

# unity面试题

## ♥C#基础

资源更新失败的原因

当我们把热更新资源传到cdn,上传成功后,测试机上下载下来的依然是过去的资源。请问这是为什么?如何解决?

出现这种情况的原因是cdn服务器有缓存,其过期时间一般难以控制。要解决这个问题,要更新cdn缓存,或在下载地址上添加?time=时间戳等字符串,使之不使用缓存而使用最新的文件。

### 随机数

```
public int GenerateRandomCard(){  
    var candidates = new int[] {2,5,6,7,10,13};  
    int resultIndex = Random.Range(0, candidates.Length);  
    return candidates[resultIndex];  
}
```

这段代码的做法可能存在什么隐患?如何解决这个隐患?

答案:计算机中的随机数都是伪随机数,即实际上是一个固定长度的序列。如果仅仅是调用随机的api,则当随机次数足够多的时候,随机结果会重复之前的序列。要解决这个隐患,最简单的方法是重置随机种子。

Unity当中,使用Random.InitState(seed)方法来重置随机种子,这个种子可以取当前的时间;

### 简述面向对象编程原则

纯概念题,回答得越多越好。一般正确答案回答这七个。

首先S.O.L.I.D

S:单一职责设计的类,接口,对象等等,都只有一个单独的职责

O:开闭原则,对修改关闭,对扩展开放

L:里氏替换原则;子类可以扩展父类的功能,但不能修改父类的功能

I:接口隔离原则,一个接口应该只有一个方法

D:依赖倒置原则。高层不应该依赖于底层,高层和底层都应该依赖于抽象。抽象不应该依赖于细节,细节应该依赖于抽象。

迪米特法则:知道得越少越好。即一个类应该保持对其他对象最小程度的了解。

合成复用法则:组合优于继承,能用组合的地方不要用继承。

### 1. 重载和重写的区别

1. 封装、继承、多态所处位置不同,重载在同类中,重写在父子类中。
2. 定义方式不同,重载方法名相同参数列表不同,重写方法名和参数列表都相同。
3. 调用方式不同,重载使用相同对象以不同参数调用,重写用不同对象以相同参数调用。
4. 多态时机不同,重载时编译时多态,重写是运行时多态。

## 2.面向对象的三大特点

1. 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  $a \blacktriangleright b; b \blacktriangleright c$ ; 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。
2. 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。
3. 多态性： 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。

## 3.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.ValueType **引用类型**包含了string, object, class, interface, delegate, array 继承自System.Object

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. 值类型的变量直接存放实际的数据，引用类型的 变量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。

## 4.请简述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
- sealed表明这个类是最后一个派生类，即无法继续继承下去，不能拥有自己的子类。

internal表明作用域范围是本程序集，也就是如果是打成了dll文件进行调用，那么调用处是无法访问到这些内容的。

sealed表明这个类是最后一个派生类，即无法继续继承下去，不能拥有自己的子类。

## 5.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

## 6.请简述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。

## 7.请简述GC（垃圾回收）产生的原因，并描述如何避免？

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

## 8. 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

## 9.请简述关键字Sealed用在类声明和函数声明时的作用

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。

## 10. 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。

## 11. .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。

## 12. 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

## 13.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。

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

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

复制

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

```
1 public int CountNumber(int num) {  
2     if (num == 1 || num == 2) {  
3         return 1;  
4     } else {  
5         return CountNumber(num - 1) + CountNumber(num - 2);  
6     }  
7 }
```

复制

## 16. 冒泡排序（手写代码）

```
1 public static void BubblingSort(int[] array) {  
2  
3     for (int i = 0; i < array.Length; i++){  
4  
5         for (int j = array.Length - 1; j > 0; j--){  
6  
7             if (array[j] < array[j - 1]) {  
8  
9                 int temp = array[j];  
10                array[j] = array[j - 1];  
11                array[j - 1] = temp;  
12            }  
13        }  
14    }  
15 }
```

复制

## 17. 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，入栈动态扩容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）
- **LinkedList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 装填因子： $\alpha = n/m = 0.72$ ，存储的数据N和空间大小M 然后通过哈希桶算法，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。 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。

```

1 //哈希表结构体
2 private struct bucket {
3     public Object key;//键
4     public Object val;//值
5     public int hash_col;//哈希码
6 }
7 //字典结构体
8 private struct Entry {
9     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
10    public int next; // 下一个元素的下标索引, 如果没有下一个就为-1
11    public TKey key; // 存放元素的键
12    public TValue value; // 存放元素的值
13 }
14
15 private int[] buckets; // Hash桶
16 private Entry[] entries; // Entry数组, 存放元素
17 private int count; // 当前entries的index位置
18 private int version; // 当前版本, 防止迭代过程中集合被更改
19 private int freeList; // 被删除Entry在entries中的下标index, 这个位置是空闲的
20 private int freeCount; // 有多少个被删除的Entry, 有多少个空闲的位置

```



```
21 private IEqualityComparer<TKey> comparer; // 比较器
22 private KeyCollection keys; // 存放Key的集合
23 private ValueCollection values; // 存放Value的集合
```

复制

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

## 18. 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

## 19. 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

## 20. 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

以上等等

在以下情况中使用委托：多用于事件处理中

## 21. 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

## 22. 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

## 23. 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。

## 24. 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决。

通过StringBuilder那进行append，这样可以减少内存垃圾

## 25. 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

## 26. 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

## 27. 给定一个存放参数的数组，重新排列数组

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

## 28. 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。

## 29. 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

## 30. 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

## 31. 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□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

## 32. 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。

## 33. foreach迭代器遍历和for循环遍历的区别

如果集合需要foreach遍历，是否可行，存在一定问题 foreach中的迭代变量item是的只读，不能对其进行修改，比如list.Remove (item) 操作 foreach只读的时候记录下来，在对记录做操作，或者直接用for循环遍历 foreach对int[]数组循环已经不产生GC，避免对ArrayList进行遍历

for语句中初始化变量i的作用域，循环体内部可见。通过索引进行遍历，可以根据索引对所遍历集合进行修改 unity中for循环使用lambda表达式注意闭包问题

foreach遍历原理 任何集合类（Array）对象都有一个GetEnumerator()方法，该方法可以返回一个实现了 IEnumerator接口的对象。这个返回的IEnumerator对象既不是集合类对象，也不是集合的元素类对象，它是一个独立的类对象。通过这个实现了 IEnumerator接口对象A，可以遍历访问集合类对象中的每一个元素对象 对象A访问MoveNext方法，方法为真，就可以访问Current方法，读取到集合的元素。

```
1 List<string> list = new List<string>() { "25", "哈3", "26", "花朵" };
2 IEnumerator listEnumerator = list.GetEnumerator();
3 while (listEnumerator.MoveNext())
4 {
5     Console.WriteLine(listEnumerator.Current);
6 }
```

复制

## 34. 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IL 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 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。
12. 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。
13. static 的使用，static 方法只能由类名调用，改变 static 变量。
14. 在构造基类上替代 C++ 初始化列表的方法。
15. Main 方法和 C++ 及 Java 中的 main 函数的声明方式不同，Main 而不能用 main
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# 从 VB 中引入了 foreach 关键字使得可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。
23. C# 中的局部变量在初始化前不能使用。
24. 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
25. 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。

26.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。

### 35. 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

### 36. 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 a Frame，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。

### 37. 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。

### 38. 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。

### 39. 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字和符号的字符编码方案。使动态字体时，Unity将不会预先生成个与所有字体的字符纹理。当需要持亚洲语或者较的字体的时候，若使正常纹理，则字体的纹理将常。

### 40. 简述StringBuilder和String的区别？（字符串处理）

String是字符串常量。StringBuffer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进改变时都需要成个新的String对象，然后将指针指向个新的对象，如果在个循环，不断的改变个对象，就要不断的成新的对象，所以效率很低，建议在不断更改String对象的地不要使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。

## 41. 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StringBuffer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 StringBuider>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

## 42. 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

## 43. 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 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，所以泛型提高了程序的安全。

## 44. 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

## 45. 能用foreach遍历访问的对象需要实现\_\_\_\_\_接口或声明\_\_\_\_\_接口的类型（C#遍历）

IEnumerable; 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方法。



## 46. 什么是里氏替换原则？（C#多态）

里氏替换原则(Liskov Substitution Principle LSP)□□□□□□□□□□□□□□□□

- 里氏替换原则中说，任何基类可以出现的地方，□类□定可以出现，作□□便扩展功能能
- 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。

## 47. 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□。

▼

```
1 实现步骤：
2 1. 导 using System.Reflection;
3 2. Assembly.Load("程序集")加载程序集,返回类型是
4   Assembly
5 3. foreach (Type type in assembly.GetTypes())
6     {
7         string t = type.Name;
8     }
9 得到程序集中所有类的名称
10 4. Type type = assembly.GetType("程序集.类名");获取
11 当前类的类型
12 5. Activator.CreateInstance(type); 创建此类型实例
13 6. MethodInfo mInfo = type.GetMethod("    ");获取
14 当前  法
15 7. mInfo.Invoke(null,    );
```

复制

## 48. 概述c#中代理和事件?

代理就是□来定义指向□法的引□。C#事件本质就是对消息的封装，□作对象之间的通信；发送□叫事件发送器，接收□叫事件接收器；

## 49. 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）。
- Dictionary 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

## 50. 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](#)。 STAThread：是单线程套间的意思，是□种线程模型。 MATAThread：是多线程套间的意思，也是□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□程可以存取。 protected：只有包含该成员的类以及派□类可以存 取。

**类修饰符：** abstract：抽象类。指示□个类只能作为其它类的基 类。 sealed：密封类。指示□个类不能被继承。理所当 然，密封类不能同时□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□法或属性没有实现。 sealed：密封□法。可以防□在派□类中对该□法的 override（□载）。不是类的每个成员□法都可以作为 密封□法密封□法，必须对基类的虚□法进□□载， 提供具体的实现□法。所以，在□法的声明中， sealed修饰符总是和override修饰符同时使□。 delegate：委托。□来定义□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□个事件。 extern：指示□法在外部实现。 override：□写。对由基类继承成员的新实现。 readonly：指示□个域只能在声明时以及相同类的内 部被赋值。 static：指示□个成员属于类型本身，□不是属于特定 的对象。即在定义后可不经实例化，就可使□。 virtual：指示□个□法或存取器的实现可以在继承类中 被覆盖。 new：在派□类中隐藏指定的基类成员，从□实现□ 9写的功能。若要隐藏继承类的成员，请使□相同名称 在派□类中声明该成员，并□ new 修饰符修饰它。

## 51. 下列代码在运行中会发生什么问题？如何避免？

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

```
4     Console.WriteLine(item * item);
5     ls.Remove(item);
6 }
```

复制

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。

## 52. 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。

## 53. 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。
- Controller（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

## 53 .在类的构造函数前加上static会报什么错?为什么?

- 在类的构造函数前加上static会导致编译错误，因为构造函数是用来创建对象的特殊函数，它不能被声明为静态的。静态函数是属于类的，而不是属于对象的，它们可以在没有创建对象的情况下被调用。因此，在构造函数前加上static会违反构造函数的特性，导致编译错误。
- 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。

---

版权声明：本文为CSDN博主「软泡芙」的原创文章，遵循CC 4.0 BY-SA版权协议，转载请附上原文出处链接及本声明。

原文链接：[https://blog.csdn.net/weixin\\_44231544/article/details/128017584](https://blog.csdn.net/weixin_44231544/article/details/128017584)

# Unity基础知识

## 1. Image和RawImage的区别

- Image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

## 2. Unity3D中的碰撞器和触发器的区别？

答：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。

## 3. 物体发生碰撞的必要条件？

答：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。

## 4. 简述四元数Quaternion的作用，四元数对欧拉角的优点？

答：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）

## 5. 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

答：Awake->OnEnable->Start

OnEnable在同一周期中可以反复地发生！

## 7. MeshRender中material和sharedmaterial的区别？

## 6. 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？

答：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。

## 8. 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，主要完成数据帧的实际发送和接收。

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。

## 9. 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

## 10. 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

## 11. 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。

## 12. 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。

## 13. 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。

## 14. 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一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。

**纹理贴图：**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贴图，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帧渲染完毕后，被送到显存帧缓冲区。

总结：GPU的工作通俗的说就是完成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。

## 15. 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。

## 16. 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

## 18. 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。
- Assetbundle：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。

## 19. 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

## 20. 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

答: OnCollisionEnter、OnCollisionStay、OnCollisionExit

## 21. 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

## 22. 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。

## 23. 物体自身旋转使用的函数?

Transform.Rotate()

## 24. 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()

## 25. 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。

答: Awake——>Start——>Update——>FixedUpdate——>LateUpdate——>OnGUI——>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Awake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Collider(碰撞体)进入trigger(触发器)时调用OnTriggerEnter。
- **OnCollisionEnter** 相对于OnTriggerEnter，传递的是Collision类而不是Collider。Collision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collisionInfo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Collider(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。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OnGUI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，OnGUI()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。

## 26. 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。

## 27. 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. 如何销毁一个UnityEngine.Object及其子类？

使用Destroy()方法;

## 29. 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。

## 30. 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

## 31. 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。

## 32. 写出光照计算中的diffuse的计算公式？

$\text{diffuse} = K_d \times \text{colorLight} \times \max(N \cdot L, 0)$ ;  $K_d$  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。

## 33. 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

LOD简单示例：[【100个 Unity踩坑小知识点】 | Unity 的 LOD技术（多细节层次）](#)

## 33. 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）

## 34. 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world transformation, view transformation和projection transformation及lighting几个过程。

### 35. 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。

### 36. 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。

### 37. .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， Mac OS等。

### 38. 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)

### 39. 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

## 40. 如何让已经存在的GameObject在LoadLevel后不被卸载掉？

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

复制

## 41. 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

## 42. 向量的点乘、叉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. 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

## 43. 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

## 44. 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）

## 45. 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

## 46. 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Rect Transform和Canvas的Canvas Scaler组件。

## 47. 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。

## 48. 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。

**区别：**动态批处理一切都是自动的，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。

## 49. 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

## 50. 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。

## 51. 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iOS、Android、Flash、Windows、Mac、Wii等平台的游戏开发，cocos2d-x支持iOS、Android、WP等。

## 52. 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向前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的时候，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。

## 53. 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



## 54. 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 。从相机到开始渲染和停止渲染之间的 距离。

## 55. 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。

## 56. 请描述为什么Unity3d中会发生 在组件上出现数据丢失的情况

剪裁平面 。从相机到开始渲染和停止渲染之间的距离。

## 57. 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

## 58. 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。

## 60. 四元数有什么作用？

对旋转角度进行计算时用到四元数

## 61. 将Camera组件的ClearFlags选项选成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。

## 62. 如何让已经存在的GameObject在LoadLevel后不被卸载掉？

```
1 void Awake()  
2 {  
3     DontDestroyOnLoad(transform.gameObject);  
4 }
```

复制

## 63. 在编辑场景时将GameObject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。

## 64. 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

## 65. 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。

## 66. 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？

答：没有区别，因为不管几张贴图只渲染一次。

## 67. 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。DrawCall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。

## 68. 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。

## 69. 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) 给定位置的表达式有两种，互为负（欧拉角有无数种表达式）

## 70. 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。

## 72. 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：

```
1 LayerMask mask = 1 << 2; 表示开启Layer2。  
2  
3 LayerMask mask = 0 << 5;表示关闭Layer5。  
4  
5 LayerMask mask = 1<<2 | 1<<8;表示开启Layer2和Layer8。  
6
```

7 LayerMask mask = 0<<3|0<<7;表示关闭Layer3和Layer7。

复制

## 73. 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lighting，Vertex 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。

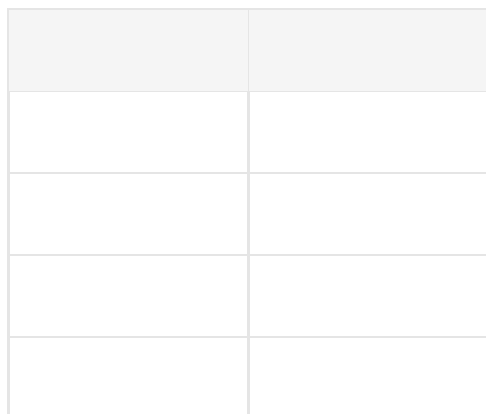
不同与顶点着色器，像素着色器不会以软件的形式来模拟像素着色器。

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

## 74. 画布的三种模式.缩放模式

- **屏幕空间-覆盖模式(Screen Space-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，UI游戏物体照样渲染
  - 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 Space-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UI 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：



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。

## 75. 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StateID和当前StateID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Idle区分肯定也不同。因此需要分别去写HumanIdleState和DogIdleState。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IdleState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Idle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

## 76. 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！

|  |  |
|--|--|
|  |  |
|  |  |
|  |  |
|  |  |

|  |  |
|--|--|
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |

## 物理系统

### 1. CharacterController和Rigidbody的区别

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

### 2. 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。

### 3. 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。

### 4. 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

### 5. 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. OnCollisionExit

### 6. 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

### 7. 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。

## 8. 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。

## 9. 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)

## 10. 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

## 1. 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UI图集不一样，因此无法合批。还有UI的动态更新会影响网格的重绘，因此需要动静分离。

## 2. Image和RawImage的区别

- Image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

## 3. 使用Unity3d实现2d游戏，有几种方式？

1. 使用本身UGUI，UGUI是Unity官方推出的最新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Unity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 Toolkit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。

## 4. 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。

## 5. 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

## 1. 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:骨骼动画的原理是，先定义一个骨骼模型，然后为每个骨骼定义动画，最后将骨骼模型与动画结合，实现动画效果；
- 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。

## 2. 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 Legacy旧版 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

## 3. 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

## 4. Animation.CrossFade 是什么？

动画淡入淡出

## 5. 写出 Animation 的五个方法

- AddClip 将 clip 添加到名称为 newName 的动画中。
- Blend 在后续 time 秒中将名称为 animation 的动画向 targetWeight 混合。
- CrossFade 在后续 time 秒的时间段内，使名称为 animation 的动画淡入，使其他动画淡出。
- CrossFadeQueued 使动画在上一个动画播放完成后交叉淡入淡出。
- IsPlaying 名称为 name 的动画是否正在播放？
- PlayQueued 在先前的动画播放完毕后再播放动画。
- RemoveClip 从动画列表中移除剪辑。
- Sample 对当前状态的动画进行采样。
- Stop 停止所有使用该动画启动的正在播放的动画。



## 6. 简述 SkinnedMesh 的实现原理

SkinnedMesh蒙皮网格动画 分为骨骼和蒙皮两部分 骨骼是一个层次结构，存储了骨骼的 Transform数据 蒙皮是mesh顶点附着在骨骼之上，顶点可以被多个骨骼影响，决定了其权重等，还有将顶点从Mesh空间变换到骨骼空间~

## 7. 动画层(Animation Layers)的作用是什么？

### 动画分层

身体部位动画分层，比如我只想动动头，身体其他部分不发生动画，可以方便处理动画区分

## 协程

### 1. 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。

### 2. 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。

### 3. 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。

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。 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。

### 4. 协同程序的执行代码是什么？有何用处，有何缺点？

```
1 官方案例)
2 function Start() {
3 // - After 0 seconds, prints "Starting 0.0"
4 // - After 0 seconds, prints "Before WaitAndPrint
5 Finishes 0.0"
6 // - After 2 seconds, prints "WaitAndPrint 2.0" // 先打印"Starting 0.0"和"Before WaitAndPrint
7 Finishes 0.0"两句,2秒后打印"WaitAndPrint 2.0" print ("Starting " + Time.time );
8 // Start function WaitAndPrint as a coroutine. And continue execution while it is running
9 // this is the same as WaintAndPrint(2.0) as the compiler does it for you automatically
10 // 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.
11 StartCoroutine(WaitAndPrint(2.0));
12 print ("Before WaitAndPrint Finishes " + Time.time );
13 }
```

```
14 function WaitAndPrint (waitTime : float) {
15 // suspend execution for waitTime seconds // 暂停执行waitTime秒
16 yield WaitForSeconds (waitTime);
17 print ("WaitAndPrint "+ Time.time );
18 }
```

复制

**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。

更多协程内容：[Unity零基础到入门 \\*| 小万字教程 对 Unity 中的 协程 ♥全面解析+实战演练♥](#)

## 数据持久化 & 资源管理

### 1. unity常用资源路径有哪些

```
1 //获取的目录路径最后不包含 /
2 //获得的文件路径开头包含 /
3 Application.dataPath; //Asset文件夹的绝对路径
4 //只读
5 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）
6 Application.persistentData ; //可读写
7
8 //资源数据库 (AssetDatabase) 是允许您访问工程中的资源的 API
9 AssetDatabase.GetAllAssetPaths; //获取所有的资源文件（不包含meta文件）
10 AssetDatabase.GetAssetPath(object) //获取object对象的相对路径
11 AssetDatabase.Refresh(); //刷新
12 AssetDatabase.GetDependencies(string); //获取依赖项文件
13
14
15 Directory.Delete(p, true); //删除P路径目录
16 Directory.Exists(p); //是否存在P路径目录
17 Directory.CreateDirectory(p); //创建P路径目录
18
19 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
20 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
```

复制

### 2. 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

### 3. 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； GetFloat();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；

### 4. 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。

### 5. AssetsBundle 打包

```
1 using UnityEditor;
2 using System.IO;
3
4 public class CreateAssetBundles //进行AssetBundle打包
5 {
6
7     [MenuItem("Assets/Build AssetBundles")]
8     static void BuildAllAssetBundles()
9     {
10         string dir = "AssetBundles";
11         if (Directory.Exists(dir) == false)
12         {
13             Directory.CreateDirectory(dir);
14         }
15
16         BuildPipeline.BuildAssetBundles(dir, //路径必须创建
17         BuildAssetBundleOptions.ChunkBasedCompression, //压缩类型***
18         BuildTarget.StandaloneWindows64); //平台***
```

```
19 }
20 }
```

复制

|  |  |
|--|--|
|  |  |
|  |  |
|  |  |

（压缩率比LZMA低，解压速度接近无压缩）|

## 6. AssetBundle加载

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

### 第一种

```
▼
1 IEnumerator Start()
2 {
3     string path = "AssetBundles/wall.unity3d";
4
5     AssetBundleCreateRequest request =AssetBundle.LoadFromMemoryAsync(File.ReadAllBytes(path));
6
7     yield return request;
8
9     AssetBundle ab = request.assetBundle;
10
11     GameObject wallPrefab = ab.LoadAsset<GameObject>("Cube");
12
13     Instantiate(wallPrefab);
14 }
```

复制

### 第二种

```
▼
1 IEnumerator Start()
```

```

2 {
3 string path = "AssetBundles/wall.unity3d";
4
5 AssetBundleCreateRequest request = AssetBundle.LoadFromFileAsync(path);
6
7 yield return request;
8
9 AssetBundle ab = request.assetBundle;
10
11 GameObject wallPrefab = ab.LoadAsset<GameObject>("Cube");
12
13 Instantiate(wallPrefab);
14 }

```

复制

### 第三种

```

1 IEnumerator Start()
2 {
3
4 string uri = @"http://localhost/AssetBundles/cubewall.unity3d";
5
6 UnityWebRequest request = UnityWebRequest.GetAssetBundle(uri);
7
8 yield return request.Send();
9
10 AssetBundle ab = DownloadHandlerAssetBundle.GetContent(request);
11
12 GameObject wallPrefab = ab.LoadAsset<GameObject>("Cube");
13
14 Instantiate(wallPrefab);
15 }

```

复制

### 第四种WWW（无依赖）

```

1 private IEnumerator LoadNoDependenceAsset()
2 {
3     string path = "";
4
5     if (loadLocal)
6     {
7 #if UNITY_EDITOR_WIN || UNITY_STANDALONE_WIN

```

```

8     path += "File:///";
9 #endif
10 #if UNITY_EDITOR_OSX || UNITY_STANDALONE_OSX
11     path += "File:///";
12 #endif
13     path += assetBundlePath + "/" + assetBundleName;
14
15     //www对象
16     WWW www = new WWW(path);
17
18     //等待下载【到内存】
19     yield return www;
20
21     //获取到AssetBundle
22     AssetBundle bundle = www.assetBundle;
23
24     //加载资源
25     GameObject prefab = bundle.LoadAsset<GameObject>(assetRealName);
26
27     //Test:实例化
28     Instantiate(prefab);
29 }

```

复制

## 第四种WWW（有依赖）

```

1 using System.Collections;
2 using System.IO;
3 using UnityEngine;
4 using UnityEngine.Networking;
5
6 public class LoadAssetsDemo : MonoBehaviour
7 {
8     [Header("版本号")]
9     public int version = 1;
10
11     [Header("加载本地资源")]
12     public bool loadLocal = true;
13     [Header("资源的bundle名称")]
14     public string assetBundleName;
15     [Header("资源的真正的文件名称")]
16     public string assetRealName;
17
18     //bundle所在的路径
19     private string assetBundlePath;
20     //bundle所在的文件夹名称

```



```
21 private string assetBundleRootName;
22
23 private void Awake()
24 {
25     assetBundlePath = Application.dataPath + "/OutputAssetBundle";
26     assetBundleRootName = assetBundlePath.Substring(assetBundlePath.LastIndexOf("/") + 1);
27
28     Debug.Log(assetBundleRootName);
29 }
30
31 IEnumerator LoadAssetsByWWW()
32
33 {
34     string path="";
35     //判断是不是本地加载
36     if(loadLocal)// loadLocal=true为本地资源
37     {
38 #if UNITY_EDITOR_WIN || UNITY_STANDALONE_WIN
39         path+="File:///";
40 #endif
41 #if UNITY_EDITOR_OSX || UNITY_STANDALONE_OSX
42         path+="File:///";
43 #endif
44     }
45     //获取要加载的资源路径【bundle的总说明文件】
46     path+=assetBundle+"/"+assetBundleRootName;
47     //加载
48     WWW www=WWW.LoadFromCacheOrDownload(path,version);
49     yield return www;
50     //拿到其中的bundle
51     AssetBundle manifestBundle=www.assetBundle;
52     //获取到说明文件
53     AssetBundleManifest manifest=manifest.LoadAsset<AssetBundleManifest>("AssetBundleManifest");
54     //获取资源的所有依赖
55     string[] dependencies=manifest.GetAllDependencies(assetBundleName);
56     //卸载Bundle和解压出来的manifest对象
57     manifestBundle.Unload(true);
58     //获取到相对路径
59     path =path.Remove(path.LastIndexOf("/")+1);
60     //声明依赖的Bundle数组
61     AssetBundle[] depAssetBundle=new AssetBundle[dependencies.Length];
62
63     //遍历加载所有的依赖
64     for(int i=0;i<dependencies.Length;i++)
65     { //获取到依赖Bundle的路径
66         string depPath=path+ dependencies[i];
67         //获取新的路径进行加载
68         www=WWW.LoadFromCacheOrDownload(depPath,version);
```

```

69  yield return www;
70  //将依赖临时保存
71  depAssetBundles[i]=www.assetBundle;
72
73 }
74  //获取路径
75  path+=assBundleName;
76  //加载最终资源
77  www=WWW.LoadFromCacheOrDownload(path,version);
78  //等待下载
79  yield return www;
80  //获取到真正的AssetBundle
81  AssetBundle realAssetBundle=www.assetBundle;
82  //加载真正的资源
83  GameObject prefab=realAssetBundle.LoadAsset<GameObject>(assetBundle);
84  //生成
85  Instantiate(prefab);
86
87  //卸载依赖
88  for(int i=0;i<depAssetBundles.Length;i++)
89  {
90      depAssetBundles[i].Unload(true);
91  }
92  realAssetBundle.Unload(true);
93 }
94
95 }

```

复制

## 7. AssetBundle卸载流程

AssetBundle.Unload(bool), 当 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 可以卸载。

或者等场景切换的时候自动调用 Resources.UnloadUnusedAssets。

## Lua 语言和 Xlua 热更

### 1. 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

### 2. 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.GetFiles(资源路径)
- 新建AssetBundleBuild对象
- 获取资源名称，并赋值对应AB名称
- 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)

## 3. 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
- 获取Lua文件

### 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StreamingAssets里 ——策略：完整更新资源放在包里 ——优点：首次更新少 ——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 ——海外游戏大部分是使用分包策略，平台规定 ——国内游戏大部分是使用整包策略
4. 文件可读写路径 ——Application.streamingAssetsPath 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downloadHandler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，但多重继承复杂性，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

- nil 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

# 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http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。例如，可以序列化一个对象，然后使该HTTP通过Internet在客户端和服务端之间传输该对象

## 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, 主要完成数据帧的实际发送和接收。

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

什么是粘包? 答: 顾名思义, 其实就是多个独立的数据包连到一块儿。

**什么情况下需要考虑粘包？** 答：实际情况如下：

1. 如果利用tcp每次发送数据，就与对方建立连接，然后双方发送完一段数据后，就关闭连接，这样就不会出现粘包问题。
2. 如果发送的数据无结构，比如文件传输，这样发送方只管发送，接收方只管接收存储就ok，也不用考虑粘包。
3. 如果双方建立连接，需要在连接后一段时间内发送不同结构数据，如连接后，有好几种结构：  
1)"good good study" 2)"day day up" 那这样的话，如果发送方连续发送这两个包出去，接收方一次接收可能会是"good good studyday day up" 这样接收方就傻了，因为协议没有规定这么奇怪的字符串，所以要把它分包处理，至于怎么分也需要双方组织一个比较好的包结构，所以一般可能会在头加一个数据长度之类的包，以确保接收。

所以说：Tcp连续发送消息的时候，会出现消息一起发送过来的问题，这时候需要考虑粘包的问题。

**粘包出现的原因** (在流传输中，UDP不会出现粘包，因为它有消息边界。)

1. 发送端需要等缓冲区满才发送出去，造成粘包 (发送端出现粘包)
2. 接收端没有及时接收缓冲区包数据，造成一次性接收多个包，出现粘包 (接收端出现粘包)

**解决粘包**

1. 缓冲区过大造成了粘包，所以在发送/接收消息时先将消息的长度作为消息的一部分发出去，这样接收方就可以根据接收到的消息长度来动态定义缓冲区的大小。（这种方法就是所谓的自定义协议，这种方法是最常用的）
2. 对发送的数据进行处理，每条消息的首尾加上特殊字符，然后再把要发送的所有消息放入一个字符串中，最后将这个字符串发送出去，接收方接收到这个字符串之后，再通过特殊标记操作字符串，把每条消息截出来。（这种方法只适合数据量较小的情况）

注：要记住这一点：TCP对上层来说是一个流协议，所谓流,就是没有界限的一串数据.大家可以想想河里的流水,是连成一片的,其间是没有分界线的，也就是没有包的概念。所以我们必须自己定义包长或者分隔符来区分每一条消息。

## **7. Socket的封包、拆包**

1. 为什么基于TCP的通信程序需要封包、拆包？ 答：TCP是流协议，所谓流，就是没有界限的一串数据。但是程序中却有多种不同的数据包，那就很可能会出现如上所说的粘包问题，所以就需要在发送端封包，在接收端拆包。
2. 那么如何封包、拆包？ 答：封包就是给一段数据加上包头或者包尾。比如说我们上面为解决粘包所使用的两种方法，其实就是封包与拆包的具体实现。

## **8. Socket 客户端 队列 的问题**

项目中采用了socket通信，通过TCP发送数据给服务器端，因为项目需要，要同时开启大量的线程去发送不同的数据给服务器端，然后服务器端返回不同的数据。由于操作频繁，经常会阻塞，或没有



接收到服务器端返回的数据；

因此考虑到使用一个队列：将同一ip下的数据存入一个队列中，通过队列协调发送；当第一条数据发送出去没有收到服务器端返回的数据时，让第二条数据插入队列中排队，当第三条数据也发送出来后，继续排队，以此类推；如果当第四条数据发出来的时候，存入队列中，第一条数据收服务器端返回数据后，队列中的第二条第三条数据就扔掉，直接发送第四条数据

## 渲染 & Shader

### 1. 什么是LightMap?

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

### 2. 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组成的件，这样的贴图被称为MipMap。

### 3. 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。

### 4. 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。

### 5. 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。

### 6. 简述水面倒影的渲染原理？

原理就是对水面的贴图纹理进行扰动，以产生波光粼粼的效果。用shader可以通过GPU在像素级别作扰动，效果细腻，需要的顶点少，速度快

### 7. 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。

### 8. 什么是渲染管道？

是指在显示器上为了显示出图像经过的一系列必要操作。渲染管道中的很多步骤，都要将任何物体从哪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。

**GPU工作流程：**顶点处理、光栅化、纹理贴图、像素处理



space。在这其间共有world transformation, view transformation和projection transformation及lighting

## 15. Unity3D Shader分哪几种，有什么区别？

表着器的抽象层次较，它可以轻松地以简洁式实现复杂着。表着器可同时在前向渲染及延迟渲染模式下正常工作。

顶点段着器可以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。

固定功能管线着器可以作为前两种着器的备选选择，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器来绘制出些基本的内容。

## 优化部分

更多优化知识学习文章：[【Unity 优化篇】 | 优化专栏《导航帖》，全面学习Unity优化技巧，让我们的Un](#)

### 1. 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个空间，如游戏中要常被大量复制的对象，子弹，敌，以及任何重复出现的对象。

### 2. 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。DrawCall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。

### 3. UI优化小知识

1. 将同一画面图片放到同一图集中
2. 图片和文字尽量不要交叉，会产生多余drawcall（相同材质和纹理的UI元素是可以合并的）
3. UI层级尽量不要重叠太多
4. 取消勾选不必要的射线检测RaycastTarget
5. 将动态的UI元素和静态的UI元素放在不同的Canvas中，减少canvas网格重构频率

### 4. 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。

### 5. 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。

**区别：**动态批处理一切都是自动的，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。

## 6. 如何优化内存？

有很多种方式，例如

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)。

## 7. 请简述GC（垃圾回收）产生的原因，并描述如何避免？

GC垃圾回收机制，避免堆内存溢出，定期回收那些没有有效引用的对象内存 GC优化，就是优化堆内存，减少堆内存，即时回收堆内存 GC属于CLR

避免：

1. 减少new的次数
2. 字符串拼接使用stringbuilder，字符串比较先定义一个变量存储，防止产生无效内存
3. list，new时候，规定内存大小
4. 如果要射线检测，应该使用避免GC的方法XXXXNoAlloc函数
5. foreach迭代器容易导致GC（目前Unity5.5已修复），使用For循环
6. 使用静态变量，GC不会回收存在的对象，但静态变量的引用对象可能被回收
7. 使用枚举替代字符串变量
8. 调用gameObject.tag=="XXX"就会产生内存垃圾；那么采用GameObject.CompareTag()可以避免内存垃圾的产生：
9. 不要在频繁调用的函数中反复进行堆内存分配，比如OnTriggerXXX，Update等函数
10. 在Update函数中，运行有规律的但不需要每一帧执行的代码，可以使用计时器，比如1秒执行一次某些代码！！

更多GC内容可查看本篇文章：[Unity零基础到进阶 \\*| Unity中的 GC及优化 超级全面解析 ☆\(>ω<\)v 建议收藏！](#)

## 8. 贴图透明通道分离，压缩格式设为ETC/PVRTC

最初我们使用了DXT5作为贴图压缩格式，希望能减小贴图的内存占用，但很快发现移动平台的显卡是不支持的。因此对于一张1024x1024大小的RGBA32贴图，虽然DXT5可将它从4MB压缩到1MB，但系统将它送进显卡之前，会先用CPU在内存里将它解压成4MB的RGBA32格式（软件解压），然后再将这4MB送进显存。于是在这段时间里，这张贴图就占用了5MB内存和4MB显存；而移动平台往往没有独立显存，需要从内存里抠一块作为显存，于是原以为只占1MB内存的贴图实际却占了9MB！

所有不支持硬件解压的压缩格式都有这个问题。经过一番调研，我们发现安卓上硬件支持最广泛的格式是ETC，苹果上则是PVRTC。但这两种格式都是不带透明（Alpha）通道的。因此我们将每张原始贴图的透明通道都分离了出来，写进另一张贴图的红色通道里。这两张贴图都采用ETC/PVRTC压缩。渲染的时候，将两张贴图都送进显存。同时我们修改了NGUI的shader，在渲染时将第二张贴图的红色通道写到第一张贴图的透明通道里，恢复原来的颜色：

```
1 fixed4 frag (v2f i) : COLOR
2
3     fixed4 col;
4
5     col.rgb = tex2D(_MainTex, i.texcoord).rgb;
6
7     col.a = tex2D(_AlphaTex, i.texcoord).r;
8
9     return col * i.color;
10
11 fixed4 frag (v2f i) : COLOR
12
13 {
14
15     fixed4 col;
16
17     col.rgb = tex2D(_MainTex, i.texcoord).rgb;
18
19     col.a = tex2D(_AlphaTex, i.texcoord).r;
20
21     return col * i.color;
22
23 }
```

复制

## 9. 关闭贴图的读写选项

Unity中导入的每张贴图都有一个启用可读可写（Read/Write Enabled）的开关，对应的程序参数是TextureImporter.isReadable。选中贴图后可在Import Setting选项卡中看到这个开关。只有打开这个开关，才可以对贴图使用Texture2D.GetPixel，读取或改写贴图资源的像素，但这就需要

系统在内存里保留一份贴图的拷贝，以供CPU访问。一般游戏运行时不会有这样的需求，因此我们对所有贴图都关闭了这个开关，只在编辑中做贴图导入后处理（比如对原始贴图分离透明通道）时打开它。这样，上文提到的1024x1024大小的贴图，其运行时的2MB内存占用又可以少一半，减小到1MB。

## 10. Unity 在移动设备上的些优化资源的方法

1. 使用assetbundle，实现资源分离和共享，将内存控制到200m之内，同时也可以实现资源的在线更新
2. 顶点数对渲染无论是cpu还是gpu都是压倒性的贡献者，降低顶点数到8万以下，fps稳定到了30帧左右
3. 只使用动态光，不是阴影，不使用光照探头粒子系统是cpu上的负担
4. 剪裁粒子系统
5. 合并同时出现的粒子系统
6. animator也是效率奇差的地方
7. 把不需要跟骨骼动画和动作过渡的地方全部使用animation，控制骨骼数在30根以下
8. animator出视口不更新
9. 删除无意义的animator
10. animator的初始化很耗时（粒子上能不能尽量不要animator）
11. 除主角外都不要跟骨骼运动apply root motion
12. 绝对禁用掉那些不带刚体带包围盒的物体（static collider）运动 NUGI的代码效率很差，基本上runtime的时候对cpu的贡献和render不相上下
13. 每帧递归的计算finalalpha改为只有初始化和变动时计算
14. 去掉法线计算
15. 不要每帧计算viewsize 和window size
16. filldrawcall时构建顶点缓存使用array.copy
17. 代码剪裁：使用strip level，使用.net2.0 subset
18. 尽量减少smooth group
19. 给美术定一个严格的经过科学验证的美术标准，并在U3D上验证

## 11. CPU端性能优化小知识点

- 逻辑和表现尽可能分离开，这样逻辑层的更新频率可以适当降低些。
- 对于一些热点函数，如mmo的实体更新、实例化，使用分帧处理，分摊单帧时间消耗。
- 做好同屏实体数量、特效数量、距离显隐等优化。



- 完善日志输出，避免没必要的日志输出，同时警惕日志字符串拼接。
- 使用骨骼烘焙 + GPUSkinning + Instance 降低CPU蒙皮骨骼消耗和drawcall。
- 开启模型的Optimize GameObjects减少节点数量和蒙皮更新消耗。
- UI拼预制做好动静分离，对于像血条名字这种频繁变动的ui，做好适当的分组。
- 减少C#和lua的频繁交互，尽量精简两者传递的参数结构。
- 使用stringbuilder优化字符串拼接的gc问题。
- 删除非必要的脚本功能函数，特别是Update/LateUpdate类高频执行函数，因为会产生C++到C#层的调用开销。对于Update里需要用到的组件、节点等提前Cache好。
- 场景里频繁使用的资源或数据结构做好资源复用和对象池。
- 对于频繁显示隐藏的UI，可以先移出到屏幕外，如果长时间不显示再进行Deactive。
- 合理拆分UI图集，区分共用图集和非共用图集，共用图集可以常驻内存，非共用图集优先按功能分类,避免资源冗余。使用IL2CPP, 编译成C++版本能极大的提升整体性能。
- 避免直接使用Material.Setxxx/Getxxx 等调用，这些调用会触发材质实例化消耗，可以考虑使用 SharedMaterial / MaterialPropertyBlock代替。
- 合并Shader里的Uniform变量。

## 12. GPU端性能优化小知识点

- 合理规划好渲染顺序，避免不必要的overdraw，如:地形（容易被其他物件遮挡）、天空盒放到较后渲染。
- 分辨率缩放,对于填充率出现瓶颈时，这个是最简单高效的。
- 避免使用GrabPass抓屏，不是所有硬件都支持，加之数据回拷和没法控制分辨率性能很差，可考虑使用CommandBuffer.blit去优化。
- 控制好地形的Blend层数，控制在4层以内，考虑到地形一般屏占面积大、贴图采样次数多，对于中低画质考虑不用normalmap。
- 做好物件、树、角色的LOD。
- 避免使用RenderWithShader类方式来定制DepthTexture,可以考虑Camera的 public void SetTargetBuffers(RenderBuffer colorBuffer, RenderBuffer depthBuffer);进行优化。
- 检查Shader的VertexInput 和 VertexOutput是否存在冗余数据.如:顶点色、多套UV。
- 警惕项目里非必要的双面材质，对于需要局部双面的地方通过加面解决。
- Shader里使用fixed、half代替float，理论上除position、uv、一些涉及depth相关计算使用float外，其他都应该使用fixed（主要是颜色值）、half。

- 对于角色皮肤这种不是特别明显的效果，考虑使用预积分这种低成本的方案。
- 对于frag里的计算过程，如果可以抽出来放到CPU应用层、顶点阶段的优先放这里计算。需要注意放到顶点阶段引起的平滑过渡问题。如：eyeVec导致高光过渡问题。
- 镜面反射类效果避免使用反射相机+RT的实现，考虑使用SSR、CubeMap类实现。
- 避免使用实时阴影，如若使用要合理控制下分辨率和阴影距离。考虑使用Projector。
- 使用统一的后处理框架代替多个Image Effect，可以共用模糊函数，减少blit操作。另外Unity自带的Postprocessing V2 支持Volume，性能还是不错的。
- Shader里避免使用分支、循环，sin、tan、pow、log等复杂数学运算。
- Unity自带的遮挡剔除因为CPU消耗和内存占用较高，加之不能Instancing，不太适合移动平台，可以考虑静态预计算(缺点是不支持动态物体)、Hi-Z等优化方案。
- 减少alpha test材质的使用，如若使用注意减小面积、控制渲染顺序。

### 13. 内存优化小知识点

- 警惕配置表的内存占用。
- 检查ShaderLab内存占用：
- 避免使用Standard材质，做好相应的variant skip。
- 排查项目冗余的Shader。
- 使用shader\_feature替代multi\_compile,这样只会收集项目里真正使用的变体组合，避免变体翻倍。
- 检查纹理资源的尺寸、格式、压缩方式、mipmap、Read & Write选项使用是否合理。
- 检查Mesh资源的Read & Write选项、顶点属性使用是否合理。代码级别的检查，如Cache预分配空间、容器的Capacity、GC等。
- 使用Profiler定位下GC，特别是Update类函数里的。如：字符串拼接、滥用容器等。
- 合理控制RenderTexture的尺寸。优化动画Animation的压缩方式、浮点精度、去除里面的Scale曲线数据。
- 减少场景GameObject节点的数量,最好支持工具监控。

## 算法

### 1. 请写出求斐波那契数列任意一位的值得算法

```
1 static int Fn(int n) {
```

```

2 if (n <= 0) {
3     throw new ArgumentOutOfRangeException();
4 }
5 if (n == 1 || n==2)
6 {
7     return 1;
8 }
9 return checked(Fn(n - 1) + Fn(n - 2)); // when n>46 memory will overflow
10 }

```

复制

## 2. 下列代码在运行中会产生几个临时对象？

```

1 string a = new string("abc");
2 a = (a.ToUpper() + "123").Substring(0,2);

```

复制

实在C#中第[][]是会出错的（Java中倒是可[]）。应该这样初始化：

```

1 string b = new string(new char[] { 'a', 'b', 'c' });

```

复制

三个临时对象：abc、ABC、AB

## 3. 冒泡排序（手写代码）

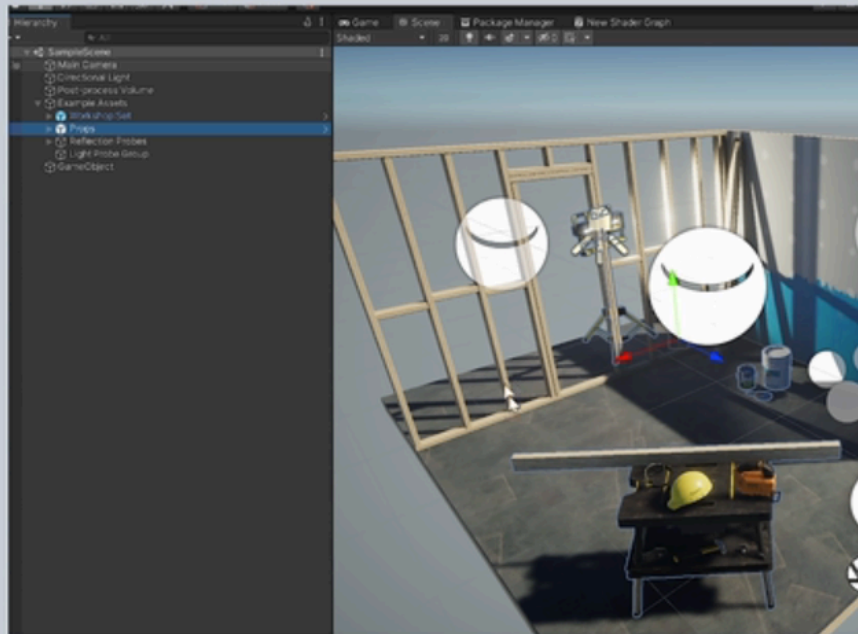
```

1 public static void BubblingSort(int[] array) {
2
3     for (int i = 0; i < array.Length; i++){
4
5         for (int j = array.Length - 1; j > 0; j--){
6
7             if (array[j] < array[i]) {
8
9                 int temp = array[j];
10                array[j] = array[i];
11                array[i] = temp;
12            }
13        }
14    }

```

## 预制体的烘焙

如图，场景中有一个预制体，我们如何在实例化预制体的时候进行正确的渲染？



专门写一个脚本进行烘焙，烘焙的时候记录下对应的光照贴图，光照设置，此预制体在光照贴图中所处的位置和缩放(lightmapoffsetScale)，实例化的时候将其重新设置回去。Unity官方有一个prefablightmapping的脚本，可以参考这个脚本来写。

自动寻路

### 动态加载资源与方式

同步：并不是按字面意思的同时或一起，而是指协同步调，协助、相互配合。是按先后顺序执行在发出一个功能调用时，在没有得到返回结果之前一直在等待，不会继续往下执行。异步：刚好和同步相反，也就是在发出一个功能调用时，不管有没有得到结果，都继续往下执行，异步加载至少有一帧的延迟。

同步的优点：管理方便，资源准备好可以及时返回。缺点：没有异步快。

异步的优点：速度快与主线程无关。缺点：调用比较麻烦，最好的做法是使用回调。

### 引擎中有哪些坐标空间

世界坐标 局部坐标 屏幕空间 UI空间

### 相机中的Clipping Plane、Near、Far数值有什么意义

相机的 Clipping Plane 是相机能够看到的最近和最远距离，而 Near 和 Far 分别是相机能够看到的最近和最远的物体到相机的距离。

在渲染场景时，所有距离相机在 Near 和 Far 范围内的物体将会被渲染出来，而距离相机小于 Near 或大于 Far 的物体则不会被渲染。

调整 Clipping Plane、Near 和 Far 可以影响相机视野的大小和渲染的效率。如果将 Near 和 Far 的值设置过小，则可能无法看到远处的物体；如果将其设置过大，则会浪费资源渲染远处不必要的物体，从而降低渲染性能。

## 四元数的作用

避免万向锁问题：在使用欧拉角或旋转矩阵进行旋转时，存在万向锁问题，即在某些情况下无法进行预期的旋转，而使用四元数可以避免这个问题。

插值平滑：在游戏中需要进行物体的插值平滑，例如在进行摄像机跟随时需要平滑过渡，四元数可以更好地实现这个效果。

通过将两个四元数进行乘法运算，可以将它们表示的旋转进行组合，得到一个新的旋转。

OnEnable、Awake、Start运行时的先后顺序

相机的移动在Update函数中好吗？物理更新一般是在Update中吗？

简述Prefab作用 简述垃圾回收机制与如何避免

场景中Static对象有什么作用 Unity实现移动有哪些方法

协程是如何实现的

## ScriptableObject如何存储在硬盘中

在Unity中，ScriptableObject是可以被序列化的对象，可以被存储为.asset文件

当使用CreateAsset()或SaveAssets()方法时，Unity会将ScriptableObject对象序列化为二进制数据，并写入磁盘文件。这些文件通常存储在项目的Assets文件夹下，可以通过Unity编辑器中的Project视图进行访问。

在运行时，可以使用AssetDatabase类的LoadAsset()或LoadAllAssets()方法来加载.asset文件中的ScriptableObject对象。此时，Unity会将二进制数据反序列化为ScriptableObject对象，并将其加载到内存中。

## BFS查找Hierarchy中游戏对象，查找中途停止搜索后如何获取到停止搜索的这个目录

创建一个队列，并将Hierarchy中的根节点（一般是场景Scene）添加到队列中。

对队列进行循环，每次从队列的头部取出一个游戏对象，并检查该对象是否为目标对象。

如果该对象是目标对象，则停止搜索并返回该对象。

如果该对象不是目标对象，则将该对象的所有子对象添加到队列的尾部。

当队列为空时，表示已经搜索完整个Hierarchy，但仍然没有找到目标对象。

SOLID原则

设计模式

算法（冒泡、二分、宽度、深度、洗牌、泛洪、A星、自动寻路等）

**\*如何优化内存**

有很多种方式，例如

- 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

同步和异步

Heap与Stack

值类型与引用类型

拆箱和装箱

反射实现原理

List与数组区别

C#委托与事件

C#中有哪些排序方法

射线检测的基本原理

## 协程函数的执行顺序与缺点

Unity中的协程是一种能够在主函数中暂停和恢复执行的功能，它使用了C#的迭代器。要实现协程，你需要继承MonoBehaviour类，并使用IEnumerator类型的函数。你还需要使用StartCoroutine()方法来开启协程，并使用yield return语句来指定协程恢复执行的条件。

首先执行协程函数，但不会阻塞当前线程，而是返回一个Coroutine对象。

然后，Coroutine对象被加入到协程队列中，等待执行。

每一帧Unity引擎都会执行协程队列中的协程，如果该协程没有被暂停，则会一直执行到协程结束，直到下一帧才会执行下一个协程。

如果该协程被暂停，那么该协程会被挂起，等待下一帧继续执行。

协程函数的缺点主要是代码可读性差，逻辑复杂时难以维护。此外，协程函数在多线程操作中可能会导致不可预测的结果，需要谨慎使用。

## 序列化是什么

序列化是将数据结构或对象转换为可在网络上传输或持久化到磁盘的格式的过程。在C#中，常用的序列化方式包括二进制序列化、XML序列化和JSON序列化。

四元数的作用

对象池

同步和异步



## Sealed关键字在声明时的作用（还有比如protected、Internal等）

sealed关键字用于声明类、方法或属性，以防止它们被子类继承或重写。一旦将类或成员标记为sealed，它们就不能被继承或重写。这通常用于确保代码的安全性或性能优化，因为它可以防止其他程序员通过继承或重写类来更改代码的行为。

internal关键字用于声明只能在同一程序集中访问的类、方法或属性。这使得程序员可以创建一些只能在程序集内部使用的辅助类或方法，从而隐藏复杂性并提高代码安全性。

## Unity协程和C#线程区别

### ArrayList与List区别

类型安全：ArrayList可以存储任意类型的对象，而List是泛型类型，可以指定存储的对象类型，提高了类型安全性和代码可读性。

性能：ArrayList是基于Object数组实现的，因此存储和检索元素时需要进行拆箱和装箱操作，影响了性能，而List是泛型类型，避免了这个问题，性能更好。

ArrayList的扩容方式是以当前容量的两倍进行扩容，而List则是按照指定的增量进行扩容，这也是List在处理大量数据时性能更好的原因之一。

### 接口和抽象类的区别

#### String与StringBuilder区别

#### 结构体和类的区别

### HashTable和字典的区别

字典是泛型的，而HashTable是非泛型的。这意味着字典可以指定键和值的类型，而HashTable则不需要。

字典在存储和检索值类型时比HashTable更快，因为没有装箱和拆箱的开销。

字典在查找不存在的键时会抛出异常，而HashTable则会返回null。

字典使用KeyValuePair<K,T>来表示键值对，而HashTable使用DictionaryEntry来表示键值对。

#### 引用参数ref和输出参数out区别

#### 数组参数是什么

【接口是否是引用类型

【栈的总体容量大小是如何计算的

【Debug.Log为什么可以后面无限添加参数

【什么是可空变量?的用法

【Using除了名称空间之外，还有在哪里使用

在C#中，using关键字可以用于引入命名空间中的类型或为它们创建别名，也可以用于定义一个范围，在该范围的末尾将释放对象。

【如何获取协程中的分别两个Yield Return

【String类型是引用类型吗

## Const和readonly的区别

const 是编译时常量，必须在声明时进行初始化，且不能被修改，而 readonly 是运行时常量，可以在声明时或构造函数中初始化，且只能在初始化的时候赋值，之后不能再修改。

const 适用于简单数据类型，如整数和枚举值，而 readonly 可以用于任何数据类型。


const 值在编译时就确定了，所以可以在编译时被嵌入到代码中，不需要在运行时进行计算，因此运行速度较快。而 readonly 值在运行时才被计算，因此会有一定的运行时开销。

const 只能在类的内部和静态成员中使用，而 readonly 可以在任何成员中使用。

总之，如果需要定义一个在运行时确定的常量，使用 readonly 更加合适；如果需要定义一个在编译时就确定的常量，使用 const 更加合适。

---

 [常见问题](#)

 [问题（无答案）](#)