C#基础

**C#基础**

**一、面向对象三大特新及其作用**

**1.封装（Encapsulation）：**

封装是面向对象编程的基本特性之一，它将数据（属性）和操作（方法）捆绑在一起，形成一个类，同时限制了外部对于类内部数据和操作的直接访问。通过封装，我们可以隐藏对象的内部状态，只暴露必要的接口给外部使用。

作用：

保护了对象的内部状态，防止外部直接修改。

提高了代码的安全性和可靠性，减少了意外的修改。

通过暴露必要的接口，简化了使用者的操作。

实例：

using System;

public class Car

{

// 私有字段，外部无法直接访问

private string model;

// 公有属性，通过属性访问私有字段

public string Model

{

get { return model; }

set { model = value; }

}

// 公有方法

public void StartEngine()

{

Console.WriteLine("Engine started for {0}.", model);

}

}

class Program

{

static void Main(string[] args)

{

Car myCar = new Car();

myCar.Model = "Toyota";

myCar.StartEngine(); // 输出: Engine started for Toyota.

}

}

**2.继承（Inheritance）：**

继承是指一个类可以派生出子类，子类会继承父类的属性和方法，同时可以在子类中添加新的属性和方法，或者重写父类的方法。

作用：

提高了代码的可重用性，避免了重复编写相似的代码。

通过建立类之间的层次关系，使得代码更易于组织和管理。

提高了代码的可扩展性，可以通过添加新的子类来扩展现有的功能。

using System;

public class Animal

{

public void Eat()

{

Console.WriteLine("Animal is eating.");

}

}

public class Dog : Animal

{

public void Bark()

{

Console.WriteLine("Dog is barking.");

}

}

class Program

{

static void Main(string[] args)

{

Dog myDog = new Dog();

myDog.Eat(); // 输出: Animal is eating.

myDog.Bark(); // 输出: Dog is barking.

}

}

**3.多态（Polymorphism）：**

多态是指同一个方法调用可以在不同的对象上产生不同的行为。它允许不同的对象对同一消息做出不同的响应。

作用：

提高了代码的灵活性和可扩展性，允许在不修改现有代码的情况下扩展程序的功能。

通过统一的接口访问不同类型的对象，降低了代码的耦合度，提高了代码的可维护性。

使得代码更容易理解和重用，提高了代码的可读性和可维护性。

using System;

public class Animal

{

public virtual void Sound()

{

Console.WriteLine("Animal makes a sound.");

}

}

public class Dog : Animal

{

public override void Sound()

{

Console.WriteLine("Dog barks.");

}

}

public class Cat : Animal

{

public override void Sound()

{

Console.WriteLine("Cat meows.");

}

}

class Program

{

static void Main(string[] args)

{

Animal myAnimal = new Animal();

Animal myDog = new Dog();

Animal myCat = new Cat();

myAnimal.Sound(); // 输出: Animal makes a sound.

myDog.Sound(); // 输出: Dog barks.

myCat.Sound(); // 输出: Cat meows.

}

}

**二、C#值类型和引用类型的区别**

在 C# 中，变量可以是值类型或引用类型，它们之间有一些重要的区别：

存储方式：

值类型：值类型的变量直接包含其数据，存储在栈上。每当你声明一个值类型的变量，系统就会为该变量分配内存空间。

引用类型：引用类型的变量存储的是对象的引用（内存地址），而不是对象本身。对象通常存储在堆上，而变量存储的是指向堆上对象的引用。当你创建一个引用类型的变量时，只会为该变量分配引用的内存空间，对象本身则存储在堆上。

传递方式：

值类型：值类型的变量在传递给函数或赋值给另一个变量时，会将其值复制一份，这意味着原始变量和目标变量是完全独立的，修改其中一个不会影响另一个。

引用类型：引用类型的变量在传递给函数或赋值给另一个变量时，实际上传递的是对象的引用，而不是对象本身。因此，原始变量和目标变量都指向相同的对象，对其中一个的修改会影响到另一个。

内存管理：

值类型：值类型的变量的生命周期与其所在的作用域相关联。当超出作用域时，变量将被销毁，释放其占用的内存。

引用类型：引用类型的变量的生命周期不仅取决于其自身的作用域，还取决于对象的引用计数或垃圾回收器的回收机制。只有当没有任何引用指向对象时，对象才会被回收。

以下是一个简单示例来说明值类型和引用类型的区别：

using System;

class Program

{

static void Main(string[] args)

{

// 值类型示例

int x = 10;

int y = x; // 值被复制到 y 中

y = 20; // 修改 y 不会影响 x

Console.WriteLine("x: " + x); // 输出: x: 10

// 引用类型示例

int[] arr1 = { 1, 2, 3 };

int[] arr2 = arr1; // 引用被复制到 arr2 中

arr2[0] = 100; // 修改 arr2 也会影响 arr1

Console.WriteLine("arr1[0]: " + arr1[0]); // 输出: arr1[0]: 100

}

}

在这个示例中，修改值类型变量 y 不会影响原始变量 x 的值，但修改引用类型变量 arr2 中的数组元素会影响原始变量 arr1 中的数组元素。

**三、ArrayList 和 List 的区别**

ArrayList 和 List 是两种不同的集合类型，它们有一些重要的区别：

命名空间：

ArrayList 属于 System.Collections 命名空间，而 List 属于 System.Collections.Generic 命名空间。

泛型支持：

ArrayList 是非泛型集合，它可以存储任意类型的对象，但在添加和检索元素时需要进行类型转换，因为它存储的是 object 类型。

List 是泛型集合，你可以在创建时指定存储的元素类型，这样在添加和检索元素时不需要进行类型转换。

类型安全性：

由于 ArrayList 存储的是 object 类型，所以在添加和检索元素时可能会出现类型转换错误的问题，这降低了代码的类型安全性。

List 使用泛型，所以在编译时就能够检测到类型错误，提高了代码的类型安全性。

性能：

由于 ArrayList 存储的是 object 类型，所以在添加和检索元素时需要进行装箱（Boxing）和拆箱（Unboxing）操作，这会带来一定的性能损失。

List 使用泛型，避免了装箱和拆箱操作，因此在性能上要优于 ArrayList。

综上所述，List 是 ArrayList 的一个更加现代化和类型安全的替代品，在大多数情况下应该优先使用 List，除非需要与遗留代码进行交互或者需要存储不同类型的对象。

**五、接口和抽象类的区别**

接口（Interface）和抽象类（Abstract Class）是面向对象编程中的两种重要的概念，它们有一些区别：

定义：

接口是一种完全抽象的类型，它只定义了成员的签名，但没有提供成员的实现。接口定义了对象应该具有的方法、属性和事件。

抽象类是一种包含了抽象成员（方法、属性、索引器、事件）的类，它可以包含抽象成员和非抽象成员的实现。抽象类可以包含方法的实现，也可以包含字段和构造函数等。

多继承：

接口支持多继承，一个类可以实现多个接口。

抽象类不支持多继承，一个类只能继承一个抽象类。

成员实现：

接口中的成员没有实现，需要实现接口的类提供具体的实现。

抽象类中的成员可以有具体的实现，也可以是抽象的，需要在子类中进行实现。

构造函数：

接口不能包含构造函数。

抽象类可以包含构造函数。

字段：

接口中不能包含字段。

抽象类中可以包含字段。

访问修饰符：

接口中的成员默认是公共的，不能包含访问修饰符。

抽象类中的成员可以有各种访问修饰符，包括公共、私有、受保护和受内部保护等。

用途：

接口通常用于定义对象的行为，描述对象应该具有的方法、属性和事件。

抽象类通常用于建模一个类的抽象概念，提供一些通用的功能，同时定义一些抽象方法让子类去实现。

综上所述，接口和抽象类都是用于实现多态性和代码重用的重要机制，它们各有特点和适用场景。一般来说，如果你需要定义一个对象的行为或者实现一种规范，应该使用接口；如果你需要定义一个类的抽象概念，并且希望提供一些通用的功能和一些需要子类实现的方法，应该使用抽象类。

**六、反射的实现原理**

反射（Reflection）是指在运行时动态地获取类型信息、访问和操作对象的属性、方法和事件等成员的能力。它允许程序在运行时探查和操作自身的结构，而不需要提前知道类型的详细信息。反射的实现原理涉及到.NET Framework 中的元数据和反射 API。

在.NET Framework中，每个程序集（Assembly）都包含了元数据（Metadata），元数据包含了程序集中定义的类型、成员、方法签名、属性等信息。这些信息是在编译时由编译器生成并嵌入到程序集中的，它描述了程序集中所有类型的结构和特性。

反射的实现原理可以简述为以下几个步骤：

加载程序集：首先，CLR（Common Language Runtime）加载程序集到内存中，然后对程序集中的元数据进行解析。

获取类型信息：通过反射 API，可以获取程序集中定义的类型信息，如类、接口、枚举等。CLR 会根据元数据提供的信息来创建 Type 对象，表示程序集中的类型。

获取成员信息：通过 Type 对象，可以获取类型中定义的成员信息，如字段、属性、方法、事件等。CLR会根据元数据提供的信息来创建 MemberInfo 对象，表示类型中的成员。

创建对象实例：通过反射，可以在运行时动态创建对象的实例。CLR根据元数据中的类型信息，使用 Activator 类或者 ConstructorInfo 类来创建对象的实例。

调用方法和访问属性：通过反射，可以在运行时动态调用对象的方法和访问对象的属性。CLR根据元数据中的方法和属性信息，使用 MethodInfo 和 PropertyInfo 等类来调用方法和访问属性。

处理特性：通过反射，可以在运行时动态获取和处理类型、成员和方法等上定义的特性。CLR会根据元数据中的特性信息，使用 Attribute 类来获取和处理特性。

总之，反射的实现原理基于CLR加载程序集和解析元数据的机制，通过反射 API来获取和操作程序集中的类型、成员和方法等信息，实现了在运行时动态地探查和操作对象的能力。反射在许多场景中都非常有用，比如动态加载程序集、动态创建对象、实现插件架构等

示例代码：

using System;

using System.Reflection;

// 定义一个简单的类

public class MyClass

{

public string Name { get; set; }

public int Age { get; set; }

public MyClass()

{

Name = "John";

Age = 30;

}

public void SayHello()

{

Console.WriteLine("Hello, my name is " + Name + " and I'm " + Age + " years old.");

}

}

class Program

{

static void Main(string[] args)

{

// 获取MyClass的Type对象

Type type = typeof(MyClass);

// 使用Type对象创建类的实例

object myClassInstance = Activator.CreateInstance(type);

// 获取Name属性的信息

PropertyInfo nameProperty = type.GetProperty("Name");

// 设置Name属性的值

nameProperty.SetValue(myClassInstance, "Alice");

// 获取Age属性的信息

PropertyInfo ageProperty = type.GetProperty("Age");

// 设置Age属性的值

ageProperty.SetValue(myClassInstance, 25);

// 调用SayHello方法

MethodInfo sayHelloMethod = type.GetMethod("SayHello");

sayHelloMethod.Invoke(myClassInstance, null);

}

}

在 C# 中，string、StringBuilder 和 StringBuffer 都用于处理字符串，但它们有一些区别：

string：

string 是不可变的字符串类型，一旦创建就不能修改。

在 .NET 中，string 类型是一个引用类型，表示一个不可变的字符序列。

string 类型适合于处理不经常变化的字符串，例如文本常量、配置信息等。

StringBuilder：

StringBuilder 是可变的字符串类型，用于频繁进行字符串拼接和修改的场景。

StringBuilder 类提供了一系列的方法来添加、删除、替换和插入字符串等操作，且操作后不会创建新的字符串对象，而是在内部的字符缓冲区中进行操作。

StringBuilder 类的性能在字符串拼接和修改时通常比直接使用 string 类型更好，特别是在需要频繁操作大量字符串时。

StringBuffer：

StringBuffer 是 Java 中的一个类，用于处理字符串。然而，在 C# 中并不存在 StringBuffer 类，而是使用 StringBuilder 来实现类似的功能。

StringBuilder 在 C# 中是用于动态修改字符串的首选类，与 Java 中的 StringBuffer 功能类似。

综上所述，string 适合于不可变的字符串，StringBuilder 适合于频繁进行字符串拼接和修改的场景，而 StringBuffer 在 C# 中没有直接对应的类，通常可以使用 StringBuilder 来实现类似的功能。

**八、Linklist和List的区别**

在 C# 中，List<T> 是动态数组的实现，而 LinkedList<T> 是双向链表的实现。它们之间的主要区别如下：

底层数据结构：

List<T>：底层实现是基于数组的动态数组，通过数组来存储元素。

LinkedList<T>：底层实现是双向链表，每个节点包含指向前一个节点和后一个节点的指针。

插入和删除操作的性能：

List<T>：在列表的中间插入或删除元素时，需要移动其他元素，时间复杂度为 O(n)；而在列表的末尾进行插入或删除操作的性能较好，时间复杂度为 O(1)。

LinkedList<T>：由于链表的特性，插入和删除操作的性能不受元素位置的影响，时间复杂度为 O(1)，即使是在列表的中间进行插入或删除操作也是如此。

随机访问的性能：

List<T>：由于底层实现是数组，因此可以通过索引进行快速的随机访问，时间复杂度为 O(1)。

LinkedList<T>：由于链表的特性，无法通过索引进行直接的随机访问，只能通过迭代器进行顺序访问，时间复杂度为 O(n)。

内存占用：

List<T>：由于底层是数组，因此在一定程度上会占用连续的内存空间，数组的大小由容量决定。

LinkedList<T>：由于底层是链表，每个元素都包含了额外的指针，因此可能会占用更多的内存空间，但不受容量限制。

综上所述，List<T> 适合于需要随机访问和频繁在末尾进行插入和删除操作的场景，而 LinkedList<T> 适合于频繁进行插入和删除操作的场景，尤其是在列表的中间位置。选择合适的容器类取决于具体的需求和场景。

1. **Delegate，Event，Action，Func的区别和联系**

Delegate：

Delegate是一种类型，用于引用方法。它允许您将方法作为参数传递给其他方法或存储对方法的引用。

Delegate允许声明具有特定签名的方法，并允许将这些方法作为参数传递给其他方法或存储到委托实例中。

Delegate可以定义多个方法的调用列表，并且可以将方法添加到调用列表中，以便在调用委托时执行所有方法。

Event：

event 是 C# 中的关键字，用于声明事件。

event是一种特殊的语言结构，它提供了更严格的封装，只允许定义了事件的类来触发事件，其他类只能订阅事件（添加或移除事件处理程序）。

event通常用于实现观察者模式（Observer Pattern），允许对象注册为事件的订阅者，以便在事件发生时接收通知。

示例：public event EventHandler MyEvent，其中 EventHandler 是一个预定义的委托类型，用于引用一个接受两个参数（sender 和 EventArgs）且无返回值的方法。

Action：

Action 是一个泛型委托类型，用于引用无返回值的方法。

Action 委托类型可以接受多个不同类型的参数（最多可以有 16 个参数），但不返回任何值。

Action 通常用于声明委托变量或作为方法的参数。

Func：

Func 是一个泛型委托类型，用于引用具有指定返回类型的方法。

Func 委托类型可以接受多个不同类型的参数（最多可以有 16 个参数），并且必须返回指定类型的值。

Func 常用于声明委托变量或作为方法的参数，特别是在需要引用具有返回值的方法时。

using System;

// 声明一个委托类型

public delegate void MyDelegate(string message);

// 定义一个包含事件的类

public class MyClass

{

// 声明一个事件

public event MyDelegate MyEvent;

// 触发事件的方法

public void RaiseEvent(string message)

{

Console.WriteLine("Event raised: " + message);

// 触发事件

MyEvent?.Invoke(message);

}

}

class Program

{

// 定义一个方法，作为委托的目标

public static void MyMethod(string message)

{

Console.WriteLine("Method called: " + message);

}

static void Main(string[] args)

{

// 创建 MyClass 实例

MyClass myClass = new MyClass();

// 创建委托实例，并将方法添加到调用列表中

MyDelegate myDelegate = new MyDelegate(MyMethod);

// 将委托实例添加到事件的调用列表中

myClass.MyEvent += myDelegate;

// 使用 Action 委托来触发事件

Action<string> action = myClass.RaiseEvent;

action("Action");

// 使用 Func 委托来触发事件

Func<string, string> func = s => s.ToUpper();

myClass.MyEvent += s => Console.WriteLine("Func called: " + func(s));

myClass.RaiseEvent("Func");

}

}

**十、JIT和AOT的区别**

JIT（Just-In-Time Compilation）和AOT（Ahead-Of-Time Compilation）都是编译技术，用于将源代码转换为机器代码以便执行。它们之间的主要区别在于编译的时机和方式：

JIT：

JIT 编译器在运行时（即程序执行期间）将源代码或中间代码编译成本地机器代码。

当程序需要执行某个方法时，JIT 编译器会将该方法的字节码或中间语言编译成本地机器代码，并且通常只编译执行路径上的代码。

JIT 编译器可以在运行时对程序进行优化，根据实际的运行环境和执行情况来生成更有效的机器代码。

AOT：

AOT 编译器在程序运行之前将源代码或中间代码编译成本地机器代码。

编译过程通常在程序部署或安装时进行，而不是在程序执行期间。

AOT 编译器生成的机器代码在程序运行时不需要再进行编译，因此可以提高程序的启动速度和运行时性能。

AOT 编译器通常会生成一份完整的机器代码，因此不需要在运行时执行编译过程。

综上所述，JIT 编译器在程序运行时将源代码或中间代码编译成机器代码，而 AOT 编译器在程序部署时就将代码编译成机器代码。JIT 编译器的优势在于可以根据实际运行情况进行优化，但可能会增加程序启动时间；而 AOT 编译器则可以提高程序的启动速度和运行时性能，但不具备 JIT 编译器的动态优化能力。

**十一、C#委托是什么有什么用处**

在 C# 中，委托（Delegate）是一种类型，它可以用于引用方法。委托可以让您将方法作为参数传递给其他方法，或者将方法存储为变量，并且在需要时调用这些方法。委托通常用于以下几个方面：

回调函数：委托可以用作回调函数，允许您将一个方法传递给另一个方法，在特定的时机调用。这在异步编程、事件处理和 GUI 编程中特别有用。

事件处理：事件是委托的特殊用法，允许对象在特定的条件下通知其他对象发生了某种状态或行为。事件处理程序（Event Handlers）就是委托，它们会在事件触发时被调用。

动态方法调用：委托允许您在运行时选择要调用的方法，这为一些动态行为提供了便利，例如插件架构或动态加载程序集。

多播委托：委托可以组合多个方法，使得可以一次调用多个方法。多播委托在事件处理、命令模式和观察者模式等场景中非常有用。

局部函数：C# 7.0 开始支持在方法内部定义局部函数，局部函数实际上是编译器为该方法生成的一个委托。

总的来说，委托为 C# 提供了一种方便的方式来处理方法的引用和调用，使得程序具有更高的灵活性和可扩展性。通过委托，您可以更轻松地实现回调、事件处理和动态方法调用等功能，从而使代码更加模块化和可维护。

**十二、foreach和for遍历的区别**

在 C# 中，foreach 和 for 是两种常用的循环语句，用于遍历集合或数组等数据结构。它们之间的区别主要体现在以下几个方面：

语法：

foreach 是一种迭代语句，其语法为 foreach (var item in collection)，其中 item 是迭代过程中的当前元素，collection 是要遍历的集合或数组。

for 是一种通用的循环语句，其语法为 for (initialization; condition; iteration)，其中 initialization 初始化循环变量，condition 是循环条件，iteration 是每次循环迭代时执行的操作。

适用范围：

foreach 通常用于遍历集合、数组等实现了 IEnumerable 接口的数据结构。它适用于遍历整个集合，并且更简洁和易读。

for 可以用于各种循环场景，包括遍历数组、执行指定次数的循环等。它更灵活，可以控制循环的开始、结束和迭代条件。

遍历过程：

foreach 在遍历集合时，每次迭代都会自动获取集合的下一个元素，并将其赋值给迭代变量。迭代过程对于调用者是透明的。

for 则需要手动管理循环变量的更新，包括初始化、条件判断和迭代操作。这使得 for 循环更加灵活，可以自定义迭代逻辑。

总的来说，foreach 更适用于遍历集合或数组等实现了 IEnumerable 接口的数据结构，它更简洁、易读，并且自动处理迭代过程；而 for 循环更灵活，适用于各种循环场景，包括遍历数组、执行指定次数的循环等，但需要手动管理循环变量的更新。

**十三、字典Dictionary的内部实现原理**

在 C# 中，Dictionary<TKey, TValue> 是一种常用的集合类型，用于存储键值对。其内部实现基于哈希表（Hash Table），具体来说，它是使用哈希表和链表（或红黑树）来实现的。

下面是 Dictionary<TKey, TValue> 的主要内部实现原理：

哈希表：

Dictionary<TKey, TValue> 内部使用了一个哈希表来存储键值对。哈希表是一种高效的数据结构，通过哈希函数将键映射到一个索引位置，从而实现快速的查找和插入操作。

哈希表通常是一个数组，每个数组元素称为一个桶（Bucket），每个桶可以存储一个或多个键值对。索引位置由键的哈希码（Hash Code）决定。

哈希冲突处理：

由于哈希函数的映射不是唯一的，可能会导致不同的键映射到相同的索引位置，这就产生了哈希冲突。

Dictionary<TKey, TValue> 内部采用了开放定址法（Open Addressing）和链地址法（Chaining）等方法来处理哈希冲突。

开放定址法：在哈希冲突发生时，尝试寻找下一个空闲的桶来存储键值对。

链地址法：在哈希冲突发生时，将键值对存储在桶中的链表（或红黑树）中，使得同一索引位置可以存储多个键值对。

扩容与重新哈希：

当哈希表中的元素数量达到一定阈值时，Dictionary<TKey, TValue> 会进行扩容操作，即增加哈希表的大小，并重新计算每个键的索引位置。

扩容过程需要重新计算每个键的哈希码，并将键值对重新分配到新的桶中。这个过程称为重新哈希。

性能：

由于哈希表具有快速的查找和插入操作，因此 Dictionary<TKey, TValue> 具有很高的性能，平均情况下的时间复杂度为 O(1)。

但在最坏情况下，即哈希冲突严重时，查找和插入操作的时间复杂度可能会退化为 O(n)。

综上所述，Dictionary<TKey, TValue> 的内部实现基于哈希表，通过哈希函数将键映射到索引位置，并采用开放定址法和链地址法来处理哈希冲突。这种实现方式保证了 Dictionary<TKey, TValue> 具有快速的查找和插入操作，并且具有较高的性能。

**十四、int?和int有什么区别?**

int? 和 int 之间的区别主要在于是否可以表示 null 值。

1. int

类型: int 是一个值类型，表示一个32位的有符号整数。

默认值: 如果未赋值，默认值为 0。

不能为 null: int 不能表示 null，因为它是一个非空值类型。

2. int?

类型: int? 是 Nullable<int> 的简写。Nullable<T> 是一个结构体，它可以包装一个值类型，使其可以表示 null 值。

默认值: int? 的默认值为 null，除非显式赋值为其他值。

可以为 null: int? 可以表示 null，这意味着它可以存储一个 int 类型的值或没有值（null）。

3. 使用场景

int: 使用在一定不需要表示空值的场景下，例如数学运算和计算中，使用 int 更直接和高效。

int?: 使用在可能需要表示未设置或空值的场景下。例如，在数据库操作中，某些整型字段可能允许 null，因此在程序中使用 int? 来表示这些字段。

4. 示例

int? nullableInt = null;

int nonNullableInt = 0;

// 可以将一个非空的 int 值赋给 int?

nullableInt = 5;

// 不能直接将 null 赋值给 int

// nonNullableInt = null; // 这会导致编译错误

// 可以通过以下方式从 int? 获取值，考虑 null 的情况

if (nullableInt.HasValue) {

nonNullableInt = nullableInt.Value;

} else {

nonNullableInt = 0; // 或者使用其他默认值

}

int? nullableInt = null;

int nonNullableInt = 0;

// 可以将一个非空的 int 值赋给 int?

nullableInt = 5;

// 不能直接将 null 赋值给 int

// nonNullableInt = null; // 这会导致编译错误

// 可以通过以下方式从 int? 获取值，考虑 null 的情况

if (nullableInt.HasValue) {

nonNullableInt = nullableInt.Value;

} else {

nonNullableInt = 0; // 或者使用其他默

}

**十五：结构体和类的区别**

[结构体](https://so.csdn.net/so/search?q=%E7%BB%93%E6%9E%84%E4%BD%93&spm=1001.2101.3001.7020" \t "https://blog.csdn.net/weixin_44231544/article/details/_blank)是一种值类型，而类是引用类型。那么结构体就是当成值来使用的，类则通过引用来对实际数据操作

**十六、Heap与Stack有何区别？**

1.heap是堆，stack是栈。

2.stack的空间由操作系统自动分配和释放，

heap的空间是手动申请和释放的，heap常用new关键字来分配。

1. stack空间有限，heap的空间是很大的自由区。
2. **简述拆箱和装箱**

装箱 就是把“值类型”转换成“引用类型”(Object)；

拆箱 就是把“引用类型”转换成“值类型”；

1. **ref参数和out参数是什么？有什么区别？**

ref引用参数和out输出参数的效果一样，都是通过关键字找到内存地址，

不同点就是输出参数必须对参数进行初始化。

1. ****C#中的排序方式有哪些？****

选择排序，冒泡排序，快速排序，插入排序，希尔排序，归并排序

代理就是用来定义指向方法的引用。

1. ****概述c#中代理和事件？****

代理就是用来定义指向方法的引用。

C＃事件本质就是对消息的封装，用作对象之间的通信；发送方叫事件发送器，接收方叫事件接收器

1. ****在类的构造函数前加上static会报什么错?为什么?****

构造函数格式为 public+类名，如果加上static会报错（静态构造函数不能有访问修饰符）

原因：静态构造函数不允许访问修饰符，也不接受任何参数；

无论创建多少类型的对象，静态构造函数只执行一次；

运行库创建类实例或者首次访问静态成员之前，运行库调用静态构造函数；

静态构造函数执行先于任何实例级别的构造函数；

显然也就无法使用this和base来调用构造函数。

1. ****数列1,1,2,3,5,8,13…第n位数是多少?用C#递归算法实现****

等差数列公式（斐波那契数列）

通项公式应该是an = a(n-1) + a(n-2) 这样就可以做递归了

**Unity基础**

**一、四元数和欧拉角的优缺点**

四元数（Quaternions）和欧拉角（Euler Angles）都是用于表示物体旋转的数学概念，它们各有优缺点：

四元数（Quaternions）：

优点：

无万向节锁（No Gimbal Lock）：四元数没有欧拉角那样的万向节锁，这意味着它们可以避免在旋转过程中丧失自由度和出现死锁的情况。

平滑插值（Smooth Interpolation）：在进行旋转插值时，四元数通常比欧拉角产生更平滑的结果。这对于动画和游戏中的物体运动非常重要。

节省内存（Memory Efficient）：四元数通常比欧拉角占用更少的内存空间。

缺点：

可读性差（Poor Human Readability）：四元数的表示方式对于人类来说不太直观，难以直接理解。这使得在调试和调整旋转时可能会更具挑战性。

复杂性（Complexity）：尽管四元数是更高维度的数学概念，但对于初学者来说，理解和操作它们可能需要更多的学习和实践。

欧拉角（Euler Angles）：

优点：

易于理解（Easy to Understand）：欧拉角的表示方式更加直观，容易被人类理解。通常使用旋转顺序（如XYZ、YXZ等）来描述物体的旋转方向。

可读性好（Readable）：由于其直观性，欧拉角通常更容易在代码中进行调试和调整。

缺点：

万向节锁（Gimbal Lock）：欧拉角存在万向节锁，在某些情况下，当一个轴的旋转角度接近90度时，会丧失一个自由度，导致旋转的限制和不稳定性。

旋转插值的困难（Interpolation Difficulty）：在欧拉角表示中，进行旋转插值可能会导致不可预测的结果，因为插值过程可能会导致旋转顺序发生变化，从而产生奇异性。

综上所述，四元数通常在游戏引擎和动画中更常用，因为它们解决了欧拉角的奇点问题和插值困难，但它们的可读性和理解难度相对较高。欧拉角则更适合简单的旋转需求和易于理解的场景，但要注意避免奇点问题。在实际应用中，根据具体情况选择合适的表示方法非常重要。

1. **万向节锁是如何导致的**

Gimbal Lock（万向节锁）是指在使用欧拉角表示物体旋转时可能发生的一种现象，导致某一轴的旋转自由度被丧失，从而限制了物体的旋转。这种情况通常发生在欧拉角旋转中的两个旋转轴（通常是相邻的两个轴）趋于平行时。

具体来说，欧拉角表示物体的旋转通常涉及三个轴：通常是绕着X轴（Roll）、Y轴（Yaw）、Z轴（Pitch）旋转。当物体的两个旋转轴趋于平行时，就可能会发生Gimbal Lock。例如，如果先绕Y轴旋转一定角度，然后绕X轴旋转，最后再绕Y轴旋转。如果在绕X轴旋转时，Y轴和Z轴趋于平行，那么在最后一次绕Y轴旋转时，就会发生Gimbal Lock。

在Gimbal Lock发生时，物体的旋转自由度将受到限制，因为两个旋转轴之间的关系变得相对固定。例如，如果发生Gimbal Lock，那么对于某个旋转轴的旋转会同时影响到另外两个轴，从而导致物体的旋转表现不再如预期那样。

Gimbal Lock是欧拉角表示的一个局限性，因为它使得旋转变得不稳定，难以精确控制。这也是为什么在某些情况下，特别是在需要高精度旋转的情况下，使用四元数等其他旋转表示方法更为合适的原因之一。

**三、Unity里旋转的顺序是什么**

在Unity中，物体的旋转顺序默认是ZYX轴顺序。这意味着物体首先绕Z轴（Roll）旋转，然后绕Y轴（Yaw）旋转，最后绕X轴（Pitch）旋转。

这种旋转顺序的意义在于物体的旋转操作将按照指定的顺序依次应用于物体的局部坐标系。例如，如果你对一个物体进行了旋转操作，首先绕Z轴旋转一定角度，然后绕Y轴旋转一定角度，最后绕X轴旋转一定角度，那么这三次旋转操作将按照ZYX的顺序依次应用于物体的局部坐标系，而不是全局坐标系。

这种旋转顺序是Unity默认的，但你也可以通过编程来更改旋转顺序。Unity的Transform组件提供了Rotate方法，可以根据自定义的旋转顺序来旋转物体。例如，可以使用Rotate(Vector3 eulerAngles, Space relativeTo)方法来指定旋转角度和旋转的坐标系。

**四、动态加载资源的几种方式**

在Unity中，可以使用多种方式实现动态加载资源，以下是几种常见的方式：

Resources.Load：

使用Resources.Load函数可以从Resources文件夹中加载资源。将资源放置在Assets文件夹下的Resources文件夹内，然后使用Resources.Load按照资源的路径加载资源。

例如：GameObject prefab = Resources.Load<GameObject>("Prefabs/MyPrefab");

AssetBundle：

AssetBundle 是一种打包和加载资源的方式，可以将一组资源打包成一个单独的文件，并在运行时动态加载。

使用AssetBundle.LoadFromFile、AssetBundle.LoadFromMemory等方法加载AssetBundle文件，然后通过加载的AssetBundle对象获取其中的资源。

详细的使用方法可以参考Unity官方文档。

Addressable Assets：

Addressable Assets System 是 Unity 提供的一种用于管理和加载资源的系统，可以在运行时动态加载资源，并支持更灵活的资源管理和加载方式。

使用 Addressable Assets System 可以根据资源的地址（Address）来加载资源，而不需要直接引用资源的路径。

这是一种更现代化和灵活的资源管理方式，适用于大型项目和需要动态加载大量资源的场景。

WebRequest：

可以使用 Unity 的WebRequest来从远程服务器下载资源，然后在运行时加载资源。

使用 UnityWebRequest 类可以方便地与网络进行交互，下载远程资源，并在下载完成后加载资源。

以上是几种常见的动态加载资源的方式，在实际开发中可以根据项目需求和场景选择合适的方式进行资源加载。

**五、MonoBehaviour生命周期**

MonoBehaviour 是 Unity 中最常用的脚本类之一，它负责管理游戏对象的行为和逻辑。MonoBehaviour 具有一系列生命周期方法，这些方法会在不同阶段调用，以便你在开发中添加自定义的行为和逻辑。以下是 MonoBehaviour 常用的生命周期方法：

Awake：

当脚本实例被加载时立即调用，通常用于初始化操作，例如查找组件、初始化变量等。这个方法在对象被激活之前调用。

OnEnable：

当脚本实例被激活时调用，通常用于在对象被激活时执行的初始化操作。

Start：

在 Awake 方法之后，在第一帧 Update 方法之前调用，通常用于初始化操作，例如设置初始状态、订阅事件等。

FixedUpdate：

每个固定的物理帧都会调用，通常用于物理相关的逻辑，例如移动、碰撞检测等。

Update：

每一帧都会调用，通常用于更新游戏逻辑和处理用户输入。

LateUpdate：

在所有 Update 方法调用之后被调用，通常用于处理其他对象的状态更新或处理。

OnDisable：

当脚本实例被禁用时调用，通常用于在对象被禁用时执行的清理操作。

OnDestroy：

当脚本实例被销毁时调用，通常用于清理操作，例如释放资源、取消订阅事件等。

此外，还有一些其他生命周期方法，例如 OnTriggerEnter、OnCollisionEnter 等，用于处理碰撞事件。

这些生命周期方法可以让你在游戏对象的不同阶段执行自定义的逻辑和行为，帮助你更好地管理游戏对象的生命周期。

**六、游戏动画有几种，分别是什么原理**

游戏动画有多种形式，主要包括关键帧动画、骨骼动画、物理动画和过程生成动画等。下面简要介绍每种动画的原理：

关键帧动画：

关键帧动画是通过定义关键帧（Keyframe）来描述动画的变化过程。在每个关键帧上，指定了物体的位置、旋转、缩放等属性。

动画系统会在关键帧之间进行插值，以生成平滑的动画过渡效果。

原理简单，易于制作，但通常需要手动创建和编辑关键帧。

骨骼动画：

骨骼动画是通过对模型的骨骼结构进行动画化，实现模型的变换和变形。

模型的骨骼结构由一系列骨骼（Bone）组成，每个骨骼都有自己的位置、旋转和缩放属性。

动画系统根据骨骼的运动轨迹计算出每个骨骼在每一帧的变换矩阵，然后应用到模型上。

骨骼动画适用于人物、动物等具有骨骼结构的模型，可以实现更加复杂和自然的动作效果。

物理动画：

物理动画利用物理引擎来模拟物体的运动和碰撞，实现更真实的动画效果。

物理引擎会根据物体的质量、碰撞体、力和约束等属性来模拟物体的运动状态，例如重力、碰撞、惯性等。

物理动画通常用于模拟物体的自然运动，如布料、发丝、碎片等，以及角色的重力、碰撞等效果。

过程生成动画：

过程生成动画是通过算法来生成动画效果，而不是基于预先录制的动画数据。

可以利用程序生成的方式来创建复杂的动画效果，例如程序生成的行走动画、流体动画等。

过程生成动画具有高度的灵活性和可扩展性，可以根据需要实时生成动画效果，适用于一些需要动态生成的场景和效果。

以上是常见的游戏动画形式及其原理，不同类型的动画在游戏开发中会根据实际需求和场景选择合适的方式来实现。

**七、矩阵相乘的意义及注意点**

矩阵相乘在计算机图形学和计算机科学中有着重要的意义，它的主要作用包括：

变换组合：在图形学中，矩阵相乘可以用于组合多个变换操作，例如平移、旋转、缩放等。将多个变换矩阵相乘可以得到一个等效的变换矩阵，从而将多个变换操作合并为一个单一的变换。

坐标变换：矩阵相乘可以用于实现坐标系之间的转换，例如从对象空间到世界空间、从世界空间到相机空间、从相机空间到裁剪空间等。

图形渲染：在图形渲染过程中，矩阵相乘用于将顶点的位置和属性从对象空间变换到屏幕空间，从而实现图形的投影和显示。

线性变换：矩阵相乘是线性代数中的基本操作，可以用于解决线性方程组、求解特征值和特征向量等问题。

在进行矩阵相乘时，需要注意以下几点：

矩阵维度匹配：两个矩阵相乘时，左边矩阵的列数必须等于右边矩阵的行数，否则无法进行相乘操作。

乘法顺序：矩阵相乘不满足交换律，即 A \* B 不一定等于 B \* A。因此，乘法顺序是非常重要的，需要根据具体的需求和应用场景来确定矩阵的顺序。

结果维度：两个矩阵相乘的结果矩阵的维度由左边矩阵的行数和右边矩阵的列数决定。例如，如果 A 是一个 m × n 的矩阵，B 是一个 n × p 的矩阵，则 A \* B 的结果是一个 m × p 的矩阵。

矩阵结合律：矩阵相乘满足结合律，即 (A \* B) \* C = A \* (B \* C)，因此可以根据需要将多个矩阵相乘的顺序进行组合。

单位矩阵：单位矩阵是矩阵相乘中的乘法单位元素，任何矩阵与单位矩阵相乘都等于它本身。

**八、协程是什么**

在C#中，协程（Coroutine）通常指的是通过迭代器（Iterator）和yield关键字来实现的一种编程技术，称为迭代器协程或生成器协程。这种协程技术允许在一个方法中暂停执行并返回一个中间结果，然后在需要时继续执行，从而实现异步、延迟加载等功能。

在C#中，迭代器方法是通过在方法声明前面加上yield return或yield break关键字来定义的。当调用这个方法时，它并不会立即执行，而是返回一个迭代器对象，只有在遍历迭代器对象时，才会执行方法中的代码，直到遇到yield return或yield break语句为止。

以下是一个简单的示例，演示了如何使用C#中的迭代器协程：

using System;

using System.Collections;

class Program

{

static void Main(string[] args)

{

// 调用协程方法

foreach (var item in MyCoroutine())

{

Console.WriteLine(item);

}

}

// 定义协程方法

static IEnumerable MyCoroutine()

{

yield return "Step 1";

Console.WriteLine("Step 2");

yield return "Step 3";

Console.WriteLine("Step 4");

}

}

在这个示例中，MyCoroutine方法是一个迭代器方法，通过yield return关键字返回多个中间结果。当调用MyCoroutine方法时，它并不会立即执行，而是返回一个迭代器对象，然后通过foreach遍历迭代器对象，依次执行协程中的代码。

C#中的迭代器协程是一种非常方便的编程技术，可以用于简化异步编程、实现延迟加载、实现状态机等。在异步编程模型中，迭代器协程经常与async/await关键字一起使用，以实现更加灵活和可读性高的异步代码。

在Python、Lua、Go等编程语言中，都提供了协程的支持，使得开发人员可以更加灵活地利用系统资源，实现高效的并发处理。在异步编程、事件驱动编程和高并发网络编程等场景中，协程通常被广泛应用。

九、协程的原理是什么，为什么不会阻塞主线程

协程的原理是基于迭代器（iterator）和状态机（state machine）的概念实现的。在 C# 中，协程通过迭代器（IEnumerator 接口）来实现，其内部使用了状态机的概念来管理协程的执行状态。

具体来说，协程的原理可以概括为以下几个步骤：

迭代器：协程函数必须返回一个 IEnumerator 接口的实现，这个接口表示了协程的执行过程。协程函数中包含了 yield 关键字，用于暂停执行并返回执行状态。当协程函数执行到 yield 语句时，会暂停执行并返回一个迭代器，等待下次执行时再继续执行。

状态机：协程的执行过程可以看作是一个状态机，根据不同的执行状态来决定下一步的操作。每次执行到 yield 语句时，状态机会暂停当前状态，并记录下一次执行的状态。当协程再次执行时，根据上次记录的状态继续执行。

主线程执行：协程是在主线程中执行的，它利用了主线程的循环来实现暂停和恢复执行的效果。当协程暂停执行时，它会将状态保存下来，并等待主线程的下一次更新来恢复执行。这种方式可以避免创建新的线程，并保证协程的执行顺序。

总的来说，协程的原理是基于迭代器和状态机的概念实现的，在执行过程中利用了主线程的循环来实现暂停和恢复执行的效果。通过 yield 关键字，协程可以在执行过程中暂停并返回执行状态，从而实现了一种灵活的异步执行机制。

协程不会阻塞主线程的原因主要有两个方面：

主线程的循环：

Unity 的主线程负责处理游戏逻辑、渲染和用户输入等任务。它通过一个循环来不断地更新游戏状态，并且在每一帧结束时执行所有的协程。

协程的执行是在主线程的循环中完成的，它利用了主线程的循环机制来实现暂停和恢复执行的效果。当协程暂停执行时，它会等待下一次主线程更新来恢复执行，而不会阻塞主线程的执行。

yield 关键字：

协程中的 yield 关键字用于暂停执行并返回执行状态。当协程执行到 yield 语句时，它会暂停执行，并返回一个迭代器，等待下次执行时再继续执行。

由于协程的执行过程中包含了 yield 关键字，它可以在需要等待的情况下暂停执行，而不会阻塞主线程。当等待的条件满足时，协程会恢复执行，并继续执行后续的逻辑。

综上所述，协程不会阻塞主线程的执行，主要是因为它利用了主线程的循环机制和 yield 关键字来实现暂停和恢复执行的效果。这种机制可以避免创建新的线程，并且保证了协程的执行顺序和主线程的同步。

**十、协程和线程有什么区别**

协程（Coroutine）和线程（Thread）是两种不同的并发编程模型，它们有一些重要的区别：

执行上下文：

协程是在单线程环境下执行的，通常是在主线程中执行。协程利用了单线程的循环机制来实现暂停和恢复执行的效果。

线程是操作系统中的一个独立执行流，它可以并行执行，并且拥有自己的执行上下文和资源。线程可以在多核处理器上并行执行，可以同时执行多个任务。

并发性：

协程是一种轻量级的并发模型，它通常用于处理异步任务、延迟执行等场景，而不是用于真正的并行处理。

线程是一种真正的并发模型，它可以在多个线程之间并行执行任务，从而提高系统的并发性能。

资源消耗：

协程通常比线程更轻量级，它不需要额外的线程上下文切换开销，并且可以利用主线程的资源。

线程需要分配额外的系统资源，比如内存和 CPU 时间，并且需要进行线程间的上下文切换，这会增加系统的开销和复杂度。

编程模型：

在编程上，协程通常使用迭代器的概念来实现暂停和恢复执行，可以更方便地编写异步逻辑。

线程通常使用多线程编程模型，需要考虑线程同步、锁机制等问题，编程复杂度较高。

总的来说，协程和线程是两种不同的并发编程模型，它们各自适用于不同的场景和需求。协程适用于处理异步任务、延迟执行等轻量级并发场景，而线程适用于需要真正的并行处理和高并发性能的场景。在 Unity 中，协程常用于处理延迟执行、动画效果等任务，而线程通常用于处理一些需要后台计算或IO操作的任务。

**十一、CLR是什么**

CLR（Common Language Runtime）是微软.NET Framework中的核心组件之一，它是一个虚拟机（类似于Java的JVM），负责.NET程序的执行和管理。CLR提供了许多重要的功能，包括：

内存管理：CLR负责分配和释放内存，实现了自动内存管理（垃圾回收），使得开发人员不需要手动管理内存，减少了内存泄漏和野指针等问题。

类型安全：CLR在运行时对代码进行类型检查，确保类型安全，防止类型转换错误和内存访问越界等问题。

异常处理：CLR提供了强大的异常处理机制，使得开发人员可以轻松地捕获和处理各种异常，保证程序的稳定性和可靠性。

代码执行：CLR负责将IL（Intermediate Language，中间语言）代码编译成本地机器代码，并执行程序的各个部分。

安全性：CLR通过安全沙箱等机制确保程序的安全性，防止恶意代码对系统造成损害。

跨语言互操作性：CLR支持多种编程语言，包括C#、VB.NET、F#等，使得不同语言编写的代码可以相互调用和交互操作。

总之，CLR是.NET Framework的核心组件之一，提供了一种高效、安全、可靠的执行环境，大大简化了开发人员的工作，并提高了程序的性能和可维护性。

1. StringBuilder添加字符串原理是什么，为什么不会产生新的字符串对象

StringBuilder是一个可变的字符串容器，用于高效地构建字符串。它内部维护了一个字符数组（char[]），用于存储字符串的内容。当我们使用StringBuilder的Append方法添加字符串时，实际上是将要添加的字符串内容追加到内部的字符数组中。

StringBuilder添加字符串的原理可以简述如下：

初始时，StringBuilder会创建一个初始容量的字符数组，用于存储字符串内容。

当我们调用Append方法添加字符串时，StringBuilder会检查当前字符数组的容量是否足够容纳新增的字符串内容。

如果当前字符数组的容量不够，StringBuilder会根据需要动态扩展字符数组的大小，以容纳新增的字符串内容。

然后，StringBuilder会将新增的字符串内容复制到字符数组的末尾，实现字符串的追加操作。

由于StringBuilder内部维护的是可变大小的字符数组，因此在添加字符串时不会创建新的字符串对象，也不会频繁地进行内存分配和复制操作，从而避免了产生大量的临时字符串对象，提高了字符串拼接的性能和效率。

另外，StringBuilder还提供了一些其他方法，如Insert、Remove、Replace等，用于在字符串中进行插入、删除和替换操作，同样也是基于内部字符数组的操作，具有高效的性能表现。

**十三. 纹理加载进内存以后占用内存如何计算？**

纹理加载进内存后占用的内存量取决于纹理的分辨率、像素格式、以及是否使用了压缩等因素。下面是计算纹理占用内存的基本方法：

1. 基本公式

纹理占用的内存大小可以通过以下公式计算：

内存占用 (字节) = 宽度 × 高度 × 每像素的字节数 \text{内存占用 (字节)} = \text{宽度} \times \text{高度} \times \text{每像素的字节数}内存占用 (字节)=宽度×高度×每像素的字节数

2. 每像素字节数 (Bytes Per Pixel)

每像素的字节数取决于纹理的像素格式（Texture Format）。不同的像素格式会影响内存占用。常见的像素格式及其每像素字节数如下：

RGBA32: 4 字节（32位，8位红色，8位绿色，8位蓝色，8位透明度）

RGB24: 3 字节（24位，8位红色，8位绿色，8位蓝色）

Alpha8: 1 字节（8位透明度）

DXT1: 0.5 字节（压缩格式，4位每像素）

DXT5: 1 字节（压缩格式，8位每像素）

3. 计算示例

假设有一个 1024x1024 像素的纹理，使用 RGBA32 格式加载：

分辨率: 1024x1024

每像素字节数: 4 字节

内存占用 = 1024 × 1024 × 4 = 4 , 194 , 304 字节内存占用 = 4 MB \text{内存占用} = 1024 \times 1024 \times 4 = 4,194,304 \text{ 字节}\text{内存占用} = 4 \text{ MB}内存占用=1024×1024×4=4,194,304 字节内存占用=4 MB

4. MipMap

如果启用了 MipMap，会额外占用内存。通常 MipMap 总共会增加约 1/3 的内存消耗。对于上述示例，如果启用了 MipMap，内存消耗大约为：

总内存 = 4 MB + ( 4 × 1 3 ) MB ≈ 5.33 MB \text{总内存} = 4 \text{ MB} + (4 \times \frac{1}{3}) \text{ MB} \approx 5.33 \text{ MB}总内存=4 MB+(4×

3

1

​

) MB≈5.33 MB

5. 压缩纹理

如果使用了压缩格式，如 DXT1 或 DXT5，内存占用会大幅减少。以 DXT1 为例：

每像素字节数: 0.5 字节

内存占用:

内存占用 = 1024 × 1024 × 0.5 = 524 , 288 字节内存占用 = 0.5 MB \text{内存占用} = 1024 \times 1024 \times 0.5 = 524,288 \text{ 字节} \text{内存占用} = 0.5 \text{ MB}内存占用=1024×1024×0.5=524,288 字节内存占用=0.5 MB

6. 总结

分辨率: 分辨率越高，占用内存越大。

像素格式: 每像素字节数越高，占用内存越大。压缩格式可以显著降低内存占用。

MipMap: 启用 MipMap 会增加内存占用。

实际计算: 在实际使用中，还应考虑纹理是否启用压缩、MipMap，以及特定平台对纹理格式的支持情况。

十四、材质、贴图、纹理的关系

在计算机图形学和游戏开发中，材质、贴图和纹理是常见的术语，它们相互关联但有着不同的用途和概念。以下是对它们关系的详细解释：

1. 纹理（Texture）

定义: 纹理是一个二维的图像或位图，用于为3D模型的表面添加细节，如颜色、图案或其他视觉效果。

作用: 纹理通常是一个平铺的图像，用来给模型表面增加真实感，如木纹、石材、皮肤等视觉细节。

存储方式: 纹理通常以图像文件的形式存储，如 PNG、JPG、TGA 等。加载到程序中时，它们会占用显存，并可以通过不同的像素格式表示（如 RGBA32、RGB24 等）。

2. 贴图（Map）

定义: 贴图是用于特定材质属性的纹理。在实际应用中，贴图是指特定功能的纹理。

类型: 常见的贴图类型包括：

颜色贴图（Color Map 或 Albedo Map）: 定义物体的基本颜色。

法线贴图（Normal Map）: 模拟高低不平的表面细节，影响光照效果。

高光贴图（Specular Map）: 控制表面反射光的强度和颜色。

环境光遮蔽贴图（Ambient Occlusion Map）: 模拟阴影效果，增强表面细节。

粗糙度贴图（Roughness Map）: 控制表面的光滑度和反射率。

凹凸贴图（Bump Map）: 使用灰度图来模拟表面的细微凹凸感。

贴图通常会被分配到材质的不同通道，以控制不同的视觉效果。

3. 材质（Material）

定义: 材质是用于定义3D模型表面属性的集合，通常包括颜色、光泽度、反射率等。材质将不同的贴图组合在一起，定义了如何在渲染时显示模型表面。

作用: 材质将纹理（或贴图）与着色器绑定在一起，以实现复杂的视觉效果。例如，材质可以定义一个金属表面是如何反射光线的、布料的表面如何散射光线等。

组成: 材质通常包含多个属性，包括颜色、光泽度、透明度等，这些属性可以由贴图（纹理）来控制，也可以由固定的数值来定义。

4. 关系

纹理 是原始的图像资源，可以用作不同类型的贴图。

贴图 是特定功能的纹理，用于影响材质的不同属性。

材质 是多个贴图的集合，定义了3D模型表面的外观和行为。

5. 应用示例

在一个 3D 模型中，如果想表现一个金属表面，可以创建一个材质，这个材质可能包含：

颜色贴图: 一个纹理文件，用于定义金属的基本颜色。

法线贴图: 一个纹理文件，用于模拟金属表面的凹凸细节。

高光贴图: 一个纹理文件，用于控制金属表面的反射强度。

这些纹理被分配到材质的不同通道，组合在一起形成最终的渲染效果。

总结

纹理 是基础图像。

贴图 是具有特定用途的纹理。

材质 是包含多个贴图的组合，定义了物体的整体外观。

材质通过使用贴图来控制3D模型的不同属性，最终决定了模型在渲染时的表现效果。

**十五、修改Time.timeScale会影响什么？**

修改 Time.timeScale 会影响 Unity 游戏中所有依赖时间的操作。Time.timeScale 是一个全局的时间缩放因子，它默认值为 1，表示正常时间流逝速度。调整 Time.timeScale 的值会影响以下方面：

1. 影响游戏的时间流逝

时间流速: Time.timeScale 控制了游戏中的时间流逝速度。

当 Time.timeScale 为 1 时，时间正常流逝。

当 Time.timeScale 小于 1 时，时间变慢。例如，将 Time.timeScale 设置为 0.5 时，时间以一半的速度流逝，相当于慢动作效果。

当 Time.timeScale 设置为 0 时，时间停止流逝，这意味着游戏中的所有时间相关的操作都暂停。常用于游戏暂停功能。

2. 影响基于 Time.deltaTime 的操作

运动和动画: Time.deltaTime 是帧与帧之间的时间间隔，通常用于控制物体的运动、动画播放等。如果 Time.timeScale 被修改，Time.deltaTime 会根据 Time.timeScale 的值相应调整。例如，当 Time.timeScale 为 0.5 时，Time.deltaTime 也会减半，导致物体移动速度减慢。

动画和粒子系统: 基于 Time.deltaTime 控制的动画和粒子系统也会受到影响。当时间缩放因子变小，动画播放速度会减慢，粒子效果也会相应变慢。

3. 影响物理引擎

物理模拟: Unity 的物理引擎使用 Time.fixedDeltaTime 来进行物理计算。Time.fixedDeltaTime 是固定的时间步长，而 Time.fixedDeltaTime 是通过 Time.timeScale 进行缩放的。因此，修改 Time.timeScale 也会影响物理模拟的速度。

例如，如果 Time.timeScale 为 0.5，物理模拟的速度也会减半。

4. 影响音频

音频影响: 在 Unity 中，AudioSource 的音频不会受 Time.timeScale 影响，除非将 AudioSource.ignoreListenerPause 设置为 false。因此，通常在暂停游戏时，您可能需要手动暂停音频播放。

5. 影响协程

协程中的等待时间: 使用 WaitForSeconds 等依赖时间的协程操作会受到 Time.timeScale 的影响。例如，如果 Time.timeScale 被设置为 0.5，一个 WaitForSeconds(2) 的等待时间会变成 4 秒。

如果使用 WaitForSecondsRealtime，则不受 Time.timeScale 的影响，仍然会等待真实的时间。

6. 影响UI动画和时间管理

如果 UI 动画或过渡效果使用 Time.deltaTime 进行控制，Time.timeScale 也会影响这些效果的速度。

同时，在使用 Time.timeScale 控制游戏时间流速时，需要小心处理时间管理逻辑，以免出现意外的逻辑错误。

总结

修改 Time.timeScale 会影响整个游戏的时间流逝速度，包括物体的运动、动画、物理模拟以及协程的执行时间。这通常用于实现游戏的暂停、慢动作效果或加速效果。需要注意的是，修改 Time.timeScale 的同时，还需要处理音频、UI 动画以及物理模拟等细节，确保游戏在不同时间缩放下的正常运行。

**十六、 Unity中生命周期函数被调用的底层原理是什么?**

Unity 中的生命周期函数（如 Awake、Start、Update、FixedUpdate、OnDestroy 等）是 Unity 引擎内部自动调用的函数，用于管理脚本在不同阶段的行为。这些函数的调用是通过 Unity 的底层引擎框架来管理的，以下是其底层原理的概述：

1. 脚本的初始化

脚本的加载与初始化: 在 Unity 中，当一个 GameObject 被加载并附加了 MonoBehaviour 脚本时，Unity 会在后台为每个脚本实例创建一个与之关联的对象。这些对象由 Unity 的 C++ 引擎管理，通过这些对象 Unity 知道每个脚本实例的存在。

2. 脚本的生命周期管理

注册脚本实例: 当一个脚本被附加到一个 GameObject 时，Unity 会将这个脚本的实例注册到一个特定的队列中。每个队列对应特定的生命周期函数，如 Awake、Start、Update 等。

事件的调度与调用: Unity 的主循环（Main Loop）会在每一帧的特定时刻检查这些队列，并根据当前的游戏状态来调用相应的生命周期函数。Unity 通过高效的调度系统来确保在正确的时间调用这些函数，例如：

Awake: 在对象被激活（即使禁用的对象也会在激活时调用）时调用，用于初始化对象。

Start: 在 Awake 之后且对象启用时调用，通常用于在所有对象都初始化完毕后再进行一些设置。

Update: 每一帧都会调用，用于处理实时的输入、动画等。

FixedUpdate: 以固定的时间步长调用，通常用于物理模拟。

OnDestroy: 当对象即将销毁时调用，用于清理资源。

3. 通过反射机制调用

反射调用: Unity 在后台使用反射来调用这些生命周期函数。当一个脚本继承自 MonoBehaviour 并且定义了一个生命周期函数（如 Update），Unity 会通过反射机制检查这个脚本是否包含这些特定函数。如果包含，Unity 会在适当的时候通过反射调用它们。

性能优化: 为了提高性能，Unity 会在脚本加载时缓存这些反射信息，并且在每一帧时只对注册过的脚本进行相应函数的调用。Unity 还会优化那些没有定义常见生命周期函数（如 Update）的脚本，以避免不必要的函数调用。

4. C++ 和 C# 之间的交互

Native 和 Managed 代码的交互: Unity 的底层引擎是用 C++ 编写的，而用户的脚本是用 C# 编写的。Unity 通过一种叫做 P/Invoke 的技术来进行 C++ 和 C# 之间的交互。引擎的主循环在 C++ 层面执行，然后通过 P/Invoke 调用托管代码中的 C# 函数。

内存管理: Unity 负责管理 C++ 和 C# 对象之间的内存，确保在不同生命周期阶段合理地创建、管理和销毁对象。

5. 事件驱动架构

Unity 的生命周期函数的调用机制是事件驱动的。这意味着在引擎的每一个主要阶段，都会触发相应的事件，例如加载场景时会触发 Awake 和 Start，每帧更新时会触发 Update，物理更新时会触发 FixedUpdate，对象销毁时会触发 OnDestroy。

总结

Unity 中生命周期函数的调用是由引擎的主循环管理的，通过对脚本实例的注册、反射机制的使用以及事件驱动的架构，Unity 在游戏运行时的不同阶段调用相应的函数。引擎通过高效的调度和内存管理来保证这些函数的调用顺序和性能，使得开发者能够专注于游戏逻辑的实现。

1. **CharacterController和Rigidbody的区别**

CharacterController自带胶囊碰撞器，里面好像封装了一个刚体

Rigidbody就是刚体，使物体带有物理的特性

1. **物体绕某点旋转使用函数叫什么**

transform.RotateAround

1. ****移动相机动作在哪个函数里，为什么在这个函数里。****

LateUpdate,在每帧执行完毕调用，他是在所有update结束后才掉，比较适合用于命令脚本的执行。官网上例子是摄像机的跟随，都是在所有update操作完才跟进摄像机，不然就有可能出现摄像机已经推进了，但是视角里还未有角色的空帧出现。

1. **一个场景放置多个camera并同时处于活动状态，会发生什么**

游戏界面可以看到很多摄像机的混合。可以用depth（深度），Layer（层）+ Culling Mask,enable = false/true来控制

1. Unity 摄像机有几种模式，成像的原理是什么？

Unity 摄像机有两种模式一种是透视摄像机 ，一种是正交摄像机，透视摄像机成像的原理是利用相似三角形来成像

**二十二、打包AssetsBundle包出来后.manifest有什么作用？**

.mainfest 是一个文本文件里面有ab包的校验码，依赖以及资源内容等相关信息，这个信息是给开发读的，同时这个信息也作为二进制打入了ab包里面,所以使用ab包资源的时候，可以不用带.manifest文件。

**二十三、如果项目需求要打空包，你如何处理？**

项目需要打空包，我们在开发的时候，场景不要放任何内容，不然随着场景打包资源也被打入进去就不是空包了。同时Resources与StreammingAssets也不能放资源包,这样才能打出空包，我们如何使用资源呢？只有一条道路，就是从服务器上下载资源Ab包，再用代码把资源运行的时候添加到场景中。

**二十四、Unity 是单线程还是多线程的？**

Unity可以支持用户自己开发使用多线程, 比如用户做网络模块的时候，可以开一个线程来做网络消息的读取等。所以Unity支持多线程开发。Unity框架里面是两个线程，一个是main thread,一个是render thread, main thread负责业务逻辑线程，跑Mono代码，物理引擎代码等，render thread负责处理绘制相关，将绘制相关数据提交给GPU。所以unity内部是多线程的，Unity的很多API与场景组件相关API必须要在main thread里面调用，不能在其他线程调用。

**二十五、如果求两个向量的夹角？**

// 获取两个向量的夹角 Vector3.Angle 只能返回 [0, 180] 的值

// 如真实情况下向量 a 到 b 的夹角（80 度）则 b 到 a 的夹角是（-80）

// 通过 Dot、Cross 结合获取到 a 到 b， b 到 a 的不同夹角

private float GetAngle(Vector3 a, Vector3 b, Vector3 dir)

{

float angle1 = Vector3.Angle(a, b); //求出两向量之间的夹角

Vector3 normal = Vector3.Cross(a, b);//叉乘求出法线向量

angle1 \*= Mathf.Sign(Vector3.Dot(normal, dir)); //求法线向量与物体上方向向量点乘，结果为1或-1，修正旋转方向

return angle1;

}

**二十六、Unity是左手坐标系还是右手坐标系？**

Unity是左手坐标系，x正方向向右，y正方向向上，z正方向向屏幕里。

**二十七、Unity绘制场景物体的渲染顺序是怎么样的？**

1:不透明物体，离摄像机近的先绘制，这样它会因为ZTest剔除掉后面的渲染内容，性能更好。

2:半透明物体，必须从后往前画，先画离摄像机远的物体，才能保证渲染结果正确。

3:shader非常复杂的模型，尽量后绘制。比如T4M的地表shader，采样了七八张图纹理，那么它最后绘制，更加容易被Z-Culling剔除掉。

4:不透明物体(2000)----AlphaTest物体(2450)----半透明物体(3000)，这个渲染顺序有利于HSR或者Early-Z的优化。Alpha-Test 在不透明物体绘制完毕之后再绘制，可以避免Alpha-Test导致EarlyZ失效，最后绘制，至少不会影响到之前不透明物体的EarlyZ。

5:RenderQueue2500，区分了不透明和半透明。半透明物体（2500以上）永远在不透明物体（2500以下）后绘制。如果物体的RenderQueue在2500的同一侧，则sortingOrder优先级更高：Camera Depth > Sorting Layer > Order In Layer > RenderQueue > 距离相机的距离。

**二十八、Unity里Mipmap的理解有哪些优点与缺点？**

优点:避免远处物体的锯齿或者闪烁感，效果表现更好。Texture Cache，纹理采样的缓存命中率上升，减少对显存的直接访问，减少带宽io，进而提高渲染效率。

缺点:多消耗了1/3的内存，增大包体积

**二十九、Unity 如何平滑过渡旋转?**

有时候我们要控制角色模型从一个方向转到另外一个方向，如果两个方向之间的夹角比较大，那么转动起来比较生硬，我们可以通过四元数的球面插值来进行平滑处理，在update里面插值来达到平滑转向的目的。

**三十、向量的点乘、叉乘以及归一化的意义是什么?**

1)点乘描述了两个向量的相似程度，结果越大两向量越相似，还可表示投影

2)叉乘得到的向量垂直于原来的两个向量

3)标准化向量：用在只关系方向，不关心大小的时候

**UI系统**

**一、Image和RawImage的区别**

在Unity中，Image和RawImage是用于显示图片的两个组件，它们之间有一些区别：

Image（UI.Image）：

Image是基于Unity的UI系统的一部分，用于显示UI元素中的图片。

它支持Sprite类型的图片，并提供了许多属性和方法来控制图片的显示，例如颜色、填充、大小、对齐等。

Image组件常用于UI界面中，例如按钮、面板等。

RawImage（UI.RawImage）：

RawImage也是用于显示图片的UI组件，但是与Image不同，它可以显示Texture类型的原始图像。

RawImage通常用于在UI界面中显示非Sprite类型的图像，例如通过代码动态加载的Texture、视频纹理等。

RawImage提供了一些属性和方法来控制原始图像的显示，但相对于Image而言，它的功能更加简单。

总的来说，Image适用于显示Sprite类型的图片，并提供了更多的控制选项，而RawImage适用于显示Texture类型的原始图像，通常用于显示动态加载的图像或视频。

**二、画布的三种渲染模式**

在 Unity 的 UI 系统中，画布（Canvas）有三种渲染模式，它们决定了画布上 UI 元素的渲染方式和相对位置的计算方式。这三种模式分别是：

Screen Space - Overlay（屏幕空间 - 覆盖）：

在这种模式下，画布以屏幕为参考坐标系，UI 元素的位置和尺寸使用屏幕的像素单位来定义。

UI 元素将渲染在屏幕上，并且不会受到场景中其他物体的遮挡。

这种模式适用于 UI 元素不需要与场景中的物体交互，且需要始终显示在屏幕上的情况，如菜单、提示信息等。

Screen Space - Camera（屏幕空间 - 相机）：

在这种模式下，画布仍然以屏幕为参考坐标系，但是可以选择一个相机作为参考来确定 UI 元素的位置和尺寸。

UI 元素将渲染在相机的视图范围内，并且会受到相机的裁剪和视角影响。

这种模式适用于需要 UI 元素与场景中的物体进行交互，例如在游戏中显示玩家状态、生命值等信息。

World Space（世界空间）：

在这种模式下，画布不再依赖于屏幕坐标系，而是与场景中的世界坐标系相对应。

UI 元素的位置和尺寸使用世界坐标单位来定义，并且随着场景中物体的移动和旋转而变化。

这种模式适用于需要将 UI 元素作为场景中的物体来处理，例如在游戏中将 UI 元素作为角色的一部分进行交互。

每种模式都有其适用的场景和特点，开发者可以根据需求选择合适的画布渲染模式来实现不同的 UI 效果。

**三、图片的TextureType选项Texture和Sprite有什么区别**

在 Unity 中，当导入图片时，你可以选择不同的 TextureType 选项，其中 Texture 和 Sprite 是两个常见的选项，它们之间有以下区别：

Texture：

当你选择 Texture 选项时，Unity 将图片导入为普通的纹理（Texture）。这意味着图片将被处理为一个单独的纹理，可以被应用到材质（Material）上。

这种情况下，Unity 会将整个图片作为一个纹理来处理，而不会考虑图片中的透明区域或九宫格（9-Slice）划分。

Sprite：

当你选择 Sprite 选项时，Unity 将图片导入为 Sprite 对象，这意味着图片将被处理为一个可在游戏中作为 2D 精灵使用的对象。

Unity 会自动检测图片中的透明区域，并将其作为精灵的轮廓进行裁剪，从而创建一个带有透明边界的精灵。

如果图片是 9-Slice 划分的，则 Unity 也会识别并应用 9-Slice 划分，使得精灵可以以不同的方式进行缩放而不失真。

综上所述，Texture 和 Sprite 选项之间的主要区别在于导入后的处理方式。如果你打算将图片用作普通纹理，应该选择 Texture 选项；如果你打算将图片用作 2D 游戏中的精灵，应该选择 Sprite 选项，这样 Unity 将会自动处理透明区域和九宫格划分。

四、camera组件的clearflags的几个参数分别有什么作用

在 Unity 中，Camera 组件的 clearFlags 参数用于定义摄像机清除屏幕的方式。它有几个常见的参数，每个参数的作用如下：

Skybox：

摄像机将使用 Skybox 渲染背景，即将 Skybox 投射到屏幕上作为背景。

这意味着在摄像机的视野范围内，未被物体遮挡的部分将呈现 Skybox 的样式。

Solid Color：

摄像机将使用指定的颜色填充屏幕作为背景。

用户可以选择自定义的颜色，用于填充背景。

Depth Only：

摄像机将只渲染深度信息，而不渲染颜色信息。

这种模式通常用于渲染阴影贴图或深度纹理，以用于后期处理或其他需要深度信息的效果。

Don’t Clear：

摄像机不会对屏幕进行清除操作，而是直接在上一帧的内容上进行渲染。

这种模式通常用于实现层叠效果或者在多个摄像机之间进行混合渲染的情况。

这些 clearFlags 参数可以根据场景需求和渲染效果选择合适的方式。例如，如果需要呈现天空盒作为背景，则可以选择 Skybox 模式；如果需要自定义背景颜色，则可以选择 Solid Color 模式。

**四、UGUI 如何实现UI物体淡入淡出?**

UGUI里面比如Image这些组件，都继承自Graphic类，这个里面提供了一个方法:  
Alpha: 目标的alpha值  
Duration: 渐变的时间  
ignoreTimeScale:是否忽略Time.scale  
void CrossFadeAlpha(float alpha, float duration, bool ignoreTimeScale);

**物理系统**

**一、两个物体发生碰撞的必要条件**

在Unity中，两个物体（GameObject）之间发生碰撞通常需要满足以下几个必要条件：

Collider组件：每个参与碰撞检测的物体都必须至少有一个Collider组件。Collider组件定义了物体的碰撞区域和形状，用于检测其他物体与之的碰撞。

刚体组件（可选）：如果至少一个参与碰撞检测的物体需要具有物理属性，例如受力、速度等，那么该物体需要附加Rigidbody（刚体）组件。刚体组件使物体成为物理系统的一部分，使得它们受到重力、力和其他物理影响。

碰撞事件处理器：你需要在至少一个参与碰撞检测的物体上添加脚本来处理碰撞事件。这些脚本通常继承自MonoBehaviour并实现Unity提供的碰撞事件接口，例如OnCollisionEnter、OnCollisionStay和OnCollisionExit等。通过这些事件，你可以在碰撞发生时执行特定的逻辑或行为。

碰撞层和碰撞体：Unity提供了层(Layer)和层蒙版(Layer Mask)的概念，通过设置物体的Layer以及定义层间的碰撞关系，你可以控制哪些物体会相互碰撞。此外，Collider还具有一个IsTrigger属性，可以将碰撞器设置为触发器，这样它们之间的碰撞将不会导致物理反应，而是会触发碰撞事件。

总之，为了在Unity中实现物体之间的碰撞，你需要确保它们都具有Collider组件，并根据需要添加刚体组件。然后，你可以编写脚本来处理碰撞事件，并通过层和触发器属性来控制碰撞行为。

**二、CharacterController 和 Rigidbody 的区别**

CharacterController 和 Rigidbody 是 Unity 中用于控制物体运动的两种不同组件，它们有以下区别：

CharacterController：

CharacterController 是一个简单的碰撞体，用于控制游戏中角色的运动，特别适用于第三人称视角的角色控制。

CharacterController 不依赖于物理引擎，因此它的运动不受物理系统的影响，例如重力、碰撞等。这意味着你可以更灵活地控制角色的移动，而不必担心物理引擎的影响。

由于 CharacterController 不使用物理引擎，因此它不会触发任何物理事件，例如碰撞事件或触发器事件。这使得它更适合于某些特定类型的角色控制，例如第三人称角色控制或平面上的2D角色控制。

Rigidbody：

Rigidbody 是用于模拟物体在物理世界中的运动的组件，它基于物理引擎进行计算，并受到物理规则的影响，例如重力、碰撞、惯性等。

Rigidbody 的运动是由物理引擎管理的，因此它受到物理引擎的影响，例如重力将会影响物体的下落，碰撞将会导致物体发生反作用力等。

由于 Rigidbody 使用物理引擎，因此它可以触发各种物理事件，例如碰撞事件、触发器事件等。这使得 Rigidbody 更适合于需要真实物理模拟的场景，例如第一人称角色控制、物体间的互动等。

综上所述，CharacterController 适用于更灵活的角色控制，不受物理引擎的限制，而 Rigidbody 适用于需要真实物理模拟的场景，受到物理引擎的影响。在选择使用哪种组件时，需要根据具体需求和场景来进行考虑。

**三、射线检测碰撞物的原理是什么**

射线检测碰撞物的原理是通过发射一条射线（或射线段）来检测它是否与场景中的物体相交。这种技术常用于检测物体之间的碰撞、获取鼠标点击位置、进行射线投射等。

射线检测碰撞物的原理可以简单概括为以下几个步骤：

定义射线起点和方向：首先需要定义射线的起点和方向。射线通常是从摄像机或者其他物体的位置出发，向特定方向发射的一条线。

射线投射：将定义好的射线投射到场景中。在 Unity 中，可以使用 Physics.Raycast、Physics.RaycastAll 或 Physics.RaycastNonAlloc 等方法来进行射线投射。

检测碰撞：一旦射线与场景中的物体相交，射线检测系统就会返回一个碰撞信息（Collision Information），包括相交的物体、相交点等信息。

处理碰撞结果：根据碰撞信息，可以执行相应的操作，比如处理碰撞效果、获取碰撞点的位置信息、执行特定的游戏逻辑等。

射线检测碰撞物的原理是基于物理引擎的碰撞检测算法实现的，它可以高效地检测出射线与场景中物体的交点，从而实现各种交互效果和游戏功能。在 Unity 中，射线检测是一个常用且强大的功能，常用于开发游戏中的碰撞检测、射线投射、击中效果等。

**网络部分**

**一、什么是黏包**

在网络通信中，“黏包”（Packet Congestion）是一种常见的问题，指的是发送方在发送数据时，将多个数据包合并成一个大的数据包发送，导致接收方在接收数据时无法正确地将这些数据包区分开来，从而产生混乱或错误的数据接收现象。

黏包问题通常发生在基于 TCP 协议的网络通信中，TCP 是一种面向连接的协议，它通过将数据流分割成数据段并进行封装、传输和重组来实现可靠的数据传输。但是，TCP 并不保证每个数据段的边界对于接收方是可见的，这就导致了黏包问题的产生。

造成黏包问题的原因主要有以下几点：

缓冲区限制：发送端和接收端都有缓冲区，当发送端的发送速度大于接收端的处理速度时，发送端的数据会在接收端的缓冲区中积累，形成一个大的数据包。

传输延迟：网络传输中存在一定的延迟，导致数据在传输过程中可能被合并成一个大的数据包。

协议设计：某些协议在设计时没有考虑到黏包问题，或者使用了固定长度的数据包格式，无法处理变长数据包的情况。

黏包问题会导致接收方无法正确解析数据，从而造成数据解析错误、数据丢失或数据混乱的情况。为了解决黏包问题，可以采用以下几种方法：

消息边界标记：在数据流中添加特定的标记来标识消息的边界，接收方根据标记来解析数据包。

消息长度字段：在数据包的开头添加一个表示消息长度的字段，接收方先读取长度字段，然后再根据长度读取对应长度的数据。

固定长度消息：将每个数据包的长度固定为一个固定的值，接收方按照固定长度读取数据。

等待超时：接收方在接收数据时等待一定的超时时间，如果在超时时间内没有接收到完整的数据包，则认为黏包发生，重新开始接收数据。

综上所述，黏包是网络通信中常见的问题，通常由于缓冲区限制、传输延迟等原因导致。为了解决黏包问题，可以采用消息边界标记、消息长度字段、固定长度消息等方法来保证数据的正确解析。

**二、tcp和udp的区别**

TCP（Transmission Control Protocol）和UDP（User Datagram Protocol）是两种常见的网络传输协议，它们之间有以下几点区别：

连接性：

TCP 是一种面向连接的协议，它在传输数据之前需要先建立连接，然后进行数据传输，最后再释放连接。TCP 提供了可靠的数据传输和错误恢复机制，能够保证数据的完整性和可靠性。

UDP 是一种无连接的协议，它在传输数据之前不需要建立连接，也不会进行数据传输的确认和重传。UDP 的传输是不可靠的，可能会出现数据丢失、乱序等情况。

数据包大小：

TCP 在传输数据时，会将数据分割成数据段（segment）并进行封装，每个数据段的大小受到 TCP 协议规定的最大传输单元（MTU）的限制。通常情况下，TCP 的数据包大小较大。

UDP 在传输数据时，将数据封装成数据报（datagram）进行传输，每个数据报的大小通常受到底层网络协议的限制。通常情况下，UDP 的数据包大小较小。

传输速度：

TCP 提供了拥塞控制和流量控制等机制，可以根据网络情况动态调整传输速度，从而保证网络的稳定性和公平性。但是，由于 TCP 的连接管理和数据确认机制，可能会导致一定的传输延迟。

UDP 没有拥塞控制和流量控制等机制，数据传输速度较快，但是不稳定。UDP 适用于对数据传输速度要求较高、对数据传输可靠性要求较低的场景。

适用场景：

TCP 适用于对数据传输可靠性要求较高的场景，比如文件传输、网页浏览、电子邮件等。

UDP 适用于对数据传输速度要求较高、对数据传输可靠性要求较低的场景，比如实时音视频传输、在线游戏、实时通信等。

综上所述，TCP 和 UDP 是两种不同的网络传输协议，它们各有优缺点，适用于不同的场景和需求。TCP 提供可靠的数据传输和错误恢复机制，适用于对数据传输可靠性要求较高的场景；UDP 提供快速的数据传输，适用于对数据传输速度要求较高的场景。

**三、TCP/IP模型分为哪五层，有什么作用**

网络分为以下五层，从底层到顶层分别是：

物理层（Physical Layer）：

物理层是网络体系结构中最底层的一层，它主要负责传输比特流，以及在物理媒介上建立和维护连接。

物理层的作用包括定义物理媒介的传输特性、确定比特流的传输方式（如电信号、光信号等）以及物理连接的建立和维护。

数据链路层（Data Link Layer）：

数据链路层位于物理层之上，主要负责将比特流组织成帧，并进行错误检测和纠正，以确保数据在传输过程中的可靠性。

数据链路层的作用包括帧的封装、地址分配、流量控制和错误检测等。

网络层（Network Layer）：

网络层是负责在不同的网络之间进行数据传输的一层，它实现了网络之间的路由和转发功能，为数据包选择合适的路径，并确保数据包能够正确地到达目的地。

网络层的作用包括地址分配、路由选择、拥塞控制和数据包转发等。

传输层（Transport Layer）：

传输层位于网络层之上，负责在网络中的端到端通信，为应用层提供可靠的数据传输服务。

传输层的作用包括建立和维护端到端的连接、数据分段和重组、流量控制和拥塞控制等。

应用层（Application Layer）：

应用层是网络体系结构中最顶层的一层，它提供了各种网络应用和服务，包括电子邮件、文件传输、网页浏览、实时通信等。

应用层的作用包括定义网络应用和服务的协议、实现用户与网络的交互、处理数据的编码和解码等。

这五层网络体系结构通常采用 OSI（Open Systems Interconnection）模型或 TCP/IP（Transmission Control Protocol/Internet Protocol）模型来描述，它们共同构成了网络通信的基本架构，为不同层次的网络服务和应用提供了统一的规范和标准。

**优化部分**

**一、如何优化内存**

优化内存是在开发游戏或应用程序时非常重要的一部分，特别是对于移动设备和性能要求较高的场景。以下是一些常见的内存优化技巧：

资源压缩：

使用压缩算法对资源文件进行压缩，例如使用纹理压缩格式（如ETC、ASTC、PVRTC等）对纹理进行压缩，以减小资源文件的尺寸。

纹理合并：

将多个小纹理合并为一个大纹理，减少纹理切换和内存开销。可以使用纹理图集工具来自动合并纹理。

动态加载和卸载：

在运行时根据需要动态加载和卸载资源，避免一次性加载所有资源，以减少内存占用。尤其是对于大型场景或大量资源的游戏，动态加载是非常重要的优化手段。

对象池技术：

使用对象池技术重复使用对象，而不是频繁地创建和销毁对象。这样可以减少内存分配和垃圾回收的开销。

内存复用：

对于一些需要频繁创建和销毁的数据结构，尽量使用对象池或对象缓存来重复利用内存，减少频繁的内存分配和释放。

内存分析和优化工具：

使用内存分析工具来检测内存泄漏和高内存占用的地方，并进行优化。例如 Unity 中的 Profiler 和 Memory Profiler 工具可以帮助你分析内存使用情况，并找出潜在的性能瓶颈。

资源优化：

优化资源的质量和尺寸，例如减少纹理的分辨率、降低模型的多边形数量等，以减小资源文件的尺寸和内存占用。

内存管理策略：

使用适当的内存管理策略，例如对象引用计数、垃圾回收等，来确保内存的合理使用和释放，避免内存泄漏和内存溢出问题。

以上是一些常见的内存优化技巧，具体的优化策略需要根据项目的具体情况和需求来进行调整和实施。

1.压缩自带类库；  
2.对象池，使用预设(Prefab)。  
3.释放AssetBundle占用的资源；  
4.降低模型的片面数，降低模型的骨骼数量，降低贴图的大小；  
5.使用光照贴图，

**二、GC产生的原因，如何避免**

GC（垃圾收集器）产生的主要原因是程序中创建了大量的对象，而这些对象在一段时间后不再被程序所引用，成为了"垃圾"。垃圾收集器会定期检查内存中的对象，回收不再被引用的对象所占用的内存空间，以便为新对象的创建提供足够的空间。主要的原因可以归结为：

动态内存分配：在运行时动态分配内存空间，创建了大量的对象。

对象的使用周期：部分对象的生命周期较短，很快就不再被程序所使用。

频繁的对象创建和销毁：频繁创建和销毁对象会导致大量的垃圾产生。

要避免或减少垃圾收集器的频繁工作，可以采取以下几种策略：

对象池技术：重用对象而不是频繁地创建和销毁它们。通过对象池技术可以避免大量的对象创建和销毁操作，减少垃圾产生。

减少对象的使用：分析程序的内存使用情况，尽量减少不必要的对象创建，尤其是一些生命周期较短的临时对象。

优化算法和数据结构：使用更加高效的算法和数据结构，减少对象的数量和内存占用。

避免内存泄漏：确保不再使用的对象能够被垃圾收集器正确回收，避免内存泄漏问题。

手动资源释放：对于一些非托管资源，可以通过手动释放资源的方式来避免不必要的内存占用，如关闭文件、释放数据库连接等。

适时释放资源：及时释放不再需要的对象引用，帮助垃圾收集器更快地回收内存。

综上所述，通过合理的内存管理和资源利用，以及优化程序设计和算法，可以有效地减少垃圾收集器的频繁工作，提高程序的性能和稳定性。

1. ****顶点数和Draw Call数？如何降低Draw Call数？****

Unity中，每次引擎准备数据并通知GPU的过程称为一次Draw Call。DrawCall越高对显卡的消耗就越大。

降低DrawCall的方法：

Dynamic Batching（动态批处理）

对足够小的网格，他在CPU层自动转换他们的坐标，将许多定点组织到一起然后一次绘制

Static Batching（静态批处理）

不需要移动的游戏对象，合并成一个大网格，然后快速绘制

1. **Unity UGUI 优化RaycastTarget。**

RaycastTarget勾选以后会消耗一些效率，为了节省效率就不要勾选它了，不仅Image组件Text组件也有这样的问题。 一般UI里也就是按钮才需要接收响应事件，那么大部分image和text是是不需要开RaycastTarget的。默认是选上的，记得关闭掉这些，可以提升性能。

1. **Unity如何优化纹理图片的体积？**

纹理图片的大小决定了包体的大小，我们要让包体小，需要优化纹理图片，纹理图片一般是由.png,jpg图片组成，我们可以通过压缩工具来压缩图片，一般我用的工具有TinyPNG。

1. **Unity编写C#代码需要注意哪些地方？（代码优化）**

1:不要使用foreach,每个foreach调用，大概会分配30B左右的内存.大量的调用会触发GC

2:尽量使用stringbuilder替代string.并且new stringbuilder(cap) 尽量预先分配好合适大小的内存.string.format的底层实现是stringbuilder，可以使用。

3:c#对于返回的变量，基本都是新new出来的.尽量少使用.比如

ICollection.ToArray(),会返回一个新new的T[]对象，ToString(),会产生string临时对象

4:不要使用Lambda表达式.每次都会有52B的内存分配. 定义函数替代之。

5:很多数据结构 比如list/Dictionary都提供了初始分配cap大小的构造函数.根据实际的需要，优先使用这样的构造函数来初始化数据结构

6:尽量不要使用值类型到System.Object的转换.c#俗称装箱.

7:不要以枚举/自定义struct作为Dictionary的key.尽量考虑用基础类型int等来替代。他们的GetHashCode都有装箱操作,每次调用TryGetValue查找都会有内存分配.

8:尽量减少new的次数,预分配/成员变量替代临时new变量等.

9:尽可能避免使用LINQ。部分功能无法在某些平台上使用，会分配大量GC Alloc。而且我很讨厌LINQ的一点就是它有可能在某些情况下无法很好的进行AOT编译。比如“OrderBy”会生成内部的泛型类“OrderedEnumerable”。这在AOT编译时是无法进行的，因为它只是在OrderBy的方法中才使用。所以如果你使用了OrderBy，那么在IOS平台上也许会报错。

10:尽可能避免使用delegate以及lambda。

1. **Unity 项目开发中 避免GC的常用技巧有哪些？**

GC是一个非常高开销的系统调用也，是大部分卡顿的主要原因，不能完全控制。因此我们要尽量减少代码堆内存分配过量防止频繁触发GC，同时也可以在Loading或者对性能不敏感的时候主动GC。

升级高版本的Unity, 减少了lamda表达式的gc

减少一些字符串拼接，使用StringBuilder代替string减少GC开销

内存池：前面说过GameObject的内存池，另外还有类对象的内存池。所有频繁反复创建删除的都应该使用。两个用途，减少加载创建释放的时间，减少内存碎片降低GC的频率。

Unity接口：AddComponent，OnGUI，UI合并频率，delegate，等。

避免频繁创建开辟空间。

****客观问题****

1. ****如果你是架构师，资源管理你会如何做？****

资源管理是Unity框架设计里面很重要的一个模块，我们先来分析需求，资源管理的需求有1:代码加载资源与资源的释放接口

2:完整的资源更新机制与管理

3:支持打空包的同时，也支持打普通的游戏包。

4:做好版本管理与渠道管理。

做到第一点需求，如果我们用Resource.load来加载，那么打空包资源更新就非常不方便。所以放弃Resources, 同时又要代码加载，我们采用纯AssetsBundle机制来进行设计。所有的资源都基于AssetsBundle来进行管理,加载,释放。AssetsBundle本身有完整成熟的更新与解决方案。打空包也方便，把ab包部署到服务器，第一次运行先下载到本地，后续从本地加载ab包。在打包Ab包的时候，分渠道，分平台，生成不同的ab包资源。如果不打空包，只要把这些ab包放入StreammingAssets目录下这样用StreammingAssets的方法来加载ab包。先从ab包下载目录加载，如果没有，就到StreammingAssets目录下加载，这样永远加载的是最新下载的，其次才是在包体所带的。

1. ****Unity项目开发中程序如何愉快的与策划沟通？****

在游戏发中，策划希望能方便的改游戏数值，游戏数据，能够灵活的编辑一些规则，这个是策划希望的，并且策划在改数据的时候，不用问程序，马上就能抛出效果，策划在改规则的时候也希望如此。规则一般提现在怪物AI等方面，数据一般提现在数值计算方面。如果是怪物AI等行为编辑比较多的，我们可以提供给策划行为决策树等编辑工具，让策划编辑行为和决策。数值计算，我们提供好数值策划表解决方案，让策划非常方便的修改里面的数据表等。游戏的地图制作，也是策划经常要干预的地方，Unity项目开发中地图编辑器要做好。

1. ****Unity项目开发如何避免多人开发是的提交冲突?****

在团队项目开发中，由于大家各自开发格子的模块或同时开发交叉的模块。就会导致了一些项目提交时候的冲突，Unity开发中如何避免冲突或者有冲突的时候非常方便的合并呢？

这边的话提供几个经验供大家参考。

1:代码冲突的合并，这个相对比较简单文本合并即可，很多版本管理工具自动就能帮你合并好。即使有冲突，人眼很快就能解决，毕竟都是文本代码。

2: 游戏场景冲突就不那么好合并了，因为游戏场景本省不是文本，同时合并的时候可能会导致一些参数要重新指定等，所以很多Unity项目不会往场景里面放任何内容，而是把内容分成一个一个的预制体，每一个人负责一个预制体（视图）。尽量避免多人同时操作一个视图。

3: 不要在视图节点上直接挂代码，也不要在代码上直接绑定视图的节点组件等,这些都不利于代码冲突解决，全部都通过代码来完成。

总结一下: 避免多人修改同一视图，视图上不挂脚本，全部通过代码来控制逻辑，代码提交冲突非常方便解决。

1. ****Unity做手游开发的时候资源规格控制标准一般是哪些？****

1:模型角色高模 6000+面，低模 3000面。顶点数不定，因为顶点存在共用的情况，所以顶点数量可能是面数的两倍，也可能比面数低。纹理使用1024大小的贴图。低模纹理大小减半。主角因为有高低模，所以不开mipmap。个别精度比较高的npc（有UI显示需求）需要开mipmap，否则场景中会有明显的闪烁。

2:场景的drawcall控制在200以下。个别场景因为美术需求，drawcall可能会更高一些，但是SetPassCall也要尽可能保证比较小（材质尽可能共用，可以有效减少SetPassCall）

3:场景的面数在100k~300k之间都是合理的数值。

4:UI的drawcall100以下，一般就是三四十常驻UI Drawcall。