Normal、Color和Tangent
    - Mesh资源的数据中经常会含有大量的Color数据、Normal数据和Tangent数据。这些数据的存在将大幅度增加Mesh资源的文件体积和内存占用。其中，Color数据和Normal数据主要为3DMax、Maya等建模软件导出时设置所生成，而Tangent一般为导入引擎时生成。

### 引擎模块自身占用

* 引擎自身中存在内存开销的部分纷繁复杂，可以说是由巨量的“微小”内存所累积起来的，比如GameObject及其各种Component（最大量的Component应该算是Transform了）、ParticleSystem、MonoScript以及各种各样的模块Manager（SceneManager、CanvasManager、PersistentManager等)...
* 真正占据较大内存开销的是这两处：WebStream 和 SerializedFile。其绝大部分的内存分配则是由AssetBundle加载资源所致。
  + WebStream的大小则是AssetBundle原始文件大小 + 解压后的数据大小 + DecompressionBuffer(0.5MB)。同时，由于Unity 5.3版本之前的AssetBundle文件为LZMA压缩，其压缩比类似于Zip（20%-25%）,所以对于一个1MB的原始AssetBundle文件，其加载后WebStream的大小则可能是5~6MB，因此，当项目中存在通过new WWW加载多个AssetBundle文件，且AssetBundle又无法及时释放时，WebStream的内存可能会很大
  + SerializedFile，则是当你使用LoadFromCacheOrDownload、CreateFromFile或new WWW本地AssetBundle文件时产生的序列化文件

AssetBundle的详细管理机制，建议查看我们之前的[AssetBundle技术文章](http://blog.uwa4d.com/archives/ABTheory.html)

### 托管堆内存占用

因为目前Unity所使用的Mono版本存在一个很严重的问题，即：Mono的堆内存一旦分配，就不会返还给系统。这意味着Mono的堆内存是只升不降的

* 高频率地 New Class/Container/Array等。
* Log输出。