委托代理

https://blog.csdn.net/hoppingg/article/details/126655024
https://blog.csdn.net/truthhk/article/details/127365986
代理可以帮助我们解决一对一或是一对多的任务分配工作。

代理分为几种

委托按照绑定委托函数的个数分为**单播**和**多播**委托，又按照是否可暴露给蓝图分为**静态**和**动态**委托，进行蓝图广播）

多播的用法

1.通过宏进行声明代理对象类型
2.使用代理类型进行构建代理对象
3.绑定回调对象，和操作函数
4.执行代理对象回调

动态代理与单播代理和多播代理的区别

1.动态代理构建类型名称需要用 F 开头（动态代理实现机制构建了类）
2.动态代理对象类型可以使用 UPROPERTY 标记，其他代理均无法使用（不加编译可过，调用会出错）
3.动态代理绑定对象的函数需要使用 UFUNCTION 进行描述（因为需要跟随代理被序列化）

```
DECLARE_DYNAMIC_DELEGATE(FGMDynDelegate);          //分号不要漏了
FGMDynDelegate GmDyDele;
GmDyDele.BindDynamic(ActorName,&AMyActor::Say);
GmDyDele.ExcuteIfBound();
```

动态代理用于蓝图

需要注意的是，在构建动态代理提供蓝图使用时，需要在代理上增加标记宏 UPROPERTY(BlueprintAssignable)
构建宏
DECLARE_DYNAMIC_MULTICAST_DELEGATE(FGMDynMulDele);
对象声明
UPROPERTY(BlueprintAssignable，BlueprintReadWrite)
FGMDynMulDele OnGmDynMulDele;
在 C++中调用

```
if(OnGmDynMulDele.IsBound())
{OnGmDynMulDele.Broadcast();}
```

虽然动态委托都支持反射，序列化（可被蓝图调用），但是只有动态多播可被蓝图绑定，通过指定 UPROPERTY(BlueprintAssignable), 来实现蓝图调用，动态单播可以使用 BlueprintReadWrite 标记，然后在蓝图中使用它的实例（

蓝图的动态多播实现的方法

多播和单播的底层实现逻辑？

1. 单播：

   - 单播是一种点对点通信方式，通过直接调用函数或方法来触发事件。
   - 在底层实现上，单播通常使用函数指针、成员函数指针或回调函数来绑定和调用事件处理函数。当事件触发时，执行相应的函数指针或回调函数。

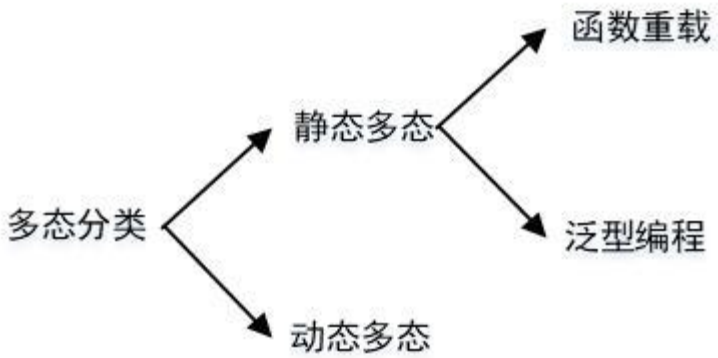
2. 多播：

   - 多播是一种一对多的通信方式，一个事件可以同时触发多个处理函数。
   - 在底层实现上，多播通常使用事件委托（Event Delegate）、函数指针数组、消息队列等机制来实现。
   - 事件委托是一种特殊的对象，它包含了一个或多个函数指针或成员函数指针，并提供了添加、删除和触发事件的方法。当注册一个处理函数时，该处理函数的指针会被添加到事件委托中。当事件触发时，事件委托会依次调用所有注册的处理函数来处理该事件。

类的多态

多态性是指用一个名字定义不同的函数，这些函数执行不同但又类似的操作，从而可以使用相同的方式来调用这些具有不同功能的同名函数。多态：具有继承关系的多种类型，称为多态类型。

形成多态的两个条件：1.**虚函数重写**。 2.**基类指针、引用去调用虚函数**。



.静态绑定：在程序的编译阶段就已经确定了程序的行为，也叫作静态多态（函数重载）

2.动态绑定：在程序运行阶段拿到具体类型在决定程序的行为，调用具体的函数，也称为动态多态（多态）

多态相关问题

1.inline 可以是虚函数吗？

可以。inline 需要展开，编译时不存在地址，但是虚函数需要将其地址存入虚表中，表现上来说，两者是互斥的。但是需要注意，inline 只是一个建议性关键字，关键取决于编译器，不会强制性执行。两者关键字存在的时候，如果是多态调用，编译器会自动忽略 inline 这个建议，因为没法将这个虚函数直接展开，这个建议无了。不是多态就可以利用此建议。

2.static 函数可以是虚函数吗？

不可以。静态成员函数没有 this 指针，直接利用类域指定的方式调用。虚函数都是为多态服务的。多态是运行时决议，而静态成员函数都是编译性决议。

3.构造函数可以是虚函数吗？

不可以。构造函数之前，虚表没有进行初始化。virtual 函数是为了实现多态，运行时去虚表找对应虚函数进行调用。对象的虚表也是在构造函数的初始化列表进行初始化的。

4.析构函数可以是虚函数。

5.拷贝构造和赋值可不可以是虚函数？

拷贝构造不可以，拷贝构造同样也是构造函数。

赋值可以，但是没有意义。

（但是可以简单实现一下父类给给子 ... ... ）但是，赋值一般是同类对象之间数据进行拷贝，这样就不存在实际价值。

6.对象访问普通函数快还是虚函数快？

不构成多态一样快，否则普通函数快。

7.虚函数表是在什么阶段生成的，存在哪里的？

构造函数初始化列表初始化的是虚函数表指针，对象中也是存的指针。

存在代码区--利用验证法，和只读常量或者静态变量的地址进行验证。

https://blog.csdn.net/weixin_61508423/article/details/127650333

https://blog.csdn.net/qq_36721032/article/details/106751173

虚函数实现：虚函数表　虚继承相关知识了解（菱形继承）

1、虚函数的作用是“运行时多态”和重载，父类中提供虚函数的实现，为子类提供默认的函数实现；子类可以重写父类的虚函数实现子类的特殊化；

2、纯虚函数：包含纯虚函数的类称为抽象类。抽象类不可以 new 出对象，只有实现了这个纯虚函数的子类才能 new 出对象。纯虚函数更像是只提供申明，不实现，是对子类的约束，是接口继承。纯虚函数是在基类中声明的虚函数，它在基类中是没有定义的，但要求任何派生类都要定义自己的实现方法。

纯虚函数表示当前类为**抽象类**，不可被实例化。被继承的派生类如果不重写此纯虚函数，那么也还是抽象类。只有被重写后，才不是抽象类。

3、普通函数：静态编译的，没有运行时多态。

注意：纯虚函数在基类中只申明不定义，但是必须在子类中加以实现，像接口。

虚函数在基类中是由定义的，即便是空，子类中可以重写。

所谓虚函数，就是 C++实现多态性的方法

友元不能是虚函数，因为虚函数必须是成员函数，而友元不是成员类。但是友元函数可以使用虚函数。

如果基类中的某些函数被说明为虚函数，那么也意味着 允许它的派生类去重新定义这个函数。

虚函数表：

C++中的虚函数的实现一般是通过虚函数表，

虚函数表实际上是一个数组，其中存储了为类对象进行声明的虚函数的地址。

对于每个含有虚函数的类，编译器都会创建一个虚函数表；

虚函数表属于类而不是对象

如果一个类中有虚函数，那么该类就有一个虚函数表。

（菱形继承）：

https://blog.csdn.net/challenglistic/article/details/127717980

泛型编程

泛型编程就是以独立于任何特定类型的方式编写代码，而模板是泛型编程的基础

泛型编程是把数据类型作为一种参数传递进来。　模板函数是在泛型编程的基础

https://blog.csdn.net/SNAKEpc12138/article/details/115977745

子系统　（单例）

https://zhuanlan.zhihu.com/p/61780989

https://www.cnblogs.com/shiroe/p/14819721.html

虚幻引擎 4（UE4）中的子系统是生命周期受控的自动实例化类`

子系统	继承自
引擎	UEngineSubsystem 类
编辑器	UEditorSubsystem 类
游戏实例	UGameInstanceSubsystem 类
本地玩家	ULocalPlayerSubsystem 类

子系统将自动公开到蓝图，带有自动理解 context 的智能节点，不需要 Cast 转型。。您可以使用标准的 UFUNCTION() 标记和规则来控制蓝图可以使用哪些 API。

Initialize 做一些初始化操作。Deinitialize()做一些清除操作。

引擎子系统，子系统将在模块的 Startup() 函数返回后执行 Initialize()，子系统将在模块的 Shutdown() 函数返回后执行 Deinitialize()。

代表引擎，数量 1。 Editor 或 Runtime 模式都是全局唯一，从进程启动开始创建，进程退出时销毁。
编辑器子系统当编辑器子系统的模块加载时，子系统将在模块的 Startup() 函数返回后执行 Initialize()，子系统将在模块的 Shutdown() 函数返回后执行 Deinitialize()。
代表编辑器，数量 1。 顾名思义，只在编辑器下存在且全局唯一，从编辑器启动开始创建，到编辑器退出时销毁。

在 GameInstance 进行初始化时，会创建所有的 GameInstanceSubsystem。在 Shutdown()函数中会执行 Deinitialize()。代表一场游戏，数量 1。 从游戏的启动开始创建，游戏退出时销毁

在 LocalPlayer 进行初始化时，会创建所有的 LocalPlayerSubsystem。在 PlayerRemoved()函数中会调用 Deinitialize()。代表本地玩家，数量可能>1

UDynamicSubsystem
UDynamicSubsystem 只代表一种可动态加载卸载的 subsystem，类中并无属性与方法。当对应模块加载/卸载时，Dynamic Subsystem 会自动动态增加/减少现有的 Subsystem 集合中元素数量。只有 UEngineSubsystems 和 UEditorSubsystems 以

动态加载。

资源加载

引用分两类：
直接属性引用（硬性引用）：直接引用 （硬加载）
间接属性引用（软性引用）：通过间接机制（如字符串形式的对象路径）来引用（软加载）
加载方式分为：
同步加载（资源过大会导致游戏程序卡顿）
异步加载 （资源加载成功后需要进行回调通知）
在 UE4 中，我们常用的软引用有以下四种 **FSoftObjectPath、FSoftClassPath、FSoftObjectPtr、TSubclassOf** 这四个。
FSoftObjectPath，FSoftClassPath 只是存储了资源的路径，使用前必须通过加载方式方可获得资源。加载方式分为同步加载（如果资源过大会导致游戏程序卡顿）和异步加载
TSoftObjectPtr 和 TSoftClassPtr 也分为同步和异步加载，针对资源拾取类别不同，使用需要注意
FSreamableManager 构建异步处理逻辑，创建在全局游戏的单例对象中，
异步加载

```
//① 声明资源加载器（构建为栈对象，需引入头文件，不要构建为堆对象）
    FStreamableManager LoadStream;
//② 回调通知函数 ，请求加载，并绑定回调函数(资源，单播)
    LoadStream.RequestAsyncLoad
```

.同步加载（会阻塞线程）

```
 LoadStream.LoadSynchronous
```

使用同步加载 LoadSynchronous 和异步加载 RequestASyncLoad，就要提到一个类

FStreamableManager，用于管理流资产并将其保存在内存中的本机类。

动态加载 移步加载

https://cuifeng.blog.csdn.net/article/details/128965978

https://www.yii666.com/blog/429808.html

https://blog.csdn.net/hoppingg/article/details/126655024

垃圾回收机制是怎么实现

虚幻引擎中的垃圾回收（Garbage Collection）机制是一种自动管理内存的机制，用于在运行时自动释放不再使用的对象和资源，避免内存泄漏和无效的内存占用。

虚幻引擎的垃圾回收机制基于引用计数和标记清除两种策略的组合实现：

1. 引用计数：通过维护每个对象的引用计数器，记录有多少个指针引用了该对象。当引用计数为 0 时，表示该对象没有被引用，可以被回收。引用计数通过增减操作在对象之间建立或断开关系。

2. 标记清除：在虚幻引擎中，引用计数机制通常与标记清除机制结合使用。标记清除过程分为两个阶段：标记阶段和清除阶段。

- 标记阶段：从根节点（如全局变量、活跃的对象等）出发，递归遍历所有可达的对象，并将其标记为活跃状态。
- 清除阶段：遍历整个堆内存，将未标记的对象进行释放，并回收其所占用的内存空间。

在虚幻引擎中，垃圾回收机制由垃圾回收器负责执行。垃圾回收器会周期性地进行垃圾回收操作，以释放不再使用的内存。

以下是虚幻垃圾回收机制的基本原理：

1. 根节点：虚幻引擎的垃圾回收器从一组特定的根节点开始遍历。这些根节点可以是全局变量、活跃的对象、堆栈中的引用等。根节点是程序中被直接或间接引用的对象的起点。

2. 标记阶段：从根节点开始，垃圾回收器通过递归或迭代的方式遍历所有可达的对象，并将其标记为活跃状态。这意味着这些对象仍然被引用着，需要保留在内存中。

3. 清除阶段：在标记阶段之后，垃圾回收器遍历整个堆内存，将未被标记的对象进行释放，并回收其占用的内存空间。这些未被标记的对象被认为是不再使用的，可以安全地释放。

4. 内存压缩（Optional）：在清除阶段之后，如果需要，虚幻引擎的垃圾回收机制可能会执行内存压缩操作。内存压缩可以将存活对象向一端移动，以减少内存碎片化并提高内存的连续性。

反射是怎么实现的

虚幻引擎中的反射（Reflection）是一种在运行时获取和操作类型信息的机制。它允许程序在不事先知道类结构的情况下，通过字符串或其他方式动态地创建、访问和修改对象。

虚幻引擎的反射系统基于元数据（Metadata）和反射 API 实现。元数据是描述类型信息的数据，包括类名、字段、属性、方法等。虚幻引擎通过编译器或脚本解释器生成并存储这些元数据。

在运行时，通过反射 API，可以根据类名或类型来获取对应的元数据。根据元数据，可以动态创建对象、调用方法、读写字段等操作。虚幻引擎的反射系统提供了一系列的接口和函数，如 UClass、UObject、UFunction 等，用于访问和操作类型信息。

具体实现上，虚幻引擎的反射系统使用了一些技术，包括但不限于：

1. 元数据生成：在编译时或脚本解析阶段，通过扫描源代码或解析脚本，生成与类结构相关的元数据信息，并将其存储在特定的数据结构中。

2. 元数据存储：虚幻引擎将元数据存储在指定的数据结构中，例如 UObject、UClass 和 UFunction 等对象。这些对象在运行时可以通过反射 API 进行访问和操作。

3. 类型查找：根据类名或类型信息，虚幻引擎的反射系统可以查找对应的元数据，并返回相应的对象或结构信息。这样可以实现动态创建对象、调用方法等功能。

4. 动态调用：基于元数据信息，反射系统可以动态地调用方法、读写字段等操作。通过函数指针或函数名称，可以在运行时执行相应的操作。

需要注意的是，虚幻引擎的反射系统是一种高级特性，它带来了灵活性和动态性，但也会带来一定的性能开销。因此，在使用反射时需要权衡其优劣，并避免滥用，以确保代码的性能和可维护性。

虚幻引擎的反射系统基于元数据（Metadata）和反射 API 实现。下面是其基本原理：

1. 元数据生成：在编译时或脚本解析阶段，虚幻引擎会通过扫描源代码或解析脚本来生成与类结构相关的元数据。元数据包括类名、字段、属性、方法等信息，用于描述类型的结构和特性。这些元数据被存储在特定的数据结构中。

2. 元数据存储：虚幻引擎将生成的元数据存储在特定的数据结构中。常见的数据结构有 UClass、UObject 和 UFunction 等对象。这些对象提供了访问和操作元数据的接口和函数。

3. 类型查找：在运行时，通过反射 API，可以根据名称、类型或其他标识符来查找对应的元数据。通过元数据，可以获取到类型的属性、方法、继承关系等信息。虚幻引擎的反射系统提供了一系列的接口和函数，如 FindClass、FindFunction 等，用于根据名称或类型来查找对应的元数据。

4. 动态创建和访问：基于元数据信息，反射系统可以动态地创建对象、调用方法、读写字段等操作。通过元数据中包含的类型信息和函数指针，可以在运行时实例化对象、调用相应的方法，并进行属性的获取和赋值操作。

UHT 和 UBT 是什么？

虚幻引擎中的 UHT 和 UBT 是两个重要的构建工具
UHT（Unreal Header Tool）：UHT 是虚幻引擎中的一个预处理器工具，用于自动生成和处理 Unreal Engine 的头文件。通过分析 C++ 源代码中的特定宏和标记，UHT 可以生成一些额外的代码、元数据和反射信息，以支持虚幻引擎的高级功能，如蓝图系统、序列化、资源管理等。
UBT（Unreal Build Tool）：UBT 是虚幻引擎中的构建工具，用于编译、链接和打包项目。它负责根据项目配置和规则，将源代码、资源文件和插件等内容编译成可运行的程序或游戏。

C++设计模式

1. 单例模式（Singleton Pattern）：确保一个类仅有一个实例，并提供全局访问点。
2. 工厂模式（Factory Pattern）：通过工厂类来创建对象，隐藏对象的具体实现。
3. 抽象工厂模式（Abstract Factory Pattern）：提供一个接口，用于创建一系列相关或依赖对象的家族，而不需要指定具体类。
4. 建造者模式（Builder Pattern）：将一个复杂对象的构建过程与其表示分离，使得同样的构建过程可以创建不同的表示。
5. 原型模式（Prototype Pattern）：通过复制已有对象来创建新对象，避免了显式的类的实例化。
6. 适配器模式（Adapter Pattern）：将一个类的接口转换成客户端所期望的另一个接口，使得原本不兼容的类能够一起工作。
7. 装饰器模式（Decorator Pattern）：动态地给一个对象添加额外的功能，同时又不改变其接口。
8. 观察者模式（Observer Pattern）：定义了一种一对多的依赖关系，当一个对象状态发生改变时，所有依赖它的对象都会得到通知并自动更新。
9. 策略模式（Strategy Pattern）：定义了一系列算法，将每个算法封装起来，并使它们可以互相替换。
10. 迭代器模式（Iterator Pattern）：提供一种顺序访问一个聚合对象中各个元素的方法，而又不需要暴露该对象的内部表示。

11. 模板方法模式（Template Method Pattern）：定义了一个算法的骨架，将一些步骤延迟到子类中实现。
12. 外观模式（Facade Pattern）：为子系统中的一组接口提供一个统一的界面，使得子系统更易于使用。
13. 状态模式（State Pattern）：允许对象在内部状态发生改变时改变其行为，看起来像是改变了对象的类。

C++11 新特性

https://blog.csdn.net/dnty00/article/details/126171485

1. 自动类型推导（auto）：通过使用 `auto` 关键字，编译器可以根据变量的初始化表达式自动推导出变量的类型。
2. 统一的初始化语法（uniform initialization）：引入了用花括号 `{}` 进行初始化的统一语法，可以用于初始化各种类型的对象。
3. 管理资源的类（RAII）：新增了 `std::unique_ptr` 和 `std::shared_ptr` 等智能指针类，用于更安全地管理动态分配的内存和其他资源。
4. 右值引用和移动语义（rvalue references and move semantics）：引入了 `&&` 运算符表示右值引用，使得在特定情况下可以高效地转移资源而不需要深拷贝。
5. lambda 表达式：提供了一种简洁的方式定义匿名函数，可以方便地在代码中创建和使用函数对象。
6. constexpr：允许声明常量表达式函数，这些函数在编译时求值，并可以用于编译期间进行常量计算。
7. foreach 循环（range-based for loop）：引入了基于范围的循环语法，使得遍历容器、数组等元素更加方便。
8. nullptr：引入了 `nullptr` 关键字，用于明确表示空指针，取代了传统的使用整型 0 或宏定义 NULL 来表示空指针。
9. 强类型枚举（strongly typed enums）：允许为枚举类型指定底层数据类型，并提供了更强的类型检查和类型安全性。
10. 并发编程支持：引入了原子操作、线程库等新特性，方便开发者处理并发编程相关的任务。

动态库和静态库区别？

库帮助我们解决代码复用问题，并且提供跨工程项目的复用结构，库是写好的现有的，成熟的，可以复用的代。库的介入方式，总体分为动态库和静态库
静态库 Libs

优点：运行速度快　编译成功后，程序可以脱离静态库，

缺点：可执行文件相对较大　当静态库修改后，可执行文件就需要重新编译

动态库 DLL

优点：可执行文件相对较小，当动态库修改后，可执行文件不需要重新编译

缺点：与静态库相比，运行速度较慢，需要依赖共享库文件

CPP 中指针是不是数据结构？

不，指针只是一种类型，而不是一种结构

CPP 中定义常量使用宏还是 const？

定义常量可以用 Const 也可以用 #define，
Define 是在编译之前进行，const 是在编译阶段处理的
const 常量有数据类型，而宏常量没有数据类型，编译器可以对 const 进行类型安全检查。#define 没有数据类型，只是单纯的替换，没有类型安全检查。常量用宏主要是提高可读性，而 const 表示这个量不应该被重新赋值，该用宏用宏，该应常量用常量，同时用也有同时用的道理。

运算符重载的意义？　　重载符 `operator`

运算符重载是对已有的运算符赋予多重含义，使同一个运算符作用于不同类型的数据时导致不同的行为，而一个类两个对象之间成员进行运算必须重新定义，让编译器在遇到对象运算时能按我们要求的进行运算，这就是运算符重载的意义

深拷贝 和浅拷贝？

1.浅拷贝： 将原对象或原数组的引用直接赋给新对象，新数组，新对象 / 数组只是原对象的一个引用
2.深拷贝： 创建一个新的对象和数组，将原对象的各项属性的“值”（数组的所有元素）拷贝过来，是“值”而不是“引用”

设计模式中单例模式的意义？

单例模式，也叫单子模式，是一种常用的软件设计模式，该模式的特点是唯一的一个，为了

保证唯一性，保证整个项目中类的实例对象有且只有一个
单例模式的使用场景非常广泛，它可以用在系统的任何地方，特别是在需要全局变量或者需要在多个模块之间共享数据的地方。例如，在一个多线程的应用程序中，可以使用单例模式来确保每个线程都使用同一个实例。
饿汉模式：静态成员在类加载时就被创建，一开始就加载到内存中，没被调用就创建，没饿就开始吃了，称为饿汉模式
懒汉模式（延迟加载）：当类加载时不创建实例，当第一个用户调用时才创建，后面调用都是调用第一次被创建的。

友元函数的优缺点？

私有成员只能在类的成员函数内部访问，如果想在别处访问对象的私有成员，只能通过类提供的接口（成员函数）间接地进行，就是想在类的成员函数外部直接访问对象的私有成员。一个类 A 可以将另一个类 B 声明为自己的友元，类 B 的所有成员函数就都可以访问类 A 对象的私有成员
优点：能够提高效率，表达简单、清晰。
缺点：友元函数破坏了封装机制，尽量不使用成员函数，除非不得已的情况下才使用友元函数。

面向对象的特效？

抽象，封装，继承，多态

面向对象的的设计原则，六个？

1. 单一职责原则：是每个类应该只有一个职责，对外只提供一个功能而引起类变化的原因应该只有一个
2. 开闭原则：一个对象对扩展开放，对修改关闭
3. 接口分离法：一个接口不需要提供太多的行为，一个接口应该只提供一种对外功能，不应该把所有的操作都封装到一个接口中
4. 里氏替换法则：同一个继承体系中的对象应该有共同的行为特征
5. 依赖倒置原则：高层模块不应该依赖于底层模块，都要依赖于抽象，编程的时候针对抽象类或者接口编程，而不是具体实现编程
6. 迪米特原则：一个对象应当对其他对象尽可能少的了解。降低各个对象之间的耦合，提高系统的可维护性，在模块之间应该只通过接口编程，而不理会模块的内部工作原理，它可以使各个模块耦合度降到最低，促进软件的复用。

针对接口编程的意义？

满足设计模式，里氏代换原则，依赖倒转原则，使得代码可以维护复用，易于扩展和修改。

智能指针的实现和意义？

程序设计中使用堆内存是非常频繁的操作，堆内存的申请和释放都由程序员自己管理。程序员自己管理堆内存可以提高了程序的效率，但是整体来说堆内存的管理是麻烦的，引入了智能指针的概念，方便管理堆内存。指针的最大问题是在复杂情况下很难管理好它指向的资源的生命周期。因此智能指针要做的就是在管理资源生命周期这件事情上更加“智能”。
智能指针只能使用于自定义类，U 类禁止使用，智能指针强调的是当前内存的使用者存在多少，当不存在时，进行回收！

智能指针分几种？

共享指针 TSharedPtr
共享引用 TSharedRef
弱指针 TWEAkPtr

智能指针怎么做到管理内存的？
智能指针本质的目的是将释放内存工作进行托管。当两个智能指针指向同一个空间，一个设置为空，另一个不会跟随为空，智能指针设置为空并不是释放内存空间，只是在减少空间引用，智能指针只能使用于自定义类，U 类禁止使用。
共享指针是怎么实现的
共享指针是虚幻中最常用的智能指针，在操作上可以帮助我们构建托管内存指针！共享指针本身非侵入式的，这使得指针的使用与操作和普通指针一致！共享指针支持主动指向空，并且共享指针是线程安全的，节省内存，性能高效

共享引用是怎么实现的
共享引用禁止为空，表明了共享引用创建后必须给予有效初始化，可以使得代码更加安全简洁，保证了对象访问的安全性。无法主动释放共享引用，可以跟随对象释放减少引用计数器

共享指针和共享引用和弱指针是怎么实现的 ？有什么区别？

1. 共享指针（std::shared_ptr）：共享指针是一种引用计数智能指针，可以跟踪有多少个共享指针共同拥有一个对象。它通过使用引用计数来判断何时释放对象的内存。当创建一个共享指针时，引用计数初始化为 1。每次拷贝构造或拷贝赋值一个共享指针时，引用计数都会增加。当引用计数减为 0 时，即没有指针指向该对象时，释放对象的内存。

2. 共享引用（std::shared_ptr、std::weak_ptr）：共享引用实际上是共享指针的一种扩展。与共享指针类似，共享引用也使用引用计数来管理对象的生命周期。共享引用有两种类型：强引用和弱引用。强引用类似于共享指针，可以访问和管理对象。而弱引用（std::weak_ptr）则是一种不拥有对象所有权的引用，它不会增加引用计数。弱引用通常用于避免循环引用问题，其主要作用是在需要时提供对对象的访问，但不会延长对象的生命周期。

3. 弱指针（std::weak_ptr）：弱指针是一种特殊的共享引用，它能够检测到对象是否已被销毁。与其他智能指针不同，弱指针并不控制对象的生命周期，而是提供对对象的非拥有性访问。通过调用 `lock()` 方法可以获得一个共享指针，如果对象仍然存在，则返回一个指向它的共享指针；否则返回一个空的共享指针。

区别：

- 共享指针和共享引用是拥有对象所有权的智能指针，它们以引用计数方式来管理对象的生命周期。
- 共享指针可以通过复制构造函数和赋值运算符进行拷贝，每次拷贝都会增加引用计数。而共享引用提供了强引用和弱引用两个类型，弱引用不会增加引用计数。
- 共享指针和共享引用可以通过 `get()` 方法来获取原始指针。
- 弱指针不拥有对象所有权，它只是提供对对象的非拥有性访问，并可以检测对象是否已被销毁。弱指针通过调用 `lock()` 方法来生成一个共享指针，用于安全地访问对象。

总结来说，共享指针用于多个指针共享拥有一个对象，共享引用扩展了共享指针的概念，提供了强引用和弱引用两种方式，而弱指针不拥有对象所有权，可以安全地检测对象是否已被销毁。

智能指针和引用的区别？

1. 所有权：智能指针拥有对对象的所有权，可以自动管理对象的生命周期。而引用只是对象的别名，不具备所有权，不能控制对象的生命周期。

2. 空值：智能指针可以持有空值（nullptr），表示没有指向有效对象。而引用必须始终指向一个有效的对象，不能为 null。

3. 可变性：智能指针可以是可变的或不可变的，具体取决于所使用的智能指针类型。通过智能指针，可以修改所拥有对象的状态。而引用是始终不可变的，不能修改所引用对象的状态。

4. 所属关系：智能指针可以有多个指针共享同一个对象，通过引用计数来管理对象的释放。引用本身不存在所属关系，它只是指向已经存在的对象。

5. 使用范围：智能指针通常用于动态分配的对象，可以方便地管理内存资源。引用通常用于传递函数参数、返回值或在容器遍历等情况下，提供更高效的访问方式。

指针数组和数组指针区别？

指针数组：是指一个数组里面装着指针，也即指针数组是一个数组；
数组指针：是指一个指向数组的指针，它其实还是一个指针，只不过是指向数组而已；

内联函数的意义？　　关键字 inline

调用时并不通过函数调用的机制，而是通过将函数体直接插入到调用处(编译阶段会实现)从而提高程序的运行效率

指针和引用的区别？

相同点:
都是地址的概念;指针指向一块内存，它的内容是所指内存的地址;引用是某块内存的别名。
区别:
指针是一个实体，而引用仅是个别名;
引用使用时无需解引用(*)，指针需要解引用;
引用只能在定义时被初始化一次，之后不可变;指针可变;
引用没有 const，指针有 const;
引用不能为空，指针可以为空;
“sizeof 引用”得到的是所指向的变量(对象)的大小，而“sizeof 指针”得到的是指针本身(所指向的变量或对象的地址)的大小;
指针和引用的自增(++)运算意义不一样;
从内存分配上看:程序为指针变量分配内存区域，而引用不需要分配内存区域。

进程和线程的区别？

线程是进程的一部分，进程是程序的一部分
进程：进程是系统分配资源和调度的基本单位，也就是说进程可以单独运行一段程序。
线程：线程是 cpu 调度和分派的最小基本单位。

共用体？

共用体 格式 union
共用体作用：共享内存，不浪费空间。
共用体类型与结构体本质上的不同：
①结构体的各成员有各自存储单元，一个结构体类型变量占用的存储单元长度与各成员所占存储单元长度有关。
②共用体类型的各成员所占存储单元的 1 长度是各成员所占用存储单元最长的长度。
共用体是将不同类型的数据项存放于同一段存储单元的一种构造数据类型。

main 函数执行前后做了什么？

首先 main()函数只不过是提供了一个函数入口，在 main()函数中的显示代码执行之前，会由编译器生成_main 函数，其中会进行所有全局对象的构造以及初始化工作。简单来说对静态变量、全局变量和全局对象来说的分配是早在 main()函数之前就完成的，所以 C/C++中并非所有的动作都是由于 main()函数引起的。

同理在 main()函数执行后，程序退出，这时候会对全局变量和全局对象进行销毁操作，所以在 main()函数还会执行相应的代码。
在上面的例子中，a 的构造函数会先执行，再执行 main，最后会调用 a 的析构函数。

平衡二叉树

规则 1：每个节点最多只有两个子节点（二叉）

规则 2：每个节点的值比它的左子树所有的节点大，比它的右子树所有节点小（有序）

规则 3：每个节点左子树的高度与右子树高度之差的绝对值不超过 1

红黑树的性质

每个结点不是红色就是黑色

根节点是黑色的

如果一个节点是红色的，则它的两个孩子结点是黑色的

对于每个结点，从该结点到其所有后代叶结点的简单路径上，均包含相同数目的黑色结点

每个叶子结点都是黑色的(此处的叶子结点指的是空结点)

平衡二叉树和红黑树的区别

平衡二叉树的左右子树的高度差绝对值不超过 1，但是红黑树在某些时刻可能会超过 1，只要符合红黑树的五个条件即可。

二叉树只要不平衡就会进行旋转，而红黑树不符合规则时，有些情况只用改变颜色不用旋转，就能达到平衡。

红黑树的左旋右旋

https://blog.csdn.net/promsing/article/details/126555227
https://blog.csdn.net/weixin_42645678/article/details/128219292

红黑树左旋右旋？

`std::map` 是一种关联容器，用于存储键值对，并按照键的顺序进行排序。`std::map` 的底层实现通常使用红黑树（Red-Black Tree）来维护有序性和平衡性。
左旋和右旋操作是红黑树中重要的平衡操作，用于保持红黑树的平衡性。

1. 左旋操作：以某个节点为支点，将其右孩子上升为新的父节点，原父节点成为新父节点的左孩子，新父节点的左孩子成为原父节点的右孩子。
2. 右旋操作：以某个节点为支点，将其左孩子上升为新的父节点，原父节点成为新父节点的右孩子，新父节点的右孩子成为原父节点的左孩子。

什么是栈？什么是堆？

栈和堆是指：栈是一种运算受限的线性表，限定仅在表尾进行插入和删除操作的线性表，这一端被称为栈顶，相对地，把另一端称为栈底；
堆是计算机科学中一类特殊的数据结构的统称，通常是一个可以被看做一棵树的数组对象。
栈区（stack）— 由编译器自动分配释放 ，存放函数的参数值，局部变量的值等
堆区（heap） — 一般由程序员分配释放， 若程序员不释放，程序结束时可能由 OS 回收 。堆是由程序员自己申请并指明大小
栈由系统自动分配，速度较快。但程序员是无法控制的。
堆是由 new 分配的内存，一般速度比较慢，而且容易产生内存碎片,不过用起来最方便.
https://www.zhihu.com/question/19729973

堆和栈怎么调用？

1. 调用堆：

    - 堆是动态分配内存的区域，主要用于存储程序运行时动态创建的对象。在大多数编程语言中，可以使用 `new` 关键字或者相应的内存分配函数（如 `malloc()`）来在堆上分配内存。
    - 要调用堆中的对象，通常需要通过对象的引用或指针进行访问。可以使用对应的语法或操作符来获取、修改或释放堆上的对象。例如，在 C++中可以使用指针操作符（`->`）来访问对象的成员方法和属性。

2. 调用栈：

    - 栈用于管理程序的执行流程和函数调用。每当函数被调用时，会在栈上创建一个新的栈帧（Stack Frame），用于保存函数的局部变量、参数和返回地址等信息。
    - 要调用栈中的数据，需要了解当前栈帧的结构以及相应的编程语言和调试工具。在大多数编程语言中，可以通过特定的语法或调试工具来访问和操作调用栈。例如，在 C/C++中，可以使用指针操作符（`*`）来访问栈上的变量。

数组和 Vector 区别？

1、内存中的位置
C++中数组为内置的数据类型，存放在**栈**中，其内存的分配和释放完全由系统自动完成；
vector，存放在**堆**中，由 STL 库中程序负责内存的分配和释放，使用方便。
2、大小能否变化

数组的大小在初始化后就固定不变，而 vector 可以通过 push_back 或 pop 等操作进行变化。

3、初始化

数组不能将数组的内容拷贝给其他数组作为初始值，也不能用数组为其他数组赋值；而向量可以。

4、执行效率

数组>vector 向量。主要原因是 vector 的扩容过程要消耗大量的时间。

vector 是 c++标准库提供的，是动态的，便于扩展，还提供了排序等更多功能。而数组一般是固定长度的，功能比较单一

vector 容器与数组相比其优点在于它能够根据需要随时自动调整自身的大小以便容下所要放入的元素。此外, vector 也提供了许多的方法来对自身进行操作。

Vector 和 list 的区别？

1，Vector

连续存储的容器，动态数组，在堆上分配空间

底层实现：数组

两倍容量增长：

vector 增加（插入）新元素时，如果未超过当时的容量，则还有剩余空间，那么直接添加到最后（插入指定位置），然后调整迭代器。

如果没有剩余空间了，则会重新配置原有元素个数的两倍空间，然后将原空间元素通过复制的方式初始化新空间，再向新空间增加元素，最后析构并释放原空间，之前的迭代器会**失效。**

2、List

动态链表，在堆上分配空间，每插入一个元数都会分配空间，每删除一个元素都会释放空间。

底层：双向链表

区别：

1）vector 底层实现是数组；list 是双向 链表。

2）vector 支持随机访问，list 不支持。

3）vector 是顺序内存，list 不是。

4）vector 在中间节点进行插入删除会导致内存拷贝，list 不会。

5）vector 一次性分配好内存，不够时才进行 2 倍扩容；list 每次插入新节点都会进行内存申请。

6）vector 随机访问性能好，插入删除性能差；list 随机访问性能差，插入删除性能好。

应用

vector 拥有一段连续的内存空间，因此支持随机访问，如果需要高效的随即访问，而不在乎插入和删除的效率，使用 vector。

list 拥有一段不连续的内存空间，如果需要高效的插入和删除，而不关心随机访问，则应使用 list。

c++vector 怎么实现的，扩容怎么扩容？

`std::vector` 是一个动态数组容器，可以根据需要自动调整大小。扩容是自动完成的，当插入新元素时，如果当前的容量不足以容纳更多的元素，`std::vector` 会自动分配一块更大的内存空间，并将现有元素拷贝到新的内存空间中

TArray 和 C++STL 里边的 vector 容器有什么区别？

`TArray` 是虚幻引擎中的一个容器类，而 `std::vector` 是 C++ 标准库中的容器类。

TMap 怎么实现的？

Unreal Engine 中的一种关联容器，用于存储键值对。它是基于红黑树（Red-Black Tree）实现的一种有序映射。

Tmap 和 map 的实现区别？

虚幻的 Tmap 实现方法是哈希表　C++的 map 实现方式是红黑树

位运算符的操作？

1. 按位与（&）：将两个操作数的对应位进行逻辑与操作，生成新的值。只有当两个位都为 1 时，结果为 1；否则为 0。

2. 按位或（|）：将两个操作数的对应位进行逻辑或操作，生成新的值。只要两个位中有至少一个为 1，结果即为 1；否则为 0。

3. 按位异或（^）：将两个操作数的对应位进行逻辑异或操作，生成新的值。当两个位相同时，结果为 0；当两个位不同时，结果为 1。

4. 按位取反（~）：对操作数的每个二进制位进行取反操作，即将 0 变为 1，将 1 变为 0。

5. 左移（<<）：将一个操作数的所有二进制位向左移动指定的位数，并在低位补 0。例如，x << n 将 x 的二进制表示向左移动 n 位。

6. 右移（>>）：将一个操作数的所有二进制位向右移动指定的位数。对于无符号数，空出的高位用 0 填充；对于有符号数，则根据最高位的符号位进行填充，称为算术右移。

时间复杂度是什么？

时间复杂度是用来衡量算法执行时间随输入规模增长而变化的量度。它描述了算法所需执行基本操作的次数或者执行时间与输入规模之间的关系。

通常使用大写字母 O 表示时间复杂度，比如 O(n)，其中 n 表示输入规模。时间复杂度可以分为几个不同的级别，常见的有：

1. 常数时间复杂度（O(1)）：无论输入规模大小，算法的执行时间都是固定的，不受输入规模的影响。
2. 线性时间复杂度（O(n)）：算法的执行时间随着输入规模线性增长，即每增加一个输入元素，算法的执行时间就会增加一个固定的单位。
3. 对数时间复杂度（O(log n)）：算法的执行时间随着输入规模呈对数增长，即每增加一倍的输入规模，算法的执行时间只增加一个固定的单位。
4. 平均时间复杂度（O(n^2)）：算法的执行时间随着输入规模的增长呈二次方增长，即每增加一个输入元素，算法的执行时间会增加一个固定的平方单位。

Vector 的插入的时间复杂度？查找的时间复杂度？

1. 插入操作的时间复杂度：
    - 在末尾插入元素的时间复杂度为平摊 O(1)，即每次插入操作的平均时间复杂度是常数。
    - 在向量中间或开头插入元素的时间复杂度为 O(n)，其中 n 是当前向量中的元素数量。这是因为插入位置之后的元素都需要向后移动一个位置。
2. 查找操作的时间复杂度：
    - 根据元素值进行查找的时间复杂度为 O(n)，其中 n 是当前向量中的元素数量。这是因为需要逐个比较元素值来确定是否匹配。
    - 根据索引进行查找的时间复杂度为 O(1)，即通过索引直接访问元素的时间是固定的。

Map 的插入的时间复杂度？查找的时间复杂度？

1. 插入操作的时间复杂度：

- 在 Map 中插入一个键值对的时间复杂度是平摊 O(log n)，其中 n 是当前 Map 中键值对的数量。这是因为 Map 通常是基于平衡二叉搜索树（如红黑树）实现的，插入操作需要维护树的平衡性。
- 注意，如果使用的是哈希表（unordered_map），则在某些情况下，插入操作的时间复杂度可能接近 O(1)，但最坏情况下仍然会退化到 O(n)。

2. 查找操作的时间复杂度：

- 根据键进行查找的时间复杂度是平摊 O(log n)，其中 n 是当前 Map 中键值对的数量。这是因为通过树的二分查找来定位键所在的位置。
- 如果使用的是哈希表（unordered_map），则在某些情况下，查找操作的时间复杂度接近 O(1)，但最坏情况下可能达到 O(n)。

List 的插入的时间复杂度？查找的时间复杂度？

1. 插入操作的时间复杂度：

- 在 List 的任意位置插入一个元素的时间复杂度是 O(1)，即在常数时间内完成。这是因为链表中的节点可以通过修改指针来进行插入操作，而不需要移动其他元素。
- 但如果要在指定位置之前或之后插入元素，则需要先进行遍历查找到该位置，因此插入操作的整体时间复杂度仍然是 O(n)，其中 n 是当前 List 中的元素数量。

2. 查找操作的时间复杂度：

- 根据索引进行查找的时间复杂度是 O(n)，其中 n 是当前 List 中的元素数量。由于链表的特性，无法像向量那样通过索引直接访问元素，必须从头开始遍历链表才能找到指定位置的元素。

树和图

树形结构：元素有层次关系，每一层上的结点只能和上一层中的至多一个结点相关，但可能和下一层的多个结点相关

图形结构：元素之间具有任意关系，任意两个元素都可能相关
树有四种遍历方式：层次遍历、先序遍历、中序遍历、后序遍历
图有两种遍历方式：深度遍历、广度遍历

树的深度优先和广度优先

深度优先遍历指的是，从树的根节点开始，先遍历左子树，然后遍历右子树。
广度优先遍历从根节点开始，沿着树的宽度依次遍历树的每个节点。

图的深度优先和广度优先

深度遍历从起点开始，先访问第一个邻接结点，然后以该结点为起点，重复该步，直到起点没有未被访问的邻接点为止，回到上一结点，继续访问下一邻接结点，重复上一步，直到访问完毕

广度遍历广度遍历有些类似与树的层次遍历：总是先遍历完一个顶点的所有邻接点再遍历下一个顶点

二分查找也称折半查找时间复杂度 O(log n)

材质描边　菲涅尔（Fresnel）　的原理

菲涅尔指的是：某种材质在不同距离上呈现出不同的反射效果。
使用菲涅尔（Fresnel）在角色或道具的材质中模拟边沿光照

通过法线判断，中间黑向四周变亮

C++集合操作之集合交集：std::set_intersection

算法 set_intersection 可以用来求两个集合的交集，此处的集合可以为 std::set，

```
std::vector<int> A = {1, 2, 2, 3, 4, 5};
    std::vector<int> B = {2, 4, 1, 2, 3};

    std::unordered_set<int> setA(A.begin(), A.end());
    std::unordered_set<int> setB(B.begin(), B.end());

    std::vector<int> result;
    std::set_intersection(setA.begin(), setA.end(), setB.begin(), setB.end(),
std::back_inserter(result));

    for (const auto& user : result) {
        std::cout << user << " ";
    }
```

冒泡排序

```
void maopao(int a[],int n){
    int temp,i,j;
    for (i=0; i<n-1; i++) {
        for(j=0;j<n-i-1;j++){
            if (a[j]>a[j+1]) {
                temp = a[j];
                a[j]=a[j+1];
```

```
                a[j+1]=temp;
            }
        }
    }
}
```

回纹

```
#include<iostream>
#include<string>
#include<algorithm>
using namespace std;
int main(){
    string a;
    cin>>a;
    int l=0,r=a.length()-1;
    while(l<r){
        if(a[l] != a[r]) break;
        l++;r--;
    }
    if (l < r) cout << a << "不是回文串" << endl;
    else cout << a << "是回文串" << endl;
    return 0;
}
```

100 个小朋友围成一个圈，设定编号为 1~100，依次按 1、2、3、4、5、6、7、8、9 循 环报数，报到 9 的出圈，直到所有小朋友出圈。请写代码打印出各个小朋友出圈顺序

```
#include <iostream>
#include <vector>

int main() {
    std::vector<int> children;
    int total = 100;

    // 初始化编号为 1~100 的小朋友
    for (int i = 1; i <= total; ++i) {
        children.push_back(i);
    }

    int count = 0; // 报数计数器
    int index = 0; // 当前小朋友的索引
```

```
    while (total > 0) {
        count++;

        if (count == 9) {
            std::cout << children[index] << " "; // 输出出圈的小朋友编号
            children.erase(children.begin() + index); // 移除出圈的小朋友

            count = 0; // 重置报数计数器
            total--; // 更新剩余小朋友数量
        } else {
            index = (index + 1) % total; // 更新当前小朋友的索引
        }
    }

    return 0;
}
```

智能指针实现（链表速度慢，怎么优化）？

关于链表速度慢的优化问题，在某些场景下，链表确实可能导致访问速度较慢，特别是对于大量数据或需要频繁随机访问的情况。为了优化链表的速度，可以考虑以下几个方面：

1. 使用更高效的数据结构：根据具体需求，可以使用其他数据结构替代链表。例如，如果需要频繁的随机访问，可以考虑使用数组或哈希表等更适合的数据结构。

2. 使用缓存：如果链表中的数据在访问时具有一定的局部性，可以使用缓存来提高访问速度。通过将链表的部分数据复制到缓存中，可以减少对链表的频繁访问。

3. 数据预处理：根据具体场景，可以在链表上执行一些预处理操作，以提前计算或缓存一些与性能相关的数据，从而减少实际访问时的计算量。

4. 分段加载：如果链表中包含大量数据，可以考虑将其分成多个段进行加载和访问，而不是一次性加载整个链表。这样可以减少内存占用和提高访问速度。

对于链表速度慢的问题，如果使用智能指针来管理链表节点，可以考虑以下优化措施：

1. 使用更高效的数据结构：链表在随机访问时的性能较差，因此可以考虑使用其他数据结构来替代链表。例如，可以使用数组、向量（vector）或哈希表等支持快速随机访问的数据结构。

2. 链表节点缓存池：为了避免频繁的内存分配和释放操作，可以使用一个链表节点缓存池来预先分配一定数量的节点，并在需要时从缓存池中获取节点，而不是每次都进行动态内存分配。这样可以减少内存分配和释放的开销，提高性能。

3. 减少智能指针拷贝操作：智能指针的拷贝操作会涉及引用计数的增加和减少，可能导致额外的开销。可以尽量避免不必要的智能指针拷贝操作，例如通过传递引用或移动语义来避免拷贝。

4. 使用弱引用指针（weak_ptr）：如果链表中的节点之间存在循环引用，可以使用弱引用指针（weak_ptr）来解决引用循环问题。弱引用指针不会增加引用计数，可以避免循环引用导致的内存泄漏。

5. 使用局部性原则：如果链表的访问具有一定的局部性特点，可以考虑使用缓存来提高访问速度。通过将链表的部分数据复制到缓存中，可以减少对链表的频繁访问，提高性能。

需要根据具体场景和需求选择合适的优化策略。以上是一些常见的优化方法，可以根据实际情况选择适合的方式来提高智能指针管理链表的性能。

C++构造函数分几种？拷贝构造函数的内存分配？拷贝构造和有参构造是一样的吗？

1. 默认构造函数（Default Constructor）：没有参数的构造函数，如果没有显式定义构造函数，则编译器会自动生成默认构造函数。它用于创建对象时不需要传递参数。

2. 参数化构造函数（Parameterized Constructor）：带有一个或多个参数的构造函数，用于通过提供参数来初始化对象的成员变量。可以根据参数的不同进行重载，以支持不同的初始化方式。

3. 拷贝构造函数（Copy Constructor）：用于通过已存在的对象创建新对象，它将已存在对象的值复制到新对象中。拷贝构造函数的参数是同类型对象的引用或常量引用。

4. 移动构造函数（Move Constructor）：C++11 引入的特性，用于实现对象的移动语义，将已存在对象的资源（如内存、文件句柄等）转移到新对象，避免不必要的拷贝操作，提高效率。

5. 隐式构造函数（Implicit Constructor）：当对象被按值传递、返回或初始化时，编译器可以隐式调用适当的构造函数。

对于拷贝构造函数的内存分配，拷贝构造函数主要是用于对象的复制，而不是进行内存分配。拷贝构造函数的主要任务是将已存在对象的值复制到新对象中，通常不涉及显式的内存分配，而是将新对象的成员变量初始化为已存在对象的值。内存分配的具体工作在构造函数之前或之后进行，可以通过其他构造函数、运算符重载或成员函数来完成。

拷贝构造函数和有参构造函数是不同的：

- 拷贝构造函数用于通过已存在的对象创建新对象，它的参数是同类型对象的引用或常量引用。
- 有参构造函数是指带有一个或多个参数的构造函数，用于通过提供参数来初始化对象的成员变量，参数可以是任意类型。

它们的作用和使用场景不同：

- 拷贝构造函数通常用于对象的复制、传递或返回，确保新对象与原对象具有相同的值。
- 有参构造函数用于在创建对象时提供初始化参数，以灵活地满足不同的需求。

需要注意的是，拷贝构造函数可以视为一种特殊的有参构造函数，但并不是所有有参构造函数都是拷贝构造函数。当对象的创建需要提供实际的初始化值时，可以使用有参构造函数，而拷贝构造函数则专门用于对象的复制初始化。

什么是数据结构？

数据结构是计算机存储、组织数据的方式，是指相互之间存在一种或多种特定关系的数据元素的集合；
集合：数据元素之间除了有相同的数据类型再没有其他的关系。

线性结构：数据元素之间是一对一的关系 ——线性表、栈、队列。

树形结构：数据元素之间是一对多的关系。

图状结构：数据元素之间是多对多的关系。

物理结构包括顺序存储结构和链式存储结构。

对 C++有什么了解吗？C++和 JAVA，VB 这些有什么区别呢？ 和 JAVA 比有哪些优势？

C++ 相对于 Java 和 VB 具有以下优势：

- 性能：由于 C++ 是一种编译型语言，可以直接编译成机器码，因此在性能方面往往比 Java 和 VB 更高。
- 内存控制：C++ 提供了对内存的直接控制，包括手动内存管理和指针操作等。这使得 C++ 在资源受限的环境中更加灵活，并且能够实现更高效的数据结构和算法。
- 底层编程：C++ 具有底层编程能力，可以直接访问硬件、操作系统和其他底层资源。这使得 C++ 更适合开发驱动程序、嵌入式系统和高性能应用。
- 生态系统：C++ 作为一门历史悠久的语言，具有强大的生态系统和广泛的应用领域。许多重要的框架、库和工具都有 C++ 版本，提供了丰富的开发资源和支持。

虚函数在内存方面是重载还是覆盖？

虚函数在内存方面实现的是覆盖（override），而不是重载（overload）。虚函数是通过虚函数表和虚函数表指针的机制来实现对虚函数的覆盖

在 C++ 中，结构体（struct）和类（class）两者非常相似，但也有一些区别：

1. 默认访问权限：在结构体中，默认的成员访问权限是公共的（public），而在类中，默认的成员访问权限是私有的（private）。这意味着结构体的成员在外部可以直接访问，而类的成员需要通过公共接口或友元才能访问。

2. 继承方式：在结构体中，默认的继承方式为公有继承（public inheritance），而在类中，默认的继承方式为私有继承（private inheritance）。

3. 成员函数：类中可以定义成员函数，包括构造函数、析构函数和其他成员函数。而结构体中只能定义成员变量，不能定义构造函数和析构函数（C++11 及之后的版本允许在结构体中定义构造函数和析构函数）。

4. 使用习惯：一般来说，结构体用于表示一组相关的数据，而类更多地用于封装数据和行为，实现面向对象的编程。

全局变量会存在哪个地方？

全局变量存放在静态存储区,位置是固定的。 局部变量在栈空间,栈地址是不固定的。
栈：就是那些由编译器在需要的时候分配，在不需要的时候自动清除的变量的存储区。里面的变量通常是局部变量、函数参数等。
堆：就是那些由 new 分配的内存块，他们的释放编译器不去管，由我们的应用程序去控制，一般一个 new 就要对应一个 delete。如果程序员没有释放掉，那么在程序结束后，操作系统会自动回收。

const 一个全局变量会存在哪个地方？

const 全局的分配内存的话都是在只读数据区，所以是不能改变的。但是 const 局部变量分配空间是在栈区，所以可以通过指针来修改 const 局部变量。

类的方法后面加不加 Override 有什么理解？

1.简单来说@override 注解是告诉编译器，下面的方法是重写父类的方法

2.如果不写@override 注解去直接重写方法，编译器是不会判断你是不是正确重写了父类中的方法的。如重写方法时参数与父类不同，程序是不会提示报错的。这会留下一

个潜在的 bug。当你写了@override 注解时，程序会判断你是否正确的重写了父类的对应方法。而且加上此注解后，程序会自动屏蔽父类的方法。

检查内存泄露?

养成良好习惯，保证 malloc/new 和 free/delete 匹配。
检查 malloc/new 和 free/delete 是否匹配，一些工具也就是这个原理。
利用宏或者钩子，在用户程序与运行库之间加了一层，用于记录内存分配情况。

在 C++中，可以使用一些工具和技术来检查内存泄漏。下面是几种常见的方法：

1. 使用内存调试工具：许多集成开发环境（IDE）和调试器提供了内存调试工具，例如 Visual Studio 的内存调试器、Valgrind 等。这些工具能够跟踪内存分配和释放的情况，并检测未释放的内存块。

2. 重载 new 和 delete 操作符：通过在代码中重载全局的 `new` 和 `delete` 操作符，可以记录每个内存分配和释放的位置，并进行统计。在程序结束时，可以检查是否有未释放的内存块。

3. 使用智能指针：C++11 引入了智能指针（如 `std::shared_ptr` 和 `std::unique_ptr`），它们可以自动管理动态分配的内存。使用智能指针可以避免手动释放内存时可能出现的错误。当没有引用指向某个对象时，智能指针会自动释放其所拥有的内存。

4. 编写测试用例和性能分析：编写测试用例并进行系统的测试，尤其是对于涉及动态内存分配和释放的部分进行更加详细的测试。同时，使用性能分析工具来监测内存使用情况和分析内存泄漏的情况。

5. 代码审查和规范：通过仔细审查代码并遵循良好的编码规范，可以减少内存泄漏的潜在问题。例如，在每次分配内存后，需要确保在适当的地方进行相应的释放操作。

在虚幻游戏中，检测和应对作弊行为是很重要的。针对玩家改变跳跃高度这种作弊行为，可以采取以下一些常见的检查方法：

1. 游戏日志分析：通过分析游戏的日志文件，可以查看玩家的行为记录，包括跳跃的次数、高度等信息。如果某个玩家的跳跃行为异常，超出正常范围，可能存在作弊行为。

2. 数据统计和异常检测：通过收集和分析大量玩家的游戏数据，可以建立基准模型或规则来判断玩家的跳跃行为是否异常。例如，通过统计其他正常玩家的跳跃高度分布，并与被怀疑作弊的玩家进行对比，如果存在明显偏差，则可能是作弊行为。

3. 服务器端验证：将跳跃逻辑放在服务器端进行处理，并对玩家跳跃请求进行验证。服务器可以检查玩家每次跳跃的高度、频率，以及与其他玩家之间的同步性等。如果玩家的跳跃行为不符合规则，可以判定为作弊。

4. 反作弊技术：虚幻引擎本身提供了反作弊技术和机制，可以使用这些技术来检测和防止作弊行为。例如，虚幻引擎的 Anti-Cheat System（ACS）模块可以监测并阻止一些常见的作弊行为，包括非法修改游戏进程、参数、脚本等。

需要注意的是，作弊行为的检测往往需要综合多种手段和算法，并且需要持续地进行改进和更新，以适应不断变化的作弊手段。此外，正确的判断作弊行为需要仔细分析和验证，避免对正常玩家产生错误的判定。因此，在游戏开发过程中，建议采用多层次的防作弊策略，并与社区玩家保持良好的沟通，共同努力维护游戏的公平性和健康环境。

哈希表是怎么实现的

哈希表（Hash Table）是一种常用的数据结构，它可以高效地实现键值对的存储和查找。哈希表的原理基于哈希函数（Hash Function）和数组（Array）。

首先，哈希函数将输入的键转换为一个固定长度的哈希值。这个哈希值通常是一个整数或者一个固定长度的字符串。哈希函数的设计要尽量使得不同的键产生不同的哈希值，并且具有均匀分布的特性。

数组的特点是：寻址容易，插入和删除困难；而链表的特点是：寻址困难，插入和删除容易。那么我们能不能综合两者的特性，做出一种寻址容易，插入删除也容易的数据结构？答案是肯定的，这就是我们要提起的哈希表，哈希表有多种不同的实现方法/

开放定址法、链地址法（拉链法）、再哈希法、建立公共溢出区等方法
拉链法：
左边很明显是个数组，数组的每个成员包括一个指针，指向一个链表的头，当然这个链表可能为空，也可能元素很多。我们根据元素的一些特征把元素分配到不同的链表中去，也是根据这些特征，找到正确的链表，再从链表中找出这个元素。

共享引用的安全性体现在，如果使用共享引用构建的对象，无法将对象空间设置为空。如果想释放内存，可以借助指向其他共享引用来减少引用计数，来释放空间
共享引用本质，无法主动减少引用计数器，只能通过被动方法，例如生命周期终结，共享引用易主

FK 和 IK　FK 是正向运动学　　IK 是逆向运动学 。

(FK 就像：运动数据给骨骼 骨骼动。IK 就像外力给动子层级，带动父层级动例如拉扯你的手臂)

FK 指的是对于有层级关系的对象来说，父级对象的动作将影响到子级对象，而子对象的动作将不会对父对象造成任何影响。当对父对象进行移动时，子对象也会同时随着移动。而子对象移动时，父对象不会产生移动。由此可见，正向运动中的动作是自上而下传递的。例如手臂动画，大臂移动或旋转，小臂和手随之移动或旋转，小臂移动或旋转，手随之移动或旋转。

IK 与正向运动学不同，反向运动学动作传递是双向的，当父对象进行位移、旋转或缩放等动作时，其子对象会受到这些动作的影响，反之，子对象的动作也将影响到父对象。例如，若要将手移到门把上，手臂中的其他关节也将发生旋转，以适应手的新位置。

动画在模拟有牵引、拖拽、支撑等动作时使用 IK 会比较方便快捷，如俯卧撑、推车等。但是在处理主动发力的的动作时，用 FK 会灵活自如，如走路时手臂自然摆动、扇扇子等。

IK & FK 优缺点

FK：

优：自带运动曲线，符合人体运动规律，动画看上去自然

缺：调节控制器较多，容易产生 Gimbal lock（万向锁）的问题

IK：

优：操作直观

缺：1.动画不容易做出运动曲线　2.移动 IK 容易使模型过度拉伸

UMG 的图集

方案 1 是利用 TexturePacker 生成一个大图，然后导入到工程中，会生成 sprites，然后赋值给 UMG 中的 Image 组件。但是这种方案有个问题，就是如果更改了大图，重新导入的时候，原来的 Sprites 会因为 UV 改变而发生变化，这个时候在面板中的 Image 组件中的 image 栏是可以看到有变化的，但是 Brush 栏是没有变化的。需要重新启动编辑器，Brush 栏才会刷新，这个时候面板预览才是正确的。

像素流

利用像素流送可以在用户不可见的电脑上远程运行虚幻引擎应用程序。简而言之，就是通过远端电脑，运行虚幻工程项目，将运行逻辑时渲染的每一帧影像进行编码到一个媒体流中，然后通过一个轻量型网页服务器进行流传递。使用者即可在任意带有 web browser 的设备上访问此项目

UDP 与 TCP　UDP 的无序问题

https://www.kmw.com/news/2862282.html

UDP 是一种数据包协议，它以包的形式存在，因此每次可以接收 100200 个数据包。在一个理想的情况下，不管有多少个 recvfrom，它都会第一次收到 100 个 recvfrom。当然，可能是因为网络的原因，如果第二个包首先到达，它可能是 200。由于网络混乱，您可能会先收到 200 个数据包，因此需要在用户定义的 UDP 协议头中添加一个序列号，以标识发送和接收数据包之间的对应关系。
TCP 是流协议，所以 recv（1000）将接收 300 个 TCP，并处理重传以确保数据包的完整性。
一般来说，TCP 协议可以保证传输的安全性，帮助您解决无序重传的问题。UDP 主要用来传输一些辅助的、不重要的、无损失的信息，提供传输性能。
在构造数据包时，在数据包中设置校验码，到达目的地后使用一定的算法重新计算校验码。通过比较二者，我们可以找出损坏的数据。由于受损和丢失的数据需要重新传输，协议必须能够使目的地在需要重新传输数据时给出源的确认信号。有些数据包不一定按顺序到达，因此协议必须能够检测出无序的数据包，临时存储它们，然后以正确的顺序将它们发送到应用层。此外，协议必须能够发现并丢弃重复的数据。一组计时器可以限制不同确认的等待时间，以便可以开始重新传输或重新建立连接。

假设要做集群效应，怎么防止 ai 扎堆，让他们之间有间隔，规整？

1. 空间分配：为每个 AI 分配独立的空间，确保它们之间有一定的距离。这可以通过在虚拟环境中设置不同的位置、区域或房间来实现。确保每个 AI 都有足够的自由空间，以避免它们过于靠近或重叠。

2. 行为限制：制定行为规则和限制条件，以确保 AI 之间的交互和移动具有一定的约束和规律。例如，设定最大移动距离、最小间隔时间或其他限制规则，使得 AI 在行动时能够保持一定的分散状态。

3. 路径规划：利用路径规划算法来引导 AI 的移动路径，确保它们在相同空间内移动时有一定的分散性。通过设计合理的路径选择策略，可以避免 AI 聚集在同一条路径上。

4. 目标差异化：为每个 AI 设定不同的目标或任务，以鼓励它们在虚拟环境中选择不同的活动方向和行为方式。确保每个 AI 都有独特的目标，这样它们就会在虚拟环境中寻找不同的机会和资源，减少扎堆现象。

锚点种类，功能，全屏锚点与中心锚点区别？

1. 全屏锚点（Full Size Anchors）：全屏锚点是一种用于将 UI 元素固定在屏幕边缘或全屏范围内的锚点。它们通常用于创建全屏覆盖的 UI 效果，例如菜单或提示框。全屏锚点可以在屏幕上的任何位置进行设置，并具有不同的锚定方式，如左上角、右下角等，以实现 UI 元素的精确定位。

2. 中心锚点（Center Anchors）：中心锚点用于将 UI 元素相对于其父级容器的中心位置进行定位。通过使用中心锚点，UI 元素可以在屏幕调整大小时保持在屏幕中央或父级容器的中心位置。这对于创建自适应、居中对齐的 UI 布局非常有用。

全屏锚点与中心锚点之间的区别主要体现在以下几个方面：

1. 定位方式：全屏锚点可以通过设置不同的锚定方式来具体指定 UI 元素的位置，例如相对于屏幕的边缘或指定的屏幕坐标。而中心锚点是将 UI 元素相对于其父级容器的中心位置进行定位。

2. 使用场景：全屏锚点通常适用于需要在屏幕上固定位置的 UI 元素，例如菜单、通知或提示框等。而中心锚点则适用于需要在屏幕调整大小时保持居中对齐的 UI 元素。

3. 定位方式的影响：由于定位方式不同，全屏锚点可以使 UI 元素精确地定位在屏幕上的指定位置，而不受其父级容器影响。而中心锚点则会受到父级容器的限制，使得 UI 元素始终相对于父级容器的中心位置进行定位。

UMG 控件子父级功能上的同一性或者区别性

同一性方面：

1. 层级关系：父级控件可以包含多个子级控件，形成层级结构。
2. 布局：父级控件可以影响其子级控件的相对位置和大小，通过设置布局规则、边距等来调整子级控件的排列方式。
3. 事件传递：父级控件可以接收并转发被子级控件触发的事件，这样事件可以在整个层级结构中进行传递和处理。

区别性方面：

1. 属性继承：子级控件可以继承父级控件的某些属性值，如颜色、字体、动画效果等。当父级控件的属性发生变化时，子级控件可以自动更新相应属性的值。
2. 响应优先级：在父级和子级控件都监听同一个事件时，如果事件同时触发，通常子级控件会具有更高的响应优先级，即子级控件的事件处理函数会先于父级控件执行。
3. 控件可见性：父级控件的可见性设置可以影响其子级控件的可见性，即如果父级控件被设置为不可见，则子级控件也会相应地变为不可见。

UE 子系统的优缺点

1. 渲染子系统（Rendering Subsystem）： 优点：

    - 提供强大的渲染功能，包括实时光照、粒子效果、后期处理等。
    - 可以创建高质量的图形效果，使游戏画面更加逼真和吸引人。
    - 支持多平台渲染，适用于不同的硬件和操作系统。

    缺点：

    - 对硬件资源要求较高，在低端设备上可能性能受限。

- 学习和掌握渲染技术需要一定的时间和经验。

2. 物理子系统（Physics Subsystem）： 优点：

- 提供了物理模拟和碰撞检测等功能，使得游戏中的物体可以根据现实世界的物理规律进行交互。
- 增加了游戏的真实感和可玩性，使得角色运动、物体交互等更加逼真。

缺点：

- 需要消耗一定的计算资源和性能，特别是在处理大量物体或复杂碰撞时。
- 物理模拟可能会导致不稳定性或计算误差，需要额外的调试和优化工作。

3. 音频子系统（Audio Subsystem）： 优点：

- 支持音效的创建、混音和空间音频等功能，给游戏带来更加沉浸式的音频体验。
- 可以增强游戏的氛围和互动性，让玩家更好地融入游戏世界。

缺点：

- 音频处理对计算资源消耗较大，需要合理管理和优化。
- 在跨平台开发时，可能需要处理不同操作系统或硬件设备上的音频兼容性问题。

4. 网络子系统（Network Subsystem）： 优点：

- 提供了网络通信和多人游戏的支持，使得玩家可以进行在线游戏和实时多人互动。
- 可以通过多种网络协议实现低延迟、高效率的数据传输和同步。

缺点：

- 在线功能通常需要与服务器进行交互，需要考虑服务器端架设和维护的成本。
- 处理网络延迟和同步问题可能需要更复杂的设计和调试。

C++进程的调用？和 C++线程的调用？

1. 进程的调用：

- 在 C++中，可以使用系统调用（如 `fork()` 和 `exec()`）来创建和执行新的进程。
- `fork()` 系统调用可以创建一个与当前进程相同的子进程，子进程将复制父进程的所有代码段、数据段和堆栈等资源。
- 在子进程中，可以使用 `exec()` 系列函数加载新的可执行程序，替换当前进程的映像，并开始执行新的程序。
- 通过系统调用，可以实现进程间的通信（如管道、消息队列、共享内存等）来进行进程间的数据交换和同步。

2. 线程的调用：

- 在 C++中，可以使用多种库来创建和管理线程，例如标准库的 `std::thread`、POSIX 线程库（pthread）等。
- 使用标准库的 `std::thread`，可以创建一个新的线程并指定要执行的函数或函数对象作为线程的入口点。
- 在线程中，可以执行各种操作，包括计算、IO 操作和同步等。
- 可以使用线程间的同步机制（如互斥锁、条件变量和信号量）来实现线程间的协调和数据共享。
- 可以通过线程库提供的函数来控制线程的创建、终止、等待和管理。

需要注意的是，进程和线程之间有一些重要的区别：

- 进程是独立的执行实体，每个进程都有自己的地址空间和资源，它们相互之间隔离。
- 线程是在进程内创建的可执行实体，共享进程的地址空间和资源。线程可以访问同一进程的数据。