一、 C++基础

1. override和overload有什么区别?

   overload（重载）：函数重载是指在同一作用域内定义多个具有相同名称但参数列表不同的函数。通过使用不同的参数类型、参数个数或参数顺序，可以重载同一个函数名来执行不同的操作。编译器根据调用时提供的参数来确定要调用的重载版本。

   override（重写）：函数重写时指在派生类中重新定义基类的虚函数，以提供特定的实现。派生类必须具有相同的函数名称、参数列表和返回类型，并使用override关键字进行标记，以表示这是对基类虚函数的重写。重写函数可以扩展或修改基类的行为，但是函数的签名必须与基类中的虚函数完全匹配。

2. 重载的返回值可以不同吗?

   函数重载的返回值可以不同，只要参数列表能够区分不同的重载版本即可。需要注意的是，仅仅改变函数的返回值类型而保持参数列表相同是无法进行函数重载的，因为函数调用时无法根据返回值类型来确定具体调用哪个函数版本。

3. 什么是多态？C++的多态是如何实现的?

   所谓多态，就是同一个函数具有多种不同的状态，或者说一个接口具有多种不同的行为。C++的多态分为编译时多态和运行时多态，编译时多态又称为静态联编，通过重载和模板实现，运行时多态又称为动态联编，通过继承和虚函数实现。

4. 虚函数的实现机制是什么?

   虚函数是通过虚函数表来实现的，虚函数表包含了一个类所有的虚函数地址，在有虚函数的类对象中，它内存空间的头部会有一个虚函数表指针，用来管理虚函数表。当子类对父类虚函数进行重写的时候，虚函数表的相应虚函数地址会发生改变，改写成这个虚函数的地址。当我们用一个父类的指针来操作子类对象的时候，它可以指明实际所调用的函数。

5. 虚函数调用是在编译时确定还是在运行时确定的？如何确定调用哪个函数?

   通过指针或者引用的方式调用虚函数是运行时确定，通过值调用的虚函数是编译期就可以确定的。

6. 虚函数是存在类中还是类对象中?

   存在类中，不同的类对象共享一张虚函数表。

7. 构造函数和析构函数能否是虚函数?

   构造函数不能是虚函数

   ①虚函数的调用需要虚函数表指针，而该指针存放在对象的内存空间中。问题是如果构造函数是虚的，就需要虚函数表的调用。但是对象还没有实例化，内存空间还没有，就没有虚函数表来调用虚函数了。

   ②虚函数的作用在于通过父类的指针或者引用来调用它的时候能够变成调用子类的函数，而构造函数是在创建对象的时候自动调用的，不可能通过父类的指针或者引用去调用，所以构造函数不能是虚函数。

   析构函数可以是虚函数

   ①因为有父类指针指向子类对象，需要析构的是子类对象，但是父类析构函数不是虚函数，那么只会调用父类的析构函数，不会调用子类的析构函数，这种情况下，子类申请的空间得不到释放会导致内存泄露的问题。

8. 在基类的构造函数和析构函数中调用虚函数会怎样?

从语法上讲，调用没有问题。但是从效果上看，往往不能达到需要的目的（不能实现多态）。因为调用构造函数的时候，是先进行父类成分的构造，再进行子类成分的构造。在父类构造期间，子类的特有成分还没有被初始化，此时下降到调用子类的虚函数，使用这些尚未初始化的数据一定会出错。同理，调用析构函数的时候，先对子类的成分进行析构，当进入父类的析构函数的时候，子类的特有成分已经销毁，此时是无法再调用虚函数实现多态的。

9. C++中类对象的内存布局是怎么样的?

   ①如果是有虚函数的话，虚函数表的指针始终存放在内存空间的头部。

   ②除了虚函数之外，内存空间会按照类的继承顺序和字段的生命顺序布局。

   ③如果有多继承，每个包含虚函数的父类会有自己的虚函数表，并且按照继承顺序布局（虚表指针+字段），如果子类重写父类虚函数，都会在每一个相应的虚函数表中更新相应地址。如果子类有自己新定义的虚函数或者非虚成员函数，也会加到第一个虚函数表的后面。

   ④如果有钻石继承，并采用虚继承，则内存空间排列顺序为：各个父类、子类、公共基类，并且各个父类不再拷贝公共基类的数据成员。

10. 钻石继承存在什么问题，如何解决?

    会存在二义性的问题，因为两个父类会对公共基类的数据和方法产生一份拷贝，因此对于子类来说读写一个公共基类数据或者调用一个方法时，不知道是哪一个父类的数据和方法，也会导致编译错误。可以采用虚继承的方法解决这个问题（父类继承公共基类时用virtual修饰），这样就只会创造一份公共基类的实例，不会造成二义性。

11. C++是如何做内存管理的?

    堆、栈、全局/静态存储区、常量存储区、自由存储区

12. 堆是用来干嘛的?

    ①动态内存分配：在程序运行时动态地分配内存和释放内存，而不需要提前知道所需内存的大小，用new。

    ②对象的动态创建和销毁：在堆上创建对象可以使他们的生命周期独立于作用域。

13. 栈使用来干嘛的?

    存储函数内部的局部变量。每当在函数内部定义一个局部变量时，变量的内存空间会在函数被调用时分配到栈上，并在函数调用结束时自动销毁。

14. 堆和栈的内存有什么区别?

    ①堆中的内存需要手动申请和手动释放，栈中内存是由操作系统自动申请和自动释放。

    ②堆能分配的内存较大，栈能分配的内存较小。

    ③在堆中分配和释放内存会产生内存碎片，栈不会产生内存碎片。

    ④堆的分配效率低，栈的分配效率高；

    ⑤堆地址从低向上，栈由高向下。

15. C++和C分别使用什么函数来做内存的分配和释放？有什么区别？能否混用?

    C++使用new和delete，C使用malloc和free

    malloc和free是C++/C的标准库函数，new/delete是C++的运算符，它们都可以用于申请动态内存和释放内存。

    但是对于非内部数据类型的对象而言，光用malloc和free无法满足动态对象的要求，对象在创建的同时要自动执行构造函数，在消亡之前自动执行析构函数。

    由于malloc/free是库函数而不是运算符，不在编译器的控制权限之内，不能够把执行构造函数和析构函数的人任务强加与malloc/free，因此C++语言需要一个能完成内存分配和初始化的运算符new和一个能完成销毁与释放内存的运算符delete。

    区别：

①new分配内存无需指定分配内存大小，malloc需要

16. 内存对齐应用

```
#include <iostream>
using namespace std;

#pragma pack(4)

class Node
{
    char a;
    int32_t b;
    char d;
    char* c;
    static int32_t e;
    virtual void test();
};

#pragma pack()

int main()
{
    cout << sizeof(Node) << endl;
    return 0;
}
```

大小：28

```cpp
#include <iostream>
using namespace std;

//#pragma pack(4)

class Node
{
    char a;
    int32_t b;
    char d;
    char* c;
    static int32_t e;
    virtual void test();
};

#pragma pack()

int main()
{
    cout << sizeof(Node) << endl;
    return 0;
}
```

大小：32

17. 为什么要进行内存对齐?

    关键在于CPU存储数据的效率问题。计算机从内存中取数据都是按照一个固定长度的。比如在32位机上，CPU每次都是取32bit数据的，也就是4字节。若不进行对齐，要取出两块地址中的数据，进行掩码和移位等操作，写入目标寄存器内存，效率很低。内存对齐一方面可以节省内存，一方面可以提升数据读取的速度。

18. C++有哪些类型转换的方法，各自有什么作用?

    static_cast:

    ①用于基本类型间的转换

    ②不能用于基本类型指针之间的转换

    ③用于有继承关系的类对象间的转换和类指针之间的转换

    dynamic_cast:

    ①用于有继承关系的类指针的转换

    ②用于有交叉关系的类指针的转换

    ③具有类型检查的功能

    ④需要虚函数的支持

    const_cast:
    ①用于去掉变量的const属性

②转换的目标必须是指针或者引用

reinterpret_const：

可以转换任何内置的数据类型转换为其他任何的数据类型，也可以转换任何指针类型为其他的类型，甚至可以转换任何内置的数据类型为指针，有着和C风格强制转换同样的能力

19. static_cast和dynamic_cast的异同点?

    二者都会做类型安全检查，static_cast在编译期进行类型检查，dynamic_cast在运行期间进行类型检查。后者需要父类具备虚函数，而前者不需要。

20. C++中的智能指针有什么作用?

    智能指针主要解决一个内存泄漏的问题，它可以自动地释放内存空间。因为它本身是一个类，当函数结束的时候会调用析构函数，并由析构函数释放内存空间。

21. 智能指针有什么?

    智能指针分为共享指针、独占指针和弱指针。

    共享指针：多个共享指针可以指向相同的对象，采用了引用计数的机制，当最后一个引用销毁的时候，释放内存空间。

    独占指针：保证同一个时间段内只有一个智能指针指向该对象。

    弱指针：用来解决共享指针相互引用时的死锁问题，如果说两个共享指针相互引用，那么这两个指针的引用计数就永远不可能下降为0，资源永远不会被释放。它是对对象的一种弱引用，不会增加对象的引用计数，和共享指针之间可以相互转换。共享指针可以直接赋值给它，它可以通过调用lock函数来获得共享指针。

22. 弱指针是为了解决共享指针的循环引用问题，那为什么不用原始指针(raw ptr)来解决这个问题?

    一个弱指针绑定到共享指针之后不会增加引用计数，一旦最后一个指向对象的共享指针被销毁，对象就会被释放，即使弱指针指向对象，也还是会释放。但是原始指针，在对象销毁之后会变成悬浮指针。

23. const的作用?

    ①定义只读变量，或者常量。

    ②修饰函数的参数和函数的返回值。

    ③修饰函数的定义体。这里的函数为类的成员函数，被const修饰的成员函数代表不能修改成员变量的值，因此const成员函数只能调用const成员函数，可以访问非const成员，但是不能修改。

24. static的作用？什么时候初始化?

    ①当用于文件作用域的时候，static以为着这些变量和函数只在本文件可见，其他文件是看不到也无法使用的，可以避免重定义的问题。

    ②当用于函数作用域的时候，即作为局部静态变量时，意味着这个变量是全局的，只会进行一次初始化。不会在每次调用的时候进行重置，但只在这个函数内可见。

    ③当用于类的声明时，即静态数据成员和静态成员函数，static表示这些数据和函数是所有类对象共享的一种属性，而非每个类对象独有。

    全局变量、文件域的静态变量和类的静态成员变量在main执行之前的静态初始化过程中分配内存并初始化。局部静态变量在第一次使用时分配内存并初始化。

25. 创建一个对象经历了哪些过程?

    ①为对象分配内存空间，创建一个空的实例化对象。

    ②让构造函数中的this指向空的实例化对象。

    ③调用构造函数，从而创建实例化对象自身的属性和方法。

26. extern的作用?

当它与“C”一起连用时，如 extern "C" void fun(int a, int b) 则告诉编译器在编译fun这个函数名时按照C的规则去翻译相应的函数名而不是C++。当他作为一个对函数或者全局变量的外部声明，提示编译器遇到此变量或函数时，在其他模块中寻找其定义。

27. explicit的作用?

    表明类的构造函数是显式的，不能进行隐式转换。

28. constexpr的嘴用?

    这个关键字告诉编译器应该去验证函数或变量在编译期是否就应该是一个常数。

29. volatile的作用?

    跟编译器优化有关，告诉编译器每次操作该变量时一定要从内存中真正取出，而不是使用已存在寄存器中的备份。

30. mutable的作用?

    可变的意思，使类中被声明为const的函数可以修改类中的非静态成员。

31. auto和deltype的作用和区别?

    用于实现类型自动推导，让编译器来操心变量的类型，auto不能用于函数传参和推导数组类型，但是deltype可以解决这个问题。

32. 什么是左值和右值?

    不是很严谨的说，左值就是既能出现在等号左边，也能出现在等号右边的变量。举例来说，我们定义的变量a就是一个左值，而malloc返回的值就是一个右值。左值就是具有可寻址的存储单元，并且能够由用户改变其值的量。它具有持久的状态，直到离开作用域才销毁。而右值表示即将销毁的临时对象，具有短暂的状态。

33. 右值引用是什么?

    右值引用用来绑定到右值，绑定到右值以后本来会被销毁的右值的生存期会延长到绑定到它的右值引用的生存期。右值引用只能绑定到即将销毁的对象上，是为了支持移动操作而引出的一个概念。右值引用的移动操作可以避免无谓的拷贝，提高性能。

34. 为什么要自己定义拷贝构造函数?

    拷贝构造函数的作用就是定义了当我们用同类型的另外一个对象来初始化本对象的时候做了什么。在某些情况下，如果我们不自己定义拷贝构造函数，使用默认的拷贝构造函数，就会出错。比如类里面有一个指针，如果使用默认的拷贝构造函数，会将指针拷贝过去，即两个指针指向同个对象，那么其中一个类对象析构之后，这个指针也会被delete掉，那么另一个类里面的指针就会变成野指针。

35. 什么是深拷贝和浅拷贝?

    浅拷贝知识简单的复制指向某个对象的指针，而不复制对象本身，新旧对象还是共享同一块内存。但深拷贝会另外创造一个一模一样的对象，新对象跟元对象不共享内存，修改新对象不会改到原对象。

36. 什么是移动构造函数，和拷贝构造函数的区别?

    移动构造函数需要传递的参数是一个右值引用，移动构造函数不分配新内存，而是接管传递过来的内存，并在移动之后把源对象销毁。移动拷贝构造函数需要传递一个左值引用，可能会造成重新分配内存，性能更低。

37. 内联函数有什么作用?

    使编译器在函数调用点上展开函数，避免函数调用的开销。

38. 内联函数存在什么缺点?

    ①可能造成代码膨胀，尤其是递归的函数，会造成大量内存开销，exe太大，占用CPU资源。

    ②不方便调试，每次修改会重新编译头文件，增加编译时间。

39. 内联函数和宏有什么区别?

    ①宏是在预处理阶段对命令进行替换，inline是在函数调用点处展开函数，节省了函数调用的开销。

    ②宏不会对参数的类型进行检查，会出现类型安全的问题。但是内联函数在编译阶段会进行类型检查。

40. 指针和引用的区别?

    ①指针的本质是一个地址，有自己的内存空间，引用只是一个别名。

    ②指针可以指向其他的对象，但是引用不能指向其他的对象，初始化之后就不能改变了。

    ③指针可以初始化为nullptr，而引用必须被初始化为一个已有对象的引用。

    ④指针可以是多级指针，引用只能是一级。

41. delete和delete[]的区别?

    如果是基本类型，delete和delete[]效果是一样的，因为系统会自动记录分配的空间，然后释放。对于自定义数据类型而言就不行了，delete仅仅释放数组第一个元素的内存空间，且仅调用了第一个对象的析构函数，但delete[]会调用数组所有元素的析构函数，并释放所有内存空间。

42. delete[]如何知道要delete多少次?

    我们在new[]一个对象数组的时候，需要保存数组的维度。C++的做法是在分配数组空间时多分配了4个字节的大小，专门保存数组的大小，这个数据应该就存在这个分配返回的指针周围，在delete[]时就可以取出这个保存的数，就知道了需要调用析构函数多少次了。

43. 在类的成员函数中能否delete this?

    可以，并且delete之后还可以调用该对象的其他成员，但是有个前提：被调用的方法不涉及这个对象的数据成员和虚函数。当一个类对象声明时，系统会为其分配存储空间。在类对象的内存空间中，只有数据成员和虚函数指针，并不包含代码内容，类的成员函数单独放在代码段中。

44. 基本类型占用的大小，不同平台上面是否有区别，有什么区别？为什么指针占的字节是8?

数据类型	**32位**	**64位**
int	4	4
unsigned int	4	4
short	2	2
long	4	8
float	4	4
double	8	8
指针	4	8
char	1	1

    指针存储的是地址，地址总线的宽度决定了CPU的寻址能力，如果计算机的地址总线是64位，那么需要64个0或1就可以找到内存中所有的地址，因此我们只需要8个字节就可以找到所有的数据。同理，在32位的计算机中，指针占4个字节。

45. 指针常量和常量指针?

    指针常量指的是指针本身是常量，但是指向的地址不是。指针本身不能被修改，但是指向的地址可以。

常量指针指的是指针指向的地址是常量，但是指针本身不是。指针本身可以被修改，但是指向的地址不可以。

46. auto a = new Clz();
    auto b = new Clz();
    a = b;
    delete a;
    delete b;这段代码有什么问题?

    这段代码存在内存泄漏的问题。首先，用new关键字创建了两个Clz类的对象，分别赋值给了指针变量a和b，然后，将指针变量a的值设置为指针变量b的值，这会导致原本指向第一个clz对象的地址丢失，无法再释放这块内存。

    而接下来的delete操作，由于之前的操作已经导致第一个clz对象的内存泄漏，因此只是放了第二个clz对象的内存，第一个对象的内存没有释放，造成了内存泄露。

二、C#基础

1. 值类型和引用类型?

   值类型：struct、enum、int、float、char、bool、decimal

   引用类型：class、delegate、interface、array、object、string

2. 装箱与拆箱?

   装箱：把值类型转换成引用类型

   拆箱：把引用类型转换成值类型

   装箱：在堆中分配一个对象实例，并将该值复制到新的对象中。

   ①新分配托管堆内存（大小为值类型实例大小加上一个方法表指针）

   ②将值类型的实例字段拷贝到新分配的内存中

   ③返回托管堆中新分配对象的地址，这个地址就是一个指向对象的引用了

   拆箱：检查对象实例，确保它是一个给定值类型的一个装箱值。将该值从实例复制到值类型变量中。装箱是不需要显式的类型转换的，不过拆箱需要显式的类型转换。

3. 垃圾回收机制（GC）

   垃圾回收机制是一种内存管理技术，它允许开发人员不必手动分配和释放内存，而是让运行时环境负责处理内存的分配和释放。

   在C#中，垃圾回收器会定期扫描运行时的对象，并标记那些仍然被引用的对象。然后，它会清理掉未被引用的对象，并回收这些内存空间，使其能够被重用。

   GC只能处理托管内存资源的释放，对于非托管资源则不能使用GC进行回收，必须由程序员手动回收，例如FileStream或SqlConnection需要调用Dispose进行资源的回收。

   垃圾回收的好处是减少了内存管理的复杂性，避免了常见的内存错误，如内存泄露和悬挂指针，它提供了更高的开发效率和更可靠的程序性能。然而，垃圾回收也可能对程序的执行性能产生一定的影响，因为垃圾回收器在清理内存时需要占用一定的计算资源。

4. 反射机制是什么?

   C#的反射机制是指在运行时动态地分析、获取并操作程序的类型信息和成员信息的能力。通过反射，我们可以在运行时获取类型的元数据，包括类、接口、字段、属性、方法等的信息，并且可以使用这些信息来创建对象、调用方法和访问成员。

   反射的主要功能包括：

   ①获取类型信息：通过type类可以获取类型的元数据，如名称、基类、接口、属性、字段、方法等。

   ②创建对象：使用反射可以动态地创建对象实例，即使在编译时不能确定类型。

③调用方法和访问成员：使用methodinfo、fieldinfo、propertyinfo等反射类可以动态地调用方法和访问对象的属性、字段等成员。

④扩展性和灵活性：通过反射可以在程序运行时动态地读取和修改程序的结构和行为，实现一些高级功能。如插件系统、反射注入等。

三、数据结构

1.

	平均	最好	最坏	空间	稳定性
快速排序	O(nlogn)	O(nlogn)	$O(n^2)$	O(logn)	不稳定
堆排序	O(nlogn)	O(nlogn)	O(nlogn)	O(1)	不稳定
冒泡排序	$O(n^2)$	O(n)	$O(n^2)$	O(1)	稳定
插入排序	$O(n^2)$	O(n)	$O(n^2)$	O(1)	稳定
选择排序	$O(n^2)$	$O(n^2)$	$O(n^2)$	O(1)	不稳定
归并排序	O(nlogn)	O(nlogn)	O(nlogn)	O(n)	稳定
希尔排序	O(nlogn)~$O(n^2)$	O(n1.3)	$O(n^2)$	O(1)	不稳定

2. STL各种容器的底层实现?

①vector，底层是一块具有连续内存的数组，vector的核心在于其长度自动可变。vector的数据结构主要由三个迭代器来完成：指向首元素的start、指向尾元素的finish、指向内存末端的end_of_storage。vector的扩容机制是：当目前可用空间不足时，分配目前空间的两倍或者目前空间加上所需新空间大小（取较大值），容量的扩张必须经过“重新配置、元素移动、释放原空间”等过程。

②list，底层是一个循环双向链表，链表节点和链表分开独立定义的，节点包含pre、next指针和data数据。

③deque，双向队列，由分段连续空间构成，每段连续空间是一个缓冲区，由一个中控器来控制。它必须维护一个map指针，还要维护start和finish两个迭代器，指向第一个缓冲区和最后一个缓冲区。deque可以在前端或者后端进行扩容，这些指针和迭代器用来控制分段缓冲区之间的跳转。

④stack和queue，栈和队列。它们都是由deque作为底层容器实现的。他们是一种容器配接器，修改了deque的接口，具有自己独特的性质。stack是deque封住了头端的开口，先进后出。queue是deque封住了尾端的开口，先进先出。

⑤priority_queue，优先队列。是以vector作为底层容器，以heap作为处理规则，heap的本质就是一个完全二叉树。

⑥set和map。底层都是由红黑树实现的。红黑树是一种二叉搜索树，但在每个节点增加一个存储位来表示节点的颜色，可以是红或者黑，非红即黑。通过对任意一条从根到叶子节点上着色方式的限制，红黑树确保了没有一条路径会比其他路径长出两倍。因此，红黑树是一种弱平衡二叉树，相对于要求严格的平衡二叉树来说，它的旋转次数少，所以对于插入、删除操作较多的情况下，通常使用红黑树。

3. STL各种容器查找、删除和插入的时间复杂度?

①vector，vector支持随机访问，时间复杂度是O(1)，如果是无序vector查找的时间复杂度是O(n)，如果是有序vector，采用二分查找则是O(logn)。对于插入操作，在尾部插入最快，在中部次之，头部最慢，删除同理。vector占用的内存较大，由于二倍扩容机制可能会导致内存的浪费，内存不足时的扩容的拷贝也会造成较大性能开销。

②list是底层链表，不支持随机访问，只能通过扫描的方式查找，复杂度为O(n)。

③deque支持随机访问，但性能比vector要低。支持双端扩容，因此在头部和尾部插入和删除元素很快，为O(1)，但是在中间插入和删除元素很慢。

④set和map，底层基于红黑树实现，增删改查的时间复杂度近似O(logn)，红黑树又是基于链表实现，因此占用内存较小。

⑤unordered_set和unordered_map，底层是基于哈希表实现的，是无序的。理论上增删改查的时间复杂度是O(1)。

4. STL怎么做内存管理的?

   STL采用了两级配置器，当分配空间大小超过128B时，会使用第一级空间配置器，直接使用malloc、realloc、free函数进行内存空间的分配和释放。当分配空间小于128B时，将使用第二季空间配置器，采用了内存池技术，通过空闲链表来管理内存。

5. 次级分配器的原理?

   每次配置一大块内存，并维护对应的自由链表。若下次再有相同大小的内存配置，就直接从自由链表中拔出。如果客户端释还小额区块，就由配置器回收到自由链表中。

6. 内存池的优势和劣势?

   优势：避免内存碎片，不需要频繁从用户态切换到内核态，性能高效。

   劣势：仍然会造成一定的内存浪费，比如申请120B就必须分配128B。

7. STL容器的push_back和emplace_back的区别?

   emplace_back使用传递来的参数直接在容器管理的内存空间中构造元素（只调用了构造函数），push_back会创建一个局部临时对象，并将其压入容器中（可能调用拷贝构造函数或移动构造函数）。

8. 各种排序算法的原理和时间复杂度?

   ①快排：一轮划分，选择一个基准值。小于该基准值的元素放到左边，大于的放在右边，此时该基准值在整个序列中的位置就确定了，接着递归地对左边子序列和右边子序列进行划分。时间复杂度O(nlogn)，最坏地时间复杂度是$O(n^2)$。

   ②堆排序：构造一个最大堆或者最小堆，以最大堆为例，那么最大的值就是根节点，把这个最大值和最后一个节点交换，然后再从前n-1个节点中构造一个最大堆，再重复上述的操作，即每次将现有序列的最大值放在现有数组的最后一位，最后就会形成一个有序数组。求升序用最大堆，降序用最小堆，时间复杂度O(nlogn)。

   ③冒泡排序：从前往后两两比较，逆序则交换，不断重复直到有序。时间复杂度$O(n^2)$，最好情况O(n)。

   ④插入排序：类似打牌，从第二个元素开始，把每个元素插入前面有序的序列中，时间复杂度$O(n^2)$，最好情况O(n)。

   ⑤选择排序：每次选择待排序序列中的最小值和未排序序列中的首元素交换，时间复杂度$O(n^2)$。

   ⑥归并排序：将整个序列划分成最小的大于等于2的等长序列，排序后再合并，再排序再合并，最后合成一个完整序列。时间复杂度O(nlogn)。

   ⑦希尔排序：插入排序的改进版。取一个步长，划分为多个子序列进行排序，再合并，时间复杂度O(n1.3)，最坏情况$O(n^2)$。

   ⑧桶排序：将数组分到有限数量的桶里，每个桶再个别排序，最后依次把各个桶中的记录列出来得到有序序列。桶排序的平均时间复杂度为线性的O(N+C)，其中C=N*(logN-logM)，M为桶的数量，最好情况下为O(N)。

9. 什么是哈希表?

   哈希表是一种根据关键码值直接访问数据的数据结构，它通过把关键码值映射到表中的一个位置来访问元素，以加快查找的速度，这个映射函数叫做哈希函数。

10. 哈希表的长度为什么要是质数?

    降低冲突发生的概率，使哈希后的数据更加均匀，如果使用合数，可能导致很多数据集中分布到一个点上，造成冲突。

11. 如何处理冲突？怎么删除一个元素?

    开放定址法、拉链法，开放定址法包括线性探测、平方探测法。

    线性探测法并不会真正的删除一个元素，而是做一个标记，否则会导致正常的查找出错。

12. map和unordered_map的区别?

    ①数据存储方式：map使用红黑树，将元素按照键的排序顺序进行存储，有序；unordered_map使用哈希表，根据键的哈希值进行存储，无序。

    ②查找效率：在查找元素时，map查找的平均时间复杂度时O(logn)，而unordered_map查找的平均时间复杂度为O(1)，使得unordered_map在大多数情况下比map更快速。

    ③内存占用：map用红黑树，每个节点需要存储键、值以及指向左右节点的指针，因此内存占用相对较高。而unordered_map使用哈希表，每个元素只需要存储键、值和一些额外的信息，所以内存占用相对较低。

13. 数据结构有什么?

    数组、链表、栈、队列、树、图、哈希表、堆

14. 快速排序什么时候会有最坏情况？怎么避免?

    最坏情况发生在待排序序列已经是有序的情况下，而选择的基准元素恰好是当前子数组的最大或最小元素。

    避免的做法：

    ①随机选择基准元素，而不是固定的选择第一个或最后一个。

    ②使用三数取中：从待排序序列的首、中、尾元素选择一个中位数作为基准元素。

    ③对小规模子数组使用其他排序算法，如插入排序。

四、计算机网络

1. TCP和UDP的区别?

    ①TCP是传输控制协议，UDP是用户数据报协议。

    ②TCP是面向连接的、可靠的数据传输协议，它要通过三次握手来建立连接。UDP是无连接的、不可靠，采取尽力而为的策略，不保证接收方一定能接受到正确的数据。

    ③TCP面向的是字节流，UDP面向的是数据报。

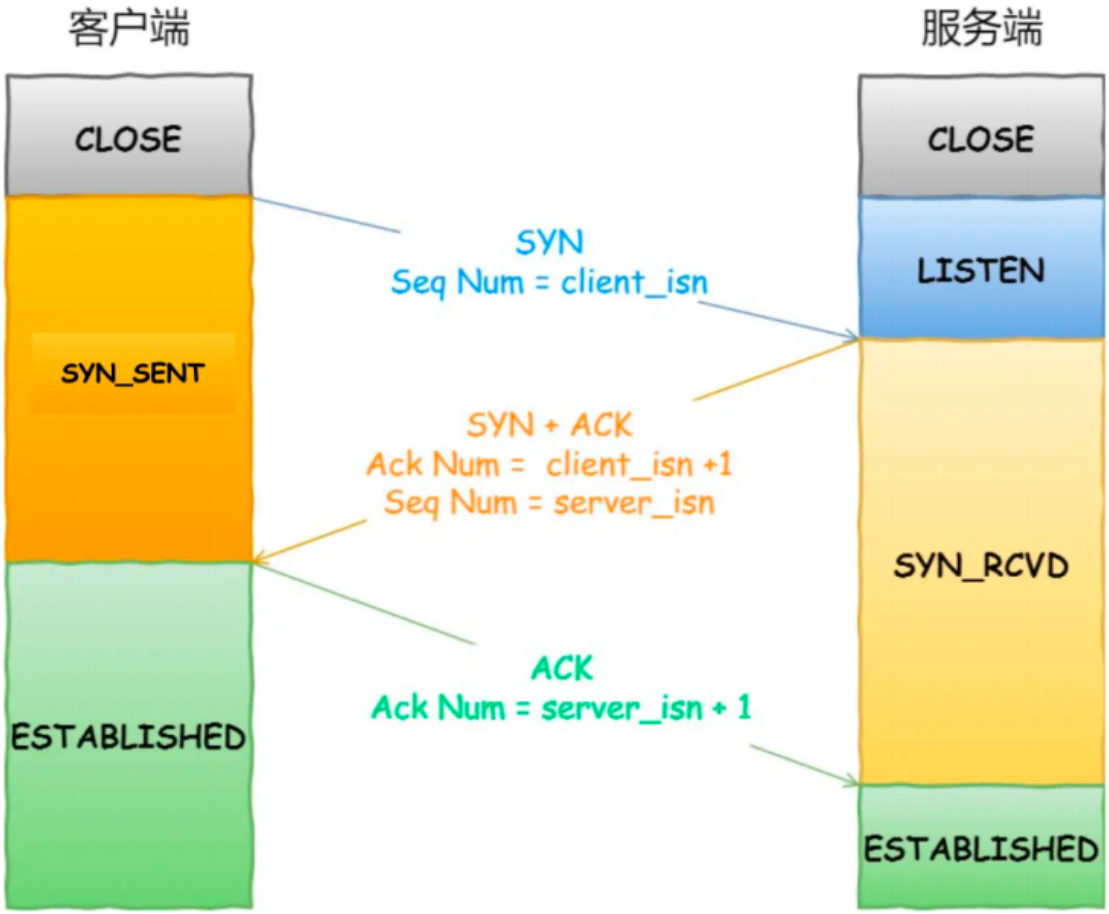
    ④TCP是点对点全双工通信，UDP支持一对一、一对多和多对多。

    ⑤TCP有拥塞控制机制，UDP没有。

    ⑥TCP首部开销20字节，UDP首部开销小，只有8字节。

2. 说说TCP的三次握手和四次挥手

    三次握手：



①一开始客户端和服务端都处于CLOSE状态，服务端主动监听客户端的某个端口，处于LISTEN状态。

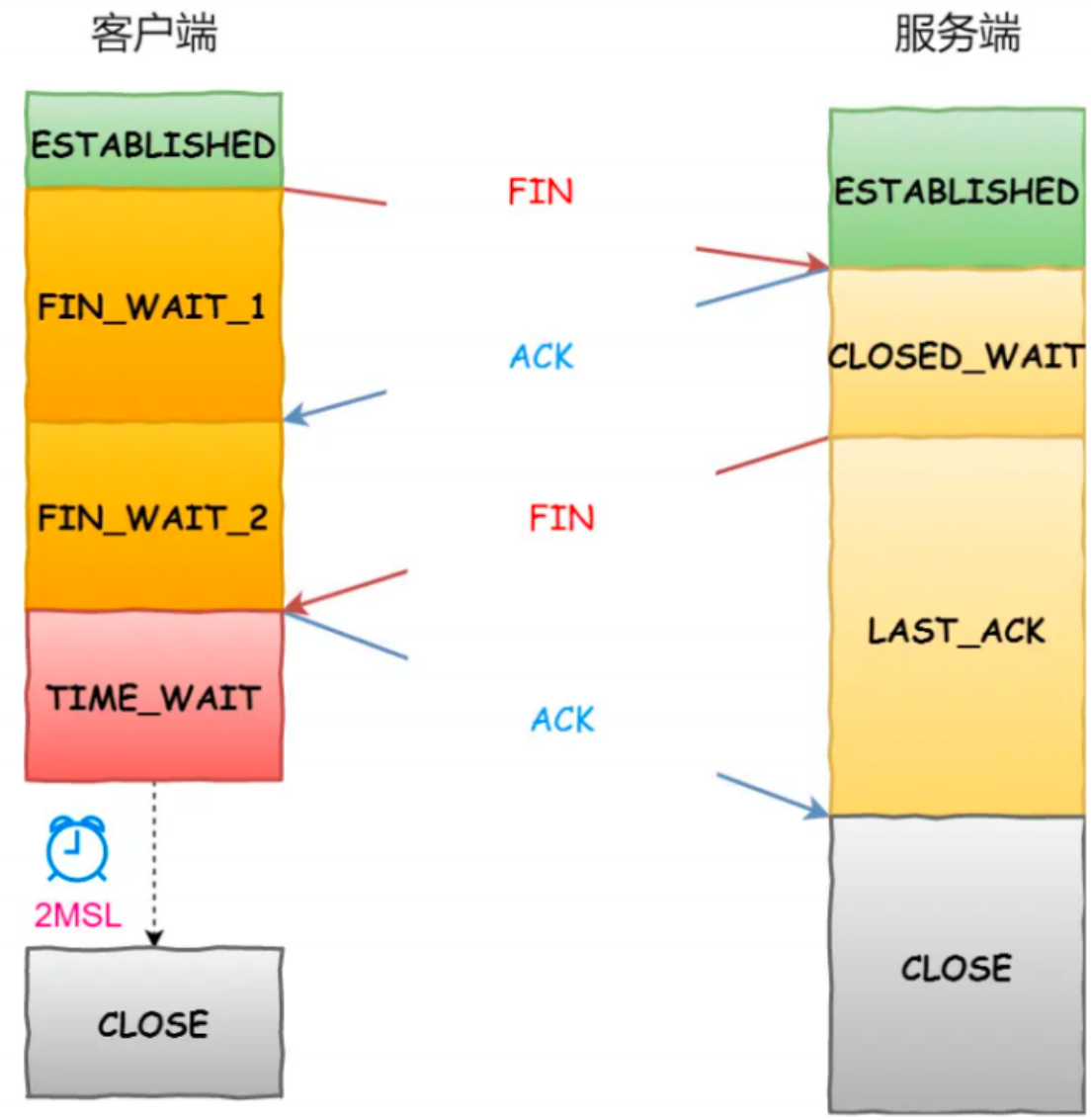
②客户端随机初始化序号(client_isn)，将该序号置于TCP首部的序号字段中，同时把SYN的标志位置为1，表示SYN报文，接着把第一个SYN报文发送给服务端，表示向服务端发起连接，该报文不包含应用层数据，之后客户端处于SYN-SENT状态。

③服务端接收到客户端的SYN报文后，首先服务端也随机初始化自己的序号(server-isn)，将此序号填入TCP首部的序号字段中，其次把TCP首部的确认应答号字段填入client_isn+1，接着把SYN和ACK标志位置为1。最后把该报文发给客户端，该报文不包含应用层数据，之后服务端处于SYN-RCVD状态。

④客户端收到服务端报文后，还要向服务端回应最后一个应答报文，首先该应答报文TCP首部ACK标志位置为1，其次确认应答号字段填入server_isn+1，最后把报文发送给服务端，这次报文可以携带客户到服务端的数据，之后客户端处于ESTABLISHED状态。

⑤服务端收到客户端的应答报文后，进入ESTABLISHED状态。

四次挥手：



①客户端打算关闭连接，此时会发送一个TCP首部FIN标志位置为1的报文，也即FIN报文，之后客户端进入FIN_WAIT_1状态。

②服务端收到报文后，就向客户端发送ACK应答报文，接着服务器进入CLOSE_WAIT状态，

③客户端收到服务端的ACK应答报文后，之后进入FIN_WAIT_2状态。

④等待服务端处理完后，也向客户端发送FIN报文，之后服务端进入LAST_ACK状态。

⑤客户端收到服务端的FIN报文后，回一个ACK应答报文，之后进入TIME_WAIT状态。

⑥服务端收到了ACK应答报文后，就进入了CLOSE状态，至此服务端已经完成连接的关闭。

⑦客户端在经过2MSL一段时间后，自动进入CLOSE状态，至此客户端也完成连接的关闭。

3. HTTP和HTTPS的区别?

①安全性：HTTP是明文传输，所有传输的数据都是未加密的，容易被窃听和篡改。而HTTPS通过使用SSL/TLS协议对传输的数据进行加密，提供了更高的安全性，保护用户的隐私和数据安全。

②默认端口：HTTP默认使用端口号80进行通信，而HTTPS默认使用端口号443进行通信。

③证书：HTTPS需要使用数字证书来验证服务器的身份。用户在访问HTTPS网站时，浏览器会检查证书的有效性，以确保连接的安全性。而HTTP不需要使用证书进行验证，存在被伪装的风险。

④性能：由于加密和解密过程的开销，HTTPS比HTTP更消耗计算资源，因此在性能上可能略逊于HTTP。

五、操作系统

1. 进程、线程、协程是什么?

进程：进程是运行时的程序，是系统进行资源分配和调度的基本单位。每个进程都有独立的内存空间，包括代码、数据、运行时栈等。进程之间是相互独立的，他们不共享内存，只能通过进程间的通信方式进行交互。进程是可以串行执行的，也是可以并行执行的。

线程：线程是进程的子单位，也成为轻量级进程。它是CPU进行分配和调度的基本单位，也是独立运行的基本单位，它实现了进程内部的并发。它们共享进程的内存和其他资源。

协程：协程是一种轻量级的线程，也称为用户态线程。与线程不同，协程的调度由程序自身控制，而不是操作系统内核控制。协程通过在函数内部保存和恢复状态来完成对多任务的切换，从而实现并发执行。它可以用来处理大量的并发任务，提高系统的并发能力。

2. 死锁是什么?

   死锁就是多个进程并发执行，在各自占有一定资源的情况下，希望获得其他进程占有的资源以推进执行，但是其他资源同样也期待获得另外进程的资源，大家都不愿意释放自己的资源，从而导致了相互阻塞、循环等待、进程无法推进的情况。

3. 死锁的条件?

   ①互斥条件（一个资源每次只能被一个进程使用）

   ②请求并保持条件（因请求资源而阻塞时，对已获得的资源保持不放）

   ③不剥夺条件（在未使用完之前，不能剥夺，只能自己释放）

   ④循环等待条件（若干进程之间形成一种头尾相接的循环等待资源关系）

4. 死锁如何防止?

   ①死锁预防，打破四个死锁条件。

   ②死锁避免，使用算法来进行资源分配，防止系统进入不安全状态，如银行家算法。

   ③死锁检测和解除：抢占资源或终止进程。

5. 什么是银行家算法?

   银行家算法在避免死锁方法中允许进程动态地申请资源，但系统在进行资源分配之前，应先计算此次分配资源的安全性，若分配不会导致系统进入不安全状态，则分配，否则等待。为实现银行家算法，系统必须设置若干数据结构。安全的状态指的是一个进程序列{P1，P2...Pn}，对于每一个进程Pi，它以后尚需要的资源不大于当前资源剩余量和其余进程所占有的资源量之和。

6. 操作系统如何管理内存?

   通过一种分页管理机制来进行内存管理。分页管理机制将程序的逻辑地址划分为固定大小的页，物理地址划分为同样大小的帧。程序加载时，可以将任意一页放入内存中任意一个帧，这些帧不必连续，从而实现了离散分离。

7. 什么是虚拟内存?

   虚拟内存是基于分页存储管理机制的，它允许程序不必将所有的页都放入内存中，而是将一部分页映射到内存中，另一部分页放在外存上（如磁盘、软盘、USB）。当引用到不在内存中的页时，系统将产生缺页中断，并从外存中调入该部分页进来，从而产生一种逻辑上内存得到扩充的感觉，实际上内存并没有增大。

8. 什么是内存碎片？内存碎片存在在物理内存还是虚拟内存?

   指在系统中存在的，被分割成多个小块的空闲内存空间。这些小块之间夹杂着已被使用的内存块，使得整个空间内存不连续，导致系统无法找到足够大且连续的内存空间来满足某些大块内存的分配请求。

   外部碎片：由于已分配的内存块和已释放的空闲内存块之间的剩余碎片堆积，导致系统无法找到足够大的连续内存区域。这会限制大内存块的分配能力，即使总的可用内存空间足够，但无法满足大块内存的需求。

   内部碎片：已分配的内存块超过了实际需要，导致未使用的内存空间浪费。例如，当一个进程请求分配10KB的内存时，系统分配了16KB的内存，然后有剩余6KB未被利用，这就属于内部碎片。

内存碎片存在在虚拟内存。

六、工程问题

1. C++源文件到可执行文件的过程?

   包括四个阶段：预处理、编译、汇编、链接

   ①预处理阶段处理头文件包含关系，对预编译命令进行替换，生成预编译文件。

   ②编译阶段将预编译文件编译，编译过程就是进行词法分析、语法分析、语义分析，生成汇编文件。

   ③汇编阶段将汇编文件转换成机器码，生成可重定位目标文件。

   ④链接阶段，将多个目标文件和所需要的库连接成可执行文件。

2. 设计模式?

   ①单例模式

   确保全局只有唯一一个自行创建的实例对象，并提供一个全局访问点来访问该实例。

   主要有两种实现方法：

   1) 懒汉式：用到的时候才会加载，线程不安全，需要加锁。

   2) 饿汉式：在main函数开始的时候即创建对象，线程安全。

   C++11标准之后的最佳选择式懒汉式，利用了局部静态变量在第一次使用时才初始化的特性。并且C++11解决了局部静态变量的线程安全问题。

   应用：

   如GameEntry、ObjectManager等。

   优点：

   1) 全局访问点：提供了一个全局访问点，可以方便地访问单例对象的实例。

   2) 节省资源：节省重复创建对象所消耗的资源。

   3) 线程安全：在多线程环境下，单例模式能够提供线程安全的访问保证。

   缺点：

   1) 全局状态：单例模式可以被任何地方访问，增加了代码的复杂性和耦合度。

   2) 难以扩展：单例模式一般只允许存在一个实例，对于后续的需求变更或者拓展，可能需要修改原有单例模式代码，比较繁琐。

   ②工厂模式

   定义一个工厂接口，具体对象创建由实现这个接口的工厂类来负责。客户端只需要通过调用工厂接口的方法来获取所需的对象，而无需关心对象的具体创建细节。

   分为简单工厂、普通工厂、抽象工厂。

   简单工厂：一个工厂生产多种产品，要指定产品的名字进行生产。

   普通工厂：将产品生产分配给多个工厂，但是每个工厂只生产一种产品。

   抽象工厂：将产品生产分配给多个工厂，每个工厂可以生产多种产品。

   应用：

   UI组件的创建是通过UI组件工厂进行管理和分发的。

   优点：

   1) 创建过程：工厂模式将对象的创建过程封装在工厂类中，客户端只需要通过工厂类来获取所需的对象，而无需关心具体的创建细节，从而降低了代码的耦合度。

2) 代码重用：可以将创建对象的逻辑集中到工厂类中，多处需要创建该对象的地方可以复用同一份代码，提高了代码的可维护性和重用性。

3) 灵活性和扩展性：工厂模式可以通过增加新的具体工厂类来扩展系统，对于新增的产品只需要添加相应的产品类和具体工厂，而无需修改已有的代码。

缺点：

1) 增加了系统的复杂度：引用工厂类会增加系统的类数量，增加了系统的复杂度。

2) 难以理解和调试

3) 增加了系统的抽象性

③观察者模式

一种行为型设计模式，允许对象之间建立一对多的依赖关系，当一个对象的状态发生改变时，与其依赖的对象会收到通知并自动更新。

应用：

事件系统

优点：

1) 松耦合性：实现了主题和观察者之间的松耦合，使它们可以独立变化。主题对象不需要知道观察者的具体细节，只需要通知它们即可，提高了系统的灵活性和可维护性。

2) 可扩展性：通过添加新的观察者对象，可以方便的扩展系统，并且不会影响到其他已有的观察者对象。这使得系统在需要新增功能时，能够更加容易地进行修改和拓展。

3) 通知机制：被观察者对象发生改变时，会自动通知相关观察者对象，从而保持了对象状态的一致性。

4) 规范化：明确了对象之间一对多关系，符合单一原则和开闭原则，使系统更加规范化和可管理。

缺点：

1) 不确定性：观察者对象对于主题对象的状态和行为是一无所知的，这可能导致观察者对象在处理变化通知时产生不必要的开销或误解。

2) 循环依赖：如果观察者和主题对象之间存在循环依赖关系，可能导致系统的死循环或递归调用。

④对象池模式

在游戏初始化时预先创建一些游戏对象，并将其保存在一个池子中，当需要使用游戏对象时，可以直接从池中获取，而不是每次都重新创建。当不再需要使用游戏对象的时候，也不会立刻销毁它，而是放回对象池中，以便后续重复利用。

优点：

频繁的创建和销毁游戏对象会导致大量的CPU和内存消耗，特别是移动设备上，这种开销更为明显。使用对象池可以优化游戏的性能，通过重复利用已经创建过的对象，避免频繁调用instantiate和destroy函数，从而减少内存分配和垃圾回收的开销。

缺点：

1) 线程安全性：在多线程环境下，对象池需要处理并发访问问题，以确保对象的正确获取和释放，这可能导致额外的同步开销和性能损失。

⑤命令模式

一种行为型设计模式，它将请求封装成一个对象，从而使得可以将请求的发送者和接收者解耦。

应用：人物移动

优点：

1) 解耦： 将请求发送者和接收者解耦，使得请求的发送者不需要知道请求执行的具体细节。

2) 可扩展性：可以方便地添加新的命令和接收者，扩展系统的功能。

缺点：

1) 增加类的数量：引入了额外的命令类，可能会增加系统中类的数量。

2) 命令调用的延迟：命令模式要求将请求封装成一个对象，因此可能会导致命令执行的延迟。

⑥状态模式

一种行为型设计模式，它允许对象在内部状态发生变化时改变其行为。状态模式将对象的行为封装在不同的状态类中，使得一个对象在不同状态下具有不同的行为。

应用：有限状态机

优点：

1) 封装性强：将不同状态的行为封装在不同的状态类中，使得每个状态类的代码相对独立，易于理解和维护。

2) 扩展性好：添加新的状态类相对容易，可以通过增加新的具体装让i类来满足系统需求的扩展。

3) 避免了大量的条件语句：提高代码的可读性和可维护性。

缺点：

1) 增加了类的数量。

2) 状态转换的逻辑复杂性：如果状态转换的逻辑较复杂，可能会导致状态类之间的耦合增加，一级状态转换的管理难度增加。

3. MVC架构的理解？

   将应用程序分成三个主要部分：模型、视图、控制器。

   模型：模型表示应用程序中的数据和业务逻辑。它负责处理数据的存储、检索和处理，并定义与数据相关的操作。模型通常不直接与用户界面交互，而是通过控制器进行通信。

   视图：视图是用户界面的表示，负责显示数据给用户，以及接受用户的输入动作。它通常包含UI元素、布局和样式等。视图应该尽量只负责展示数据，而不涉及业务逻辑。

   控制器：控制器是模型和视图的桥梁，负责协调数据和用户界面的交互。它接受来自视图的用户输入，并根据输入更新模型，并更新视图来反映模型的变化。控制器也可以包含一些逻辑判断和处理。

七、项目问题

1. 介绍一下你做的最喜欢的项目？

   我做过最喜欢的项目是在实习期间做的.....，它是我目前为止接触过的比较大型的并且上线了的游戏，也学习到了很多东西。

   首先，这个游戏使用了gameframework游戏框架，对我来说是比较新的东西，这个框架对游戏的常用模块进行了封装，规范了开发过程、加快了开发速度。它的功能包括资源管理、对象池、数据表、事件、声音、界面、日志工具等等。

   其次，这个项目运用到了很多设计模式，比如单例模式、工厂模式、观察者模式、状态模式、对象池模式、命令模式等。

   然后，这个游戏的UI部分用的UGUI开发，剧情用的TimeLine进行实现，动画效果通过dotween呈现。

   最后，这个游戏用到了寻路算法，实现了人物在导航网格中进行寻路。

2. 了解过什么寻路算法？

   A星算法、迪杰斯特拉算法、广度优先、深度优先

3. A星算法的原理？

A星算法是一种经典的寻路算法，基于启发式搜索的思想，它综合考虑了从起始点到目标点的代价和估计的剩余代价，并选择最优的路径。A星算法的基本原理是通过计算每个节点的F值（F = G + H）来确定最佳路径。其中，G值表示当前节点到起始节点的代价，H值表示当前节点到目标节点的预估代价。通过对节点进行开放列表和关闭列表的追踪，不断更新选择最优路径，直到找到目标节点或搜索完成。

4. 迪杰斯特拉的原理?

   是一种在加权图中找到最短路径的算法，该算法通过逐步扩展节点集合和更新节点的距离值来确定最短路径。

   具体原理：

   ①创建一个距离列表，其中存储了每个节点到起始节点的距离值。初始时，起始节点的距离值为0，其他节点的距离值设置为无穷大。

   ②创建一个未访问列表，用于跟踪待处理的节点。

   ③从未访问列表中选择距离值最小的节点，将其标记为当前节点。

   ④如果计算得到的距离值小于邻居节点当前存储的距离值，则将新的距离值更新到邻居节点上。

   ⑤将当前节点标记为已访问，并从未访问列表中删除。

   ⑥当所有节点被访问完后，计算完成。可以反向追溯从目标节点到起始节点的父节点来计算最短路径。

5. NavMesh的原理?

   navmesh时一种用于游戏和虚拟环境中的路径规划和导航技术，它主要用于在三维环境中为角色或物体生成可行走路径。

   navmesh的实现原理如下：

   ①场景建模：将游戏场景转化为一系列多边形或三角形区域来表示地面和障碍物。

   ②导航网格生成：根据场景建模结果，通过算法将多边形或三角形链接起来，形成一个连通的导航网格。

   ③导航网格修剪：对导航网格进行修剪，去除不必要的小区域，优化网格结构，以提高路径搜索的效率。

   ④导航数据计算：对修剪后的导航网格进行计算，计算出每个区域的邻居关系、障碍物信息等。

   ⑤路径规划：当需要寻找路径时，根据角色当前位置和目标位置在导航网格上进行搜索，找到一条最佳的路径。

   ⑥路径跟踪：根据路径规划的结果，进行路径跟踪，使角色沿着路径移动。

6. 帧同步和状态同步的理解?

   帧同步：在游戏中，服务器每隔一定时间将当前帧的所有状态信息发送给客户端，客户端接收到后进行更新并渲染显示。在帧同步中，服务器负责所有的游戏逻辑计算和决策。而客户端仅负责接受服务器发送的状态信息，并根据这些信息进行精确的模拟和表现。帧同步可以保证客户端之间具有相同的游戏状态，从而保证游戏的公平性。

   状态同步：在游戏中，服务器周期地将游戏状态发送给客户端，包括玩家位置、角色属性、动作等，客户端通过接收服务器的状态信息来更新游戏状态。在状态同步中，服务器和客户端共同承担了游戏逻辑计算和决策的责任。状态同步能够减少网络数据的传输量，但可能会导致客户端之间存在一定的差异，因为客户端的运行环境和处理能力可能不同。

7. 分别在什么情景下适用?

   帧同步：

   实时竞技游戏：射击游戏、格斗游戏

   多人协作游戏

状态同步：

大型多人在线游戏mmorpg，状态同步可以减少对带宽和服务器资源的压力。

离线模式游戏

8. dotween的原理？

   通过编程实现动画效果，利用插值计算使属性值平滑过渡，并通过时间管理和时间回调来控制动画的播放和交互。

9. 异步加载是什么？

   异步加载是一种编程模式，用于在程序运行过程中加载和处理大量数据、资源或执行耗时操作，而不会阻塞主线程导致程序卡顿。它通过并发或并行的操作，使得程序能够继续执行其他任务，同时在后台加载或处理需要的数据。

10. 为什么会先执行lua代码再执行C#代码？

    unity游戏开发中，目前代码热更应用最多的是unity+lua的方式，主要因为C#是编译型语言，C#会被编译成IL（中间语言），IL（中间语言）转换为机器码的过程可以在运行前执行也可以在运行时执行，但是IOS不允许获取具有可执行权限的内存空间，这就直接要求IT（即时编译）以full AOT模式，这种模式会在生成之前把IL（中间语言）翻译成机器码而不是在运行期间，这就限制了C#所有平台的更新能力。

    而lua是使用C编写的脚本语言，它在运行时读入lua编写的代码，在解释lua字节码时不是翻译为机器码，而是使用C代码进行解释，不用开辟特殊的内存空间，也不会有新代码在执行，执行的是lua虚拟机，用C写出来的虚拟机，这和C#的机制是完全不同的，因为lua是基于C的脚本语言。

    基于lua是解释型的脚本语言，不用编译。可以把lua脚本像asset下面的其他资产一样，边加载边运行。

11. 热更新是怎么实现的？

    实现原理大致可以分为两个阶段：编译阶段和运行阶段。

    ①编译阶段：

    xlua会将lua脚本转换为对应的IL（中间）代码，生成一个C#动态程序集。这个过程中，xlua会根据不同操作类型生成不同的C#调用代码，并将其作为函数体塞到对应的方法中去。例如，如果lua脚本需要执行一个C#方法，则xlua会生成一个调用该方法的代码，然后将其定义在一个被标记为monobehavior的C#类里面。这样，在运行时，我们就可以将这个类挂载到gameobject上，然后通过调用它的方法来执行lua脚本中需要的逻辑。

    ②运行阶段：

    在运行阶段，我们将从服务器上下载新的lua脚本，然后加载为一个Xlua.luaEnv对象。这个luaEnv对象封装了虚拟机，用来执行lua脚本，同时还包含其他附加功能，比如调试器、GC释放等。

    当我们需要执行某个lua脚本的时候，xlua会先到缓存池中查找有没有已经编译好的monobehavior的实例，若没有则会重新编译生成。然后，xlua会使用lua委托将编译好的C#代码封装在C#中，并通过luaEnv对象执行该lua委托。

    这样的话，在运行阶段，我们就可以通过修改服务器上的lua脚本文件夹来更新游戏逻辑和ui等内容，而不需要重新编译和发布整个项目。同时，由于xlua框架生成的C#动态程序集会被缓存下来，因此在一定程度上也提高了游戏的性能。

八、unity问题

1. ui性能会受什么因素影响？

   ①对象的数量和层次：大量的对象和复杂的层次结构会增加渲染和处理的开销。尽量减少不必要的ui元素，优化层次结构，合并ui元素。
   ②图片和纹理使用：过多过大的图片和纹理会增加内存占用和加载时间、降低性能。尽量优化和压缩图片，避免不必要的图像资源开销。

③动画和布局计算：频繁的动画和实时布局计算会增加CPU使用率和GPU负载，导致性能下降。尽量减少复杂的动画和布局计算，并使用恰当的技术（ui批处理）来减少绘制调用次数。

④UI时间和交互处理：频繁的UI监听和处理会增加CPU负载，尽量只注册必要的事件。

⑤UI更新：避免过渡频繁的UI更新，可以使用延迟更新、数据绑定、条件更新等技术，避免无效的重绘和布局操作。

2. drawcall是什么?

   绘制调用是计算机图形渲染中的一个概念，指的是向图形处理单元（GPU）发出一次命令来执行渲染操作的操作。

   当需要在屏幕上渲染一个或多个对象时，需要将绘制命令发送给GPU。每个绘制命令通常对应于一个物体、一个模型、一个图元或一组顶点等。GPU根据这些绘制命令执行相应的渲染操作。

   绘制调用的数量对于图形渲染能力有着重要的影响。较大的绘制调用数量可能导致GPU频繁地切换渲染状态和资源，增加了开销和延迟。

   常见的减少绘制调用的手段包括利用批处理技术，将多个物体或模型合并成一个绘制调用。使用着色器来实现渲染效果的共享。使用视锥体剔除等。通过减少绘制调用的数量，可以提高图形渲染性能和效率。

3. UI合批是什么?

   UI合批是一种优化技术，用于减少图形渲染中UI绘制调用（drawcall）的数量，以提高UI渲染性能。

   在UI渲染中，每个UI元素（如按钮、文本框等）通常需要进行一次绘制调用。如果UI元素过多，绘制调用的数量会显著增加，从而降低绘制性能。为了提高性能，可以使用UI合批技术将多个UI元素的渲染操作合并成一个绘制调用。

   UI合批的原理是将属于同一材质和纹理的UI元素分组，并将它们作为一个批次一起渲染。通过合并多个UI元素的渲染操作，可以减少绘制调用的数量，提高渲染效率。

   例如，如果多个按钮都使用了相同的纹理和材质，则可以将它们合并为一个批次进行渲染，而不是单独绘制每个按钮的调用。这种方式可以减少绘制调用的数量，从而提高性能。

4. 内存泄露会对游戏产生什么影响?

   ①内存占用增加：内存泄漏会导致游戏占用的内存逐渐增加，最终可能达到系统可用内存的极限。过高的内存占用可能导致系统变慢、性能下降、游戏崩溃或导致游戏无法运行。

   ②内存资源浪费：内存泄露会导致游戏资源的浪费，被泄露的资源无法被回收和重用。

5. 为什么性能会下降?

   ①当内存泄漏时，分配的内存空间没有被正确释放，这样就会再堆内存中留下一些不再使用的内存块。随着程序的运行，这些未释放的内存块就会逐渐累积。这些内存块无法被重复利用，无法提供给新的内存分配请求。

   正常情况下，内存分配器会从堆中找到一块合适大小的连续内存块来满足内存分配请求。然后，由于存在内存泄露导致的未释放内存块，堆内存中可能会出现大量的零散、不连续的小内存块，这就是内存碎片。

   内存碎片化会导致内存分配和释放变得更加复杂和低效。由于堆内存中存在大量的小块空闲内存，但这些小块无法提供给较大的内存分配请求，分配操作可能会失败或需要搜索并合并分散的内存块，增加了时间和计算开销。这样的过程会影响内存分配和释放的效率，最终导致游戏性能下降。

   ②会导致频繁进行垃圾回收（GC）

   这些对象本应该被释放以腾出内存空间供其他对象使用，但由于内存泄露而继续占用内存。垃圾回收期会在每次执行垃圾回收过程时尝试回收无用的、不可达的对象，但却无法回收泄露的对象。

   因此，在内存泄漏的情况下，垃圾回收器被迫频繁地执行垃圾回收操作，希望通过清理其他可回收对象来释放一部分内存。这会消耗大量的CPU资源和时间，影响游戏的运行性能。

频繁的垃圾回收操作会造成额外的开销，降低游戏的响应性和流畅度。

6. update和lateupdate有什么区别?

   ①顺序：update在每一帧先于lateupdate调用

   ②用途：update常用于更新游戏对象的逻辑、移动等操作，而lateupdate常用于处理跟随或相机相关的逻辑。因为在lateupdate中，已经完成了所有对象的update，完成了对象的位置和旋转操作，避免相机抖动或位置不准确的情况。

7. 游戏中的物理模拟对性能有什么影响？如何优化?

   主要是由于计算和渲染复杂的物理效果所需要的计算资源消耗。

   ①CPU负载：物理模拟涉及到计算对象之间的碰撞、重力、运动等物理效果，这些计算通常在CPU上进行。如果物理模拟非常复杂或者有大量物体参与，会增加CPU的计算负载，导致性能下降。

   ②内存使用：物理模拟可能需要保存和处理大量的物理数据，比如刚体信息、碰撞形状等。这些数据通常存储在内存中。较高的内存使用量可能会影响游戏的性能，并导致更多的内存访问和数据传输。

   ③渲染开销：物理模拟通常需要将物体的位置和姿态信息传递给渲染引擎，以正确渲染物体的外观和状态。如果物理模拟非常频繁，会产生大量的渲染开销，影响渲染性能。

   优化策略：

   ①降低物理模拟的复杂度和精度，使用简化的碰撞代替复杂形状的碰撞检测，减少物理模拟的帧率。

   ②对于不需要进行物理模拟的物体，可以将其设置为静态对象，避免不必要的物理计算和碰撞检测。

   ③进行批处理和优化，尽量减少物理模拟对象的数量和复杂度，合并物体以减少计算开销。

8. unity的动态资源加载?

   ①Resource.Load：可以使用Resource.Load函数来加载位于“Resource”文件夹中的资源。这种方式使用于经常需要加载、卸载的小型资源，如贴图、音效等。使用该方法时，必须确保资源放置在正确的目录下，且需要通过资源名称进行加载。

   ②AssetBundle：AssetBundle是一种打包和加载资源的方式，在构建时以独立文件的形式生成，可以按需加载和卸载。使用AssetBundle可以将资源打包成各种形式（例如场景、材质、模型等），然后再运行时根据需要进行加载。在运行时使用AssetBundle.LoadFromFile进行加载。

   ③UnityWebRequest：可以从远程服务器上下载资源，可以通过发送GET请求来获取需要的资源并加载到游戏中。

9. AssetBundle是什么?

   AssetBundle是unity中一种用于打包和卸载资源的工具，它允许将游戏所需的资源（如模型、贴图、音效等）打包成独立的二进制文件，并在运行时根据需要加载和卸载。

10. unity的生命周期?

    awake：第一次被创建的时候调用，初始化。

    onenable：当非活动状态转为活动状态时调用。

    start：只会执行一次，初始化。

    fixedupdate：每隔固定时间间隔调用一次，用于处理物理相关的计算和更新。

    update：每一帧都会调用update函数，用于更新对象的逻辑和状态。

    lateupdate：在所有update函数调用完，在渲染之前调用。用于处理跟随相机等操作。

    ondisable：从活跃状态变为非活动状态。

    ondestroy：被销毁时调用。

11. destroy和destroyimmediate的区别?

①destroy：异步执行，当前帧结束后销毁目标对象，在下一帧被标记为不活动并在内存管理上释放。在回调函数update、lateupdate、ondestroy仍可以执行。不能在非主线程中使用。

②destroyimmediate：同步执行，立即销毁目标对象，被销毁的对象将立即在内存管理上释放，性能开销可能较大。被销毁的对象立即变为不活跃状态并释放内存，无法在destroyimmediate后的代码中继续访问被销毁的对象。

12. 动态合批和静态合批?

    动态合批：

    灵活：在运行时根据需要动态合并多个小型物体或网格，减少渲染调用的数量，对于动态场景中频繁变化的物体很有用。

    减少内存：物体在渲染前被合并，动态合批可以减少渲染调用和渲染状态的切换，从而减少了额外的内存开销。

    额外的CPU：动态合批通常需要在运行时进行计算和合并操作，这会增加CPU的负载。对于大量物体频繁进行合批操作可能会引起性能问题。

    静态合批：

    高效的渲染性能：静态合批是在编辑器中预先计算和优化的，可以将多个静态物体或网格合并为一个批次，减少渲染调用和状态切换的开销。这对于静态场景或几乎不变的物体非常有效。

    低CPU开销：静态合批是在编辑器阶段完成的，运行时只需要简单地绘制合并后的批次，相对于动态合批而言对CPU开销较低。

    内存占用：静态合批会生成新的合并网格数据，这可能会增加内存的使用量。

九、场景题

1. 点乘和叉乘的意义?

   点乘：A*B = |A||B|cosθ

   叉乘：A*B = |A||B|sinθ

2. 怎么判断足球一定能射进球门?

   使用点乘。知道球和两个门边界的坐标值，计算它们的点乘。从而可以得到它们的cos值，知道是否能够射进。

十、其他问题

1. 你的个人职业生涯规划是怎么样的?

   我把职业规划分成了几个阶段，第一个阶段我希望我能够熟练的运用unity，对它的基本使用要十分熟悉，因此我在4月份的时候去参加了实习，在实习的过程中我比较熟练的运用了unity，并且实现了很多需求，跟完游戏的每一个版本并修复好bug。在第二个阶段我想要去了解尽可能多的unity的各种组件和高级用法，对于大部分游戏的功能实现要做到心中有数，并且要时刻注意代码的规范和复用性。第三个阶段，要更多注重性能优化。要多研究哪些用法会更多的消耗性能，并有什么方法能够优化。第四个阶段，学习计算机图形学，往渲染方向研究。

2. 对于这个岗位你有什么优势?

   ① 我参与过游戏公司的实习，对整个游戏开发的流程比较熟悉，知道在什么环节应该要和策划、美术确认什么东西。对团队协作的工具比较熟悉，比如git的使用。

   ② 我的沟通能力强，适应性比较好。上一份实习虽然是我第一次参与游戏开发的工作，但是我融入的很快。在评审会的时候我能立马理解策划提出的需求的意思，也能够依照策划的想法和美术调整模型、动作的要求，并更好的完成需求。大家对我的评价也不错。

   ③ 细心、敢于提出自己的想法。我能够敏锐的捕捉到一些需求之间的关联，或者说策划可能忽视的一些细节，在完成新需求的时候也会更多的注意到是否会牵扯到已有的功能。如果发生冲突，我会首先考虑到最优的解决方式，然后与策划商讨是否满意，或者改换其他方案。