Unity内存管理机制

1. **Unity主要采用自动内存管理的机制**

Unity内部有两个内存管理池：堆内存、堆栈内存

堆栈内存（stack）主要存储 较小 和 短暂 的数据

堆内存（heap）主要用来存储 较大 和 存储时间较长 的数据

Unity中的 变量 只会在 堆栈内存 或者 堆内存 上进行内存分配

只要变量处于 激活 状态，则占用的内存就会被标记为使用状态

一旦变量不再激活，则其所占用的内存就会被回收

堆栈上的内存回收及其快速

堆上的内存并不是及时回收 此时其对应的内存依然会标记为使用状态

垃圾回收（GC）主要是指 堆上的内存 分配和回收

Unity中会定时对堆内存进行GC操作

1. **堆栈内存分配和回收机制**

十分快捷简单 因为只存储 短暂较小 数据

堆栈的运行方式其本质就是一个数据的集合 数据存储到堆栈上时，只需要简单的在其后进行扩展，数据失效时只需从堆栈上移除

1. **堆内存分配和回收机制**

较复杂，因为堆内存上既可以存储 短期较小数据也可以存储大数据 其内存分配和回收顺序不可控。

堆上的变量存储主要分以下几步：

1. Unity先检测是否有足够的 闲置内存单元 来存储数据，有的话就分配对应大小的内存
2. 如果没有，unity会触发GC来释放不再使用的堆内存，这是一步很缓慢的操作 GC完成后有足够大小内存 则进行分配
3. 如果还没有，unity会扩展堆内存的大小 这步操作会很缓慢

堆内存的分配有可能会变得十分缓慢 所以通常需要减少这样的操作次数

1. **垃圾回收时的操作**

当堆内存上一个变量不再处于激活状态时，其占用内存并不会立刻被回收，只有在GC时才会被回收。

每次GC时 主要进行下面的操作：

1. GC会检查堆内存上每个存储的变量 看其引用是否处于激活状态
2. 如果变量的引用不再处于激活状态 则会被标记为 可回收
3. 被标记的变量会被移除 其占用的内存会被回收到堆内存中

GC操作是一个极其耗费的操作 堆内存上的变量或者引用越多 运行操作会更多 耗时更长

1. **何时触发GC**

主要三个操作触发：

1. 在堆内存上进行 内存分配 而不够的时候 会触发GC来利用闲置的内存
2. GC自动触发 不同平台 运行频率不一样
3. GC被强制执行
4. **GC操作带来的问题**

GC操作耗时长 使得游戏运行缓慢

堆内存的 碎片化 一个内存单元从堆内存分配出来是根据其存储的变量大小 在回收时有可能使得堆内存被分割成碎片化的单元

也就是说 堆内存总体内存单元是很大的 但是单独的内存单元很小 在下次分配的时候找不到合适大小的单元 又会触发GC

那么堆内存碎片化造成两个结果：一个是游戏占用内存越来越大 二是GC会更频繁地被触发

1. **如何避免GC**

我们可以通过三种方式来减少GC对游戏的影响：

1. 减少GC的时间

2. 减少GC的频率

3. 在合适的时机触发GC 比如游戏加载场景的时候

尽量确保更少的堆分配以及对象引用，这样能保证在GC时会更快。

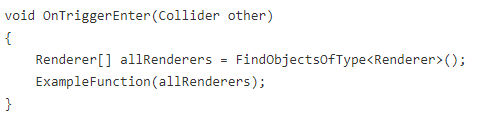
尽量减少堆分配和释放的频率，这样能保证更少的触发GC。

\*\* 减少代码垃圾的产生量 \*\*

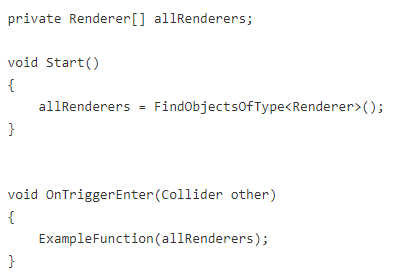
1. **缓存机制**

应当尽量避免重复调用产生堆分配的函数，存储这些对象并复用他们。

比如下面这段方法每次执行都会引起堆分配



我们应当把数组在函数外定义好 缓存起来进行使用



1. **尽量不要在频繁调用的函数中分配内存**

比如update() lateUpdate() 这样每帧都调用的地方，在这里分配会导致垃圾迅速累积。 我们应当在 start() 或者 awake() 当中缓存对象的引用，然后再使用。或者保证分配内存的代码只在需要分配的时候才执行。



1. **清空容器**

对于像List之类的数据结构，并不需要每次都重新创建去new一个，只需要执行clear操作即可。

1. **使用对象池**
2. **注意字符串的使用**

**c#中字符串是引用类型，所以我们在创建和丢弃字符串的时候会产生垃圾。**

**c#中的字符串是不可变的，也就是说他们的值在第一次创建之后就不能改变了。那么如果我们用 + 来连接两个字符串，unity会创建一个新的字符串并且丢弃旧的字符串，这样就会产生垃圾。**

**所以我们尽量避免是用string的 + 连接操作。可以使用 stringBuilder 来代替。**

**减少不必要的字符串创建，多次使用相同的字符串时应当利用一个字符串创建并缓存使用。**

**减少不必要的字符串操作，比如有一个经常要更新的Text组件，这个text是一个连接的字符串。那么我们可以考虑使用两个Text组件。**

**数组**  优点：在内存中连续存储 索引速度很快 赋值 修改元素也简单

缺点：分配在一块连续的内存 分配空间必须确定大小 必须连续这样导致存储效率低 插入 删除麻烦 声明数组的时候要指定数组大小 长度过长又会造成内存浪费 过短又会数据溢出 不清楚数组长度情况下很棘手

1. ArrayList 优点：大小是动态扩充和收缩的 声明的时候不用指定大小 很方便添加 插入 移除

缺点： 在ArrayList中插入不同类型的数据是允许的 因为ArrayList把插入的数据都当做object来处理 类型不安全 同时存在 装箱 拆箱 操作 带来很大的性能损耗

值类型 ----> 引用类型 装箱操作 引用类型 -----> 值类型 拆箱操作

Ps. 值类型直接存储其值 eg. Int bool 等等 引用类型存储对其值的引用 eg. String class

值类型变量声明后 不管有没有赋值都直接为其分配了内存

引用类型 先在栈上分配一小片内存用于容纳一个地址 当new的时候 分配堆上的空间同时把地址保存到栈上的内存

1. 泛型List 优点： 同ArrayList 并且同时避免了装箱 拆箱操作 保证了类型安全
2. 哈希表

找不到返回 null 需要进行装箱拆箱操作所以比字典慢 所有成员是线程安全的

不是泛型类型 可以多线程读取

适用条件: 某些数据会被高频率查询 数据量比较大 查询字段包含字符串类型 数据类型不唯一

1. 字典

找不到返回 error 不拆箱 装箱比hashtable快 只有公共的静态成员都是线程安全的 dictionary 是一个泛型 可以使用它与任何数据类型 字典单线程读取

**构造函数 :** 一种特殊方法，主要用来在创建对象时初始化对象，可以为对象成员变量赋初始值，一个类可以有多个构造函数 根据参数个数和类型不同区分 也就是构造函数的重载

**重写：** 当一个子类继承一个父类 而子类中的方法与父类中的方法名称 参数都一样 那么子类中的这个方法就是重写了 父类中的这个方法

**重载：** 一个类中的方法与另一个方法同名 但是参数表不同 这种方法称之为 重载

（override）重写通常是派生类继承基类的方法 有时需要派生类中继承的方法有不同的实现 那么就需要重写基类方法。

（overload）重载是在一个类中用相同名称 不同参数类型创建一个以上的过程 实例构造函数或属性

**抽象类**

是一种特殊的类 不能被实例化

抽象类中包括 抽象方法 且不包含任何实现 派生类必须 重写 抽象方法

抽象类可以 派生于 另一个抽象类 可以覆盖基类的抽象方法 也可以不覆盖

如果不覆盖的话 则下面的派生类必须 覆盖他们

A\*寻路

F = G + H

G - 当前点与起始点的距离 H - 当前点与目标点的距离

1. 把起始点 A添加到 开启列表
2. 寻找起点周围所有可达的点 加入 开启列表
3. 从开启列表中删除 A 放到关闭列表中 关闭列表保存所有不需要再次检查的点
4. 再把新确定的点 从 开启列表中 删除 添加到关闭列表
5. 继续检查 新的点 周围不在关闭列表以及可通过的点 添加到开启列表
6. 如果周围的某个点已经在 开启列表中 则需要重新计算 G 值 如果G值更低 那么把该点

进程是Process

线程是Thread

线程可以说是小于进程.

一个进程至少有一个线程,也可以有多个线程.

进程是具有一定独立功能的程序关于某个数据集合上的一次运行活动,进程是系统进行资源分配和调度的一个独立单位.

线程是进程的一个实体,是CPU调度和分派的基本单位,它是比进程更小的能独立运行的基本单位.线程自己基本上不拥有系统资源,只拥有一点在运行中必不可少的资源(如程序计数器,一组寄存器和栈),但是它可与同属一个进程的其他的线程共享进程所拥有的全部资源.

***设计模式***

1. **单例模式**

保证一个类只有一个实例 提供一个全局访问

public static Class Singleton

{

private Singleton instance;

private Singleton() {} // 防止外部实例化

//提供外部访问入口

public static Singleton GetInstance()

{

If( instance == NULL)

instance = new Singleton();

return instance;

}

}

关于多线程访问 加个锁

public static Class Singleton

{

private Singleton instance;

private Singleton() {} // 防止外部实例化

private static readonly object locker = new object();

//提供外部访问入口

public static Singleton GetInstance()

{

If( instance == NULL)

{

If( locker )

{

If( instance == NULL)

instance = new Singleton();

}

}

return instance;

}

}

1. **简单工厂模式**

优点：

1.简单工厂模式解决了客户端直接依赖于具体对象的问题，客户端可以消除直接创建对象的责任，而仅仅是消费产品。简单工厂模式实现了对责任的分割。

2.简单工厂模式也起到了代码复用的作用

缺点：

1. 工厂类集中了所有产品创建逻辑，一旦不能正常工作，整个系统都会受到影响。
2. 系统扩展困难，一旦添加新产品就不得不修改工厂逻辑，这样就会造成工厂逻辑过于复杂。

适用条件： 1.当工厂类创建对象比较少的时候

2.客户如果只知道传入工厂的参数，对如何创建对象的逻辑不关心

class Program

{

static void Main(string[] args)

{

Factory.CreateFood("Tomato").Print();

Console.ReadKey();

}

public abstract class Food

{

public abstract void Print();

}

public class Tomato : Food

{

public override void Print()

{

Console.WriteLine("西红柿炒饭");

}

}

public class Potato : Food

{

public override void Print()

{

Console.WriteLine("土豆炒饭");

}

}

public class Factory

{

public static Food CreateFood(string foodType)

{

Food food = null;

if(foodType=="Tomato")

{

food = new Tomato();

}

else if(foodType=="Potato")

{

food = new Potato();

}

return food;

}

}

}

**3.工厂方法模式**

优点： 不同于简单工厂模式 能更好的扩展

缺点： 一个工厂只能创建单个产品

class Program

{

static void Main(string[] args)

{

Food myFood = new TomatoFactory().CreateFactory();

myFood.Print();

Console.ReadKey();

}

public abstract class Food

{

public abstract void Print();

}

public class Tomato : Food

{

public override void Print()

{

Console.WriteLine("西红柿炒饭");

}

}

public class Potato : Food

{

public override void Print()

{

Console.WriteLine("土豆炒饭");

}

}

public abstract class Factory

{

public abstract Food CreateFactory();

}

public class TomatoFactory:Factory

{

public override Food CreateFactory()

{

return new Tomato();

}

}

public class PotatoFactory : Factory

{

public override Food CreateFactory()

{

return new Potato();

}

}

}

4. **抽象工厂模式**

优点：解决工厂方法模式 一个工厂只能创建单个产品的问题

class Program

{

static void Main(string[] args)

{

NanChangFactory nf = new NanChangFactory();

nf.CreateYaBo().print();

nf.CreateYaJia().print();

ShangHaiFactory sf = new ShangHaiFactory();

sf.CreateYaBo().print();

sf.CreateYaJia().print();

Console.ReadKey();

}

public abstract class Factory

{

public abstract YaBo CreateYaBo();

public abstract YaJia CreateYaJia();

}

public class NanChangFactory : Factory

{

public override YaBo CreateYaBo()

{

return new NanChangYaBo();

}

public override YaJia CreateYaJia()

{

return new NanChangYaJia();

}

}

public class ShangHaiFactory : Factory

{

public override YaBo CreateYaBo()

{

return new ShangHaiYaBo();

}

public override YaJia CreateYaJia()

{

return new ShangHaiYaJia();

}

}

public abstract class YaBo

{

public abstract void print();

}

public abstract class YaJia

{

public abstract void print();

}

public class NanChangYaBo : YaBo

{

public override void print()

{

Console.WriteLine("南昌鸭脖 辣辣辣");

}

}

public class NanChangYaJia : YaJia

{

public override void print()

{

Console.WriteLine("南昌鸭架 辣辣辣");

}

}

public class ShangHaiYaBo : YaBo

{

public override void print()

{

Console.WriteLine("上海鸭脖 甜甜甜");

}

}

public class ShangHaiYaJia : YaJia

{

public override void print()

{

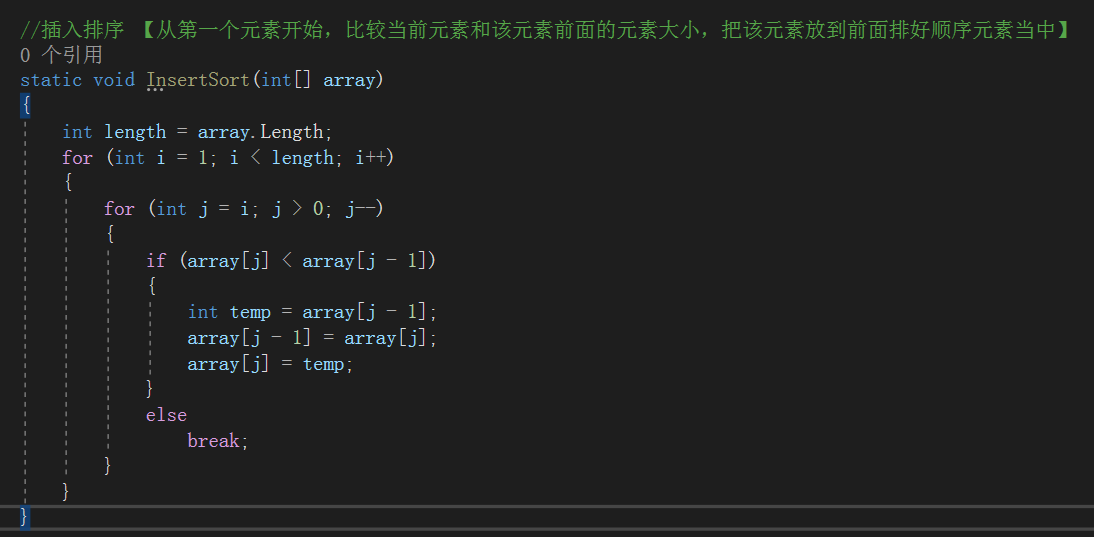
Console.WriteLine("上海鸭架 甜甜甜");

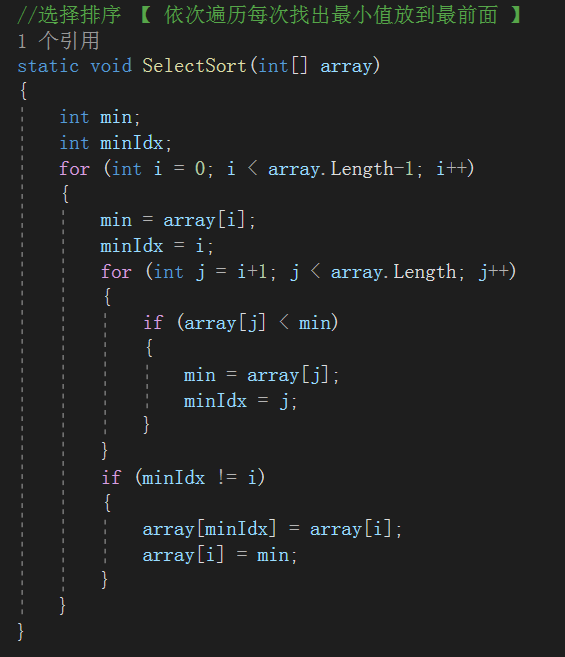
}

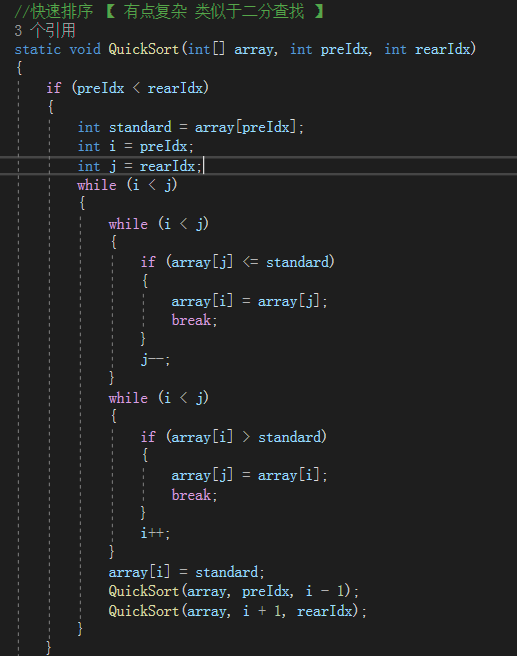
}

}

排序算法



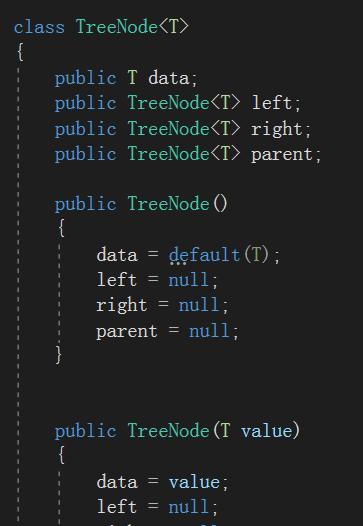




========== 二叉树 ==========

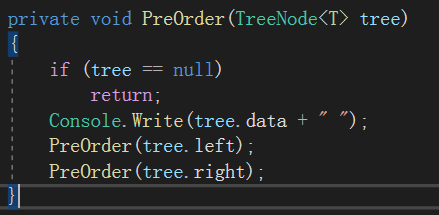
1. 基本概念就不说了...

树节点模板类定义：

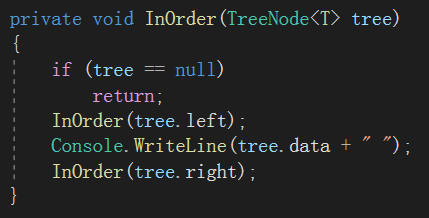


满二叉树 完全二叉树...

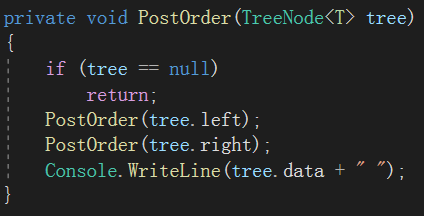
1. DFS ( Depth First Search ) 深度优先
2. 先序遍历 【 节点 - 左 - 右 】



1. 中序遍历 【 左 - 节点 - 右 】



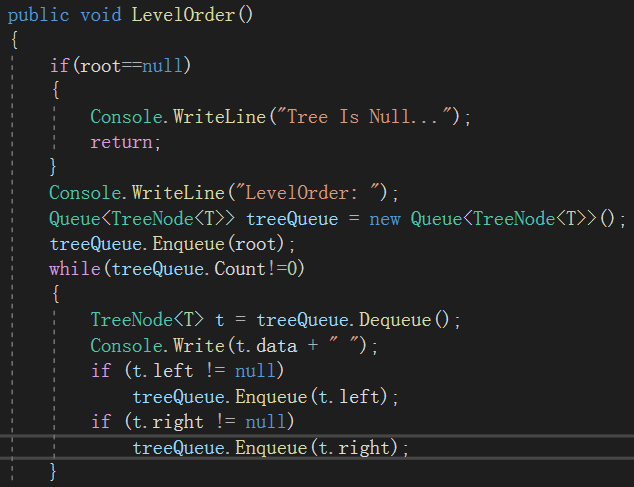
1. 后序遍历 【 左 - 右 - 节点 】



1. BFS ( Breadth First Search ) 广度优先

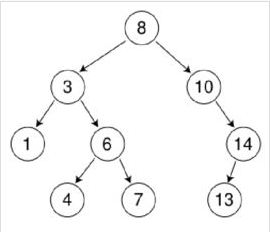
层次遍历 【 按照层级遍历 】

利用队列 先进先出输出



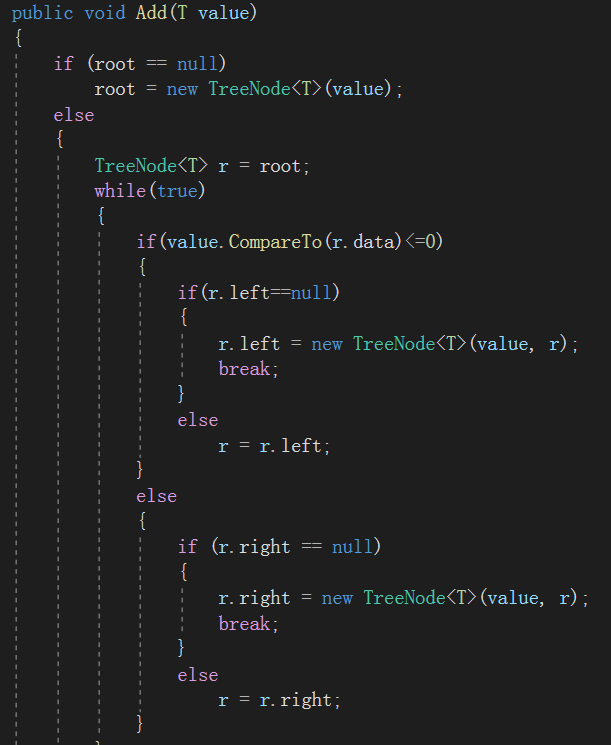
1. **二叉排序树**

**特点： 左子树的值比根节点小，右子树的值比根节点大，并且左右子树都遵循该规则。 如图：**



1. 添加节点

遵循规则添加，遍历整棵树，小的往左放，大的往右放...



1. 删除节点

有下列几种情况：

1. 删除的节点为叶节点

只要把节点直接删除即可 因为删除叶节点不会改变原二叉排序树的顺序

1. 删除的节点不是叶节点
2. 只有左子树，也就是删除的节点只有左子树没有右子树

直接把节点删除，把左子树提上来，放到原来删除节点的位置

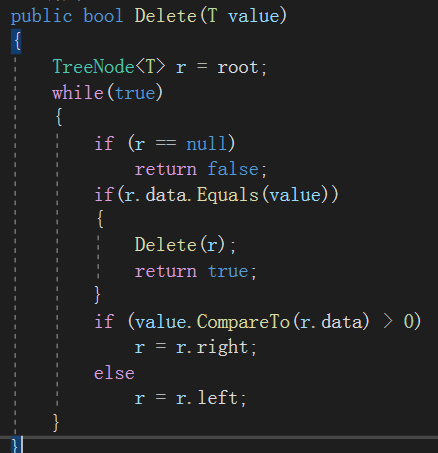
1. 只有右子树，也就是删除的节点只有右子树没有左子树

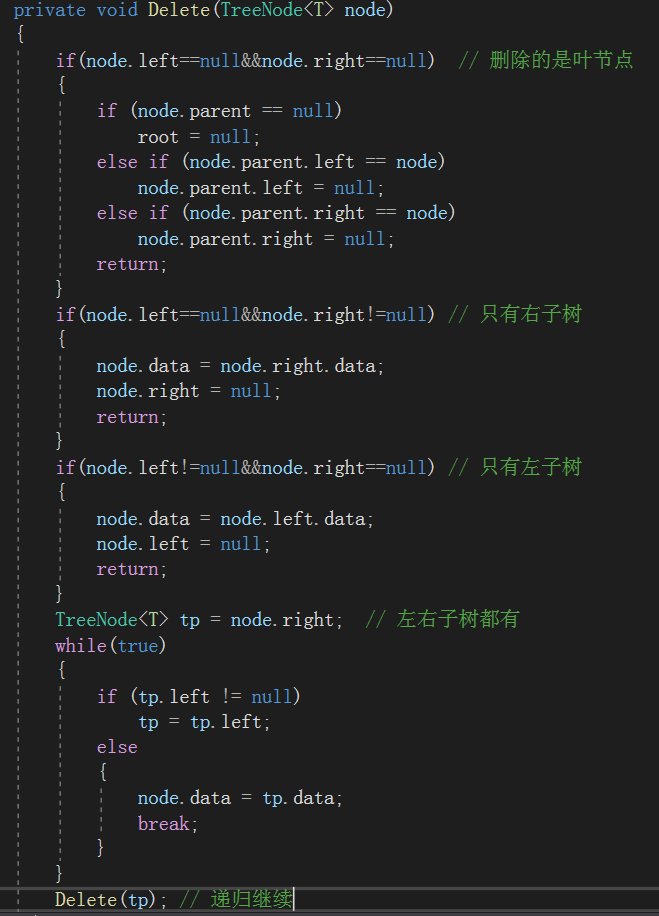
直接把节点删除，把右子树提上来，放到原来删除节点的位置

1. 左子树和右子树都有

这种情况就把右子树中的最小值找出来，然后把最小值放到被删除的节点位置...

因为二叉排序树的特性，左边的是比节点小的值，右边的是比节点大的值，所以找右边中最小的值！理解一下...





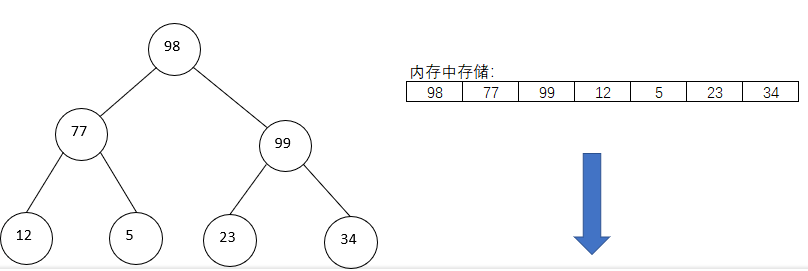
1. **堆排序**

**这里的堆指的是一棵堆积树，是完全二叉树，区别于堆栈的堆...**

**内存中以数组方式顺序存储**

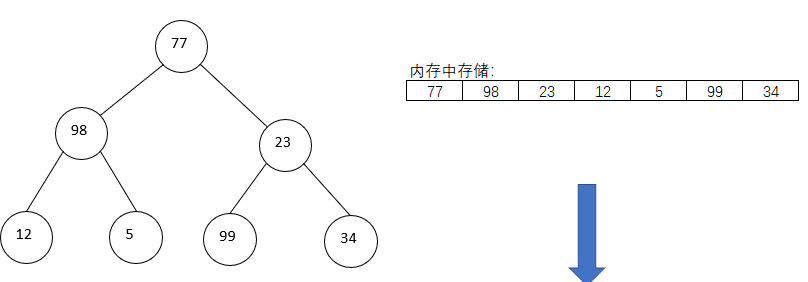
**大根堆**

每个节点的值都比父节点的值小，也就是最大值在堆顶



**小根堆**

每个节点的值都比父节点的值大，也就是最小值在堆顶



由于数据是顺序存储，我们可以知道对于节点 K 来说，他的左节点位于 2K+1; 右节点位于 2K+2;

一个长度为 count 的堆，他的最后一个非叶节点位于 count/2 - 1 \*针对于完全二叉树

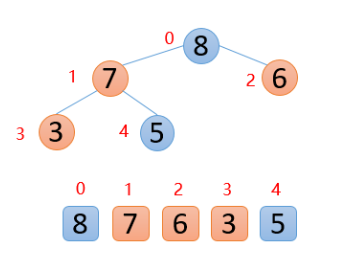
1. 堆排序的思路

利用大小根堆的特性，把一个堆进行排序，例如这里用大根堆排序。

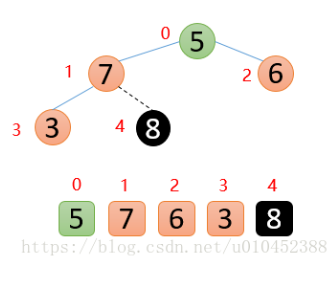
首先排一次序，找出最大值，然后把这个最大值和最后一个元素交换顺序，接着对剩下的 n-1 个元素进行大根堆排序，再次找到最大值和最后一个元素交换，这里这个元素就交换到了倒数第二个的位置...

就这样重复进行，直到最后所有元素排完...

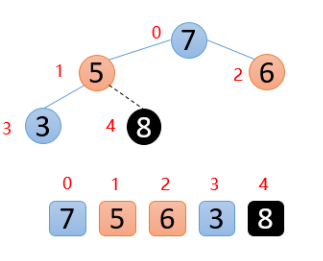
① 假设第一次排完如下 最大值 8 已经被排到堆顶

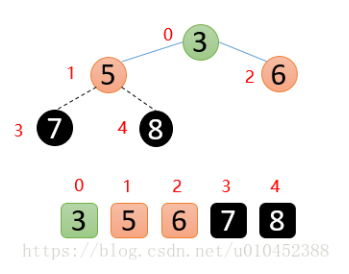


② 把 8 沉下去，也就是和最后元素 5 交换，交换完之后如下， 此时将固定 8 ，也就是接下来对除了 8 之外的其他元素再次进行排序，找到最大值。

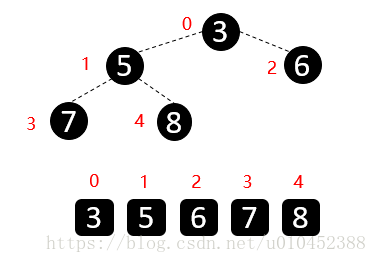


③ 再次排到最大值 7 到堆顶， 再次和最后元素 3 交换。 注意这里的 8 是不参与排序当中的，所以最后一个是 3

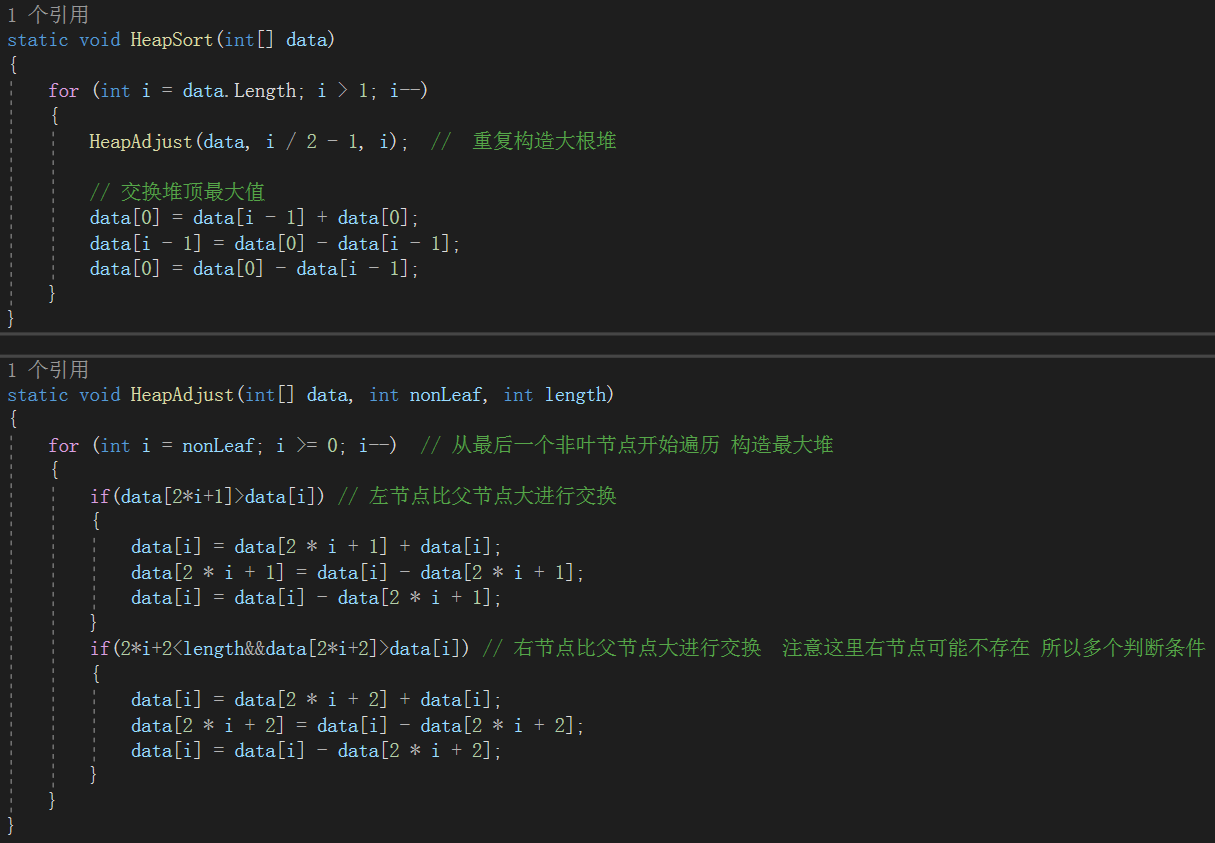




④ 依次类推.... 最终结果

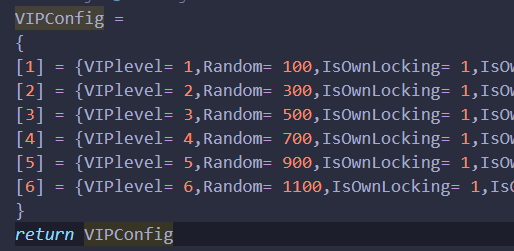


具体代码：

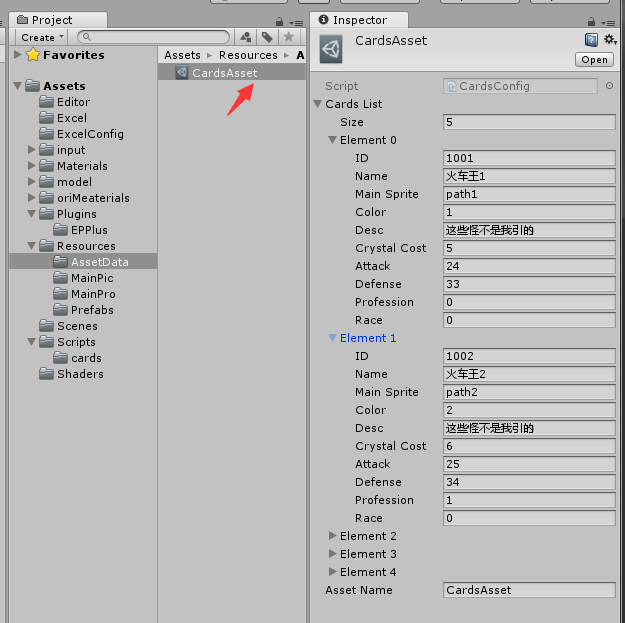


**关于Excel配置表的读取**

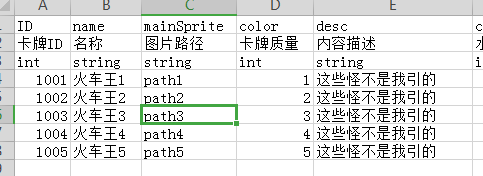
在项目中有看到读取Excel表自动生成Lua的Config脚本，在编码时便可以通过读取该config拿到配置的数据。



那如果是在c#中该怎么保存读取的数据呢？于是便动手写了一套...

基本思路： 通过unity的ScriptableObject来保存成asset文件

1. 先解析Excel表，一般来说配置表都这样配置：字段名 + 字段含义 + 字段类型



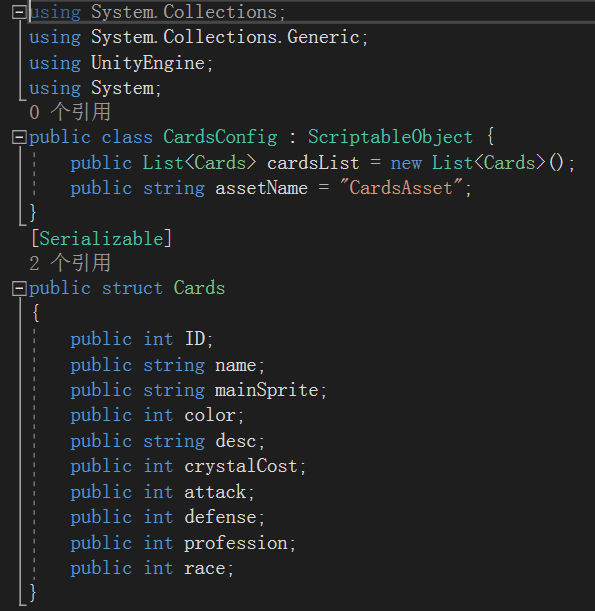
在解析Excel表时通过 字段名 + 字段类型，自动生成一个config cs脚本。 这个脚本由 一个定义了所有字段的struct , 一个struct链表, 一个保存成asset文件的文件名 组成。类似这样：

注意的是这个 config 脚本必须继承 ScriptableObject，不然无法生成asset文件。

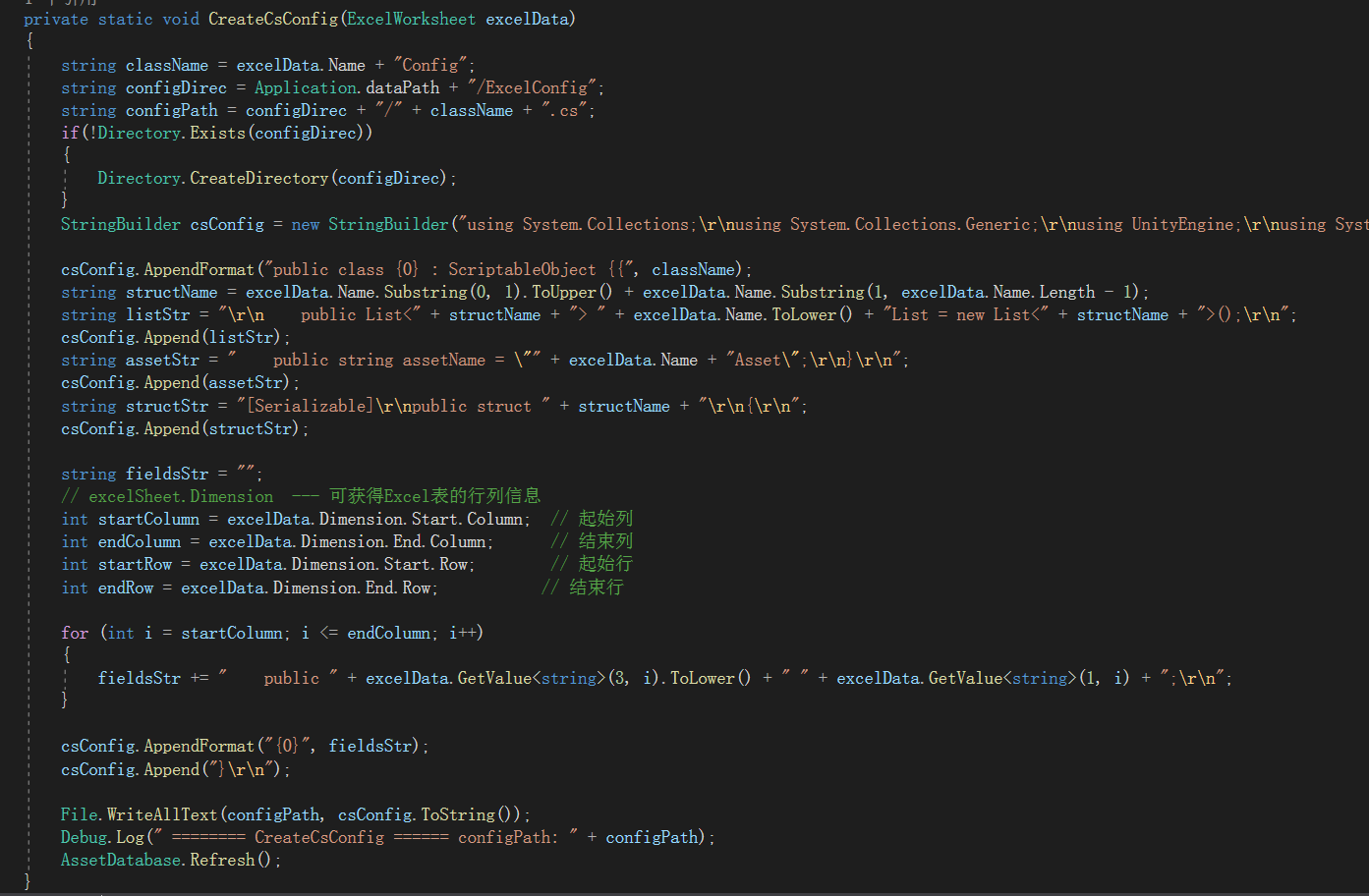
cardList : 这个链表就保存着解析后的配置数据，以供编码时获取使用。

assetName: 这个名称是保存成asset文件后的 asset文件名称，目的是为了在编码时 Resource.load 用，因为想要封装一层，利用反射获取到该字段直接load对应的asset文件，不想要自己每次都写一个名字。

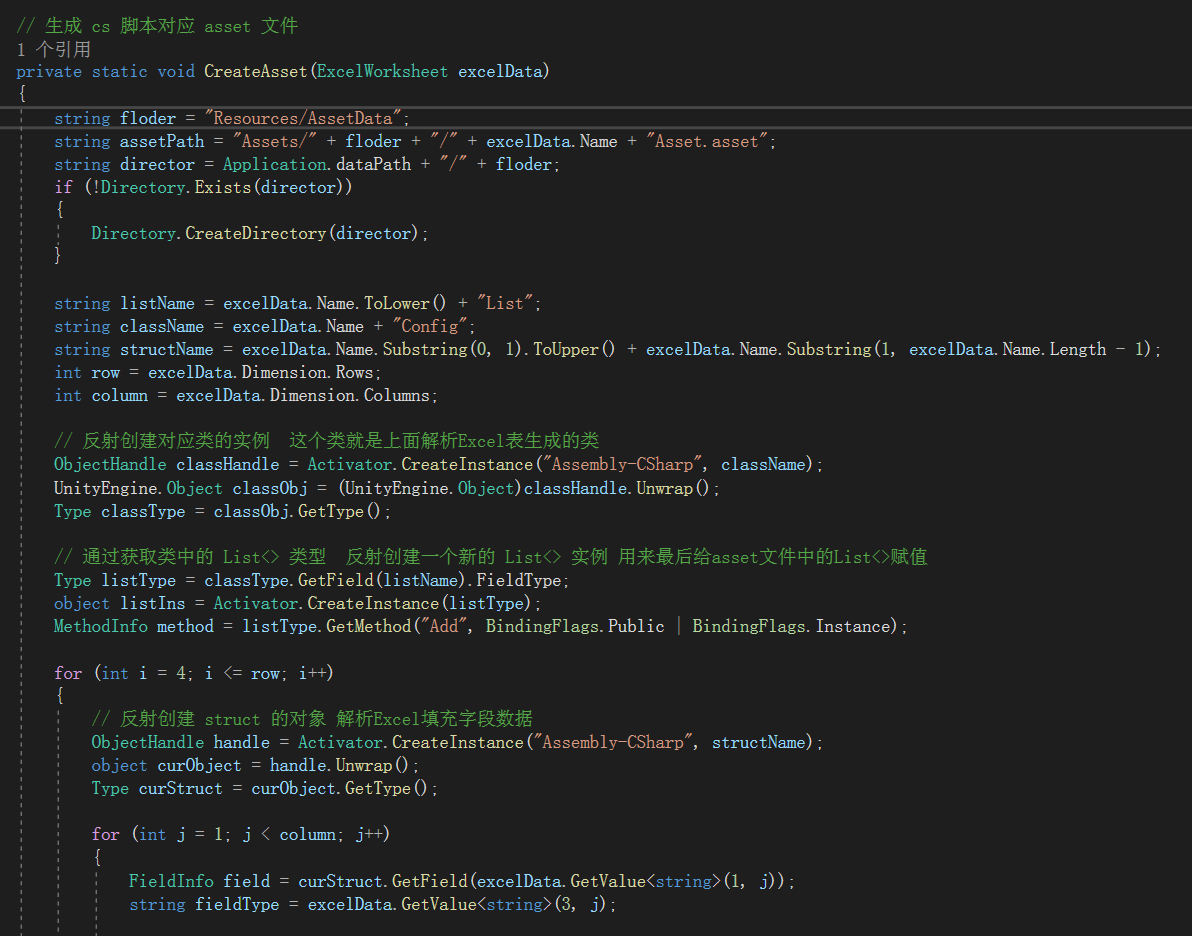
Cards: 这个结构体就包含了所有字段信息

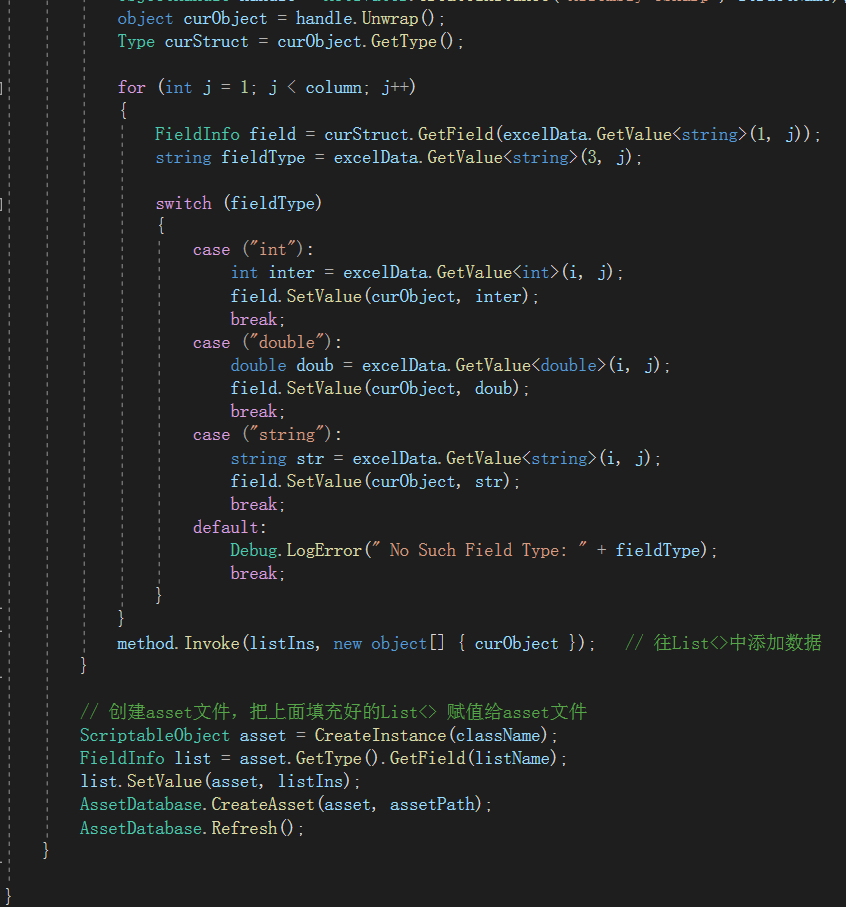


这里是自动生成config文件：



1. 生成完config脚本之后，就可以解析Excel表填充配置数据，解析完后生成asset文件。





1. 上述准备工作完成后，那在编码中如何使用呢

这里封装一层 通过 LoadAsset 使用配置文件

可以看到注释的那一行，就是未封装直接调用resource.load使用...

