参考来源

2023年Unity面试题大全，共十万字面试题总结【收藏一篇足够面试，持续更新】 unity面试题2023-CSDN博客

kimi AI

C#基础

重载和重写的区别

	重载（overload）	重写（override）
封装、继承、多态位置	同类	父子类
定义方式	方法名相同，参数列表不同	方法名相同，参数列表相同
调用方式	使用相同对象以不同参数调用	使用不同对象以相同参数调用
多态时机	编译时多态	运行时多态
类的关系	同一个类中方法之间的关系，是平值关系	子类和父类之间的关系，是继承关系

重载：方法重载。

重写：虚拟方法重写、抽象方法重写

面向对象的三大特点

继承

- 提高代码重用度，增强软件可维护性的重要手段，符合开闭原则。继承最主要的作用就是把子类的公共属性集合起来，便与共同管理，使用起来也更加方便。你既然使用了继承，那代表着你认同子类都有一些共同的特性，所以你把这些共同的特性提取出来设置为父类。
- 继承的传递性:传递机制 a>b; b>c;c具有a的特性。
- 继承的单根性:在C#中一个类只能继承一个类，不能有多个父类。

封装

- 封装是将数据和行为相结合，通过行为约束代码修改数据的程度，增强数据的安全性，属性是C#封装实现的最好体现。就是将一些复杂的逻辑经过包装之后给别人使用就很方便，别人不需要了解里面是如何实现的，只要传入所需要的参数就可以得到想要的结果。
- 封装的意义在于保护或者防止代码(数据)被我们无意中破坏。

多态

- 多态性是指同名的方法在不同环境下，自适应的反应出不同得表现，是方法动态展示的重要手段。
- 多态就是一个对象多种状态，子类对象可以赋值给父类型的变量。
- (个人理解：不同子类继承了同一个子类，重写的某个函数不同子类是不同的，这就是多态的表现。)

简述值类型和引用类型有什么区别

值类型：包含了所有简单类型(整数、浮点、bool、char)、struct、enum。继承自System.ValueTyoe

引用类型：包含了string，object，class，interface，delegate，array。继承自System.Obiect

- 1.值类型存储在内存栈中，引用类型数据存储在内存堆中，而内存单元中存放的是堆中存放的地址。
- 2.值类型存取快，引用类型存取慢。
- 3.值类型表示实际数据，引用类型表示指向存储在内存堆中的数据的指针和引用。
- 4.栈的内存是自动释放的，堆内存是.NET 中会由 GC 来自动释放。
- 5.值类型继承自 System.ValueType,引用类型继承自 System.Object。
- 6.值类型的变5直接存放实际的数据，而引用类型的变5存放的则是数据的地址，即对象的引用。
- 7.值类型变量直接把变量的值保存在堆栈中，引用类型的变量把实际数据的地址保存在堆栈中。

整理的表格：

特性/差异点	值类型	引用类型
包含类型	简单类型（整数、浮点、bool、char）、struct、enum	string、object、class、interface、delegate、array
继承自	System.ValueType	System.Object
存储位置	内存栈	内存堆
存储方式	直接存储实际数据	存储数据的地址（引用）
存取速度	快	慢
内存管理	自动释放	由GC自动释放
变量存储内容	变量的值	实际数据的地址

特性/差异点	值类型	引用类型
继承关系	继承自System.ValueType	继承自System.Object

简述private，public，protected，internal的区别

- public:对任何类和成员都公开，无限制访问
- private:仅对该类公开
- protected:对该类和其派生类公开
- internal:只能在包含该类的程序集中访问该类
- protected internal: protected + internal

整理的表格：

访问修饰符	描述	访问范围
private	私有成员	仅在定义它们的类内部
public	公共成员	任何类都可以访问
protected	受保护的成员	定义它们的类和派生类
internal	内部成员	同一程序集中的任何类
protected internal	组合访问修饰符	同一程序集中的受保护成员或派生类
private protected	组合访问修饰符	同一程序集中的私有成员或派生类

protected internal和private protected是C# 7.2版本中新增的访问修饰符，它们提供了更细粒度的访问控制。

C#中所有引用类型的基类是什么

- 引用类型的基类是System.Object
- 值类型的基类是 System.ValueType同时
- 值类型也隐式继承自System.Object

简述C#中ArrayList和 List的主要区别

- ArrayList 不带泛型 数据类型丢失
- List 带泛型 数据类型不丢失
- ArrayList 需要装箱拆箱 List不需要

ArrayList存在不安全类型(ArrayList会把所有插 入其中的数据都当做Object来处理)装箱拆箱的 操作(费时)IList是接口，ArrayList是一个实现了 该接口的类，可以被实例化

List类是ArrayList类的泛型等效类。它的大部分用法都与ArrayList相似，因为List类也继承了IList接口。最关键的区别在于，在声明List集合时，我们同时需要为其声明List集合内数据的对象类型。

整理表格：

特性/差异点	ArrayList	List
泛型支持	不支持泛型	支持泛型
数据类型安全	可能丢失，所有元素视为object	保持数据类型，不丢失
性能	可能需要装箱和拆箱，影响性能	不需要装箱拆箱，性能较高
类型安全	由于元素视为object，类型不安全	元素类型明确，类型安全
接口实现	实现了`IList`接口	继承自`IList<T>`接口
声明方式	声明时不需要指定元素类型	声明时需要指定元素类型
适用场景	适用于不确定元素类型或多种类型的场景	适用于元素类型固定且单一的场景

编程中，推荐使用泛型集合，如`List<T>`，以避免类型转换和提高代码的可读性和维护性。

简述GC(垃圾回收)产生的原因，描述如何避免?

GC为了避免内存溢出而产生的回收机制

避免:
1.减少 new 产生对象的次数
2.使用公用的对象(静态成员)
3.将 String 换为 StringBuilder

请描述Interface与抽象类之间的不同

1.接口不是类 不能实例化 抽象类可以间接实例化
2.接口是完全抽象 抽象类为部分抽象
3.接口可以多继承 抽象类是单继承

简述关键字Sealed用在类声明和函数声明时的作用

Sealed：封闭的，密封的

类声明时可防止其他类继承此类

在方法中声明则可防止派生类重写此方法

反射的实现原理?

可以在加载程序运行时，动态获取和加载程序集，并且可以获取到程序集的信息反射即在运行期动态获取类、对象、方法、对象数据等的一种重要手段

主要使用的类库:System.Reflection

核心类:

1. Assembly描述了程序集
2. Type描述了类这种类型
3. ConstructorInfo描述了构造函数
4. MethodInfo描述了所有的方法
5. FieldInfo描述了类的字段
6. PropertyInfo描述类的属性

通过以上核心类可在运行时动态获取程序集中的类，并执行类构造产生类对象，动态获取对象的字段或属性值，更可以动态执行类方法和实例方法等。

Net与 Mono 的关系?

.Net是一个语言平台，Mono为.Net提供集成开发环境，集成并实现了.NET的编译器、CLR 和基础类库，使得.Net既可以运行在windows也可以运行于 linux，Unix，Mac Os 等。

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、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
- 一个类只能有一个静态函数，如果有静态变量，系统也会自动生成静态函数
- （ai说）：总结来说，不能在构造函数前加上 `static` 关键字，因为构造函数是用来创建和初始化对象的，而静态上下文与对象实例无关。静态构造函数是隐式存在的，不需要（也不能）显式声明。

C# String类型比 stringBuilder 类型的优势是什么?

- 如果是处理字符串的话，用string中的方法每次都需要创建一个新的字符串对象并且分配新的内存地址，而 stringBuilder 是在原来的内存里对字符串进行修改，所以在字符串处理
- 方面还是建议用stringBuilder这样比较节约内存。但是 string 类的方法和功能仍然还是比 stringBuilder类要强。
- string类由于具有不可变性(即对一个 string 对象进行任何更改时，其实都是创建另外一个 string 类的对象)，所以当需要频繁的对一个 string 类对象进行更改的时候，,建议使用StringBuilder 类，StringBuilder 类的原理是首先在内存中开辟一定大小的内存空间，当对此 StringBuilder 类对象进行更改时，如果内存空间大小不够， 会对此内存空间进行扩充，而不是重新创建一个对象，这样如果对一个字符串对象进行频繁操作的时候，不会造成过多的内存浪费，其实本质上并没有很大区别，都是用来存储和操作字符串的，唯一的区别就在于性能上。
- String主要用于公共 API，通用性好、用途广泛、读取性能高、占用内存小。
- StringBuilder主要用于拼接 String，修改性能好,
- 不过现在的编译器已经把String的+操作优化成 StringBuilder了，所以一般用String 就可以了
- String是不可变的，所以天然线程同步。
- StringBuilder可变，非线程同步

整理成表格的形式：

特性/优势点	String	StringBuilder
可变性	不可变	可变
内存分配	每次操作可能分配新内存	在原有内存空间修改，扩充内存
线程安全	线程安全	非线程安全
性能	读取性能高，修改性能低	修改性能好，拼接效率高
用途	公共API，用途广泛	主要用于字符串拼接
内存占用	占用内存小	初始内存分配，可能较大
适用场景	频繁读取，少量修改	频繁修改，如循环拼接字符串
编译器优化	编译器优化了`+`操作	编译器优化了大量字符串操作
通用性	通用性好，使用广泛	特定场景下使用

"编译器优化"指的是现代C#编译器在编译期间会优化字符串连接操作，避免在运行时使用大量的中间字符串对象，从而提高性能。但是，如果涉及到大量的字符串操作，特别是在循环中，使用StringBuilder仍然是更优的选择，因为它避免了每次修改字符串时都创建新对象的开销。

C#函数 Func(string a, string b)用 Lambda 表达式怎么写?

```
(a,b) => {};
```

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

```
public int CountNumber(int num) {
       if (num == 1 || num == 2) {
           return 1;
       } else {
           return CountNumber(num -1) + CountNumber(num-2);
       }
  }
```

冒泡排序(手写代码)

```
public static void BubblingSort(int[]array) {

      for (int i = 0; i < array.Length; i++){

          for (int j = array.Length - 1; j > 0; j--){

              if (array[j] < array[i]) {
```

```
                int temp = array[j];
                array[j] = array[j-1];
                array[j - 1] = temp;
            }
        }
    }
}
```

C#中有哪些常用的容器类，各有什么特点

List,HashTable, Dictionary, Stack, Queue

- **stack栈:**先进后出，入栈和出栈，底层泛型数组实现，入动态扩容2倍
- **Queue队列:**先进先出，入队和出队，底层泛型数组实现，表头表尾指针，判空还是满通过size比较

  Queue和Stack主要是用来存储临时信息的
- **Array数组:**需要声明长度，不安全
- **ArrayList数组列表:**动态增加数组，不安全，实现了IList接口(表示可按照索引进行访问的非泛型集合对象)，Object数组实现
- **List列表:**底层实现是泛型数组，特性，动态扩容，泛型安全将泛型数据(对值类型来说就是数据本身，对引用类型来说就是引用)存储在一个泛型数组中添加元素时若超过当前泛型数组容量，则以2倍扩容，进而实现List大小动态可变。(注:大小指容量，不是Count)
- **LinkList链表:**

  1、数组和List、ArrayList集合都有一个重大的缺陷，就是从数组的中间位置删除或插入一个元素需要付出很大的代价，其原因是数组中处于被删除元素之后的所有元素都要向数组的前端移动。

  2、LinkedList(底层是由链表实现的)基于链表的数据结构，很好的解决了数组删除插入效率低的问题，且不用动态的扩充数组的长度。

  3、LinkedList的优点:插入、删除元素效率比较高;缺点:访问效率比较低。
- **HashTable哈希表(散列表)**

  概念:不定长的二进制数据通过哈希函数映射到一个较短的二进制数据集，即Key通过HashFunction函数获得HashCode，然后通过哈希桶算法，将HashCode分段，每一段都是一个桶结构，一般是HashCode直接取余。桶结构会加剧冲突，解决冲突使用拉链法，将产生冲突的元素建立一个单链表，并将头指针地址存储至Hash表对应桶的位置。这样定位到Hash表桶的位置后可通过遍历单链表的形式来查找元素。

  装填因子:a=n/m=0.72,存储的数据N和空间大小M

  1、Key-Value形式存取，无序，类型Object，需要类型转换。

  2、Hashtable查询速度快，而添加速度相对慢

  3、Hashtable中的数据实际存储在内部的一个数据桶里(bucket结构体数组)，容量固定，根据数组索引获取值。

```
//哈希表结构体
private struct bucket {
   public Object key;//键
    public Object val;//值
```

```
    public int hash_col;//哈希码
}
//字典结构体
private struct Entry {
    public int hashCode;    // 除符号位以外的31位hashCode值，如果该Entry没有被使用，那么为-1
    public int next;        // 下一个元素的下标索引，如果没有下一个就为-1
    public TKey key;        // 存放元素的键
    public TValue value;    // 存放元素的值
}

private int[] buckets;      // Hash桶
private Entry[] entries;    // Entry数组，存放元素
private int count;          // 当前entries的index位置
private int version;        // 当前版本，防止迭代过程中集合被更改
private int freeList;       // 被删除Entry在entries中的下标index，这个位置是空闲的
private int freeCount;      // 有多少个被删除的Entry，有多少个空闲的位置
private IEqualityComparer<TKey> comparer;   // 比较器
private KeyCollection keys;     // 存放Key的集合
private ValueCollection values;     // 存放Value的集合
```

性能排序:

- 插入性能: LinkedList>Dictionary>HashTable >List.
- 遍历性能: List>LinkedList>Dictionary>HashTable
- 删除性能: Dictionary>LinkedList>HashTable >List

C#中常规容器和泛型容器有什么区别，哪种效率高?

不带泛型的容器需要装箱和拆箱操作速度慢，所以泛型容器效率更高数据类型更安全

有哪些常见的数值类?

简单值类型:包括 整数类型、实数类型、字符类型、布尔类型

复合值类型:包括 结构类型、枚举类型

C#中委托 和 接口有什么区别?各用在什么场合?

接口(interface)是约束类应该具备的功能集合，约束了类应该具备的功能，使类从千变万化的具体逻辑中解脱出来，便于类的管理和扩展，同时又合理解决了类的单继承问题。

C#中的委托 是约束方法集合的一个类，可以便捷的使用委托对这个方法集合进行操作。

在以下情况中使用接口:
1.无法使用继承的场合
2.完全抽象的场合
3.多人协作的场合

委托多用于事件处理中

C#中unsafe关键字是用来做什么的?什么场合下使用?

非托管代码才需要这个关键字一般用在带指针操作的场合。

项目背包系统的任务装备栏使用到

（ai说：）

在C#中，`unsafe`关键字用于定义一个代码区域，在这个区域内可以执行一些通常不允许的操作，比如直接访问和操作内存地址。使用`unsafe`代码可以进行低级内存操作，这通常在以下场合下使用：

1. **性能优化**:
   - 某些情况下，为了提高性能，可能需要绕过C#的安全性和抽象性，直接操作内存。
2. **指针操作**:
   - C#通常不允许使用指针，但在`unsafe`代码块中可以声明和使用指针。
3. **与非托管代码交互**:
   - 当需要与C或C++等非托管代码交互时，可能需要使用指针或其它低级内存操作。
4. **位操作**:
   - 虽然C#提供了位操作符，但在某些复杂的情况下，可能需要更直接的位字段访问。
5. **特定系统编程**:
   - 在开发操作系统、驱动程序或嵌入式系统软件时，可能需要直接访问硬件或内存。
6. **内存密集型应用**:
   - 在内存使用非常关键的应用中，如大型数据处理或图形处理，`unsafe`代码可以用来手动管理内存。

使用`unsafe`代码时需要非常小心，因为它绕过了C#的许多安全特性，容易导致程序崩溃或安全漏洞。例如，如果不正确地使用指针，可能会导致访问违规（Access Violation）异常。

下面是一个简单的`unsafe`代码块示例：

```
unsafe
{
    int* ptr = stackalloc int[10]; // 分配内存
    for (int i = 0; i < 10; i++)
    {
        ptr[i] = i * i;
    }

    // 使用指针访问和操作内存
    fixed (int* p = &ptr[0])
    {
        // p 现在指向 ptr[0]
    }
}
```

在上面的代码中，`unsafe`代码块允许我们使用指针`ptr`来访问和修改内存。`stackalloc`关键字用于在栈上分配内存，而`fixed`关键字用于固定内存地址，使其在`fixed`语句块内保持不变。

总的来说，`unsafe`关键字是一个强大的工具，但应该谨慎使用，只在确实需要时使用。

C#中ref和out关键字有什么区别?

ref修饰引用参数。参数必须赋值，带回返回值，又进又出out修饰输出参数。参数可以不赋值，带回返回值之前必须明确赋值引用参数和输出参数不会创建新的存储位置
如果ref参数是值类型，原先的值类型数据，会随着方法里的数据改变而改变如果ref参数值引用类型，方法里重新赋值后，原对象堆中数据会改变，如果对引用类型再次创建新对象并赋值给ref参数，引用地址会重新指向新对象堆数据。方法结束后形参和新对象都会消失。实参还是指向原始对象，值不够数据改变了

For，foreach，Enumerator.MoveNext的使用，与内存消耗情况

for循环可以通过索引依次进行遍历，foreach和Enumerator.MoveNext通过迭代的方式进行遍历内存消耗上本质上并没有太大的区别。
但是在Unity中的Update中，一般不推荐使用foreach 因为会遗留内存垃圾。

函数中多次使用string的+=处理，会产生大量内存垃圾(垃圾碎片)，有什么好的方法可以解杂。

通过StringBuilder那进行append，这样可以减少内存垃圾

当需要频繁创建使用某个对象时，有什么好的程序设计方案来节省内存?

设计单例模式进行创建对象或者使用对象池

JIT和AOT区别

Just-In-Time -实时编译
执行慢安装快占空间小一点
Ahead-Of-Time -预先编译
执行快安装慢占内存占外存大

给定一个存放参数的数组，重新排列数组

```
void SortArray(Array arr){Array.Sort(arr);}
```

Foreach循环迭代时，若把其中的某个元素删除，程序报错，怎么找到那个元素?以及具体怎么处理这种情况?(注:Try...Catch捕捉异常，发送信息不可行)

foreach不能进行元素的删除，因为迭代器会锁定迭代的集合，解决方法:记录找到索引或者key值迭代结束后再进行删除。

GameObject a=new GameObject()GameObject b=a实例化出来了A，将A赋给B，现在将B删除，问A还存在吗?

存在，b删除只是将它在栈中的内存删除，而A对象本身是在堆中，所以A还存在

C#中 委托和事件的区别

大致来说，委托是一个类，该类内部维护着一个字段，指向一个方法。事件可以被看作一个委托类型的变量，通过事件注册、取消多个委托或方法。

- 委托就是一个类，也可以实例化，通过委托的构造函数来把方法赋值给委托实例
- 触发委托有2种方式: 委托实例.Invoke(参数列表)，委托实例(参数列表)。
- 事件可以看作是一个委托类型的变量
- 通过+=为事件注册多个委托实例或多个方法
- 通过-=为事件注销多个委托实例或多个方法
- EventHandler就是一个委托

结构体和类有何区别?

结构体是一种值类型，而类是引用类型。(值类型、引用类型是根据数据存储的角度来分的)就是值类型用于存储数据的值，引用类型用于存储对实际数据的引用。

那么结构体就是当成值来使用的，类则通过引用来对实际数据操作

C#的委托是什么?有何用处?

委托类似于一种安全的指针引用，在使用它时是 当做类来看待而不是一个方法，相当于对一组方 法的列表的引用。

用处:使用委托使程序员可以将方法引用封装在 委托对象内。然后可以将该委托对象传递给可调 用所引用方法的代码，而不必在编译时知道将调 用哪个方法。与C或C++中的函数指针不同，委托 是面向对象，而且是类型安全的。

foreach迭代器遍历和for循环遍历的区别

（ai说：）

foreach 遍历

- **语法简洁**：`foreach` 提供了一种简洁的方式来遍历集合中的每个元素。
- **只读访问**：在 `foreach` 循环中，迭代变量（如 `item`）是只读的，不能在循环内修改。
- **使用场景**：适合于只需要读取集合元素而不需要修改集合的场景。

- **原理**：`foreach` 通过调用集合的 `GetEnumerator()` 方法来获取一个 `IEnumerator` 对象，然后通过调用 `MoveNext()` 和 `Current` 属性来遍历集合。

for 循环遍历

- **灵活性高**：`for` 循环提供了更高的灵活性，可以在循环中修改集合。
- **索引访问**：`for` 循环通过索引访问集合元素，允许对元素进行修改或删除。
- **使用场景**：适合于需要修改集合或需要索引信息的场景。
- **原理**：`for` 循环通常使用一个索引变量来控制循环次数，直接通过索引访问集合元素。

特殊情况和注意事项

- **修改集合**：在 `foreach` 循环中直接修改集合（如使用 `list.Remove(item)`）是不允许的，因为这可能导致运行时错误。如果需要修改，可以在 `foreach` 循环中记录需要修改的元素或索引，然后在循环外进行操作，或者使用 `for` 循环。
- **性能**：对于数组或 `ArrayList` 等，使用 `for` 循环可以减少垃圾回收（GC）的压力，因为 `foreach` 可能会在每次迭代时创建迭代器对象。
- **Unity 和 Lambda 表达式**：在 Unity 中使用 `for` 循环和 Lambda 表达式时，需要注意闭包问题，这可能会导致意外的行为或性能问题。

```
// 使用 foreach 遍历 List
List<int> numbers = new List<int> { 1, 2, 3 };
foreach (int item in numbers)
{
    Console.WriteLine(item); // 只能读取，不能修改 item
}

// 使用 for 循环遍历 List 并修改集合
for (int i = 0; i < numbers.Count; i++)
{
    if (numbers[i] == 2)
    {
        numbers.RemoveAt(i);
        i--; // 调整索引以适应集合大小变化
    }
}
```

`foreach` 遍历提供了一种简单且安全的方式来读取集合元素，而 `for` 循环则提供了更高的灵活性

C#和C++的区别

简单的说:C# 与C++ 比较的话，最重要的特性 就是C# 是一种完全面向对象的语言，而C++不 是，另外C# 是基于I 中间语言

和.NET Framework CLR 的，在可移植性，可维 护性和强壮性都比C++ 有很大的改进。C# 的设 计目标是用来开发快速稳定可扩展的应用程序，当然也可以通过Interop和Pinvoke 完成一些底层操作

具体对比:

1. 继承:C++支持多继承，C#类只能继承一个基类中的实现但可以实现多个接口。
2. 数组:声明 C# 数组和声明 C++ 数组的语法不同。在 C# 中，“[]”标记出现在数组类型的后面。
3. 数据类型:在C++中bool类可以与整型转换，但C#中bool 类型和其他类型(特别是int)之间没有转换。long 类型:在 C# 中，long 数据类型为 64 位，而在 C++ 中为 32 位。

4. struct 类型:在 C# 中，类和结构在语义上不同。struct 是值类型，而 class 是引用类型5.switch 语句:与 C++ 中的
5. switch 语句不同，C# 不支持从一个 case 标签贯穿到另一个 case 标签。
6. delegate 类型:委托与 C++ 中的函数指针基本相似，但前者具有类型安全，是安全的。
7. 从派生类调用重写基类成员。 base
8. 使用 new 修饰符显式隐藏继承成员。
9. 重写方法需要父类方法中用virtual声名，子类方法用override 关键字,
10. 预处理器指令用于条件编译。C# 中不使用头文件。 C# 预处理器指令11.异常处理:C#中引入了 finally 语句，这是C++没有的。
11. C# 运算符:C# 支持其他运算符，如 is 和 typeof。它还引入了某些逻辑运算符的不同功能。
12. static 的使用，static方法只能由类名调用，改变static变量。
13. 在构造基类上替代 C++初始化列表的方法。
14. Main 方法和 C++ 及Java中的 main 函数的声明方式不同，Main而不能用main16.方法参数:C# 支持 ref和 out 参数，这两个参数取代指针通过引用传递参数。
15. 在 C# 中只能在unsafe不安全模式下才使用指针。
16. 方法参数:C# 支持 ref 和 out 参数，这两个参数取代指针通过引用传递参数。
17. 在 C# 中只能在unsafe不安全模式下才使用指针。
18. 在 C# 中以不同的方式执行重载运算符。
19. 字符串:C# 字符串不同于 C++ 字符串。
20. foreach:C#從VB中引入了foreach关键字使得以循环访问数组和集合。
21. C# 中没有全局方法和全局变量:方法和变量必须包含在类型声明(如 class 或 struct)中。22.C# 中没有头文件和 #include 指令:using 指令用于引用其他未完全限定类型名的命名空间中的类型。
22. C# 中的局部变量在初始化前不能使用。
23. 析构函数:在 C# 中，不能控制析构函数的调用时间，原因是析构函数由垃圾回收器自动调用。析构函数
24. 构造函数:与 C++ 类似，如果在 C# 中没有提供类构造函数，则为您自动生成默认构造函数。该默认构造函数将所有字段初始化为它们的默认值。
25. 在 C# 中，方法参数不能有默认值。如果要获得同样的效果，需使用方法重载。

整理成表格的形式：

特性/差异点	C#	C++
面向对象	完全面向对象	不完全面向对象，支持多继承
中间语言和运行环境	基于.NET Framework和CLR的中间语言	编译成机器码，直接运行
继承	只能继承一个类，但可以实现多个接口	支持多继承
数组声明	`类型[] 数组名`	`类型 数组名[]`
基本数据类型	`bool` 不与整型隐式转换，`long` 为64位	`bool` 可与整型隐式转换，`long` 为32位
struct类型	`struct` 是值类型	`struct` 在语义上与 `class` 相同，都是值类型
switch语句	不支持case贯穿	支持case贯穿
委托	类似函数指针，类型安全	函数指针，非类型安全
重写基类成员	使用 `base` 关键字	使用 `::` 和 `base` 概念
隐藏继承成员	使用 `new` 修饰符	无直接对应机制
重写方法	使用 `virtual` 和 `override` 关键字	使用 `virtual` 和 `override` 概念
预处理器指令	不使用头文件，使用 `#define` 等指令	使用预处理器和头文件 `#include`
异常处理	有 `finally` 块	没有 `finally` 块

特性/差异点	C#	C++
运算符	支持 `is` 和 `typeof`，某些逻辑运算符功能不同	运算符功能不同，不支持 `is` 和 `typeof`
static使用	`static` 方法和变量通过类名直接调用	`static` 成员通过类名或对象调用
构造函数	可以自动生成默认构造函数	可以自动生成默认构造函数，但可手动定义多个构造函数
Main方法	`static void Main()`	`int main()`
方法参数	支持 `ref` 和 `out` 参数，不支持指针	支持指针，不支持 `ref` 和 `out`
指针	只能在 `unsafe` 上下文使用	广泛使用
运算符重载	可以重载运算符	可以重载运算符，但规则不同
字符串	字符串不可变，类型为 `System.String`	字符串可变，通常使用字符数组
循环访问	引入 `foreach` 关键字	使用 `for` 或 `while` 循环
全局方法/变量	没有全局方法和变量	可以有全局方法和变量
头文件和#include	不使用头文件和 `#include`	使用头文件和 `#include`
局部变量	必须初始化后才能使用	可以不初始化直接使用
析构函数	由垃圾回收器自动调用，不可手动调用	可以手动调用析构函数
方法参数默认值	不支持方法参数默认值	支持方法参数默认值

C#引用和C++指针的区别

C#不支持指针，但可以使用Unsafe，不安全模式，CLR不检测

C#可以定义指针的类型、整数型、实数型、!struct结构体

C#指针操作符、C#指针定义使用fixed，可以操作类中的值类型

相同点:
都是地址
指针指向一块内存，它的内容是所指内存的地址;而引用则是某块内存的别名

不同点:
指针是个实体，引用是个别名
sizeof 引用”得到的是所指向的变量(对象)的大小，而“sizeof 指针”得到的是指针本身的大小;引用是类型安全的，而指针在不安全模式下

整理成表格的形式：

特性/差异点	C# 引用	C++ 指针
支持情况	默认安全，无指针概念，但在`unsafe`上下文中可以使用指针	直接支持指针
类型定义	不直接支持指针类型定义，但可以在`unsafe`上下文中定义	可以定义各种类型的指针，如整型、实数型、结构体等
操作符	使用`&`获取引用地址，`*`用于解引用（在`unsafe`上下文）	使用`&`获取地址，`*`进行解引用
操作限制	引用总是指向某个对象，不能有NULL引用；操作受CLR管理	指针可以是NULL，可以指向任意内存地址，包括非法内存
大小计算	`sizeof`引用得到的是引用类型的大小	`sizeof`指针得到的是指针本身的大小
类型安全	是，引用提供了类型安全	否，指针操作在不安全模式下可能导致未定义行为
实体与别名	引用是别名，本质上是对象的另一个名字	指针是实体，存储了对象的内存地址
内存管理	由CLR自动垃圾回收，无需手动管理	需要程序员手动管理内存分配和释放
使用场合	主要用于常规对象访问，`unsafe`上下文用于特定低级操作	广泛用于底层内存操作和系统编程

请注意，C#的`unsafe`上下文提供了一种方式来执行指针操作，但这通常是受限的，并且不推荐在没有充分理由的情况下使用，因为它绕过了CLR的内存安全检查。而C++的指针是语言的核心部分，提供了更高的灵活性和控制能力，但同时也增加了出错的风险。

堆和栈的区别?

通常保存着我们代码执行的步骤，如在代码段1中 AddFive()方法，int pValue变量，int result变量等等。 而堆上存放的则多是对象，数据等。(译者注:忽略编 译器优化)我们可以把栈想象成一个接着一个叠放在 一起的盒子。当我们使用的时候，每次从最顶部取走 一个盒子。栈也是如此，当一个方法(或类型)被调 用完成的时候，就从栈顶取走(called aFrame，译 注:调用帧)，接着下一个。堆则不然，像是一个仓 库，储存着我们使用的各种对象等信息，跟栈不同的 是他们被调用完毕不会立即被清理掉。

整理成表格的形式：

特性/差异点	栈（Stack）	堆（Heap）
存储内容	局部变量、方法调用时的上下文信息	对象实例、数据等
内存管理	自动管理，遵循后进先出（LIFO）原则	手动管理，需要显式分配和释放内存
访问速度	访问速度快，因为内存连续	访问速度慢，因为内存可能分散
生命周期	短暂，方法调用结束后立即释放	较长，需要显式回收或由垃圾回收机制处理
存储结构	可以想象成叠放的盒子，每次使用从顶部取一个	可以想象成仓库，存储各种信息，不立即清理
数据访问	直接通过变量名访问	通常需要通过指针或引用访问
内存大小	相对较小，受限于系统或编译器设定	相对较大，受限于物理内存大小
清理方式	方法调用结束后自动清理	需要程序员手动清理或由垃圾收集器回收

请注意，表格中描述的堆和栈的特点是基于一般情况，实际行为可能会因编程语言、编译器优化、运行时环境等因素而有所不同。例如，在某些编程语言中，对象也可能在栈上分配（例如，C++中的小对象优化）。而在C#等语言中，所有的对象都是通过垃圾回收机制管理的，即使它们存储在堆上。

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。
3.stack空间有限，heap 的空间是很大的自由区。

整理成表格的形式：

特性/差异点	栈（Stack）	堆（Heap）
定义	用于存储程序执行期间的局部变量和调用栈	用于存储动态分配的对象和数据
内存管理	自动分配和释放	手动申请和释放，使用`new`关键字分配
空间大小	相对有限，大小固定或有上限	相对很大，受限于系统内存
访问速度	较快，因为栈内存靠近处理器	较慢，因为可能分散在物理内存

特性/差异点	栈（Stack）	堆（Heap）
生命周期	短暂，函数调用结束后自动释放	较长，需程序员显式释放或由垃圾回收机制处理
数据管理	后进先出（LIFO）	不确定，取决于分配和释放的顺序
作用域	只在当前函数调用中有效	可以跨多个函数调用持续存在
存储内容	局部变量和函数调用的上下文信息	对象实例、大型数据结构等
内存分配策略	编译器确定，通常与函数调用相关	运行时确定，由程序员控制

请注意，C#中的垃圾回收机制（GC）会定期清理堆内存中不再使用的对象，而栈内存则在函数调用结束后自动释放。此外，`unsafe`代码块中可能使用指针直接操作堆内存，但这通常受限于特定的场景和安全考虑。

Mock和Stub有何区别?

Mock与Stub的区别:Mock:关注行为验证。细粒度的 测试，即代码的逻辑，多数情况下用于单元测试。Stub:关注状态验证。粗粒度的测试，在某个依赖系 统不存在或者还没实现或者难以测试的情况下使用，例如访问文件系统，数据库连接，远程协议等。

整理成表格的形式：

特性/差异点	Mock	Stub
关注点	行为验证	状态验证
测试粒度	细粒度	粗粒度
测试目的	验证代码逻辑	模拟依赖系统的状态
使用场景	单元测试，需要验证方法调用逻辑	集成测试或系统测试，模拟尚未实现的系统或难以测试的依赖
功能	可以模拟对象并验证方法是否被调用以及参数	返回预定义的结果，不关注方法调用的细节
实现方式	使用Mock框架，如Moq、NSubstitute等	实现接口或继承基类，提供简单的默认实现

请注意，Mock和Stub都是测试中使用的模拟对象，但它们的使用目的和方式有所不同。Mock对象主要用于验证被测试代码的行为，确保代码按预期执行，而Stub对象则用于模拟外部依赖的返回状态，以便在测试中隔离外部系统或组件。

为什么dynamic font 在 unicode环境下优于staticfont(字符串编码)

Unicode是国际组织制定的可以容纳世界上所有文字和符号的字符编码方案。使用动态字体时，Unity将不会预先生成一个与所有字体的字符纹理。当需要支持亚洲语言或者较大的字体的时候，若使用正常纹理，则字体的纹理将非常大。

简述StringBuilder和String的区别?(字符串处理)

String是字符串常量。StringBufer是字符串变量，线程安全。StringBuilder是字符串变量，线程不安全
String类型是个不可变的对象，当每次对String进行改变时都需要生成一个新的String对象，然后将指针指向一个新的对象，如果在一个循环里面，不断的改变一个对象，就要不断的生成新的对象，所以效率很低，建议在不断更改String对象的地方不要使用String类型。
StringBuilder对象在做字符串连接操作时是在原来的字符串上进行修改，改善了性能。这一点我们平时使用中也许都知道，连接操作频繁的时候，使用StringBuilder对象。

整理成表格的形式：

特性/差异点	**String**	**StringBuilder**
可变性	不可变	可变
线程安全性	安全	非线程安全
内存效率	低效，每次修改都可能产生新对象	高效，原地修改减少内存分配
性能	低，特别是在循环中频繁修改字符串	高，适合频繁修改和拼接字符串操作
用途	适用于不经常变动的字符串操作	适用于需要频繁修改的字符串操作，如循环拼接
内部实现	每次修改都可能涉及新的内存分配	内部使用可扩展的字符数组，减少内存分配

请注意，`String`在C#中是不可变的，每次对字符串进行修改操作时，实际上都会生成一个新的`String`对象。而`StringBuilder`是为了优化字符串的修改和拼接操作而设计的，它在内部维护一个可增长的字符数组，可以在不生成新对象的情况下修改字符串内容。由于`StringBuilder`是非线程安全的，所以在多线程环境下使用时需要额外的同步处理。

string、stringBuilder、stringBuffer

- **String**不变性，字符序列不可变，对原管理中实例对象赋值，会重新开一个新的实例对象赋值新开的实例对象会等待被GC。
  string拼接要重新开辟空间，因为string原值不会改变，导致GC频繁，性能消耗大
- **StringBuffer**是字符串可变对象，可通过自带的StringBuffer,方法来改变并生成想要的字符串。对原实例对象做拼接的实例，不会生成新的实例对象。拼接使用StringBuilder和StringBuffer，只开辟一个内存空间，这是性能优化的点。
- **StringBuilder**是字符串可变对象，基本和StringBuilder相同,唯一的区别是StringBuffer是线程安全，相关方法前带synchronized关键字，一般用于多线程StringBuilder是非线程安全，所以性能略好，一般用于单线程

三者性能比较 StringBuilder>StringBuffer>String

1. 如果要操作少量的数据 =string
2. 单线程操作字符串缓冲区 下操作大量数据=StringBuilder
3. 多线程操作字符串缓冲区 下操作大量数据=StringBuffer

整理成表格的形式：

特性/差异点	**String**	**StringBuilder**	**StringBuffer**
可变性	不可变	可变	可变
线程安全性	非线程安全	非线程安全	线程安全
性能	低，每次修改都可能产生新对象	高，适合单线程下频繁修改	高，适合多线程下频繁修改
内存效率	低，字符串拼接时频繁GC	高，原地修改减少内存分配	高，原地修改，线程同步开销
用途	适用于不经常变动的字符串操作	适用于单线程中的频繁字符串操作	适用于多线程中的频繁字符串操作
内部实现	每次修改都涉及新的内存分配	内部使用可扩展的字符数组	内部使用同步机制的可扩展字符数组
垃圾回收	频繁，因为每次修改都生成新对象	较少，修改在同一对象上	较少，修改在同一对象上，但有线程同步开销
推荐使用场景	少量数据操作	单线程大量数据操作	多线程大量数据操作

请注意，`StringBuffer`在C#中实际上是`System.Text.StringBuilder`类的一部分，它提供了额外的同步机制来确保线程安全。在.NET中没有独立的`StringBuffer`类，因此上表中的`StringBuffer`实际上指的是使用`StringBuilder`时考虑线程安全的情况。在.NET中通常直接使用`StringBuilder`，如果需要线程安全的字符串操作，可以通过其他机制（如`lock`语句）来确保。

字典Dictionary的内部实现原理

泛型集合命名空间using System.Collections.Generic;任何键都必须是唯一
该类最大的优点就是它查找元素的时间复杂度接近O(1)，实际项目中常被用来做一些数据的本地缓存，提升整体效率。

实现原理

1. 哈希算法:将不定长度的二进制数据集给映射到一个较短的二进制长度数据集一个Key通过HashFunc得到HashCode
2. Hash桶算法:对HashCode进行分段显示，常用方法是对HashCode直接取余
3. 解决碰撞冲突算法(拉链法):分段会导致key对应的桶会相同，拉链法的思想就像对冲突的元素，建立一个单链表，头指针存储到对应的哈希桶位置。反之就是通过确定hash桶位置后，遍历单链表，获取对应的value

泛型是什么

多个代码对 【不同数据类型】 执行 【相同指令】的情况泛型:多个类型共享一组代码泛型允许类型参数化，泛型类型是类型的模板
5种泛型:类、结构、接口、委托、方法
类型占位符 T来表示泛型

泛型类不是实际的类，而是类的模板
从泛型类型创建实例
声明泛型类型》通过提供【真实类型】创建构造函数类型》从构造类型创建实例类<T1,T2>泛型类型参数

性能:泛型不会强行对值类型进行装箱和拆箱，或对引用类型进行向下强制类型转换，所以性能得到提高

安全:通过知道使用泛型定义的变量的类型限制，编译器可以在一定程度上验证类型假设，所以泛型了程序的类型安全

Mathf.Round和Mathf.Clamp和Mathf.Lerp含义?

- Mathf.Round:四舍五入
- Mathf.Clamp:左右限值
- Mathf.Lerp:插值

能用foreach遍历访问的对象需要实现接口或声明方法的类型(C#遍历)

Enumerable; GetEnumerator

List和Dictionary类型可以用foreach遍历，他们都实现了IEnumerable接口，申明了GetEnumerator方学

什么是里氏替换原则?(C#多态)

里氏替换原则(Liskov Substitution Principle LSP)面向对象设计的基本原则之一

- 里氏替换原则中说，任何基类可以出现的地方，子类一定可以出现，作用方便扩展功能
- 子类可以实现父类的抽象方法，但是不能覆盖父类的非抽象方法。
- 子类中可以增加自己特有的方法。
- 当子类覆盖或实现父类的方法时，方法的前置条件(即方法的形参)要比父类方法的输入参数更宽松。
- 当子类的方法实现父类的抽象方法时，方法的后置条件(即方法的返回值)要比父类更严格

反射的实现原理?

可以在加载程序运行时，动态获取和加载程序集，并且可以获取到程序集的信息反射即在运行期动态获取类、对象、方法、对象数据等的一种重要手段
主要使用的类库:System.Reflection

核心类:

1. Assembly描述了程序集
2. Type描述了类这种类型
3. ConstructorInfo描述了构造函数
4. MethodInfo描述了所有的方法
5. FieldInfo描述了类的字段
6. PropertyInfo描述类的属性

通过以上核心类可在运行时动态获取程序集中的类，并执行类构造产生类对象，动态获取对象的字段或属性值，更可以动态执行类方法和实例方法等。
审查元数据并收集关于它的类型信息的能力。

审查元数据并收集关于它的类型信息的能力。

实现步骤:

1. 导入using System.Reflection;
2. Assembly.Load("程序集")加载程序集,返回类型是一个Assembly
3. foreach (Type type in assembly.GetTypes())
   {
   string t = type.Name;
   }
   得到程序集中所有类的名称
4. Type type = assembly.GetType("程序集.类名");获取当前类的类型
5. Activator.CreateInstance(type); 创建此类型实例
6. MethodInfo mInfo = type.GetMethod("方法名");获取当前方法

7. mInfo.Invoke(null,方法参数);

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，接收方叫事件接收市

哈希表与字典对比

字典:内部用了Hashtable作为存储结构

- 如果我们试图找到一个不存在的键，它将返回/抛出异常。
- 它比哈希表更快，因为没有装箱和拆箱，尤其是值类型。
- 仅公共静态成员是线程安全的。
- 字典是一种通用类型，这意味着我们可以将其与任何数据类型一起使用(创建时，必须同时指定键和值的数据类型)。
- Dictionay 是 Hashtable 的类型安全实现，Keys和Values是强类型的，
- Dictionary遍历输出的顺序，就是加入的顺序

**哈希表:

- 如果我们尝试查找不存在的键，则返回 null。
- 它比字典慢，因为它需要装箱和拆箱。
- 哈希表中的所有成员都是线程安全的，
- 哈希表不是通用类型，
- Hashtable 是松散类型的数据结构，我们可以添加任何类型的键和值。
- HashTable是经过优化的，访问下标的对象先散列过，所以内部是无序散列的

C#中四种访问修饰符是哪些?各有什么区别?

1. 属性修饰符
2. 存取修饰符
3. 类修饰符
4. 成员修饰符

属性修饰符:

- Serializable:按值将对象封送到远程服务器。
- STATread:是单线程套间的意思，是一种线程模型。
- MATAThread:是多线程套间的意思，也是一种线程模型。

存取修饰符:

- public:存取不受限制。
- private:只有包含该成员的类可以存取。
- internal:只有当前工程可以存取。
- protected:只有包含该成员的类以及派生类可以存取。

类修饰符:

- abstract:抽象类。指示一个类只能作为其它类的基类。

- sealed:密封类。指示一个类不能被继承。理所当然，密封类不能同时又是抽象类，因为抽象总是希望被继承的。

成员修饰符:

- abstract:指示该方法或属性没有实现。
- sealed:密封方法。可以防止在派生类中对该方法的override(重载)。不是类的每个成员方法都可以作为密封方法密封方法，必须对基类的虚方法进行)载，提供具体的实现方法。所以，在方法的声明中，sealed修饰符总是和override修饰符同时使用。
- delegate:委托。用来定义一个函数指针。C#中的事件驱动是基于delegate +event的。
- const:指定该成员的值只读不允许修改。
- event:声明一个事件。
- extern:指示方法在外部实现。
- override:)写。对由基类继承成员的新实现
- readonly:指示一个域只能在声明时以及相同类的内部被赋值。
- static:指示一个成员属于类型本身，而不是属于特定的对象。即在定义后可不经实例化，就可使用。virtual:指示一个方法或存取器的实现可以在继承类中被覆盖。
- new:在派生类中隐藏指定的基类成员，从而实现)写的功能。 若要隐藏继承类的成员，请使用相同名称在派生类中声明该成员，并用 new 修饰符修饰它,

下列代码在运行中会发生什么问题?如何避免?

```
List<int> ls = new List<int>(new int[]{ 1, 2, 3, 4, 5 });
          foreach (int item in ls)
          {
              Console.WriteLine(item * item);
              ls.Remove(item);
          }
```

会产生运行时错误，因为foreach是只读的。不能一边遍历一边修改。

使用For循环遍历可以解决。

什么是装箱拆箱，怎样减少操作

C#装箱是将值类型转换为引用类型，拆箱是将引用类型转换为值类型。牵扯到装箱和拆箱操作比较多的就是在集合中，例如:ArrayList或者HashTable之类.

MVC

MVC全名是Model View Controller，是模型(model)-视图(view)-控制器(controller)的缩写，一种软件设计典范。

用一种业务逻辑、数据、界面显示分离的方法，将业务逻辑聚集到一个部件里面，在改进和个性化定制界面及用户交互的同时，不需要重新编写业务逻辑。MVC被独特的发展起来用于映射传统的输入、处理和输出功能在一个逻辑的图形化用户界面的结构中。

- Model(模型)是应用程序中用于处理应用程序数据逻辑的部分通常模型对象负责在数据库中存取数据。
- View(视图)是应用程序中处理数据显示的部分。通常视图是依据模型数据创建的。
- Controler(控制器)是应用程序中处理用户交互的部分。通常控制器负责从视图读取数据，控制用户输入，并向模型发送数据

unity基础

lmage和Rawlmage的区别

- lmgae比Rawlmage更消耗性能
- Image只能使用Sprite属性的图片，但是Rawlmage什么样的都可以使用。
- Image适合放一些有操作的图片，裁剪平铺旋转什么的，针对Image Type属性。
- Rawlmage就放单独展示的图片就可以，性能会比Image好很多

Unity3D中的碰撞器和触发器的区别?

碰撞器是触发器的载体，而触发器只是碰撞器身上的一个属性。

当Is Trigger=false时，碰撞器根据物理引擎引发碰撞，产生碰撞的效果，可以调用OnCollisionEnter/Stay/Exit系数:

当Is Trigger=true时，碰撞器被物理引擎所忽略，没有碰撞效果，可以调用OnTriggerEnter/Stay/Exit函数。

如果既要检测到物体的接触又不想让碰撞检测影响物体移动或要检测一个物件是否经过空间中的某个区域这时就可以用到触发器。

物体发生碰撞的必要条件?

两个物体都必须带有碰撞器Colider，其中一个物体还必须带有Rigidbody刚体,

简述四元数Quaternion的作用，四元数对欧拉角的优点?

四元数用于表示旋转

相对欧拉角的优点:能进行增量旋转、避免万向锁、给定方位的表达方式有两种，互为负(欧拉角有无数种表达方式)

如何安全的在不同工程间安全地迁移asset数据?三种方法

1. 将Assets和Library一起迁移
2. 导出包package

3. 用unity自带的assets Server功能

OnEnable、Awake、Start运行时的发生顺序?哪些可能在同一个对象周期中反复的发生?

Awake->OnEnable->Start

OnEnable在同一周期中可以反复地发生!

MeshRender中material和sharedmaterial的区别?

修改sharedMaterial将改变所有物体使用这个材质的外观，并且也改变储存在工程里的材质设置。不推荐修改由sharedMaterial返回的材质。如果你想修改渲染器的材质，使用material替代。

TCP/IP协议栈各个层次及分别的功能?

网络接口层:这是协议栈的最低层，对应OSI的物理层和数据链路层，主要完成数据的实际发送和接收。

网络层:处理分组在网络中的活动，例如路由选择和转发等，这一层主要包括IP协议、ARP、ICMP协议等。

传输层:主要功能是提供应用程序之间的通信，这一层主要是TCP/UDP协议

应用层:用来处理特定的应用，针对不同的应用提供了不同的协议，例如进行文件传输时用到的FTP协议，发送email用到的SMTP等，

Unity提供了几种光源，分别是什么?
四种。
平行光:Directional Light
点光源:Point Light
聚光灯:Spot Light
区域光源:Area Light

简述一下对象池，你觉得在FPS里哪些东西适合使用对象池?

对象池就存放需要被反复调用资源的一个空间
比如游戏中要常被大量复制的对象，子弹，敌人，以及任何重复出现的对象。

特点:用内存换取cpu的优化

CharacterController和Rigidbody的区别?

Rigidbody具有完全真实物理的特性，而CharacterController可以说是受限的的Rigidbody，具有一定的物理效果但不是完全真实的。

移动相机动作在哪个函数里，为什么在这个函数里?

LateUpdate，是在所有的Update结束后才调用，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，都是所有的Update操作完才进行摄像机的跟进，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。

简述prefab的用处

在游戏运行时实例化，prefab相当于一个模板，对你已经有的素材、脚本、参数做一个默认的配置以便于以后的修改，同事prefab打包的内容简化了导出的操作，便于团队的交流。

GPU的工作原理?

简而言之，GPU的图形(处理)流水线完成如下的工作:(并不一定是按照如下顺序)

顶点处理:这阶段GPU读取描述3D图形外观的顶点数据并根据顶点数据确定3D图形的形状及位置关系，建立起3D图形的骨架。在支持DX8和DX9规格的GPU中，这些工作由硬件实现的Vertex Shader(定点着色器)完成。

光栅化计算:显示器实际显示的图像是由像素组成的，我们需要将上面生成的图形上的点和线通过-定的算法转换到相应的像素点。把一个矢量图形转换为一系列像素点的过程就称为光栅化。例如，一条数学表示的斜线段，最终被转化成阶梯状的连续像素点。

纹理帖图:顶点单元生成的多边形只构成了3D物体的轮廓，而纹理映射(texture mapping)工作完成对多变形表面的帖图，通俗的说，就是将多边形的表面贴上相应的图片，从而生成"真实"的图形。TMU(Texture mapping unit)即是用来完成此项工作。

像素处理:这阶段(在对每个像素进行光栅化处理期间)GPU完成对像素的计算和处理，从而确定每个像素的最终属性。在支持DX8和DX9规格的GPU中，这些工作由硬件实现的Pixel Shader(像素着色器)完成。

最终输出:由ROP(光栅化引擎)最终完成像素的输出，1帧染完毕后，被送到显存缓冲区。总结:GPU的工作通俗的来说就是完成3D图形的生成，将图形映射到相应的像素点上，对每个像素进行计算确定最终颜色并完成输出。

什么是渲染管道?

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

主要步骤有: 本地坐标->视图坐标->背面裁剪->光照->裁剪->投影->视图变换->光栅化。

如何优化内存?

1. 压缩自带类库;
2. 将暂时不用的以后还需要使用的物体隐藏起来而不是直接Destroy掉;
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，使用多层次细节(LOD)，使用着色器(Shader)，使用预设(Prefab)
6. 代码中少产生临时变量

动态加载资源的方式?

- instantiate:最简单的一种方式，以实例化的方式动态生成一个物体，
- Assetsbundle:即将资源打成 asset bundle 放在服务器或本地磁盘，然后使用WWW模块get 下来，然后从这个bundle中load某个obiect，unity官方推荐也是绝大多数商业化项目使用的一种方式。
- Resource.Load:可以直接load并返回某个类型的Object，前提是要把这个资源放在Resource命名的文件夹下，Unity不管有没有场景引用，都会将其全部打入到安装包中
- AssetDatabase.loadasset :这种方式只在editor范围内有效，游戏运行时没有这个函数，它通常是在开发中调试用的。

使用Unity3d实现2d游戏，有几种方式?

1.使用本身的GUI、UGUI
2.把摄像机的Projection(投影)值调为Orthographic(正交投影)，不考虑z轴;
3.使用2d插件，如:2DToolKit、NGUI

在物体发生碰撞的整个过程中，有几个阶段，分别列出对应的函数 三个阶段

OnCollisionEnter、OnCollisionStay、OnCollisionExit

Unity3d的物理引擎中，有几种施加力的方式，分别描述出来

- rigidbody.AddForce
- rigidbody.AddForceAtPosition

什么叫做链条关节?

Hinge Joint，可以模拟两个物体间用一根链条连接在一起的情况，能保持两个物体在一个固定距离内部相互移动而不产生作用力，但是达到固定距离后就会产生拉力。

物体自身旋转使用的函数?

Transform.Rotate()

Unity3d提供了一个用于保存和读取数据的类(PlayerPrefs)，请列出保存和读取整形数据的函数

PlayerPrefs.SetInt()、PlayerPrefs.GetInt()

Unity3d脚本从唤醒到销毁有着一套比较完整的生命周期，请列出系统自带的几个重要的方法。

Awake-->Start-->Update-->FixedUpdate-->LateUpdate-->OnGUl-->OnDisable->OnDestroy

主要执行顺序：
编辑器->初始化->物理系统->输入事件->游戏逻辑->场景渲染->GUI渲染->物体激活或禁用->销毁物体>应用结束

主要函数介绍：

- Reset 是在用户点击检视面板的Reset按钮或者首次添加该组件时被调用。此函数只在编辑模式下被调用。Reset最常用于在检视面板中给定一个最常用的默认值。
- Awake 用于在游戏开始之前初始化变量或游戏状态。在脚本整个生命周期内它仅被调用一次.Awake在所有对象被初始化之后调用，所以你可以安全的与其他对象对话或用诸如GameObject.FindWithTag 这样的函数搜索它们。每个游戏物体上的Awke以随机的顺序被调用。因此，你应该用Awake来设置脚本间的引用，并用Start来传递信息 ,Awake总是在Start之前被调用。它不能用来执行协同程序。
- OnDisable 不能用于协同程序。当对象变为不可用或非激活状态时此函数被调用。
- Start 在behaviour的生命周期中只被调用一次。它和Awake的不同是Start只在脚本实例被启用时调用。
- 你可以按需调整延迟初始化代码。Awake总是在Start之前执行。这允许你协调初始化顺序。
- FixedUpdate 当MonoBehaviour启用时，其在每一帧被调用。处理Rigidbody时，需要用FixedUpdate代替Update。例如:给刚体加一个作用力时，你必须应用作用力在FixedUpdate里的固定帧，而不是Update中的帧。(两者帧长不同)。
- ·OnTriggerEnter 可以被用作协同程序，在函数中调用yield语句。当Colider(碰撞体)进入trigger(触发器)时调用OnTriggerEnter。
- OnCollisionEnter 相对于OnTriggerEnter，传递的是Colision类而不是Colider。Colision包含接触点，碰撞速度等细节。如果在函数中不使用碰撞信息，省略colisionlnfo参数以避免不必要的运算。注意如果碰撞体附加了一个非动力学刚体，只发送碰撞事件。可以被用作协同程序。当鼠标在GUIElement(GUI元素)或Colider(碰撞体)上点击时调用OnMouseDown。·Update 是实现各种游戏行为最常用的函数。
- yield 一个协同程序在执行过程中,可以在任意位置使用vield语句。vield的返回值控制何时恢复协同程序向下执行。协同程序在对象自有帧执行过程中堪称优秀。协同程序在性能上没有更多的开销。StartCoroutine函数是立刻返回的,但是yield可以延迟结果。直到协同程序执行完毕
- LateUpdate 是在所有Update函数调用后被调用。这可用于调整脚本执行顺序。例如:当物体在Update里移动时，跟随物体的相机可以在LateUpdate里实现。渲染和处理GUI事件时调用。这意味着你的0nGU!程序将会在每一帧被调用。要得到更多的GUI事件的信息查阅Event手册。如果Monobehaviour的enabled属性设为false，OnGUl()将不会被调用。
- OnApplicationQuit，当用户停止运行模式时在编辑器中调用。当web被关闭时在网络播放器中被调用。

物理更新一般放在哪个系统函数里?

FixedUpdate，每固定帧绘制时执行一次，和Update不同的是FixedUpdate是渲染帧执行，如果你的渲染效率低下的时候FixedUpdate调用次数就会跟着下降。
FixedUpdate比较适用于物理引擎的计算，因为是跟每帧渲染有关。 Update就比较适合做控制。

在场景中放置多个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。
28.如何销毁-个UnityEngine.Object及其子类?使用Destroy()方法;

请描述游戏动画有哪几种，以及其原理?

主要有关节动画、骨骼动画、单一网格模型动画(关键帧动画)。
关节动画:把角色分成若干独立部分，一个部分对应一个网格模型，部分的动画连接成一个整体的动画，角色比较灵活，Quake2中使用这种动画;
骨骼动画，广泛应用的动画方式，集成了以上两个方式的优点，骨骼按角色特点组成一定的层次结构，有关节相连，可做相对运动，皮肤作为单一网格蒙在骨骼之外，决定角色的外观;
单一网格模型动画由一个完整的网格模型构成，在动画序列的关键帧里记录各个顶点的原位置及其改变量，然后插值运算实现动画效果，角色动画较真实。

请描述为什么Unity3d中会发生在组件上出现数据丢失的情况

一般是组件上绑定的物体对象被删除了

alpha blend工作原理?

Alpha Blend 实现透明效果，不过只能针对某块区域进行alpha操作，透明度可设.。

alpha blend工作原理?

Alpha Blend 实现透明效果，不过只能针对某块区域进行alpha操作，透明度可设

写出光照计算中的diffuse的计算公式?

difuse = Kd x colorLight x max(N*L,0);Kd 漫反射系数、colorLight光的颜色、N 单位法线向量、L由点指向光源的单位向量、其中N与L点乘，如果结果小于等于0，则漫反射为0。

LOD是什么，优缺点是什么?

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

缺点:增加了内存

两种阴影判断的方法、工作原理?

本影和半影:

- 本影:景物表面上那些没有被光源直接照射的区域(全黑的轮廓分明的区域)
- 半影:景物表面上那些被某些特定光源直接照射但并非被所有特定光源直接照射的区域(半明半暗区域)工作原理:从光源处向物体的所有可见面投射光线，将这些面投影到场景中得到投影面，再将这些投影面与场景中的其他平面求交得出阴影多边形，保存这些阴影多边形信息，然后再按视点位置对场景进行相应处理得到所要求的视图(利用空间换时间，每次只需依据视点位置进行一次阴影计算即可，省去了一次消隐过程)

Vertex Shader是什么，怎么计算?

顶点着色器 是一段执行在GPU上的程序，用来取代fixed pipeline中的transformation和lighting,Vertex Shader主要操作顶点。

Vertex Shader对输入顶点完成了从local space到homogeneous space(齐次空间)的变换过程，homogeneous space即projection space的下一个space。在这其间共有worldtransformation, view transformation和projection transformation及lighting几个过程

MipMap是什么，作用?

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

请描述Interface与抽象类之间的不同

语法不同处:
1.抽象类中可以有字段，接口没有。
2.抽象类中可以有实现成员，接口只能包含抽象成员。
3.抽象类中所有成员修饰符都可以使用，接口中所有的成员都是对外的，所以不需要修饰符修饰。

用法不同处:
1.抽象类是概念的抽象，接口关注于行为。
2.抽象类的子类与父类的关系是泛化关系，耦合度较高，而实现类和接口之间是实现的关系，耦合度比泛化低。
3.一个类只能继承一个类，但是可以实现多个接口。

.Net与Mono的关系?

mono是.net的一个开源跨平台工具，就类似java虚拟机，java本身不是跨平台语言，但运行在虚拟机上就能够实现了跨平台。

.net只能在windows下运行，mono可以实现跨平台编译运行，可以运行于Linux，Unix，MacOs等

简述Unity3D支持的作为脚本的语言的名称?

Unity的脚本语言基于Mono的.Net平台上运行，可以使用.NET库，这也为XML、数据库、正则表达式等问题提供了很好的解决方案。

Unity里的脚本都会经过编译，他们的运行速度也很快。这三种语言实际上的功能和运行速度是一样的，区别主要体现在语言特性上。

Unity支持的语言:C#，JavaScrip(不在使用)

Unity3D是否支持写成多线程程序?如果支持的话需要注意什么?

支持:如果同时你要处理很多事情或者与Unity的对象互动小可以用thread,否则使用coroutine。

Unity3d没有多线程的概念，不过unity也给我们提供了StartCoroutine(协同程序)和LoadLevelAsync(异步加载关卡)后台加载场景的方法。

注意:仅能从主线程中访问Unity3D的组件，对象和Unity3D系统调用。C#中有lock这个关键字,以确保只有一个线程可以在特定时间内访问特定的对象

如何让已经存在的GameObject在LoadLevel后不被卸载掉?

```
DontDestroyOnLoad(transform.gameObject);
```

U3D中用于记录节点空间几何信息的组件名称，及其父类名称

Transform 父类是 Component

向量的点乘、又乘以及归一化的意义?

- 叉乘 几何意义:得到一个与这两个向量都垂直的向量，这个向量的模是以两个向量为边的平行四边形的面积
- 点乘 几何意义:可以用来表征或计算两个向量之间的夹角，以及在b向量在a向量方向上的投影

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.又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

矩阵相乘的意义及注意点?

用于表示线性变换:旋转、缩放、投影、平移、仿射
注意矩阵的蠕变:误差的积累!

当一个细小的高速物体撞向另一个较大的物体时，会出现什么情况?如何避免?

穿透(碰撞检测失败)(例如CS射击游戏，可以使用开枪时发射射线，射线碰撞到则掉血击中)

为何大家都在移动设备上寻求U3D原生GUI的替代方案

不美观，OnGui很耗费时间，使用不方便

请简述如何在不同分辨率下保持UI的一致性

多屏幕分辨率下的UI布局一般考虑两个问题:
1.布局元素的位置，即屏幕分辨率变化的情况下，布局元素的位置可能固定不动，导致布局元素可能超出边界;
2.布局元素的尺寸，即在屏幕分辨率变化的情况下，布局元素的大小尺寸可能会固定不变，导致布局元素之间出现重叠等功能。
为了解决这两个问题，在Unity GUI体系中有两个组件可以来解决问题，分别是布局元素的RectTransform和Canvas的Canvas Scaler组件。

请简述OnBecameVisible及OnBecameInvisible的发生时机，以及这一对回调函数的意义?

当物体是否可见切换之时。可以用于只需要在物体可见时才进行的计算。

什么叫动态合批?跟静态合批有什么区别?

如果动态物体共用着相同的材质，那么Unity会自动对这些物体进行批处理动态批处理操作是自动完成的，并不需要你进行额外的操作。
不需要做任何操作，而且物体是可以移动的，但是限制很多。静区别:动态批处理一切都是自动的，态批处理:自由度很高，限制很少，缺点可能会占用更多的内存，而且经过静态批处理后的所有物体都不可以再移动了。

Unity提供了几种光源，分别是什么?

四种。
平行光:Directional Light
点光源:Point Light
聚光灯:Spot Light
区域光源:Area Light

什么是LightMap?

LightMap:就是指在三维软件里实现打好光，然后渲染把场景各表面的光照输出到贴图上，最后又通过引擎贴到场景上，这样就使物体有了光照的感觉。

Unity和cocos2d的区别

Unity3D支持C#、javascript等，cocos2d-x支持c++、Html5、Lua等
cocos2d 开源 并且免费
Unity3D支持i0S、Android、Flash、Windows、Mac、Wi等平台的游戏开发，cocos2d-x支持iOSAndroid、WP等。

Unity3D shader分哪几种，有什么区别?

1.表面着色器的抽象层次比较高，它可以轻松地以简洁方式实现复杂着色。表面着色器可同时在前向渲染及延迟渲染模式下正常工作。
2.顶点片段着色器可以非常灵活地实现需要的效果，但是需要编写更多的代码，并且很难与Unity的渲染管线完美集成。
3.固定功能管线着色器可以作为前两种着色器的备用选择，当硬件无法运行那些酷炫Shader的时还可以通过固定功能管线着色器来绘制出一些基本的内容。

获取、增加、删除组件的命令分别是什么?

获取:GetComponent
增加:AddComponent。
删除:Destroy

Unity中，照相机的Clipping Planes的作用是什么?调整Near、Far两个值时，应该注意什么?

剪裁平面 。从相机到开始染和停止渲染之间的 距离。

简述prefab的用处
在游戏运行时实例化，prefab相当于一个模板，对你已经有的素材、脚本、参数做一个默认的配 置以便于以后的修改，同时prefab打包的内容 简化了导出的操作，便于团队的交流。

请描述为什么Unity3d中会发生在组件上出现数据丢失的情况

在Unity3D中，组件上出现数据丢失的情况可能是由于多种原因，其中包括:

1. 对象被删除:当组件所绑定的物体对象被删除时，与之关联的数据也会一并丢失,。
2. 内存不足:如果设备的内存不足以容纳所有数据，可能会导致数据丢失或损坏。
3. 文件损坏:如果Unity3D项目的文件受到损坏，可能会导致组件上的数据出现问题。
4. 未正确保存:如果在编辑器中未正确保存对组件所做的更改，可能会导致数据丢失。
5. 使用了不兼容的插件或脚本:某些第三方插件或脚本

可能与Unity3D编辑器或游戏引擎不兼容从而导致数据丢失或组件功能异常。

为了避免在Unity3D中出现数据丢失的情况，建议定期备份项目、及时保存更改、确保足够的内存和磁盘空间、以及避免使用未经测试或不信任的第三方插件或脚本。

如何在Unity3D中查看场景的面数，顶点数和Draw Call数?如何降低Draw Call数?

在Game视图右上角点击Stats。降低Draw Call 的技术是Draw Call Batching

请问alpha test在何时使用?能达到什么效果?

Alpha Test,中文就是透明度测试。

简而言之就是V&F shader中最后fragment函数输出的该点颜色值(即上一讲frag的输出half4)的alpha值与固定值进行比较。Alpha Test语句通常于Pass,中的起始位置。Alpha Test产生的效果也很极端，要么完全透明，即看不到，要么完全不透明。

四元数有什么作用?

对旋转角度进行计算时用到四元数

将Camera组件的ClearFlags选顶选成Depth only是什么意思?有何用处?

仅深度，该模式用于对象不被裁剪。

如何让已经存在的GameObject在LoadLevel后不被卸载掉?

```
void Awake()
{
    DontDestroyOnLoad(transform.gameObject);
}
```

在编辑场景时将Game0bject设置为Static有何作用?

设置游戏对象为Static将会剔除(或禁用)网格对象当这些部分被静态物体挡住而不可见时。因此在你的场景中的所有不会动的物体都应该标记为Static。

有A和B两组物体，有什么办法能够保证A组物体永远比B组物体先渲染?

把A组物体的渲染对列大于B物体的渲染队列

将图片的TextureType选项分别选为Texture和Sprite有什么区别

Sprite作为UI精灵使用，Texture作用模型贴图使用。

问一个Terrain，分别贴3张，4张，5张地表贴图，渲染速度有什么区别?为什么?

没有区别，因为不管几张贴图只渲染一次。

什么是DrawCall? DrawCall高了又什么影响?如何降低DrawCall?

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

- Dynamic Batching
- Static Batching
- 高级特性Shader降级为统一的低级特性的Shader。

实时点光源的优缺点是什么?

可以有cookies-带有 alpha通道的立方图(Cubemap)纹理。点光源是最耗费资源的。

简述四元数的作用，四元数对欧拉角的优点?

四元数用于表示旋转，对旋转角度进行计算时用到四元数相对欧拉角的优点:
1)能进行增量旋转
2)避免万向锁
3)给定方位的表达方式有两种，互为负(欧拉角有无数种表达方式)

Addcomponent后哪个生命周期函数会被调用

对于AddComponent添加的脚本，其Awake，Start，OnEnable是在Add的当前帧被调用的其中Awake，OnEnable与AddComponent处于同一调用链上Start会在当前帧稍晚一些的时候被调用，Update则是根据Add调用时机决定何时调用:如果Add是在当前帧的Update前调用，那么新脚本的Update也会在当前帧被调用，否则会被延迟到下一帧调用。

层剔除

用layermask，通过位运算的方式去设置

在代码中使用时如何开启某个Layers?
LayerMask mask=1<< 你需要开启的Layers层。
LayerMask mask=0<< 你需要关闭的Layers层,

举几个例子:

- LayerMask mask = 1 << 2; 表示开启Layer2。
- LayerMask mask = 0 << 5;表示关闭Layer5。
- LayerMask mask = 1<<2|1<<8;表示开启Layer2和Layer8。
- LayerMask mask = 0<<3|0<<7;表示关闭Layer3和Layer7。

分别解释顶点着色器和像素着色器是什么

顶点着色器是一段执行在GPU上的程序，用来取代 fixed pipeline中的transformation和lightingVertex Shader主要操作顶点。

像素着色器实际上就是对每一个像素进行光栅化的处理期间，在GPU上运算的一段程序.

不同与顶点着色器，像素着色器不会以软件的形式来模拟像素着色器。

像素着色器实质上是取代了固定功能流水线中多重纹理的环节，而且赋予了我们访问单个像素以及访问每一个像素纹理坐标的能力

画布的三种模式.缩放模式

- **屏幕空间-覆盖模式(Screen $pace-Overlay)**，Canvas创建出来后，默认就是该模式，该模式和摄像机无关，即使场景内没有摄像机，l游戏物体照样渲染

  1.屏幕空间:电脑或者手机显示屏的2D空间，只有x轴和y轴。

  覆盖模式:UI元素永远在3D元素的前面

- **屏幕空间-摄像机模式(Screen $pace-Camera)**，设置成该模式后需要指定一个摄像机游戏物体，指定后UGUI就会自动出现在该摄像机的“投射范围"内，和NGUI的默认U Root效果一致，如果隐藏掉摄像机，UGUI当然就无法渲染
- **世界空间模式(Worldspace)**，设置成该模式后UGUI就相当于是场景内的一个普通的“Cube 游戏模型”，可以在场景内任意的移动UGUI元素的位置，通常用于怪物血条显示和VR开发

缩放模式:

Property:	**Function:**
Ul Scale Mode	Canvas中UI元素的缩放模式
Constant Pixel Size	使UI保持自己的尺寸，与屏幕尺寸无关。
Scale With Screen Size	屏幕尺寸越大，UI越大
Constant Physical Size	使UI元素保持相同的物理大小，与屏幕尺寸无关

Constant Pixel Size、Constant Physical Size实际上他们本质是一样的，只不过Constant Pixel Size 通过逻辑像素大小调节来维持缩放，而 Constant Physical Size通过物理大小调节来维持缩放。

FSM有限状态机

FSM是一种数据结构，它由以下几个部分组成:

1. 内在的所有状态(必须是有限个)
2. 输入条件
3. 状态之间起到连接性作用的转换函数

为什么要用FSM?

因为它编程快速简单，易于调试，性能高，与人类思维相似从而便于梳理，灵活且容易修改

FSM的描述性定义:
一个有限状态机是一个设备，或是一个模型，具有有限数量的状态。它可以在任何给定时间根据输入进行操作，使得系统从一个状态转换到另一个状态，或者是使一个输出或者一种行为的发生，一个有限状态机在任何瞬间只能处于一种状态，

State 状态基类，定义了基本的Enter，Update，Exit三种状态行为，通常在这三种状态行为的方法里会写一些逻辑。每个State都会有StateID(状态id，可以是枚举等)，FSMControl(控制该状态的状态控制器的引用)，Check方法(用来进行状态判断，并返回StateID，通过FSMControl驱动)

FSMControl，包含了一下FSMMachine，封装层。
FSMMachine，驱动它的State列表，Update方法调用当前State的Check方法来获得StateID，当currentState的Check方法返回的StatelD和当前StatelD不同，则切换状态

这是一个简单的FSM状态机系统，根据需要自己写个Control继承FSMControl来驱动状态。因为Check是State的职责，所以每一个不同对象的行为如Human的Idle和Dog的Idel区分肯定也不同。因此需要分别去写HumanldleState和DogldleState。如果还有Cat，Fish，可想而知代码量会有多么庞大

因此我将FSMControl抽象为一个公共基类，把State的Check具体实现作为FSMControl的Virtual方法。这样在ldleState里的Check方法就不用写具体的状态切换判断逻辑，而是调用它FSMControl子类(自己写的继承自FSMControl的Control类)的重写方法

这样每次添加的新对象只要有Ide这个状态，就可以用一个公用的StateIdle，状态切换的逻辑差异放在Control层

使用过哪些Unity插件

因人而异，可以去简单了解一下要说的插件，没用过也可以，至少你知道这个插件了!

插件名	作用
shader graph	制作shader光影效果
cinemachine+timeline+postprocessingstack	制作过场动画
nodecanvas	制作怪物ai
easytouch	手游触摸控制
DoTween	动画插件
Fungus	对话插件
3D WebView	浏览器插件
Vectrosity	划线插件
AVPro Video	视频播放插件

物理系统

CharacterController和Rigidbody的区别

Rigidbody具有完全真实物理的特性，而CharacterController可以说是受限的Rigidbody，具有一定的物理效果但不是完全真实的。

射线检测碰撞物的原理是?

答:射线是3D世界中一个点向一个方向发射的一条无终点的线，在发射轨迹中与其他物体发生碰撞时，它将停止发射 。

什么叫做链条关节?

Hinge Joint，可以模拟两个物体间用一根链条连 接在一起的情况，能保持两个物体在一个固定距 离内部相互移动而不产生作用力，但是达到固定 距离后就会产生拉力。

物体发生碰撞的必要条件?

两个物体都必须带有碰撞器Collider，其中一个物体还必须带有Rigidbody刚体

在物体发生碰撞的整个过程 中，有几个阶段，分别列出对应的函数 三个阶段

1.OnCollisionEnter
2.OnCollisionStay
3.OnCollis ionExit

Unity3d中的碰撞器和触发器的 区别?

碰撞器是触发器的载体，而触发器只是碰撞器身 上的一个属性。当Is Trigger=false时，碰撞器根据物理引擎引发 碰撞，产生碰撞的效果，可以调用 OnColisionEnter/Stay/Exit函数;
当Is Trigger=true时，碰撞器被物理引擎所忽略，没有碰撞效果，可以调用OnTriggerEnter/Stay/ Exit函数。如果既要检测到物体的接触又不想让碰撞检测影 响物体移动或要检测一个物件是否经过空间中的 某个区域这时就可以用到触发器

射线检测碰撞物的原理是?

射线是3D世界中一个点向一个方向发射的一条无,终点的线，在发射轨迹中与其他物体发生碰撞 时，它将停止发射

Unity3d的物理引擎中，有几种 施加力的方式，分别描述出来

rigidbody.AddForce/AddForceAtPosition，都在 rigidbody系列函数中。

当一个细小的高速物体撞向另一个较大的物体时，会出现什么情况?如何避免?

穿透(碰撞检测失败)

射线Raycast原理

从一个起点向一个方向发射一条物理射线，返回碰撞到的物体的碰撞信息

UI&2D部分

UGUI 合批的一些问题

简单来说在一个Canvas下，需要相同的material，相同的纹理以及相同的Z值。例如UI上的字体Texture使用的是字体的图集，往往和我们自己的U图集不一样，因此无法合批。还有U的动态更新会影响网格的重绘，因此需要动静分离。

Image和RawImage的区别

。Imgae比RawImage更消耗性能
·Image只能使用Sprite属性的图片，但是RawImage什么样的都可以使用
。Image适合放一些有操作的图片，裁剪平铺旋转什么的，针对Image Type属性
RawImage就放单独展示的图片就可以，性能会比Image好很多

使用Unity3d实现2d游戏，有几种方式?

1.使用本身UGUI，UGUI是duUnity官方推出zhi的最新UI系统，UI就是UserInterface。
2.把摄像机的投影改为正交投影，不考虑Z轴
3.使用Untiy自身的2D模式，在2d模式中，层级视图中只有一个正交摄像机，场景视图选择的是2D模式
4.使用2D TooKit插件，2D Toolkit是一组与Unity环境无缝集成的工具，提供高效的2D精灵和文本系统。

将图片的TextureType选项分别选为Texture和Sprite有什么区别

Sprite作为UI精灵使用，Texture作用模型贴图使用。

请简述如何在不同分辨率下保 持UI的一致性

屏幕分辨率的自适应 性，原理就是计算出屏幕的宽高比跟原来的预设 的屏幕分辨率求出一个对比值，然后修改摄像机 的size。

动画系统

请描述游戏动画有哪几种，以及其原理?

主要有关节动画、骨骼动画、单一网格模型动画(关键 帧动画)。
关节动画:把角色分成若干独立部分，一个 部分对应一个网格模型，部分的动画连接成一个整体 的动画角色比较灵活，Quake2中使用这种动画;
骨骼动画，广泛应用的动画方式，集成了以上两个方 式的优点，骨骼按角色特点组成一定的层次结构有关节相连，可做相对运动，皮肤作为单一网格蒙在骨 骼之外，决定角色的外观;
单一网格模型动画由一个完整的网格模型构成，在动 画序列的关键帧里记录各个顶点的原位置及其改变量，然后插值运算实现动画效果，角色动画较真实。

Avator的作用

用户提供的模型骨架和Unity的骨架结构进行适配，是一种骨架映射关系。方便动画的重定向
AnimationType有三种类型Humanoid人型:可以动画重定向，游戏对象挂载animator，子类原始模型+重定向模型，设置原始模型和使用模型的AnimationType为Humanoid类型
Generic非人型
Legacyl日版
Avator Mask身体遮罩，身体某一部分是否受到动画影响反向动力学 IK，通过手或脚来控制身体其他部分

反向旋转动画的方法是什么?

1.将动画速度调成-1
2.改代码animation.speed=-1

Animation.CrossFade 是什么?

动画淡入淡出

协程

Unity 协程 Coroutine 的作用

协程Coroutine在Unity中一直扮演者重要的角色。可以实现简单的计时器、将耗时的操作拆分成几个步骤分散在每一帧去运行等等，而尽量不阻塞主线程运行。

什么是协同程序?

在主线程运行时同时开启另一段逻辑处理，来协助当前程序的执行。换句话说，开启协程就是开启一个线程。可以用来控制运动、序列以及对象的行为，

Unity3D的协程和C#线程 之间的区别是什么?

多线程程序同时运行多个线程，而在任一指定时刻只 有一个协程在运行，并且这个正在运行的协同程序只 在必要时才被挂起。
除主线程之外的线程无法访问Unity3D的对象、组件、方法。Unitv3d没有多线程的概念，不过unitv也给我们提供了 StartCoroutine(协同程序)和LoadLevelAsync(异步 加载关卡)后台加载场景的方法。
StartCoroutine为什 么叫协同程序呢，所谓协同，就是当你在 StartCoroutine的函数体里处理一段代码时，利用yield 语句等待执行结果，这期间不影响主程序的继续执 行，可以协同工作。

协同程序的执行代码是什么?有何用处，有何缺点?

```
//(官方案例)
function Start() {
// - After 0 seconds, prints "Starting 0.0"
// - After 0 seconds, prints "Before WaitAndPrint
Finishes 0.0"
// - After 2 seconds, prints "WaitAndPrint 2.0" // 先打印"Starting 0.0"和"Before
WaitAndPrint
Finishes 0.0"两句,2秒后打印"WaitAndPrint 2.0" print ("Starting " + Time.time );
// Start function WaitAndPrint as a coroutine. And continue execution while it is
running
// this is the same as WaintAndPrint(2.0) as the compiler does it for you
automatically
// 协同程序WaitAndPrint在Start函数内执行,可以视 同于它与Start函数同步执行.
StartCoroutine(WaitAndPrint(2.0));
print ("Before WaitAndPrint Finishes " + Time.time );
}
function WaitAndPrint (waitTime : float) {
// suspend execution for waitTime seconds // 暂停执行waitTime秒
yield WaitForSeconds (waitTime);
```

```
print ("WaitAndPrint "+ Time.time );
}
```

作用:一个协同程序在执行过程中,可以在任意位置使用yield语句。yield的返回值控制何时恢复协同程序向 下执行。协同程序在对象自有帧执行过程中堪称优 秀。协同程序在性能上没有更多的开销。 缺点:协同程序并非真线程，可能会发生堵塞。

数据持久化&资源管理

unity常用资源路径有哪些

```
//获取的目录路径最后不包含  /
//获得的文件路径开头包含 /
Application.dataPath; //Asset文件夹的绝对路径
//只读
Application.streamingAssetsPath;  //StreamingAssets文件夹的绝对路径（要先判断是否存在这个文件夹路径）
Application.persistentData ; //可读写

//资源数据库 (AssetDatabase) 是允许您访问工程中的资源的 API
AssetDatabase.GetAllAssetPaths; //获取所有的资源文件（不包含meta文件）
AssetDatabase.GetAssetPath(object) //获取object对象的相对路径
AssetDatabase.Refresh(); //刷新
AssetDatabase.GetDependencies(string); //获取依赖项文件


Directory.Delete(p, true); //删除P路径目录
Directory.Exists(p);  //是否存在P路径目录
Directory.CreateDirectory(p); //创建P路径目录

AssetDatabase //类库，对Asset文件夹下的文件进行操作，获取相对路径，获取所有文件，获取相对依赖项
Directory //类库，相关文件夹路径目录进行操作，是否存在，创建目录，删除等操作
```

如何安全的在不同工程间安全 地迁移asset数据?三种方法

1.将Assets目录和Library目录一起迁移
2.导出包
3.用unity自带的assets Server功能

unity 提供了一个用于保存读取数据的类，(playerPrefs)，请列出保存读取整形数据的函数

PlayerPrefs类是一个本地持久化保存与读取数据的类PlayerPrefs类支持3中数据类型的保存和读取，浮点型，整形，和字符串型。
分别对应的函数为:
SetInt();保存整型数据;GetInt();读取整形数据;SetFloat();保存浮点型数据; GetFlost();读取浮点型数据,SetString();保存字符串型数据; GetString();读取字符串型数据

动态加载资源的方式?

·instantiate:最简单的一种方式，以实例化的方式动态生成一个物体。
。Assetsbundle:即将资源打成 asset bundle 放在服务器或本地磁盘，然后使用WWW模块get 下来，然后从这个bundle中load某个object，unity官方推荐也是绝大多数商业化项目使用的一种方式。
Resource.Load:可以直接load并返回某个类型的Object，前提是要把这个资源放在Resource命名的文件夹下，Unity不管有没有场景引用，都会将其全部打入到安装包中AssetDatabase.loadasset:这种方式只在editor范围内有效，游戏运行时没有这个函数，它通常是在开发中调试用的。

AssetsBundle 打包

```
using UnityEditor;
using System.IO;

public class CreateAssetBundles //进行AssetBundle打包
{

    [MenuItem("Assets/Build AssetBundles")]
    static void BuildAllAssetBundles()
 {
    string dir = "AssetBundles";
    if (Directory.Exists(dir) == false)
    {
      Directory.CreateDirectory(dir);
    }

    BuildPipeline.BuildAssetBundles(dir, //路径必须创建
    BuildAssetBundleOptions.ChunkBasedCompression, //压缩类型***
    BuildTarget.StandaloneWindows64);//平台***
 }
}
```

6.AssetBundle加载

1.LoadFromMemory(LoadFromMemoryAsync)
2.LoadFromFile(LoadFromFileAsync)
3.UnityWebRequest
4.LoadAssetsByWWW(LoadFromCacheOrDownload)

第一种

```
IEnumerator Start()
{
  string path = "AssetBundles/wall.unity3d";

  AssetBundleCreateRequest request
=AssetBundle.LoadFromMemoryAsync(File.ReadAllBytes(path));

  yield return request;

  AssetBundle ab = request.assetBundle;

  GameObject wallPrefab = ab.LoadAsset<GameObject>("Cube");
```

```
  Instantiate(wallPrefab);
}
```

第二种

```
IEnumerator Start()
{
string path = "AssetBundles/wall.unity3d";

  AssetBundleCreateRequest request = AssetBundle.LoadFromFileAsync(path);

  yield return request;

  AssetBundle ab = request.assetBundle;

  GameObject wallPrefab = ab.LoadAsset<GameObject>("Cube");

  Instantiate(wallPrefab);
}
```

第三种

```
IEnumerator Start()
{

  string uri = @"http://localhost/AssetBundles/cubewall.unity3d";

  UnityWebRequest request =   UnityWebRequest.GetAssetBundle(uri);

  yield return request.Send();

  AssetBundle ab = DownloadHandlerAssetBundle.GetContent(request);

  GameObject wallPrefab = ab.LoadAsset<GameObject>("Cube");

  Instantiate(wallPrefab);
}
```

第四种WWW(无依赖)

```
private IEnumerator LoadNoDepandenceAsset()
    {
        string path = "";

        if (loadLocal)
        {
#if UNITY_EDITOR_WIN || UNITY_STANDALONE_WIN
            path += "File:///";
#endif
#if UNITY_EDITOR_OSX || UNITY_STANDALONE_OSX
            path += "File://";
#endif
            path += assetBundlePath + "/" + assetBundleName;
```

```
            //www对象
            WWW www = new WWW(path);

            //等待下载【到内存】
            yield return www;

            //获取到AssetBundle
            AssetBundle bundle = www.assetBundle;

            //加载资源
            GameObject prefab = bundle.LoadAsset<GameObject>(assetRealName);

            //Test:实例化
            Instantiate(prefab);
        }
```

第四种WWW(有依赖)

```
using System.Collections;
using System.IO;
using UnityEngine;
using UnityEngine.Networking;

public class LoadAssetsDemo : MonoBehaviour
{
    [Header("版本号")]
    public int version = 1;

    [Header("加载本地资源")]
    public bool loadLocal = true;
    [Header("资源的bundle名称")]
    public string assetBundleName;
    [Header("资源的真正的文件名称")]
    public string assetRealName;

    //bundle所在的路径
    private string assetBundlePath;
    //bundle所在的文件夹名称
    private string assetBundleRootName;

    private void Awake()
    {
        assetBundlePath = Application.dataPath + "/OutputAssetBundle";
        assetBundleRootName =
assetBundlePath.Substring(assetBundlePath.LastIndexOf("/") + 1);

        Debug.Log(assetBundleRootName);
    }

IEnumerator LoadAssetsByWWW()

{
   string path="";
```

```
//判断是不是本地加载
   if(loadLocal)// loadLocal=true为本地资源
   {
#if UNITY_EDITOR_WIN || UNITY_STANDALONE_WIN
        path+="File:///";
#endif
#if UNITY_EDITOR_OSX || UNITY_STANDALONE_OSX
        path+="File://";
#edif
   }
    //获取要加载的资源路径【bundle的总说明文件】
   path+=assetBundle+"/"+assetBundleRootName;
   //加载
   WWW www=WWW.LoadFromCacheOrDownload(path,version);
   yield return www;
   //拿到其中的bundle
   AssetBundle manifestBundle=www.assetsBundle;
   //获取到说明文件
   AssetBundleManifest manifest=manifest.LoadAsset<AssetBundleManifest>
("AssetBundleManifest");
   //获取资源的所有依赖
   string[] dependencies=manifest.GetAllDependencies(assetBundleName);
   //卸载Bubdle和解压出来的manifest对象
   manifestBundle.Unload(true);
   //获取到相对路径
   path =path.Remove(path.LastIndexOf("/")+1);
   //声明依赖的Bundle数组
   AssetBundle[] depAssetBundle=new AssetBundle[dependencies.Length];

   //遍历加载所有的依赖
   for(int i=0;i<dependencies.Length;i++)
{  //获取到依赖Bundle的路径
   string depPath=path+ dependencies[i];
   //获取新的路径进行加载
   www=WWW.LoadFromCacheOrDownload(depPath,version);
   yield return www;
   //将依赖临时保存
   depAssetBundles[i]=www.assetsBundle;

}
   //获取路径
   path+=assBundleName;
   //加载最终资源
   www=WWW.LoadFromCacheOrDownload(path,version);
   //等待下载
   yield return www;
   //获取到真正的AssetBundle
   AssetBundle realAssetBundle=www.assBunle;
   //加载真正的资源
   GameObject prefab=realAssetBundle.LoadAsset<GameObject>(assetBundle);
   //生成
   Instantiate(prefab);

   //卸载依赖
   for(int i-0;i<depAssetBundle.Length;i++)
```

```
{
    depAssetBundle[i].Unload(true);
}
    realAssetBundle.Unload(true);
}

}
```

AssetBundle卸载流程

AssetBundle.Unload(bool)，T
true卸载所有资源
false只卸载没使用的资源，而正在使用的资源与AssetBundle依赖关系会丢失，调用Resources.UnloadUnusedAssets可以知载
或者等场景切换的时候自动调用Resources.UnloadUnusedAssets。

Lua语言和Xlua热更

Lua如何调用C#

三种方式
第一种:官方不推荐
第二种:如果Resource文件下的Lua文件，使用Lua的Require函数即可
第三种:如果Lua文件是下载的，使用自定义Loader可满足

资源如何打包?依赖项列表如何生成?

1.查找指定文件夹ABResource里的资源文件

- Directory.GetFile(资源路径)
- 新建AssetBundleBuild对象0
- 获取资源名称，并赋值对应AB名称0
- 获取各个资源的依赖项:通过UnityEditor.AssetDataBase类获取各个资源的依赖项

2.使用Unity自带的BuildPipeline进行构建AB包

- BuildPipeLine.BuildAssetBundles(输出AB包路径)
- File.WriteAllLines(将依赖项写入文件里)

如何解析版本文件?如何加载AB包资源?具体流程是怎么样的?

1.解析版本文件列表

- File.ReadAllLines(读取文件列表资源路径URL)
- 获取资源名称，获取AB包名称，获取依赖项，字典容器存储0
- 获取Lua文件0

2.加载资源

- 异步加载资源AB包，AssetBundleRequest请求，AssetBundle.LoadFromFileAsync。
- 先检查依赖项，再异步加载AB包依赖项。
- 加载成功后都有对应的回调方法，将资源作为参数传入

4.热更新方案有哪些?以及具体热更流程

1.整包:存放在上SteamingAssets里
一策略:完整更新资源放在包里
-优点:首次更新少
一缺点:安装包下载时间长，首次安装久

2.分包
策略:少部分资源放在包里，大部分更新资源存放在更新资源器中
优点:安装包小，安装时间短，下载快
缺点:首次更新下载解压缩包时间旧

3.适用性
平台规定-海外游戏大部分是使用分包策略
-国内游戏大部分是使用整包策略

4.文件可读写路径
-Application.streamingAssestsPath 只读目录
-Application.persistentDatapath 可读写目录
资源服务器地址URL

5.【从资源服务器】下载单个文件或多个文件
-NetWorking.UnityWebRequest获取URL,HTTP GET,连接资源服务器获取到downloadHander的文件数据Data，完成后会回调方法，将文件Data作为参数传出

6.检查是否初次安装

5.简述Lua实现面向对象的原理

1. 表table就是一个对象，对象具有了标识self，状态等相关操作
2. 使用参数self表示方法的该接受者是对象本身，是面向对象的核心点,冒号操作符可以隐藏该self参数
3. 类(Class):每个对象都有一个原型，原型(lua类体系)可以组织多个对象间共享行为
4. setmetatable(A,{ index=B})把B设为A的原型
5. 继承(Inheritance):Lua中类也是对象，可以从其他类(对象)中获取方法和没有的字段
6. 继承特性:可以重新定义(修改实现)在基类继承的任意方法
7. 多重继承:一个函数function用作 Index元方法，实现多重继承，还需要对父类列表进行查找方法，但多继承复杂性，性能不如单继承，优化，将继承的方法赋值到子类当中
8. 私有性(很少用)基本思想:两个表表示一个对象，第一个表保存对象的状态在方法的闭包中，第二个表用来保存对象的操作(或接口)，用来访问对象本身。使第一个表完成内容私有性。

6.简述Lua有哪8个类型?简述用途

- null空--可以表示无效值，全局变量(默认赋值为nil)，赋值nil ，使其被删除
- number 整数
- table表-
- string 字符
- userdata 自定义
- function 函数

- bool 布尔
- thread线程

7.lua中pairs和ipairs的区别，从table内存分布方面来考虑

在 Lua 中， pairs 和 ipairs 是两个用于遍历表(table)的内置函数。从内存分布的角度来看，它们的主要区别在于如何处理表中的数字键。

1、pairs:

- pairs 遍历表中的所有键，包括字符串键和数字键。
- 当数字键作为迭代器返回时，它们会被转换为字符串形式。例如，如果表中有数字键1，2，3，使用 pairs 遍历时，它们会被返回为字符串键"1"，"2"，"3"
- 因此，使用 pairs遍历表时，字符串和数字键都会被考虑，并且数字键会被转换为字符串。

2、ipairs:

- ipairs 专门用于遍历表的数字键(从1到n的整数)
- 当使用 ipairs遍历表时，只有数字键会被考虑，并且这些数字键不会被转换为字符串。
- 如果表中有非数字的键或没有数字键，ipairs将不会返回任何值。

内存分布方面的考虑:

- 在遍历表的字符串键时，使用 pairs 和 ipairs 的内存消耗是相似的，因为它们都需要存储这些键的值。
- 但是，对于数字键，由于 ipairs 只考虑数字键，所以它通常会使用更少的内存。特别是当表中有大量数字键时，使用 ipairs 可以节省内存。
- 另一方面，如果表中有大量的非数字键或混合类型的键(字符串和数字)，使用 pairs 可能更为合适因为它可以处理所有类型的键。

总结:从内存分布的角度来看，选择 pairs 或 ipairs 取决于你的具体需求。如果你主要关心数字键并且确信表中只有数字键，那么使用 ipairs 可以节省内存。如果你需要处理所有类型的键或不确定表中是否只有数字键，那么使用 pairs可能更为合适。

网络

1.客户端与服务器交互方式有几种?

1. socket通常也称作"套接字",实现服务器和客户端之间的物理连接，并进行数据传输，主要有UDP和TCP两个协议。Socket处于网络协议的传输层。
2. 协议传输的主要有htp协议 和基于http协议的Soap协议(web service),常见的方式是 http 的post 和get请求，web 服务。

2.概述序列化

序列化 简单理解成把对象转换为容易传输的格式的过程。比如，可以序列化一个对象，然后使用HTTP通过Internet在客户端和服务器端之间传输该对象

3.UDP/TCP含义，区别

UDP协议全称是用户数据报协议

1. 面向无连接
2. 面向报文
3. 不可靠性
4. 有单播，多播，广播的功能
5. 头部开销小，传输数据报文时是很高效的。

TCP协议全称是传输控制协议是一种面向连接的、可靠的、基于字节流的传输层通信协议。三次握手、四次挥手

TCP:

1. 面向连接
2. 仅支持单播传输
3. 面向字节流
4. 可靠传输
5. 提供拥塞控制
6. TCP提供全双工通信

4.TCP/IP协议栈各个层次及分别的功能?

网络接口层:这是协议栈的最低层，对应OSI的物理层和数据链路层，主要完成数据的实际发送和接收。网络层:处理分组在网络中的活动，例如路由选择和转发等，这一层主要包括IP协议、ARP、ICMP协议等。传输层:主要功能是提供应用程序之间的通信，这一层主要是TCP/UDP协议。应用层:用来处理特定的应用，针对不同的应用提供了不同的协议，例如进行文件传输时用到的FTP协议，发送email用到的SMTP等。

5.写出WWW的几个方法

- WWW.LoadFromCacheOrDownload:可被用于将Assets Bundles自动缓存到本地磁盘
- Www.Dispose:释放现有的 WWW 对象。
- WWW.isDone:是否完成下载?(只读)
- WWW.progress:下载进度(只读)。

6.Socket粘包

渲染&Shader

优化部分

算法