# 腾讯光子客户端一面

1.一个数字转换为二进制，其中有多少个1

int count\_ones\_bitwise(int n) {

int count = 0;

unsigned int num = static\_cast<unsigned int>(n); *// 处理负数（转换为无符号数）*

while (num > 0) {

count += num & 1; *// 检查最低位是否为1*

num >>= 1; *// 右移一位*

}

return count;

}

使用位运算

2. 二维空间中三角形，怎么判断一个点是否在其内部？

计算向量AE与向量AB的叉乘，如果每条边计算出的结果的符号相同，那么则表明E在所有边的同一侧，即在三角形内。

计算重心坐标，如果重心坐标中的任意一个值小于0则表明其在三角形外面。

计算重心坐标的方法又有面积法和方程两种方法

# 腾讯天美客户端一面

**1.帧同步和状态同步**

帧同步，客户端上传操作，服务器只负责转发操作，具体的操作在客户端执行。战斗逻辑在本地，会产生外挂。在逻辑帧上传操作，不是渲染帧，可能是几个渲染帧才上传一个。

采用延迟锁步来实现确保操作同步，客户端的操作会延迟几帧才实际执行，在这个过程中会等待所有玩家的操作信息，如果出现丢包那么采用预测来执行如果预测错误那么回退。

状态同步，客户端上传实时操作，服务器计算操作并将结果返回客户端，客户端只是显示层。

回放系统难以实现，需要保存大量数据。且对服务器的压力非常大。

在服务器每隔一段时间就保存一次游戏的完整状态，如果已经出现了不同步的，那么回退到服务器保存的快照重新执行玩家的操作。

[【网络同步】浅析帧同步和状态同步 - 知乎 (zhihu.com)](https://zhuanlan.zhihu.com/p/357973435)

<https://zhuanlan.zhihu.com/p/357973435>

**2.map和hash\_map**

map底层使用红黑树o(logn)的时间复杂度，hash\_map使用hash表，查找的时间复杂度为O

0(1)。

红黑树是平衡条件为最长路径不超过最短路径两倍的平衡树，而AVL树是子树高度差不超过2的平衡树。

红黑树使用着色来维持平衡条件，而AVL使用深度来维持。

红黑树最关键的两个条件是红节点不能与红节点相连，根节点到叶节点的不同路径上黑节点的数目相同。

[红黑树详解-CSDN博客](https://blog.csdn.net/hlzs_01/article/details/147254146)

<https://blog.csdn.net/hlzs_01/article/details/147254146>

**3.CPP的编译链接**

编译：预编译，编译，汇编

预编译：执行预处理指令，将引用的头文件完整拷贝到代码中，并将宏定义进行替换。

编译：将预编译产生的代码编译为汇编代码。

汇编：将编译产生的文件汇编为二进制机器语言。

链接：将每一个符号表相对应的符号进行连接。包括静态连接和动态连接。

头文件：只包含函数原型和类的定义，有部分内联函数和模版函数的实现。便于通过编译并为后续的连接提供需要的符号。

库文件：包括函数的实现的具体代码。

[【C语言/C++】C/C++程序的编译+链接\_c语言怎么连接-CSDN博客](https://blog.csdn.net/m0_71580879/article/details/143488527)

<https://blog.csdn.net/m0_71580879/article/details/143488527>

**4. 头文件的函数在cpp中没有实现会报错吗**

不会，如果没有实现那么会在连接阶段去库文件中查找并解析。

**5. 协程及其底层**

协程本质是一个轻量级的用户态线程，由程序进行管理，可以用来实现异步的操作。

C#的协程的底层是一个简单的状态机。C#调用StartCoroutine方法，将IEnumerator对象传入C++层，通过反射来找到对象的moveNext和current方法，然后创建一个Coroutine对象，将这两个方法交给该对象，随后执行Run方法调用moveNext方法一次执行到yield时将对象保存到阻塞链表中，在每一帧都会检测一次阻塞链表中是否有满足条件的coroutine对象，有的话就执行它的Run函数，等待Movenext函数返回false时协程结束。

协程中核心的点有两个，一个是movenext函数执行当前段的方法，另一个是current保存当前执行到的阶段以及等待的相关信息，当执行到yield时会将当前的协程函数挂入阻塞列表，每一帧检查是否可以执行。

[Unity/C++/C# 多线程与协程(Coroutine)原理与用法详解\_c# coroutine-CSDN博客](https://blog.csdn.net/m0_73952999/article/details/145962680?utm_medium=distribute.pc_relevant.none-task-blog-2~default~baidujs_baidulandingword~default-8-145962680-blog-116240688.235%5ev43%5epc_blog_bottom_relevance_base2&spm=1001.2101.3001.4242.5&utm_relevant_index=11)

[https://blog.csdn.net/m0\_73952999/article/details/145962680?utm\_medium=distribute.pc\_relevant.none-task-blog-2~default~baidujs\_baidulandingword~default-8-145962680-blog-116240688.235^v43^pc\_blog\_bottom\_relevance\_base2&spm=1001.2101.3001.4242.5&utm\_relevant\_index=11](https://blog.csdn.net/m0_73952999/article/details/145962680?utm_medium=distribute.pc_relevant.none-task-blog-2~default~baidujs_baidulandingword~default-8-145962680-blog-116240688.235%5ev43%5epc_blog_bottom_relevance_base2&spm=1001.2101.3001.4242.5&utm_relevant_index=11)

[反编译C#代码来看看协程是什么 - 知乎 (zhihu.com)](https://zhuanlan.zhihu.com/p/35195150)

<https://zhuanlan.zhihu.com/p/35195150>

**6. 移动的实现，用了插值吗**

移动的实现有两种逻辑，增加速度和直接改变位置

直接改变位置：

每帧直接在原本的位置的基础上增加位移

使用Lerp函数，规定移动的总时间和移动的末位置

void Update()

{ //update每帧更新(调用)

time += Time.deltaTime;

//计算当前到开始的时间，每调用一次update就加一次两帧间的时间

t = Mathf.Clamp01(time / totalTime);

//每次调用update重新计算当前的比例系数,clamp钳制到【0，1】区间，相当于映射到【0，1】

transform.position = Vector3.Lerp(start, target, t);

//Lerp的A,B值需要是固定的，t系数值要动态跟随当前时间比例变化，即可实现匀速运动

}

使用Slerp函数，实现弧线运动

void Update()

{

time += Time.deltaTime; //计算当前到开始的时间，每调用一次update就加一次两帧间的时间

t = Mathf.Clamp01(time / totalTime); //钳制到[0，1]

transform.position=Vector3.Slerp(startPos, targetPos, t);

}

上面的两种函数的系数t都是当前的比例以移动的时间比上总的时间就可以实现匀速的运动

通过刚体来设置速度：

rigibody.velocity=new Vector3(speed.x,speed.y,speed.z)

直接在每一帧中设置游戏物体的速度来实现移动

跳跃之类的瞬间速度是rig.addforce()

[unity 实现移动\_unity移动脚本-CSDN博客](https://blog.csdn.net/liangqianjin_1/article/details/145398473)

<https://blog.csdn.net/liangqianjin_1/article/details/145398473>

**7.渲染管线**

应用阶段：传递渲染需要的数据并发出DrawCall指令让GPU进行计算

几何阶段：

顶点着色器：发生MVP矩阵变换以及通过齐次除法得到NDC坐标

裁剪：将不会被摄像机渲染到的点剔除

屏幕映射：将顶点NDC映射到屏幕坐标

光栅化阶段：

三角形设置：将几何阶段得到的点连接三角形

三角形遍历：哪些像素被三角形覆盖，被覆盖的三角形生成一个片元

片元着色器：根据片元的数据计算片元实际的颜色

逐片元操作：通过透明度，深度等一系列测试计算像素的实际颜色。

**8.TCP UDP**

UDP无连接，只需要知道目标的IP和端口号就可以发送。不可靠，没有确认机制，无法得知是否成功传输。

TCP有连接，即使发送失败也会通知进程，有序，流量控制。三次握手，两次只能确定一方的发送接受功能正常，四次挥手。

TCP延迟大于UDP的原因不是在于三次握手的时间开销而是源自于其可靠机制在丢包时的处理会导致后续所有数据发生等待。比如每一帧都接收信息发生丢包后TCP会导致后续几帧都出现延迟但是UDP只会在这个丢包的帧出现问题。

[TCP与UDP协议详解：可靠性、报文格式与区别-CSDN博客](https://blog.csdn.net/qq15035899256/article/details/126073927?ops_request_misc=%257B%2522request%255Fid%2522%253A%252278ed2c9c6869ed8f8f85f6da75c22738%2522%252C%2522scm%2522%253A%252220140713.130102334..%2522%257D&request_id=78ed2c9c6869ed8f8f85f6da75c22738&biz_id=0&utm_medium=distribute.pc_search_result.none-task-blog-2~all~top_positive~default-1-126073927-null-null.142%5ev102%5epc_search_result_base6&utm_term=TCPUDP&spm=1018.2226.3001.4187)

[https://blog.csdn.net/qq15035899256/article/details/126073927?ops\_request\_misc=%257B%2522request%255Fid%2522%253A%252278ed2c9c6869ed8f8f85f6da75c22738%2522%252C%2522scm%2522%253A%252220140713.130102334..%2522%257D&request\_id=78ed2c9c6869ed8f8f85f6da75c22738&biz\_id=0&utm\_medium=distribute.pc\_search\_result.none-task-blog-2~all~top\_positive~default-1-126073927-null-null.142^v102^pc\_search\_result\_base6&utm\_term=TCPUDP&spm=1018.2226.3001.4187](https://blog.csdn.net/qq15035899256/article/details/126073927?ops_request_misc=%257B%2522request%255Fid%2522%253A%252278ed2c9c6869ed8f8f85f6da75c22738%2522%252C%2522scm%2522%253A%252220140713.130102334..%2522%257D&request_id=78ed2c9c6869ed8f8f85f6da75c22738&biz_id=0&utm_medium=distribute.pc_search_result.none-task-blog-2~all~top_positive~default-1-126073927-null-null.142%5ev102%5epc_search_result_base6&utm_term=TCPUDP&spm=1018.2226.3001.4187)

**9.gc了解吗**

标记清除压缩

**10.HTTP和HTTPS协议**

HTTP协议是明文，可以被截取破解。

HTTPS协议在HTTP的基础上增加了加密和身份验证，加密是使用的密钥，对称密钥和非对称密钥。身份验证是验证服务器的数字证书。

[Http 和 Https的区别（图文详解） - 知乎 (zhihu.com)](https://zhuanlan.zhihu.com/p/701521059)

<https://zhuanlan.zhihu.com/p/701521059>

# 腾讯魔方客户端一面

**1.C#和C++中结构体有什么不同？**

C++的类和结构体没有本质的区别，只是在默认访问和继承属性上有区别，类的继承属性和默认访问都是private的。他们主要的区别还是在于用法，结构体用来封装数据，类用来实现复杂的逻辑。

[C++ 中类（class）和结构体（struct）的区别\_c++结构体和类的区别-CSDN博客](https://blog.csdn.net/chenai886/article/details/144132525)

<https://blog.csdn.net/chenai886/article/details/144132525>

C#的结构体进一步发扬了结构体作为轻量级的类的作用，将结构体作为值类型在栈上分配空间，而类的对象在堆上分配空间。

C++的类的对象是值类型，在赋值时进行拷贝赋值。

C#的对象是引用类型，在赋值时进行地址的传递。

C++相较于C#更加底层，所以它的效率更高但是作为一个工具来说C++没有C#现代。

C++开放了更多的功能给程序员，C++认为程序员不会犯错，而C#认为程序员会犯错。

具体来说，C++将内存的管理完全交给程序员，程序员的不当操作会出现内存泄漏，而C#有自带的GC（标记回收压缩）大大降低了内存泄漏的风险。

同时，这两个语言的运行也有较大的不同，C++直接编译为机器语言在本地运行效率更高，而C#需要编译为中间的IL语言在虚拟机上进行运行。

**2.C++11的常见新特性有哪些?**

Atuo关键字，智能指针，Lambda表达式

shared\_ptr的引用计数是线程安全的但是对于shared\_ptr对象本身的修改不是线程安全的，同时对于shared\_ptr引用对象的修改也不是线程安全的， shared\_ptr不会对被引用的对象本身提供任何保障。

Lambda表达式在底层中会被编译器转化为一个重载（）运算符的类对象，捕获列表作为构造函数的参数，参数作为重载函数的参数。



**3.外部排序（归并排序）**

假设内存中能够容纳三个块，那么将两个块作为输入缓存的块，一个块作为输出缓存的块，并进行归并排序，将结果写回硬盘形成由两个块构成的一个归并段，这个段是有序的，随后对所有块进行该操作，得到有序的归并段，随后对两个归并段进行归并排序，先将两个段中分别的低地址的块调入内存进行归并排序，再将高地址的块调入内存进行归并排序，并对后面所有归并段重复该操作，然后文件的每4个块都是有序的，再不断重复上面的过程就得到了有序的文件。

[数据结构（十六）----外部排序-CSDN博客](https://blog.csdn.net/weixin_69884785/article/details/138917062)

<https://blog.csdn.net/weixin_69884785/article/details/138917062>

# 腾讯天美游戏客户端一面

**1.Int类型和long类型的区别**

他们都是用于表示整数类型，只是在内存中占用的大小不同，int占用4个字节而long在不同的系统中长度不同，所以在有跨平台的场合不建议使用该数据类型。

long在windows始终下为4字节，但是在64位的linux和macOS系统中是8字节，在32位时又是4个字节。

**2.Const的用法，修改const的 变量，const的全局常量存放在哪里？**

const表示该值不可以重新赋值，本身只是一个修饰符不会影响变量的存储位置。如果是全局的变量那么会存储到静态存储区，如果是局部的变量那么会被存储到存放到栈区。如果是类成员，那么存储的位置跟随对象。

但是有一个例外是如果变量的值在编译时即可确定那么编译器会采取优化，将该变量视为编译器常量，直接用值来进行替换，这种情况下变量甚至不会出现在内存空间中。但是如果对这个变量进行了取地址，那么会导致优化失效，强制为变量分配内存空间。

[C++内存划分&const常量存储位置\_const存储是栈还是堆-CSDN博客](https://blog.csdn.net/weixin_61256992/article/details/146295634)

<https://blog.csdn.net/weixin_61256992/article/details/146295634>

**3.** **引用类型是否可以做函数的返回类型，使用的时候有哪些需要注意的地方**

可以，最需要注意的点在于返回的被引用的变量的生命周期需要足够长，不然会导致未定义的行为。

对于返回值为临时对象时会有优化，直接将临时对象放在在函数外构造，然后作为参数传递，避免在返回过程中产生的临时对象。这个优化的效果是和右值引用相似的。

**4. 右值引用和移动构造，移动构造函数里面步骤**

移动语义的作用是面对一些即将被销毁的对象不是复制他而是窃取他的资源。右值引用是为了实现移动语义。

移动构造函数和赋值构造函数在函数内部是相同的都是通过传入的参数初始化当前对象，只是在传入参数时有所区别，移动构造函数传参使用右值引用直接剥夺原有资源的所有权，而拷贝构造函数使用左值引用不会剥夺原有资源的所有权。

[【C++ | 移动构造函数】C++11的 移动构造函数 详解及例子代码-CSDN博客](https://blog.csdn.net/wkd_007/article/details/139633287)

<https://blog.csdn.net/wkd_007/article/details/139633287>

**5.** **函数调用栈，栈上面有那些内容。函数的参数入栈顺序是怎么样的，函数的返回值怎么转递**

函数的栈帧就是函数调用过程中在程序的调用栈所开辟的空间，保存函数的参数，函数的返回值，临时变量，以及上下文信息。将参数从左至右压入栈中，函数的返回值一般是通过寄存器的方式来进行传递，但是如果返回值较大则会在调用者的帧栈中单独开辟一块用来保存返回值的空间并将这块空间的地址传递。

栈的速度比堆快，因为栈是在内存空间连续分配，并且以堆栈的形式进行管理，分配与回收只需要进行指针的移动。

堆在内存空间中是非连续分配的，涉及内存查找和碎片整理所以速度较慢。

[C语言：函数栈帧-CSDN博客](https://blog.csdn.net/2401_88251472/article/details/147362419)

<https://blog.csdn.net/2401_88251472/article/details/147362419>

**6.** **析构函数为什么一般定义为虚函数，如果子类只定义一些基础的类型变量，是否会内存泄漏**

在实现运行时多态的过程中，如果未将析构函数定义为虚函数，那么会执行父类的析构哈数来释放对象的内存，这样会导致子类增加的内存空间无法被释放造成内存溢出。基础类型的变量不会造成内存泄露，因为其被存放在栈空间上由计算机进行管理。只有当子类使用new或者malloc来分配堆空间时才有可能造成内存的泄露。

**7. 构造函数里面可不可以调用虚函数，如果在构造函数里面调用了虚函数会有什么情况**

可以，只是vptr此时仍指向父类的虚函数表，所以会调用父类的函数导致多态失效。

**8. New操作符除了构造对象还有其他用法吗，new操作符是否可以重载，有什么用途。**

还可以构造数组以及在预先分配的内存空间上构造对象。

可以重载，以实现自定义的内存管理，new操作符本身也就是对malloc的封装，这个封装的过程由我们自己来实现也是可行的。

**9. Static\_cast和dynamic\_cast的区别，转换失败会发生怎么情况**

Static\_cast在编译时进行检查，适用于基本类型的转换和向上转型，可以进行向下转型但是必须保证是正确的，因为其不会进行运行时检查，其效率高于dynamic\_cast但是无法保证类型是绝对安全的。

dynamic\_cast在运行时的进行类型的检查，主要用于向上转型。

static\_cast是在编译时进行检查所以在转换失败时会报出编译错误。而对于dynamic\_cast而言，其在运行时进行检查，在指针转换时转换失败会返回一个空指针，在引用转换失败时会抛出异常。

也就是说static\_cast是在确认安全的时候使用，而dynamic\_cast是在不确定安全的时候使用。

**10.** **ClassA和ClassB没有任何关联，强行转换可以吗，加一些定义就可以转换？比如加一些特殊的构造函数，或者重载一些特殊的运算符。**

直接使用static\_cast和dynamic\_cast进行强制转换是不可行的。有两种常见的实现无关联类的转换，可以自定义一个函数纯逻辑性的进行转换或者重载类型转换运算符。

class ClassA {

public:

int value;

operator ClassB() const {

ClassB b;

b.value = value;

return b;

}

};

ClassA a{42};

ClassB b = a;

**11.** **MAP和unordered\_map关联和时间复杂度**

Map和unordered\_map在实现上实际没有关联，关联可能在于他们都是使用键值对的形式进行的存储，Map底层使用红黑树，其增删改查的时间复杂度是O(logn)，而unordered\_map底层使用Hash表，时间复杂度是O(1)

**12.** **Vector里面的push\_back和empalce\_back的区别**

Push\_back()是将已经构造完成的对象加入容器中，而empalce\_back是在容器的内部进行对象的构造，减少了一次拷贝，效率更高尤其在构造大型对象时更加明显。

**13. Shared\_ptr是不是线程安全的，避免循环引用**

Shared\_ptr本身是线程安全的但是不保证引用的对象的线程安全。对于Shared\_ptr出现的循环引用的问题，我们引入了weak\_ptr来解决。

Shared\_ptr会改变引用对象的生命周期，只有引用归零时生命周期才会结束。而对象销毁时会解除其中所有shared\_ptr的引用。

**14.** **编译链接的过程，链接库有有几种方法**

预编译：处理代码中的预处理指令

编译：将代码编译为汇编代码

汇编：将汇编代码汇编为机器代码

链接：将代码中的符号与符号表进行连接

有两种方式，静态链接和动态链接，静态链接直接将库中的代码复制打包到最后的可执行文件中生成独立的可执行文件；而动态链接会在可执行文件中保留符号，在运行时参照符号表动态链接库文件进行调用。

**15.在两个CPP中分别定义int A，能否通过**

不能，在链接阶段会报重复定义的错误。

链接阶段会去查询每一个.o文件产生的符号表并进行合并，在合并时会发现有两个int a，不知道应该采用哪一个于是出现报错。

可以使用Static将int a局限于本文件，或者使用inline来使两个定义都生效，但是如果这样定义必须完全相同不然违反单一定义原则还是会报错。

**链接阶段（Static Linking）**

* 链接器接收所有目标文件 .o 和静态库 .a。
* 读取并合并符号表，建立全局符号表。
* 检测重复定义和未定义符号错误。
* **符号解析**：未定义符号匹配到静态库或其他目标文件中的定义。
* **地址分配**：为所有代码和数据段分配虚拟地址，合并 .text、.data、.bss 等段。
* **重定位**：修改所有引用符号的机器码或数据地址，替换为分配后的真实地址。
* 从静态库中提取需要的目标文件 .o，其余忽略。
* 将所有代码和数据合并成一个整体。

**16.** **UDP进程用了一个80端口，TCP进程还能使用吗**

可以的，端口号是和协议相绑定的，不同的协议可以使用相同的端口号，但是相同的协议不能。

**17. 输入网址到加载网页内容的过程**

本地解析网址->本地DNS查询IP地址->三次握手建立TCP连接->发送HTTP连接->接收并解析HTML文件->加载子资源

**18.操作系统里面一个多线程的程序怎么进行同步，无锁编程有了解过吗**

目前主要使用的方法有锁和信号量。无锁编程实际上是使用硬件层面提供的原子操作指令来保证线程安全，没有循环等待的过程所以效率较高。

**19.系统调用有了解过吗？主要有什么作用？**

用户态进程使用内核功能的接口，是用户程序访问操作系统内核提供的服务的唯一方式。

线程操作，访问内存和磁盘等面向于硬件的操作我们自己写的程序实际上是没有权限执行的，所以需要借助系统调用来实现。

**20.多线程程序会有几个栈，系统调用里面会有栈切换**

只考虑用户态的话，多线程程序每一个线程都会有一个独立的栈来执行函数，所以线程的数量就是栈的数量，但是如果考虑内核态，那么每一个线程还有一个内核态的栈，也就是栈的数量是线程的两倍。使用系统调用会进入内核态，栈也会从线程的栈进入内核态的栈。

**21.单例模式**

懒汉式在使用时才进行创建，可能会出现多个线程同时使用所以存在线程不安全，而饿汉式在程序启动时就加载好，线程安全但是有一点点内存的浪费。

**22.工厂模式**

工厂模式实际上是对对象创建过程的封装，提供更加简单的对象创作方法，就像语法糖一样。

# 灵犀互娱 平平无奇工作室 游戏客户端开发

**1.C++多态有哪几种实现方式？**

多态有编译时多态和运行时多态两种，编译时多态的是模版，函数重载和运算符重载，而运行时多态是基于虚函数实现的。

**2.C++虚函数**

造函数不可以声明为虚函数，而析构函数则建议声明为虚函数以实现子类内存空间的释放，使用new生成的空间独立的一块空间不是对象的一部分，所以需要在析构函数中速手动释放。构造函数可以调用虚函数，但是由于虚指针没有被初始化仍指向父类的虚表此时会调用父类的虚函数。构造函数不能被声明为虚函数是因为空间没有被初始化，虚指针没有被正确赋值，虚函数执行时需要确定其类型但是构造函数执行时还未确定其类型。

**3.vector中插入元素会发生什么？插入类的元素时是否会调用构造函数？**

会先判断容量是否够，如果够的话插入元素时会导致插入位置后面的所有元素都进行后移导致效率较低，不够会发生整个vector位置的迁移然后再进行插入。插入对象时会调用，如果使用push\_back会在对象时调用一次，拷贝构造的时候调用一次，如果使用enplace\_back的话就只会调用一次。

**4.map元素是否有序**

有序map的底层是使用的红黑树来实现，而红黑树本质是平衡条件更加宽松的二叉平衡树所以他是有序的。

**5.哈希冲突是什么,如何解决哈希冲突**

哈希冲突指的是多个键值被映射到同一个下标，解决哈希冲突可以采用开放地址法或者拉链法。

**6.编译过程，为什么部分符号不会出现在最后的符号表，静态链接发生在哪一个阶段？**

预编译：执行代码中的预处理指令

编译：将代码编译为汇编代码

汇编：将汇编代码汇编为二进制机器语言

连接：解析各目标文件的符号表，解决未定义符号与其他目标文件或者库文件中已定义符号的对应关系，然后生成一张新的符号表，静态链接会将库文件中对应的代码拷贝进最后的可执行文件中生成独立的可执行文件，动态链接会在运行时根据符号表在库中动态加载

两种情况，常见的是使用static来主动避免符号进入全局符号表，static会采用内部链接，另一种情况是编译器优化，未使用的符号会被剔除。

Static的作用：1.让变量只进行一次初始化2.让变量属于类而非某一个具体的对象3.让变量变成内部链接无法被其他文件访问

**7.TCP和UDP**

TCP，稳定，3次握手4次回收，有确认机制，所以延迟较高

UDP，不稳定，不可靠，丢包不管，所以延迟低

**8.帧同步，帧同步要解决的问题**

帧同步，客户端本地计算操作，由服务器分发操作。如果出现丢包会带来较大的影响同时作弊也比状态同步更加容易，延迟较低。

检查到未收到包则等待一段时间，如果没有收到操作那么进行预测使用上一帧或者空操作来代替以让游戏正常进行，同时客户端补发操作，如果预测的操作与实际的操作不同那么进行回滚重新计算。

状态同步，客户端上传操作，由服务器进行计算后分发。服务器压力较大延迟较高。

**9.行为树相较于状态机的优点是什么？有哪些常见的节点？**

结构清晰，扩展性强，能够更加优雅的处理复杂的AI逻辑不会像状态机一样出现状态爆炸。

常见的节点有  
Selector：从左到右执行节点，执行成功时立刻返回

Sequence：从左到右执行节点，每一个都执行成功时才返回成功，任意节点执行失败时停止执行并返回失败

Condition：条件判断，不满足停止执行并立刻返回

Action：实际的逻辑节点，包含游戏物体的实际的运行逻辑

behaviorTree = new SelectorNode(new List<BTNode>

{

new SequenceNode(new List<BTNode>

{

new ConditionNode(IsPlayerInAttackRange),

new ActionNode(AttackPlayer)

}),

new SequenceNode(new List<BTNode>

{

new ConditionNode(IsPlayerVisible),

new ActionNode(MoveTowardsPlayer)

}),

new ActionNode(Patrol)

});

}

**10.MVP变换指的是什么？**

MVP指的是Model，View，Projection，分别是将模型从局部空间移动到世界空间，从世界空间移动摄像机空间，再从摄像机空间移动到裁剪空间

**11.如何实现人物描边**

外扩模型，采用两边渲染，第一次渲染剔除正面，将顶点按法线移动后将模型渲染成描边的颜色形成外壳，第二次剔除背面正常渲染。

**12.如何在100万个数据中找到字典序前1000的数据？**

采用大顶堆来实现，堆顶保存最大的数据，对每一个数据进行一次比较。假如每一次都需要进行插入那么时间的消耗为1，000，000\*log1000 = 10^7在可以接受的范围内用户基本难以感知到延迟

# 米哈游客户端一面

**1.说说什么是线程和进程，他们之间的区别是什么，线程之间的通讯和进程之间的通讯的区别是什么？**

进程是程序的一次运行的实例，操作系统分配资源的最小单位。

线程是程序执行的最小单位，多个线程共享进程的内存资源。

由于线程之间共享资源，所以其可以直接通过信号量和锁来实现，但是进程之间彼此独立所以必须使用操作系统提供的IPC来实现，包括管道和共享内存区之类的。

**2.什么是虚拟内存，主要解决什么问题？**

虚拟内存是一种内存管理技术可以让程序认为自己拥有完整而连续的内存空间，实际上这些内存被存放在物理内存和磁盘中。

主要解决了物理内存有限以及物理内存之间相互干扰的问题

**3.什么时候会出现缺页中断？出现缺页中断时**

当访问的内存块不存在于物理内存中时，常见的就是程序首次访问某块虚拟内存，访问的页已经被替换出了物理内存。

因为用户态实际上是不能进行对内存和磁盘的操作，所以出现缺页中断时会暂停程序随后进入内核态进行异常处理。在异常处理的过程中首先会进行现场以便于完成后继续执行原本的程序，随后判断访问是否合法，如果合法那么查找该页在磁盘中的位置，随后分配物理内存将内存页从磁盘中调如物理内存，然后更新页表恢复现场，程序继续执行。

**4.C++的编译步骤分为几步**

1.预编译：处理预编译指令

2.编译：将源代码编译为汇编代码

3.汇编：将汇编代码汇编为二进制机器语言

4.链接：解析目标文件的符号表，将文件中的未定义符号与其他目标文件或者库文件中的已定义符号建立对应关系。有两种链接，静态链接将代码直接拷贝进可执行文件中生成独立的可执行文件，动态链接在运行时根据符号表动态加载。

**5.STL用过吗？用过哪些?**

Vector，底层为动态数组，C++11之后实际上再大多数的情况下都建议使用vector来替代数组进行使用。

Set，map底层实现一样，为红黑树。

unordered\_set,unordered\_map底层实现都是哈希表。

**6.vector的resize()和reserve()的区别？**resize()是元素层面的修改，而reserve()是修改分配的空间，在增加的元素个数超过原本的容量时会触发reserve的内部来重新分配空间。如果给一个容器分配小于其实际需要空间的空间那么什么都不会发生。

**7.Map和Multimap的底层原理？红黑树的基本概念？红黑树有几个要注意的点？复杂度是多少？**

Map和MultiMap的底层都是使用红黑树，红黑树本质上是平衡条件更加宽松的AVL树，不像AVL要求子树高度差不超过2，只要求根节点到叶子结点的最长路径不超过最短路径的两倍。平衡条件更加宽松也就意味着其对调更加友好。首先根节点必须是黑色，然后红色节点不能接着红色节点，根节点到任意叶子节点路径上的黑色节点数必须相同。

增删改查的时间复杂度都是O(logn)。

**8.多态对你来说意味着什么？编译时多态和运行时多态举例?编程的时候什么时候会用到？**

多态意味着运行时多态和编译时多态，编译时多态有函数重载和模板，运行时多态有虚函数。实现多种敌人时就会使用到运行时多态来根据敌人的具体类型来调用不同的方法，而编译时多态通常用于实现框架性的内容，比如ECS框架中实现对各个组件的统一操作就会使用到模板。

**9.int \*/int&/int /const int 之间可以形成函数重载吗？是直接在编译时报错吗？**

Int\*/int&/int可以形成函数的重载，因为他们本质上是不同的类型，但是int/const int 不能形成函数的重载，他们都是int类型const只是一个修饰词并不会修改变量原本的类型。是直接报错，无法通过编译的语义分析。

**10.指针和引用的区别，初始化有区别吗，指针和引用的对象可以更改吗？**

一个保存对象的地址，一个直接就是对象，指针可以不用初始化但是引用必须进行初始化，指针指向的对象可以更改但是引用的对象不可以更改。

**11.什么是野指针？**

指向的空间不合法的指针。没有初始化或者原本指向的空间被回收。

**12.const int \* 和int\* const 的区别**

Int\* cosnt是指针本身无法修改，也就是只能始终指向一个对象。Cosnt int\* 是无法通过指针修改该对象

**13.智能指针中哪一个用的多一点？如果自己实现一个智能指针你会怎么去实现？除了将对象空间的分配放到构造和析构函数还有什么要注意的地方吗？**

Unique\_Ptr用得多一点，我会创建一个类将需要托管的对象的空间的管理放到对象的构造函数和析构函数中做。重载运算符防止出现二次释放同一资源。

**14.C++的虚函数了解吗？C++如何处理虚函数？虚函数表存的是什么内容？虚函数指针呢？虚函数表存在内存的哪一个位置？**

虚函数是C++中实现运行时多态的核心机制。当类中存在虚函数时，会生成一张虚函数表和一个虚函数指针，调用时通过虚指针查找虚函数表调用实际类型的指针。

虚函数指针保存虚函数表的地址。虚函数表中保存了该类所有虚函数的位置。纯虚函数

Virtual void func() = 0;其作用是强制子类实现该函数，否则子类也会变为无法实例化的抽象类。

**15.Static关键字的用法？**

1.修饰全局变量/函数时会将变量/函数的链接属性修改为内部链接限制作用域为当前文件。

2.修饰类的变量和函数时，修饰的内容会属于整个类而非对象。

3.修饰函数内的局部变量时会将变量存储到静态存储区生命周期延长到整个程序，只会被初始化一次。

**16.设计模式有了解过吗？单例模式实现过吗？如何确定全局只存在一个实例。**

在项目完成的过程中了解设计模式来进行代码结构层面的优化是不可避免的，最常使用的就是单例和观察者，最近实现过的是工厂模式。

单例模式有懒汉和恶汉的两种实现方式，

单例本身是依赖于类本身的，全局只会存在一个类所以也就保证了全局只存在一个实例。

**17.观察者模式有了解过吗？Unity自带的观察者模式有哪些？Unity如何实现观察者模式？**

采用通知而非轮询的方式来触发事件处理。

Unity中UI的eventSystem主体就是使用了观察者模式。

ImputModule监听鼠标等输入事件，通过RayCast来确定当前输入作用的UI元素，EventSystem调用UI元素上实现的事件对应的接口，随后执行具体的逻辑。

**18.C++的类型转换？无关类型是否可以进行转换？**

(int)这种C风格的做法，好用但是不够现代化。

C++更加现代化的写法是使用Static\_Cast()进行值类型和安全的向下转型，Dynamic\_Cast()进行不安全的向下转型，向上转型也可以但是不建议。

无关类型可以进行转换但是C++本身并没有提供相关的方法需要自己实现。单独定义一个函数或者重载类型转换运算符。

# 腾讯光子客户端二面

**1.类的对象大小由什么决定？**

类中非静态成员的大小加根据内存对齐机制多出来的空间。

内存对齐机制：每一个变量的地址偏移都需要是自身大小的整数倍，整个对象大小必须是对齐边界(整个结构体中最大的类型)的整数倍。

**2.虚继承**

避免菱形继承中重复继承基类导致的资源浪费的问题，用得不太多，继承两个基类的话一般使用组合，而且在C#中根本不会出现菱形继承。

**3.inline关键字**

Inline会告知编辑器此处展开函数而非调用，对于多次执行的函数可以优化性能。但是如果函数比较复杂也就是存在递归等内容那么编辑器可能会忽略inline直接调用函数。

# 腾讯光子一面

**1.MVP矩阵**

Model-View-Projection，将模型从模型空间移动到摄像机观察空间，再从摄像机观察空间移动到裁剪空间。

**2.Z-Buffer的作用**

记录深度缓存用于后续逐片元操作时计算遮盖关系。

**3.接口和抽象类的区别**

主要是功能层面的区别，接口相当于没有任何实际内容的规范而抽象类则是实现了部分的实际内容。

**4.透明物体为什么要从后往前渲染**

因为从后向前渲染后导致颜色的叠加错误，因为计算透明物体的渲染公式中需要知道背景的颜色，如果从前向后渲染会导致背景的颜色还不确定。

最终颜色 = 当前像素颜色 × 透明度 + 背后颜色 × (1 - 透明度)

# 游卡客户端一面

**1.XLua和C#的双向的交互？Lua调用C#的参数和返回值是如何传递的呢？C#获取到Lua中的string并修改那么会影响Lua中的string吗？原因是什么？Lua中的Table传递给C#那么C#是否可以修改这个Table吗这个时候Table会影响原本的Lua中的Table？Lua中的闭包指的是什么？**

XLua通过一个Lua虚拟机来实现C#与Lua的通讯，C#通过DOString来加载Lua的脚本，而Lua脚本中的全局变量和函数会被注册到Lua虚拟机中的全局表，这些注册到全局表的元素可以被C#通过键值直接进行访问。

而Lua访问C#可以直接通过命名空间来进行访问有两种访问的方式，一种是通过反射进行访问效率比较低，一种是通过桥接文件进行调用，效率远高于通过反射访问，但是需要提前生成。

参数从C#传递到Lua时会由XLua进行类型的转换，基本类型会直接映射，而List会映射到顺序表，Dictionary会映射到键值表，这一个映射的过程将会被XLua自动执行，而自定义复杂类型需要生成桥接文件才行。

参数和返回值的数据传递都是通过虚拟机的Lua栈来进行的。

Lua的String，Number，Boolean会被当做值类型传递到C#，所以C#只会得到一个复制的副本其修改无法影响Lua中的原始版本。而Table，Function，UserData会被视为引用类型进行传递，C#层修改的就是变量本身。

闭包指的是函数和其保存的运行环境，Lua的闭包机制可以延长外层函数的临时变量的生命周期，避免内层函数访问时出现未定义行为。

**2.C#委托指的是什么？那如果是一个异步调用的话应该是如何实现的呢？多线程之间的同步有哪些方式？那死锁是怎么发生的呢？**

本质是经过封装的函数指针。在多播委托中每一个方法仍然被封装为单播委托，多播委托的对象中有一个List保存所有单播委托。

异步调用主要的关键字是async声明方法为异步方法，await 等待异步方法的完成，Task封装异步方法，作用是让耗时的方法比如寻路和碰撞检测到其他线程去进行，在这个过程中最主要的关注点在于非Unity主线程外的线程都无法使用UnityMonobehaviour的生命周期的内容。

多线程共用资源所以可以直接通过变量或者数据结构来进行同步的操作。

死锁发生的原因是出现了循环等待，导致所有进程都无法执行。

死锁的条件是：

互斥：资源一次只能被一个线程占用

不可剥夺：资源在使用完成前无法被剥夺

请求与保持：在持有资源的时候又去申请其他资源

循环等待：存在资源获取的环状结构

1. **对象池一般是用在哪些地方，作用是什么？**

对象池的作用是避免频繁创建和销毁对象带来的开销，也是空间换时间的一种方法。

**4.之前有用过状态机吗？状态机最主要的几个接口是什么？对于状态突然中断的处理？**

有，状态机最主要的接口是进入，更新，退出。

**5.ECS指的是什么？他们的作用是什么？**

ECS指的是E：entity实体一个标识，C：Conponent组件保存实体相关的数据，S：System执行具体的逻辑操作。

ECS框架最主要的作用避免OOP中数据离散存储带来的读写开销, ECS 通过让组件数据连续存放，利用空间局部性降低查询开销；同时保证缓存中存放的都是同类数据，减少了缓存缺失率，提升了整体性能，在Unity中使用时还能避免对Monobehaviour的过度使用带来的性能开销。

**6.渲染的标准流水线？光栅化的过程？漫反射的数学模型？**

应用阶段：CPU传递渲染数据并发出DrawCall指令

几何阶段：

顶点着色器：计算顶点的光照UV坐标等信息，并将模型从世界空间移动到摄像机观察空间。

投影：将模型从摄像机观察空间移动到裁剪空间

裁剪：将模型不会被渲染的部分进行剔除，并记录深度缓存，得到归一化坐标

屏幕投影：将点映射到屏幕坐标

光栅化阶段：

图元组装：将几何阶段得到的点连接成图元

三角形遍历：检测屏幕上的一个像素是否被一个三角形覆盖，如果被覆盖那么生成一个片元。

片元着色器：通过插值计算每一个片元的数据

逐片元操作：通过深度测试，透明度混合等测试检验一个片元是否会被输出.

传统的光照使用Blinn-phong模型来进行

实际的颜色 = 环境光 + 漫反射强度系数 \* 漫反射强度 + 高光强度系数 \* 高光强度

环境光：背景光线，保证即使没有直射光物体也不会完全表现为黑色

漫反射颜色：通常来源于纹理采样，也就是物体本身的颜色

漫反射强度：KD（漫反射颜色）\* ID（入射光强度）\* max（0，N（法线向量）·L（指向光源的单位向量））

高光颜色：通常提前定义的颜色，高光通常表现为纯白色

高光强度：KS（高光反射颜色）\* IS（入射光强度）\*max（0，N（法线向量）\*L（半程向量））^n（高光指数，越小光照的衰减越慢，光照的范围越大）

半程向量：normalize（L（指向光源的单位向量）\*V（指向观察者的单位向量））

1. **MVC的理解？**

M：Model，保存数据以及对数据的操作，V：ViewUI显示的相关逻辑，C：Controller，Model和View沟通的桥梁。

**8.贝塞尔曲线实现的算法可以简单描述一下吗？**

**9.Lua的Table是什么样的数据结构？里面是具体怎么组织数据的？**

Table是一种十分灵活的数据结构，在Lua中有两种组织数据的方式一种是顺序也就是数组，另一种是键值对也就是字典在XLua的数据转化中，Lua实际上也会被映射为这两种数据类型，Lua的Table可以同时是这两种Table但是在向C#进行映射的时候其中顺序表的部分只能被list接受键值对的部分只能被字典的部分接受。

1. **AB包应该如何进行分包？**

**11.项目中遇到的问题？**

**12.简单描述Shader收集工具？**

**13.C#中闭封包和拆包指的是什么？**

封包指的是将值类型封装为引用类型并保存在堆上的过程，而拆包则是将值类型还原。这个过程中会带来性能的开销，装箱会分配堆内存，拆箱需要类型检查。其作用在于将值类型作为引用类型来使用统一的接口。

# 网易雷火一面

**1.实现碰撞检测的算法**

以Unity为例，碰撞体中存在凸体选项，选择后将会简化模型为凸体此时Unity底层会使用GJK算法和SAT算法来进行检测，如果不选择勾选那么会对每一个三角形面进行检测，这样带来的开销是巨大的并且两个碰撞体都没有选择凸体选项那么Unity不会将他们进行碰撞检测，因为这样会带来巨大到难以承受的开销。

在采用凸体的前提下可以采用两种算法来生成：

GJK算法：计算两个模型当前方向与反方向的最远点的差得到一个单形，如果单形没有覆盖原点，那么取单形最接近圆点的边的法向量为新的方向，如果在迭代过程中不再出现新的最远点那么迭代停止。二维的单形使用三角形来覆盖，三维的单形使用四面体。

SAT算法：计算所有顶点在某一个方向向量上的投影，以长方形包围盒为例，这个方向来源于两个包围盒不平行面的法向量(每个包围盒3条)，以及不平行边两两相交形成的面的法向量(3\*3 = 9条)，则共有15条边要进行检测，那么8\*8\*15共960次计算。如果赠右游戏物体的话带来的开销也也是巨大的，所以需要使用SAP或者四叉树来进行初过滤。

# 途游一面

1. **Lua实现Unity中代码的更新的原理**

XLua实现热更新是通过[Hotfix]属性，这个属性和[LuaCallCSharp]是两个完全无关的属性，在XLua中操作的按钮也不同[Hotfix]是单独的生成热更新按钮而[LuaCallCSharp]是生成桥接文件的按钮。

[Hotfix]本身的实现原理是IL注入程序集，Lua层使用hotfix方法来将所有要替换的Lua函数注册到一张全局表中，而C#层在注入完成后所有带有[Hotfix]属性的方法在执行之前会判断全局表中是否有该函数，如果有那么调用该函数而非原本的C#方法。在替换完成后，这个过程中就存在C#对于Lua的调用，所以需要提前生成桥接文件，这样效率比较高。

[深入理解xLua基于IL代码注入的热更新原理\_luaenv.realstateptr-CSDN博客](https://blog.csdn.net/m1234567q/article/details/134082574)

<https://blog.csdn.net/m1234567q/article/details/134082574>

1. **C#和Lua内存交互的原理**

桥接文件的作用在于原本运行时做的反射查找工作，提前转成编译期生成的静态代码。

1. **Unity开发时经常会用到的生命周期方法**

Awake：在脚本实例化即进入场景的时候调用一次

OnEnable：在每一次脚本激活时调用

Start：在第一次Update之前调用，也是必须在脚本启用时才会生效

Update：每一帧调用一次

LateUpdate：在Update调用之后再进行调用

FixedUpdate：固定时间调用，常用于物理系统的运算

OnDisable：在每一次取消激活时调用

OnDestroy：对象被销毁时调用

1. **协程的原理**
2. **ECS架构的优势**
3. **分析羊了个羊和黑神话悟空两个游戏**
4. **大致的职业规划**

# 途游二面

1. **大概讲一讲自己的项目**
2. **C++的Map各自的底层的数据结构，插入时会发生什么动作，哪一些会产生扩容**

Map和MultiMap底层是红黑树，在插入时会通过一系列的操作进行调整。Unordered\_map是底层哈希表也是动态数组，在插入值的时候会发生动态的扩容

1. **Vector在插入时可能会发生什么操作？**
2. **静态多态和动态多态？虚函数介绍一下**
3. **静态多态的偏特化是什么？如何判断一个类中是否存在一个函数**

使用概念和约束来实现

1. **Move和Forward分别的作用**
2. **写过Shader吗，项目的具体实现**
3. **手撕，保存1~1000的乱序数组，删除其中的一个数字如何通过一次遍历得到缺失的数字**

直接求和然后相减

# 雷火一面A面

**1.template<class T> void f(T){**

**Static int i = 0;**

**Cou<< ++i;**

**}**

**Int main(){**

**f(1);**

**f(1.0);**

**f(1);**

**}**

**输出是什么？**

输出是112，每一个参数类型都会实例化出一个具体的函数，实例化出的函数会被放到.o文件中所以才被称为编译时多态。

**2.** **64位系统中，int\*\* a[4][4]，sizeof(a)占多少字节**

本质的数据类型就是指针，而在64位的系统中指针的大小固定为8个字节，所以16\*8=128字节。

**3.** **struct A{**

**int a;**

**char b ;**

**int c;**

**char d;**

**}**

**struct p{**

**struct A w[2];**

**stort b;**

**struct A\* p;**

**}**

**4.** **class A{**

**public :**

**A(){printf(“A”);}**

**A(const A&){printf(“b”);}**

**A(A&&){printf(“c”);}**

**A operator = (const A&){printf(“d”);}**

**};**

**class B:public A**

**{**

**public:**

**B():A(){printf(“1”);}**

**B(const B& b):A(b) {printf(“2”);}**

**B(B&& b):A(b){printf(“3”);}**

**B operator = (const B&){printf(“4”);}**

**};**

**int mian()**

**{**

**B b;**

**B m(b);**

**B n=b;**

**B q(std::move(b));  
}**

基类的构造函数先于子类的构造函数调用。所以调用的顺序是

：A的无参构造函数，B的无参构造函数

：A的有参构造函数，B的有参构造函数

：注意，赋值运算符仅在变量定义完成后才生效，在变量定义时使用赋值运算符执行的是构造函数，所以此处执行的顺序的A的有参构造函数，B的有参构造函数。

：左值在通常的情况下是不能引用右值的，但是const的左值即可以引用右值也可以引用左值。

**5.从1~9这9个数字中随机选出3个不重复的数字，和为奇数的组合有几个？**

排列组合即可

**6.有一个A物体和一个B物体，如何判断B物体在A物体的前方？（从向量角度考虑这个问题）**

B的中心位置减去A的中心位置，然后将得到的向量点乘方向向量，如果结果为正则表明B在A的前方

**7.讲一讲Lua中的静态函数和静态变量**

Lua里面是没有静态函数和静态变量的

**8.MVC架构的优势和劣势**

优势在于更加清晰的系统结构以及更加明确的功能边界划分

劣势在于更高的复杂度

# 雷火一面B面

**1.如何判断两个球体是否发生了碰撞，如何获得碰撞点？**

计算球心的距离和半径之和，如果球心的距离小于半径之和则发生了碰撞。

计算球心连线的单位向量和半径之和，两张相乘再加球心即为碰撞的点

**2.如何获得点到射线之间的距离？**

取射线起点与目标点连线的向量点乘射线方向的单位向量得到三角形的两条边再用勾股定理求垂直边。

直接求方向向量和射线起点与目标点连线的向量叉乘得到平行四边形的面积，又方向向量是单位向量所以此面积就是就距离。

# 灵犀互娱一面

**1.有了解过Vector的底层的实现吗？怎么扩容？迁移的过程是什么？会触发什么样操作？比如原本的对象你就不管他就直接迁移了吗？会调用构造函数吗？对象有一些堆的内存，那么在迁移后会发生什么呢？比如有一个数组，那么在迁移后会产生一个新的数组吗？这个是我能定制化的吗？**

扩容的过程是确定新容量；申请新的空间；在新的内存空间上拷贝/移动构造新的对象，所以实际上迁移后的对象是原本对象的副本；调用原本内存对象的析构函数，随后回收内存；更新指针和容量。这个过程是可以通过修改拷贝构造函数和移动构造函数来实现的。如果在堆上面那么在移动的过程中会再次分配内存带来巨大的性能开销，而原本在堆上分配的内存会在原本对象析构的时候被释放掉。数组会逐元素复制。

**2.List有实现过吗？List的底层是什么？假如你要实现一个快速排序，那么你会选择使用**Vector还是list呢？为什么使用Vector？Vector访问下标的时间复杂度是多少？List呢？

List的底层是双向的链表，实现快速排序必须使用Vector，因为快速排序需要计算下标分段，只有vector才支持随机访问。vector访问的时间复杂度是O(1)，list访问的时间复杂度是O(n)。

**3.Map底层的实现？红黑树的优点？查询的时间复杂度？**Map的底层实现是红黑树。其性能开销比较平衡，查找和插入删除的性能都比较优秀。查找/删除/插入的时间复杂度都是O(logn)

**4.快速排序的时间复杂度？是稳定的吗？快排的最差情况是什么？其时间复杂度是多少？**

快速排序的平均情况是O(nlogn)，是不稳定的，最差的时间复杂的是O(n)，每次的选定值都是最大值或者最小值。

**5.TopK问题**

维护一个小顶堆。

对于前K个值直接插入建堆，插入到最后然后逐级向上比较。

建堆完成后，将新加入的值与顶部的值进行比较，如果新插入的值更大，那么进行替换并进行调整，调整的过程是直接比较子节点中较小的值，如果父节点更大那么与其交换，并递归的进行这一步骤。

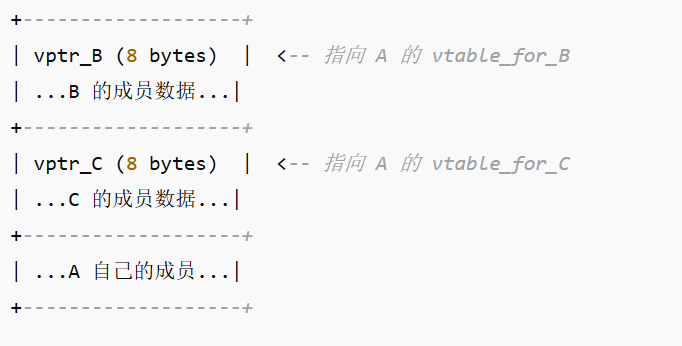
**6.虚函数有了解过吗？如果我不用虚函数而是直接去重写这个函数怎么样？在什么时候会发生多态失效？**

虚函数的关键点在于虚函数表和虚函数指针。那么函数会被直接覆盖，直接执行声明类型的函数而不是实际类型的函数，虚函数的多态只有在使用指针的时候会生效。

**7.虚函数表存放的位置？那么这个算是一个静态的多态吗？如果A继承至B，C那么会有两个虚函数表吗？**

虚函数表在编译阶段生成在静态存储区的只读段，还是算是运行时多态。如果A同时继承至B，C那么会有两个虚函数指针分别指向B部分的虚函数表和C部分的虚函数表。继承的实质是创建子对象，继承后在实例化对象的时候会在对象内存中先实例化出一个基类的对象，如果同时继承至多个类，那么会实例化出多个基类的对象，所以会有多个虚函数指针。

而子类自身是不带有虚函数指针的，其所含有的虚函数指针都是来源于基类的对象。



**8.有了解过左值右值吗？移动语义有什么作用？移动构造函数跟普通拷贝构造函数构造出来的对象有什么区别吗？剥夺资源指的是什么？**

左值就是引用，相当于对资源取了一个别名。

在C++中右值包含两个部分，一个是纯右值，也就是完全没有地址的纯粹的临时变量，另一个是将亡值，也就是即将被销毁的变量。

移动语义的作用实际上是声明该变量可以被安全剥夺，而不是实际的剥夺的逻辑，声明该变量可以被安全的剥夺的话就可以通过移动构造函数来进行实际的剥夺的逻辑。

在移动构造函数中会调用每一个成员的移动构造函数，如果没有移动构造函数那么就会退化为拷贝构造函数。

**9.堆上的数据和栈上的数据的区别？那他们的回收会有什么区别吗？在栈上自动会收的话什么时候会回收呢？生命周期是怎么样的呢？能举一个例子吗？如果一个函数返回一个对象的话那么生命周期是怎么样的呢？会调用几次构造函数呢？**

栈上的内存由计算机自动分配，空间比较小但是由于数据连续所以查找的速度更快；堆的内存由程序员手动分配和释放，空间更大但是由于数据离散存放查找的速度更慢。

栈上的内存在对象的生命周期结束的时候被计算机自动回收，而堆上的内存需要程序员自行进行回收。

函数返回一个临时对象的话实际上这个临时对象会在函数调用者的内存空间中再复制一个临时对象，再将这个临时对象拷贝给外面的对象。则在这个过程中一共调用了三次构造函数，如果开启编译器优化的话会将临时对象直接在接收对象的位置进行构造，整个过程只会调用一次移动构造函数。

**10.C#和C++的区别**

C#和C++最主要的区别来源于C#需要编译为IL中间语言在CLR虚拟机上面运行，C++直接编译为机器语言运行。这也是为什么C#的编译效率更高但是执行效率更低，C++编译的效率更低但是执行的效率更低。

**11.讲讲C#的GC机制？每一次都要从根节点开始遍历吗？**

标记：从根节点出发，标记所以仍然被访问的对象

清除：清除所有没有被访问的对象

压缩：对内存进行整理，减少碎片

GC会有迭代机制

**12.有听说过他们的装箱拆箱吗？性能开销体现在什么地方？**

装箱/拆箱指的是将值类型转换为引用类型再将引用类型转换会值类型的操作。

性能开销主要在于向装箱的过程会在堆上面开辟新的内存空间并将值类型的值进行复制，拆箱时又会将堆上的值重新复制到栈上。

**13.值类型和引用类型的区别？哪些是值类型？字符串算是什么类型吗？**

值类型在栈上面分配内存变量直接存放实际的值，引用类型在堆上面分配内存变量存放的是对象的引用，而这个变量是在栈上面存放的。基础变量和结构体是值类型，自己声明的类一般是引用类型。字符串是引用类型。

**14.TCP和UDP的区别？什么算是不太关键的问题？TCP的延迟出现在哪里？ TCP的拥塞控制？滑动窗口的作用？可以发窗口长度数量的包？丢包会发生什么？快速重传是什么？**

**为什么王者荣耀使用UDP保证安全？**

最大的区别在于TCP稳定有链接和重传机制，UDP不稳定没有重传机制。TCP的延迟有三次握手，和丢包重传带来的延迟，主要是丢包重传。

TCP的拥塞控制解决网络传输路径过载的问题，全局，由拥塞窗口进行控制；流量控制解决网络传输过载的问题，端对端的局部，由接受窗口进行控制。发送方实际的发送窗口的大小同时取决于拥塞窗口和接受窗口的大小，不能大于这两者。

流量控制中的接受窗口大小由接收方当前的接受缓存决定，接受缓存大说明可以接受更多的包那么接受窗口的大小就较大。

TCP的确认机制是累加确认，在接收方接受到包的时候那么会返回下一个希望接受的包的ACK号。不需要等发送窗口中的所有包都被接受了再滑动而是接受一部分就滑动一部分。

TCP拥塞控制的阶段：

1. 慢启动：每一个RTT发送窗口的大小变为原本的两倍。
2. 拥塞避免：当发送窗口的大小大于拥塞阈值的时候每一个RTT发送窗口的大小加一
3. 快速重传：如果丢了一个包，正常情况需要等超时才重传，效率低所以引入了快速重传机制，如果出现三次重复ACK，那么也会判定为丢包
4. 丢包的处理：在超时时，认为当前的拥塞非常的严重所以，cwnd直接变为1，拥塞阈值变为当前cwnd的1/2。如果接受到3次重复ACK则认为当前的拥塞不是很严重，拥塞阈值仍然变为当前cwnd的1/2，而cwnd直接变为拥塞阈值的大小从拥塞避免阶段开始。

也就是说超时是我们常规的情况，而三次重复ACK的快速重传机制是我们引入的优化内容。

**15.线程和进程的区别？线程会持有资源吗？比如哪些资源？**

线程是基于进程的，进程是资源分配的基本单位，线程是资源调度的基本单位。

线程栈，保存临时变量，函数调用过程。线程的本质还是函数的执行，所以也会有像函数调用一样的栈帧。

程序计数器，保存当前执行的指令。

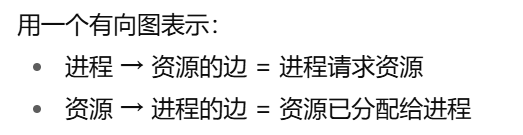
**16.什么会导致死锁。如何避免死锁？如何检查出现了死锁？出现死锁如何解决？如果直接剥夺资源的话不会打断当前线程正在执行的内容吗？**

互斥条件。不剥夺条件。请求保持条件。循环等待条件。

检测出现死锁：

使用银行家算法，判断当前是否存在一条可以安全执行的路径

使用资源分配图，存在环时则表明出现死锁。

****

解除死锁常用的方法有资源剥夺，进程撤销，进程回退。

进程回退需要提前保存安全状态来确定回退的位置。

**17.Unity的协程和线程的区别？**

协程是更加轻量级的线程，其本身也是运行在Unity的主线程上。其本质是一个迭代器对象，通过current来标定当前执行的阶段和继续执行的条件，通过movenext来执行下一个阶段的函数，执行到yield的时候将函数挂入阻塞队列中，同时每一帧检测当前协程是否可以执行。

**18.讲讲Unity的渲染管线？MVP的计算？矩阵要修改哪几个值？矩阵乘法交换顺序的话会不会影响结果？**

应用阶段：CPU获取渲染所需要的信息并发出DrawCall指令让GPU进行渲染。

几何阶段

顶点着色器：通过MVP等一系列矩阵变换将模型从局部空间移动到裁剪空间，并通过齐次得到NDC坐标。

裁剪：将不会被渲染的点裁剪。

屏幕映射：将模型的坐标映射到屏幕上

光栅化阶段

三角形设置：将前面得到的点连接成三角形

三角形遍历：判断一个像素是否可以被一个三角形覆盖，如果是那么生成一个片元

片元着色器：计算每一个片元的实际颜色信息

逐片元操作：经过深度测试等一系列测试判断片元是否会被输出

V矩阵实际上是将游戏物体从世界坐标移动到摄像机观察空间，通过数学计算我们可以得到坐标点的变换只需要将原始点左乘坐标轴的变换，所以在这个过程中我们只需要得到坐标轴的变换。

取世界坐标的UP向量叉乘相机位置指向观察点的方向向量得到右方，再将右方叉乘方向向量得到上方，就可以得到点需要进行的旋转和平移的矩阵。

P矩阵本质是一个相似三角形的投影的过程，关键坐标c是相机的z坐标。

会影响的，就比如在View变换中，我们的旋转变化的中心点是原点，而不是物体的中心点，先移动再旋转会导致物体发生位移。

**19.为什么要将透明物体和不透明物体区分开的渲染？不透明物体重叠会出现什么问题？假如记录片元的深度和颜色数据再进行渲染会有什么问题？**

因为透明物体要关闭深度写入并与底色混合。

假如有三个游戏物体ABC，B是不透明的游戏物体，那么在渲染的过程中会先渲染B并写入深度缓存和帧缓存，到渲染透明物体的时候C的深度小于深度缓存不进行渲染，A物体读取帧缓存中的值并计算颜色得到最后的实际颜色再写入帧缓存。

会导致无法进行排序。

最核心的解决方法就是进行片元级别的排序。有英伟达提出的深度剥离技术，通过多个pass将不透明物体分层渲染最后混合；或者逐像素链表技术，将一个像素的所有片元进行保存，最后通过一个Pass进行深度排序混合；这两个都可以精准的得到结果，但是开销都比较大。

[(35 封私信 / 52 条消息) 顺序无关的半透明混合（OIT）相关方法 - 知乎](https://zhuanlan.zhihu.com/p/353940259)

深度剥离技术：

第一个Pass开启深度写入并得到一层颜色

后面的所有Pass关闭深度写入并将离上一层最近的更深的片元渲染为一层，同时得到一个当前层的深度纹理，后面的Pass重复当前的过程，注意每一个Pass都是一个完整的渲染过程，在一个Pass中直接使用上一层的深度来做测试即可。

最后从后往前进行混合。

在这个过程中需要使用一个单独的深度纹理，并且保存每一层的渲染结果，同时还要完整渲染很多遍，时间和空间的开销都很大。

也可以使用近似的方法，加权混合，把距离作为一个参数进行考虑，效率更加优秀但是效果没有上面两个好。

[顺序无关半透明物体渲染OIT | Blurred code](https://www.blurredcode.com/2022/05/4c17b61d/)

**20.模板测试会用到什么地方？**

模板测试是程序员定义的测试，在光栅化之后进行。

最常用的地方是用来遮罩效果。这个过程是不可以编程的，但是可以配置的。其中主要起作用的点是模板缓冲中的模板值，片元自身携带的参考值，比较函数以及在比较完成后对模板缓冲进行修改的操作函数。

使用两遍Pass，第一遍Pass将需要绘制的部分的模板值设置为1，后续再在模板值为1的地方进行实际的绘制。

**21.热更新工具是什么？Shader变体收集工具的用处？**

热更新工具。

在一个Shader变体首次进行使用的时候会进行编译，这会带来一定的卡顿。所以我们可以将项目中使用到的shader变体存入一个SVC文件在构建阶段就将其提前编译以避免使用时卡顿。

**22.为什么使用Lua来进行热更新？如何在运行时做更改？如果想要自动检查修改的文件应该如何做？**

因为lua是一个解释语言，不需要提前进行编译，是在执行的过程中进行编译。

直接重新加载文件即可。

自动检查修改的文件可以通过git等版本管理的工具来实现。

Package.loaded这个表保存了所有已经被require的包，require的时候会去这张表里面查，如果没查到再去加载，只要将其中的目标项设置为nil后再require即可实现重新加载package.loaded[“A”] = nil;

**23.有了解过面Lua向对象吗？用过其他元表吗？\_\_newindex的作用？在有值的时候还会调用吗？如何封装一个外界可以访问但是无法修改的表？如何跨过\_\_newIndex来访问\_\_index指向的表？**

我认为Lua的面向对象很难被称为常规的面向对象，因为对象实例本身与类是在运行时强耦合的。

用过\_\_index,\_\_newIndex,和一些运算相关的原表。

\_\_index 是取值缺失时调用，\_\_newindex是赋值缺失时调用。

不会进行调用。

通过将封装表的\_\_index设置为实际的数据表，\_\_newIndex设置为其他表，通过这一层封装表的访问就可以实现只读的效果。

通过getmetatable来得到元表，注意一个表的元表只能有一个通过setmetable来设置的话会将原本的原表直接替换掉，得到原表后可以直接通过\_\_index字段得到封装的数据表，随后就可以直接修改数据表。

**24.公司的项目会在什么地方使用ECS？**

公司项目的战斗部分使用了ECS，但是这个部分确实没有过多的研究因为不是我负责的模块吗，只是在做有关内容的时候有看过一点。

**25.UGUI优化的方式？有什么会打断合批？位置的改变打断合批本质是什么？那么UI会有深度测试吗？那么会有冲突的问题吗？为什么不会出现闪烁的效果？**

1.将UI元素进行分层，经常修改的元素和静态的元素分层处理。Unity在处理UI的渲染时会将一个Canvas下的所有UI元素建立为一个Mesh网格，如果其中一个元素的顶点或者其他数据发生变化那么会重建整个网格，将整个Canvas进行重新绘制带来巨大的开销。注意image的颜色也是顶点数据，修改时也会导致重建。

2.启用合批。UI在GPU层面本质都是矩形的Mesh，像正常的模型Mesh一样。在启用合批的情况下可以将使用相同纹理的UI元素的Mesh进行合并并一同进行渲染，减少DrawCall的次数。注意判断哪些元素可以合批的逻辑是判断当前的UI元素与前一个(在Unity层级窗口的顺序)UI元素的纹理是否相同，注意是前一个，所以如果一个Canvas下的UI元素不全是一个图集的话就会发生出现DrawCall没有减少的情况，打断合批就是指的这个，即中断当前的批次，再开一个新的批次。合批要求两个UI元素的渲染信息完全一样，即贴图，Shader等影响渲染的因素。

UI不会有深度测试，因为UI元素本身的z轴数据相同，如果使用深度测试来决定遮挡关系会出现问题，比如浮点数精度导致闪烁现象。所以UI元素在渲染的过程中是关闭了深度测试的，依靠渲染的顺序来决定当前像素的实际颜色，而渲染的顺序又是由Canvas排序和物体在Unity层级窗口中的顺序决定的。

# 腾讯一面

**1.简单介绍ECS架构具体写了什么内容。预先缓存写了什么内容？**

编写了实体，组件和系统的各个模块。其实是将所有的需要在代码中进行访问的游戏物体提前缓存到UIBase脚本的一个字典中，从而在查找的时候提高性能**。**

**2.Lua和C#相互调用速度慢的原因是什么？**

最主要的是Lua和C#的内存空间和数据结构完全不同在使用的时候需要进行转换，而这个转换就会带来开销。Lua和C#的参数传递是通过Lua栈来实现的。

其次是虚拟机的上下文切换。

**3.简单描述Shader变体的过程？如果变体没有收集全会发生是什么？如何避免这些没有收集到的Shader在运行时编译带来性能开销？**

收集整个项目中的所有Shader和模型，然后把它们应用到每一个场景，同时通过反射调用Unity自身的静态私有方法进行收集并写入SVC文件。

Shader在运行的时候会进行编译带来开销，我们可以提前将项目中所有使用到的Shader提前编译以降低运行时开销。

没法避免。

GPU是单指令多数据的架构，同时只能执行一条指令处理多条数据，如果使用ifelse语句那么路径不同时会被视为不同指令对并行带来非常大影响。可使用lerp来替代逻辑判断，t为1则完全为A，为0则完全为B。

1. [封私信 / 56 条消息) shader中用for，if等条件语句为什么会使得帧率降低很多？ - 知乎](https://www.zhihu.com/question/27084107)

**4.渲染流程**

应用阶段：CPU将准备好的数据传递给GPU并发出DrawCall指令

几何阶段

顶点着色器：通过MVP变化将模型从局部坐标系移动到裁剪空间，同时通过齐次除法得到NDC坐标

裁剪：将不会被渲染的点进行裁剪。

屏幕映射：将点坐标映射到屏幕。

光栅化阶段

三角形设置：将前面得到的点链接为三角形。模型文件中保存的只是顶点索引数据而非真实的顶点数据。

三角形遍历：判断一个像素是否为三角形覆盖，被覆盖则生成一个片元。

片元着色器：计算片元的实际信息。

逐片元操作：通过模板测试，深度测试和颜色混合等最后将片元写入帧缓存。

**5.多态。模板可以给特定的类型参数指定一个特定的类型实现吗？如果在构造函数中调用虚函数会发生什么？如果继承了两个类那么虚函数指针的位置会在哪里？那么会有几个虚函数表？如果BC同时继承A，D继承至BC那么D中会有几份A。**

可以也就是偏特化，甚至可以实现对具有特定数据成员的类指定实现。如果在构造函数中调用虚函数，那么此时虚函数还没被覆盖，会指向父类的虚函数表直接执行父类的虚函数导致多态失效。

继承的本质是在子类的内存空间生成基类的子对象，所以继承两个类的话会在两个子对象各自内存开始的地方存放虚指针。两个虚函数表。两份A，就向前面所说，会生成完整的子对象，也就是会有两份。

注意如果此时A中也有虚函数的话那么也是两个虚函数指针，两张虚函数表，虽然也会有A的子对象，但是A的子对象中的虚指针会被覆盖，BC中的虚指针实际也就是A的虚指针。

**6.智能指针了解过吗？有哪些智能指针？自己实现一个的话你会怎么做？Unique是如何实现不让别人拷贝他的资源？shared\_ptr你认为是如何实现的？**

了解过，可能会使用RAII技术，也就是说将资源的申请放到类的构造函数中，资源的释放放到类的析构函数中。

需要删除掉拷贝构造函数以及重载赋值运算符，仅保留移动构造函数。

使用一个引用计数来保存当前对于资源的引用，仅引用归0时才释放资源。

**7.有了解过右值吗？什么是右值？右值的应用有哪些？可以使用左值引用右值吗？可以创建引用的引用吗？对于编辑器来说他会怎么处理引用的引用？左值引用的右值引用是一个什么东西？即创建一个左值引用的类，使用右值引用引用这个会发生什么？**

右值在C++中有两种，一种是纯右值，各种字面量和执行过程中产生的临时变量；另一种是将亡值，表示可以被剥夺资源的左值。

右值引用本质是一个标识，表示当前的资源可以被剥夺。当引用一个临时变量时，那么会将该临时变量的生命周期与右值引用同步；引用一个字面值时会创建一个临时变量然后再进行引用；将亡值大部分情况下是需要通过move来手动转换的。

可以，右值本质就是一个没有具名，编辑器不希望我们进行修改的临时值，所以我们使用const int& 也可以实现相同的效果。

可以，本质上是传递原始对象，所以实际上就是对原始对象进行操作 。

左值引用的右值引用部分可能是在考察引用折叠的相关内容。

**8.线程和进程的区别？进程之间通讯的机制？什么时候会进入内核态？**

线程基于进程的，同时线程是系统调度的基本单位，进程是资源分配的基本的单位。

进程之间通讯可以通过管道，公共存储区或者网络通讯也就是socket那一套，网络通讯的本质也是进程间的通讯。socket本身是一种系统调用。

我们编写的程序本身是无法调用比较底层的功能的，比如进程管理，内存管理和磁盘操作。这些内容都需要通过系统调用来实现，而系统调用就会使操作系统进入内核态。除此之外中断也会使操作系统进入内核态进行处理。

**9.虚拟内存的概念有了解过吗？为什么我们要做虚拟内存？通过逻辑地址访问物理地址这件事情是怎么做的？每一次都要查页表吗？CPU一般有几级Cache？为什么不使用一个大的Cache而是采用分层？除了Cache还有别的什么地方可以有缓存？离CPU最近的地方对比其他Cache有没有什么特别的地方？**

虚拟内存的作用在于从逻辑层面扩充物理内存，让程序认为自己有一块连续而完整的内存空间，但是实际上这些内存空间可能离散存放。直接计算块号然后通过页表或是快表查找对应的物理块号，再通过块内偏移得到实际的物理地址。不一定，在查页表之前会先查找页表，如果快表中有缓存那么不会查找页表。一般有三级Cache。Cache要求查找的效率够高，采用一个大Cache的话会降低查找的效率同时也会增大成本，大Cache的组号需要解码更多行同时Tag的比较数量也会增加。还有快表和页缓存。离CPU最近的Cache会将指令和数据分区存放进一步提升访问的效率。

**10. 多核CPU如何保证数据的一致性？单核CPU同时只能跑一条指令还是多条指令？有没有什么情况导致不能使用流水线？分支指令会提前跑还是等待判断条件执行？**

**没有正确的预测会发生什么？**

CPU访问的是相同的内存，但是前两级Cache中的缓存是每一个CPU所独有的，所以可能会出现Cache中的副本与内存中的实际值不同的情况，所以引入了缓存一致性协议，每一个Cache行都有一个标记位，在CPU修改内存数据后会通知所有其他CPU将该行的标记位修改为已污染。写Cache后可以直接每一次都同时写入内存，也可以在写完后被替换时再写入内存，这个方法需要缓存一致性协议，更新数据可以直接从另一个Cache拿。

多条，可以使用流水线。数据相关，控制相关，硬件冲突。数据相关也就是后续的指令需要使用前面的指令的结果。控制相关是指出现了分支跳转指令，此时CPU不知道下一条指令的正确地址只能采取预测的形式来执行下一条指令，但是如果预测出错就会出现回退导致流水线在这个过程中所做的处理失效。硬件冲突，硬件的数量不足以支撑所有计算并行处理。

分支指令会通过预测的方式提前执行，如果预测出线错误那么再进行回退，也就是前面提到的失效。

**11.数组查找和插入的时间复杂度是多少？如果希望实现一个插入和查找效率均衡的内容你会如何实现？**

数组按值查找的时间复杂度为O(n)，如果是通过下标随机访问的话那么时间复杂度为O(1)。

我可能会使用红黑树。

**12.哈希表的访问和插入的时间复杂度是多少？解决哈希冲突的方法？**

哈希表平均的访问和插入的时间复杂度都是O(1)但是在最坏的情况下哈希表会退化为链表，此时的时间复杂度为O(1)。

解决冲突的方法有：开放地址法和链表法。

**13. 用两个栈实现一个队列你会怎么实现？每一次都需要Pop吗？**

一个栈用于Push，一个栈用于Pop，如果用于Pop的栈的元素个数为0，那么将另一个栈的元素全部加入Pop栈中。

**反问：**

**有什么不足？**

**对于项目中的技术可以了解得更加深一些。**

**对于操作系统的内容也可以了解得更深一些，对于概念的理解也可以更加透彻。**

**对于代码能力要再加强一点。**

**希望招一个什么样的人？**

**不太看重项目，更加注重基础和学习能力。**

**面评？**

# 抖音一面

**1.const关键字(修饰变量和函数)**

const表示该值不可以重新赋值，本身只是一个修饰符不会影响变量的存储位置。如果是全局的变量那么会存储到静态存储区，如果是局部的变量那么会被存储到存放到栈区。如果是类成员，那么存储的位置跟随对象。

但是有一个例外是如果变量的值在编译时即可确定那么编译器会采取优化，将该变量视为编译器常量，直接用值来进行替换，这种情况下变量甚至不会出现在内存空间中。但是如果对这个变量进行了取地址，那么会导致优化失效，强制为变量分配内存空间。

const本身无法修饰全局的函数，只能修饰成员函数，在修饰成员函数时会导致成员函数的this指针变为const，从而无法访问其他成员函数和成员变量。

**2.Static关键字**

修饰成员变量时这个成员变量将属于类，在所有对象之间共享。

修饰成员函数时，该函数不再具有this指针，只能访问静态的成员变量。

修饰全局变量或者函数时，该变量或者函数的连接属性会变为内部，不再被其他文件引用。

修饰局部变量时，该变量会在整个过程中只初始化一次。

**3.C++内存布局**

栈，堆，静态存储区，代码段。

**4.new和malloc的区别**

new是一个运算符，本质是对malloc的封装。

new在malloc的基础上增加了更多上层的功能，比如调用构造函数可以分配数组的空间。

new在分配失败时直接抛出异常，malloc在分配失败时返回null。

new返回实际类型的指针，而malloc返回万能指针，需要自己进行转换。

new分配数组和分配变量的指针本质完全一样，只是在语义上有区别

T\* a;a++;本身是重载了++运算符的，实际增加的大小取决于T的大小。

**5.野指针和悬空指针**

野指针指未初始化或者说指向非法地址的指针。而悬空指针指的是指向对象已经被释放的地址的指针。

**6.智能指针**

**7.C++多态**

**8.构造函数能否声明为虚函数？**

**9.Stl的map和Unordered\_map的底层的实现，哈希表和红黑树的增删改查的时间复杂度**

哈希表的增删改查的平均时间复杂度都是O(1);

红黑树的增删改查的时间复杂度都是O(logn);

**10.快速排序和堆排序。排序算法的时间复杂度。**

快速排序和堆排序的时间复杂度都是O(nlogn)。

快速排序采用的了分治的思想，将数组视为多个数组进行排序。

常见的排序一共有三种时间复杂度级别，插入排序，选择排序，冒泡排序都是O(n^2)，归并排序，快速排序，堆排序O(nlogn)，基数排序O(n);

堆排序借助了堆顶始终是最大最小值的思想，底层使用数组。

建堆：从后往前判断每一个非叶子节点是否均大于其子节点，如果不大于那么将此节点与叶子节点中较大的值进行交换，然后再判断交换后是否仍满足。

排序：建堆后直接将堆顶元素与堆尾元素交换，同时将堆的规模减一再进行调整，并不断重复这个过程直到规模变为1。

基数排序没有进行比较

每一轮遍历将当前位的值挂到对应数字下面，然后按序取出进行下一次比较。

**11.在浏览器输入一个网址并返回网页的过程。为什么不是两次握手三次挥手？**

本地解析URL。检查本地缓存是否有该域名对应的IP地址，有则直接使用。没有则进入DNS解析。

DNS解析URL。递归的查询IP地址。

三次握手建立TCP连接。

发送HTTP请求。

服务器响应HTTP请求，返回响应报文。

解析HTML文件加载子资源。

四次挥手断开TCP连接。

三次握手的过程：seq表示当前发送的序列号，ack表示希望接受到的包号。

客户端发送：SYN=1，seq=x

服务器发送：SYN=1，seq=y，ack=x+1

客户端发送：seq=x+1，ack=y+1

两次握手主要会导致两个问题，会出现幽灵连接的情况，以及缺乏双向确认机制可能导致服务器和客户端机制不同。

四次挥手的过程：服务器发送数据和客户端接受数据是两个独立的通道，在关闭TCP连接的时候需要将这两个通道进行独立的关闭。同时两个通道的关闭都是相互独立的，可以服务器先关也可以客户端先关。

以客户端先关为例：

客户端发送：FIN=1，seq=x

服务器发送：ACK=x+1

服务器发送：FIN=1，seq=y

客户端发送：ACK=y+1

**11.TCP的可靠传输，流量控制和拥塞控制**

TCP的可靠传输的核心点在于序列号和累加确认。

TCP滑动窗口的实际大小取决于接受窗口的大小以及拥塞窗口的大小，这两个窗口实际上也就是流量控制和拥塞控制的体现。

接受窗口的大小取决于接受方的缓存区，缓存区空闲大的话那么接受窗口就大。

拥塞窗口的大小是动态的，有慢启动和拥塞避免两个阶段，以及三次重复ACK带来的快速重传快速回复机制。

**12.Git的pull和fetch的区别**

fetch是更新本地的远端分支，而pull是在更新的同时将本地的远端分支和本地分支进行合并。

# Lua相关面试内容

**1.Lua脚本的作用是什么？**

在游戏行业最主要的用处在于热更新。

Unity在PC端时会进行正常的构建流程，也就是先将C#代码编译为IL然后再在第一次运行时由JIT将IL构建为机器语言，这个过程其实是可以通过修改程序集实现热更新的，因为JIT会将新的程序集进行编译。但是Unity在打包为移动端时会将IL代码直接转为C++然后再转为机器码，此后的整个运行环境中就没有IL和JIT了，即使将新的DLL下载下来也无法编译为机器语言执行。

即使实现了在移动平台上通过JIT实现热更新，但是也只仅限于安卓，IOS明确禁止了JIT，因为JIT会在第一次执行时到具体函数时才编译，这样会生成新的可执行代码，这对于IOS来说是禁止的。

而Lua本质是一个文本文件，通过虚拟机在运行时将其直接解释为字节码运行，依靠虚拟机执行跨平台能优秀。

解释语言指的是将代码编译为中间语言交给虚拟机执行，编译语言将代码编译为机器码直接交给CPU执行，即时编译语言将代码提前编译为中间代码再交给虚拟机编译为机器码执行。

**2.简述元表和元方法的概念和使用**

元表是为普通表附加特殊行为的表，元方法是定义这些特殊行为的方法。

在一个表里面增加\_\_index和\_\_add的字段，然后通过setmetable将其直接设置为另一张表的元表。

**3.ipairs和pairs的区别**

ipairs只能遍历表中顺序的部分，而pairs可以遍历表中的所有的部分，只是不保证有序。

pairs本质是底层存储结构的遍历，但是有部分数字键被保存在了哈希的部分就会导致看起来无序。

而ipairs是按顺序键值访问，如果n不在数组部分那就去哈希部分找，如果还没有就中断查找。

如果n小于数组的容量那么表明键值n在数组中，直接去数组部分找，不在则去哈希部分找。

**4.阐述lua表的原理和表的结构，以及数组部分和哈希表部分如何增长**

表是lua中的唯一数据结构，lua中的所有数据结构都是由表来构成的。其底层由顺序部分和哈希部分构成。数组部分在增长的会比较当前插入的键是否足够密集，如果比较稀疏的话会将其插入哈希的部分，而哈希表的部分就是正常的哈希表，在比较拥挤的时候会触发重新分配空间，此时会将整个表重新来分配一遍，也要进行上述的密集判定。无论是哈希部分还是数组部分触发扩容时都会重新进行一次分配。

**5.Lua的面向对象**

Lua中的面向对象的继承和创建对象的本质都是相同的，最后的结果都是得到一张表。

实现的过程也就是将子类或者对象的\_\_index指向父类。

**6.Lua热更新**

以XLua为例，Lua热更新的本质是IL注入，在所有需要进行Lua热更新的C#代码部分提前打上[hotfix]标签，随后这个方法对应的IL部分会增加方法是否被Lua层重载的条件判断，在Lua直接使用xlua.hotfix(类，方法，新方法(self)

end)

后这个方法会被注册到热更新的表，对应的C#方法在执行前会通过热更新的全局表判断当前方法是否有被Lua热更新，如果有那么调用Lua层。

**7.Lua的基础数据类型有哪些？**

String，Number，Bool ，UserData，Function，Table。

但是与常规的强类型语言不同，这里的类型指的是数据的类型而非变量的类型。

Lua向C#层传递的UserData和Table是引用其他的是副本。

**8.Lua中的loadfile，dostring，dofile，require的区别**

loadfile只是编译加载文件的内容，dostring将字符串编译为一个函数，dofile在loadfile的基础上还多了执行，require将模块加载入全局表。

**9.Lua中哪些数据类型使用GC机制**

Table,func,userdata。这些存放在栈上的数据类型。

**10.Lua的GC机制**

Lua的GC机制和C#的机制有一点像，都是采用了标记清除，但是Lua更加轻量，其不具有压缩的这个过程同时也不具有C#GC的分代机制。

# 腾讯面试

**1.cosnt关键字的作用，当const修饰函数时起到什么作用？**

const关键字在C++中用于将变量声明为只读，其无法修饰非成员函数，所以在这里我们仅讨论成员函数，在const修饰成员函数时会将this指针引用的对象变为const导致无法修改对象的成员变量。

**2.const变量存储于内存的什么位置？**

cosnt本身只是一个修饰符，不会修改变量的存储位置，所以变量实际存储的位置就是变量原本的位置。

但是有一种例外，如果变量的值在编译时即确定，编译器会将该只读变量视为编译器常量，直接将变量替换为值，此时在内存空间中根本不会存在这个变量。

**3.explicit关键字修饰构造函数的作用**

explicit是用来禁止隐式转换，在C++中只能用于修饰类型转换运算符的重载和构造函数，在修饰构造函数时禁止通过传入构造函数的参数来隐式构造对象，修饰类型转换运算符时禁止隐式的转换。

#include <iostream>

using namespace std;

class A {

public:

A(int x) { cout << "A(int) called\n"; }

};

void foo(A a) {}

int main() {

foo(10); // ✅ 自动把 int 转换成 A

}

修饰类型转换运算符时

struct B {

explicit operator bool() const { return true; }

};

int main() {

B b;

// if (b) { } // ✅ OK，条件语句里允许用

bool flag = b; // ❌ 错误，因为 explicit 禁止隐式转换

bool flag2 = static\_cast<bool>(b); // ✅ 必须显式转换

}

**4.列举一些整型字面量的后缀，分别对应哪些数据类型**

U：unsigned int表示的范围为0~2^32-1

L：long，不建议使用，在Windos上面始终保持为4字节但是在linux和mac上64位为8字节，32字节。

UL：unsigned long int

LL：long long

ULL：unsigned long long

**5.inline函数和宏定义之间有什么区别**

宏定义的本质是在编译时文本级别的替换，十分底层没有任何其他限制。

inline函数本质是函数展开，是一个真正的函数，更加的上层，而且实际是否会发生展开是由编译器决定的。

太过大（循环非常多），递归函数和虚函数编译器不会展开。

**6.dynamic\_cast,static\_cast,reinterprert\_cast之间的区别**

dynamic\_cast是运行时检查，更加安全但是效率更低。

static\_cast是编译时检查，只在能提前确认安全的时候使用。

两个都可以用于向上/向下转型，由于向上转型是安全的所以我们通常使用static\_cast，向下转型是不安全的，所以我们通常使用dynamic\_cast。

注意dynamic\_cast必须要多态类型，所以其只能用来转换引用和指针而不能用来转换基本类型。在转换指针出错时返回空指针，而转换引用报错时直接报错。

reinterprert\_cast直接将内存地址当做目标类型来访问，完全不安全所以基本不用。

**7.move函数的作用，底层的原理**

move函数本身是获得一个左值的右值引用并将其返回。底层本质上是通过static\_cast将左值进行类型转换。所以在这个过程中没有将左值转换为右值的过程，只有转换为一个右值引用。

static<remove\_reference<T>::type&&>来将其传入的内容转换为原始类型的右值引用。

**8.左值和右值有什么区别**

左值有名字，可以取地址。右值没有名字，通常不可以取地址。同时两者在使用上的最大区别在于调用移动构造函数还是拷贝构造函数。

**9.完美转发的概念**

通过万能引用和forward将参数原封不动的进行传递

template<typename t>

void wraper(T&& x)

{

add(forward<T>(x));

}

forward还可以在这里使用

void wrapper(std::string&& s) {

// s 是一个有名字的变量，表达式里就是左值

process(s); // ❌ 调用的是左值重载，而不是右值重载

process(std::forward<std::string>(s)); // ✅ 恢复右值身份，调用右值重载

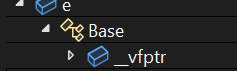
}

即将右值引用还原为原本的实际类型。

**9.多继承的情况下类的内存结构**

在常规的菱形继承的情况下，D中会有两份A，此时会出现二义性报错，所以我们使用虚继承来避免此问题，在使用虚继承的情况下所有被虚继承的基类会产生一个共享对象而非向原本的子对象

普通继承：



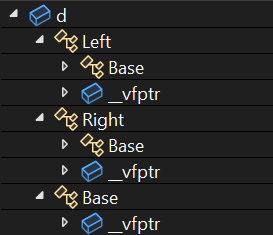
虚继承：



此时子类的对象的虚指针不再覆盖父类的虚指针，而是产生一个完整的基类子对象，自己再增加一个虚指针，也就是说虚继承的情况下会产生一个完整的基类子对象。

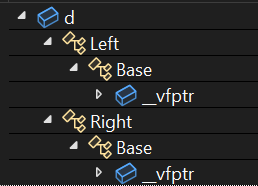
在菱形继承的情况下

虚继承：



此时的D对象中有三个虚指针，共享的完整的Base子对象中有一个虚指针，两个子父类分别有一个虚指针指向自身的虚函数表和一个指向共享Base的指针，注意此时父子类不再覆盖原本的父类的虚指针而是产生新的虚指针，同时Base类的子对象也会只产生一个共享的。

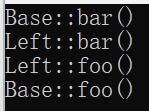
常规情况下：



两个父子类中分别有一个Base对象，共有两个Base子对象，同时父子类不单独产生虚指针，直接覆盖Base类的虚指针

**10.构造函数中和析构函数中可以调用虚函数吗**

可以但是会导致多态失效，谁的构造函函数/析构函数就调用谁的版本。



在子类的构造/析构函数中调用虚函数不会出现问题，此时的虚指针已经被覆盖了能够执行正确的函数，只是要注意数据成员的初始化顺序。

但是在基类的构造函数中调用虚函数，此时虚指针还没有被子类进行覆盖仍然指向基类的虚函数表，所以会调用基类的虚函数。

在基类的虚构函数中调用虚函数时，此时子类的析构函数已经被执行（注意，析构函数一定要被写为虚函数，不然只会执行基类的构造函数），子类的部分已经被回收，其虚函数无法被调用，同时在进入基类的析构函数的时候会将虚函数指针再指向基类的虚函数表，所以此时会调用基类的虚函数。

**11.基类的析构函数需要设置为虚函数吗？哪些情况下不需要声明为虚函数。**

需要，不然在多态的场合会进行普通函数的调用逻辑直接去调用实际的类型的函数。

在A\* a = &b;a.fun();会去A中找对应的函数，找不到再去B中找，注意，只要有一个满足就会触发调用，即使B和A的同名函数参数不同。而在B\* b = &a;情况下会去B中找。

注意，函数的重载不发生在基类和父类之间，基类和父类之间根据指针的类型调用函数，重载发生在同一个作用域中的同名函数之间。

**12.placement\_new和new的区别，placement\_new可以用在什么地方？**

placemen\_new是new运算符的一种重载的形式，其在提前分配的指定的空间构造对象，会调用构造函数。主要的作用在于在栈上面分配空间，注意在栈上面分配的空间不用手动的回收，只要在希望回收的时候调用构造函数即可a->~A();如果本身是在堆上面分配的空间那么可以使用delete来进行回收。

**13.SharedPtr的底层实现，unique\_ptr的底层实现**

使用一个引用计数来保存当前资源被多少个SharedPtr引用了，注意这个SharedPtr中的引用计数是一个int\* usecount = new int;的指针以支持在不同的对象中共用计数。

注意重载构造函数，拷贝构造函数，赋值运算符。

unique\_ptr注意删除拷贝赋值函数，拷贝构造函数，实现移动赋值函数和移动构造函数。

**14. shared\_ptr底层的引用计数器如何实现线程安全**

通过原子操作来实现，这也是无锁编程的实现方式，使用原子操作来让硬件进行加锁保证操作执行的不可中断。

无锁编程指的是软件层面的没有锁，通过CPU提供的原子操作和内存一致性原理来实现同步。软件层面的锁包括互斥锁和自旋锁，互斥锁是在资源被其他线程使用时将当前进程挂入阻塞队列，自旋锁是是不断查询当前资源是否被释放。

原子操作：保证一段内存读–改–写在 CPU 层面不可分割**。**

**15.ref,out,in的作用和区别**

ref和out都是使函数内的影响传递到函数外的变量，区别在于ref要求变量必须初始化后传入，out要求变量必须在传入函数后才能使用，主要区别还是语义层面。而in相当于是const，保证变量不会被修改。上面三个的底层原理都是将变量当做引用来传递变量的地址。

**16. const和readonly关键字的区别**

const在编译时确定值，所以必须初始化。readonly在运行时确定值可以不用初始化，可以在构造函数中赋值。

**17.保守GC和精确GC**

保守GC不知道哪些是指针哪些是具体的数据，只能通过数据的值来粗略的判断，所以没法实现压缩。精准GC明确知道哪些是指针哪些是数据所以可以实现压缩。

**18.HTTP和HTTPS**

HTTP是常规的超文本传输协议，HTTPS是带有安全考量的版本。其通过服务器证书验证和加密来实现安全传输。

非对称加密阶段：

服务器发送公钥，客户端随机生成私钥，通过公钥加密后发送给服务器，服务器解码得到私钥。

对称加密阶段：  
服务器使用私钥加密数据并发送给客户端。

客户端使用私钥解密数据。