# 实验目的

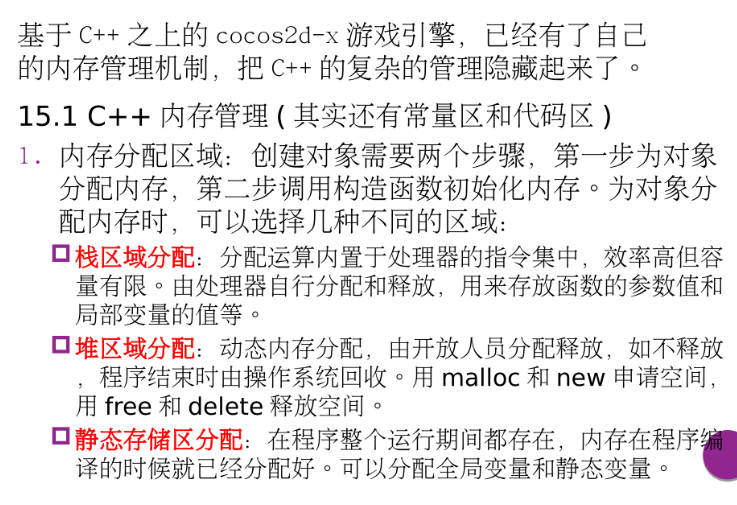
根据讲义总结cocos的内存管理机制。

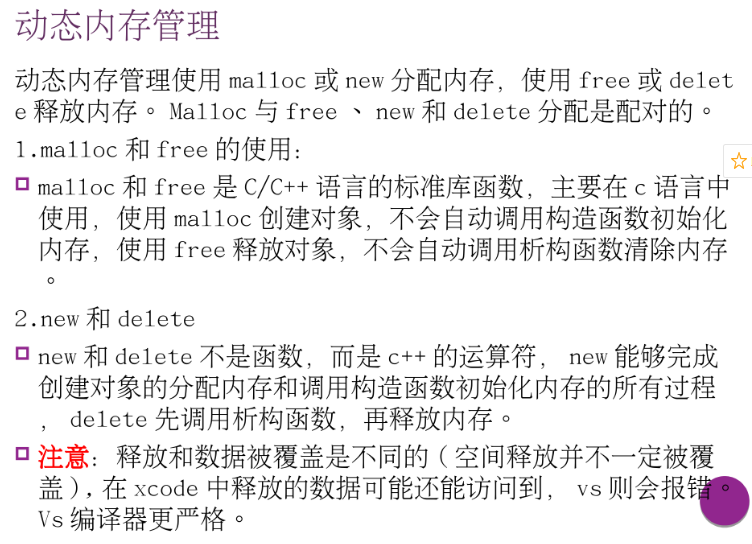
# 实验内容

根据课堂内容，讲义以及前面所讲授的关于纹理缓存、动画缓存等知识，深入分析总结一下关于cocos2d-x的**内存管理机制**及**优化策略**，并提交总结报告。

# 实验心得Part1：

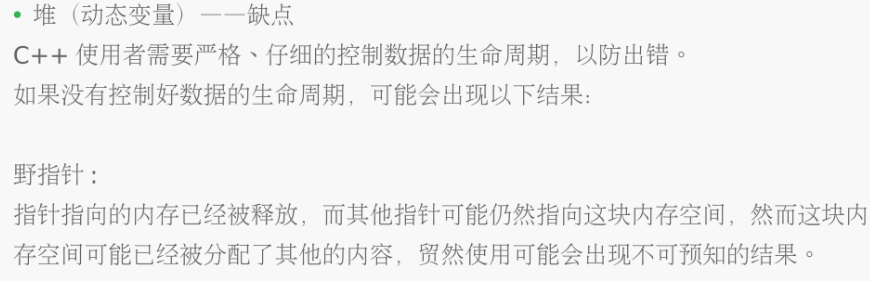
## C/C++中动态内存管理



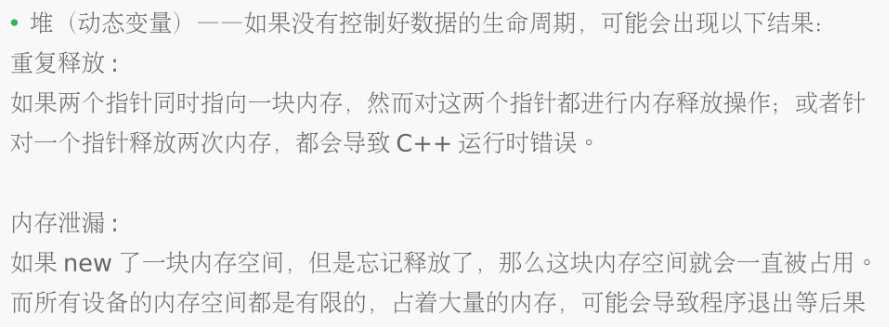


### 堆内存使用缺点

#### 野指针



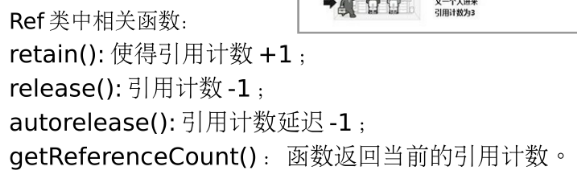
#### 内存泄漏

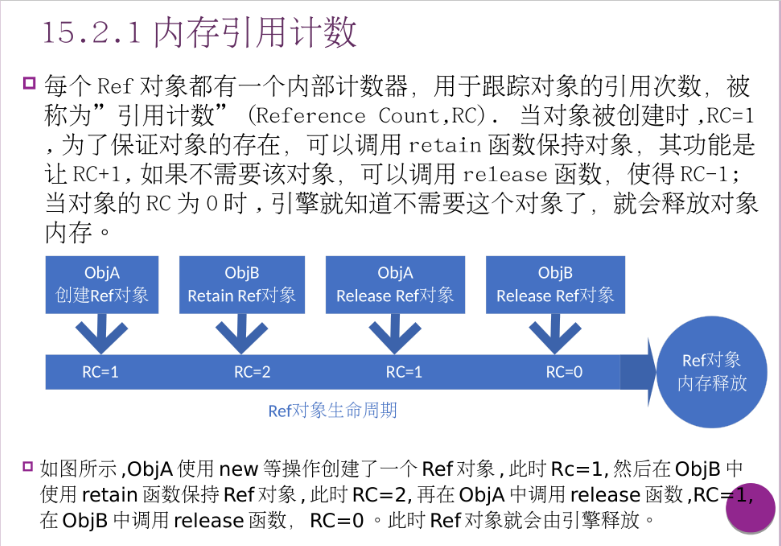


## Ref内存管理



### 引用计数reference count

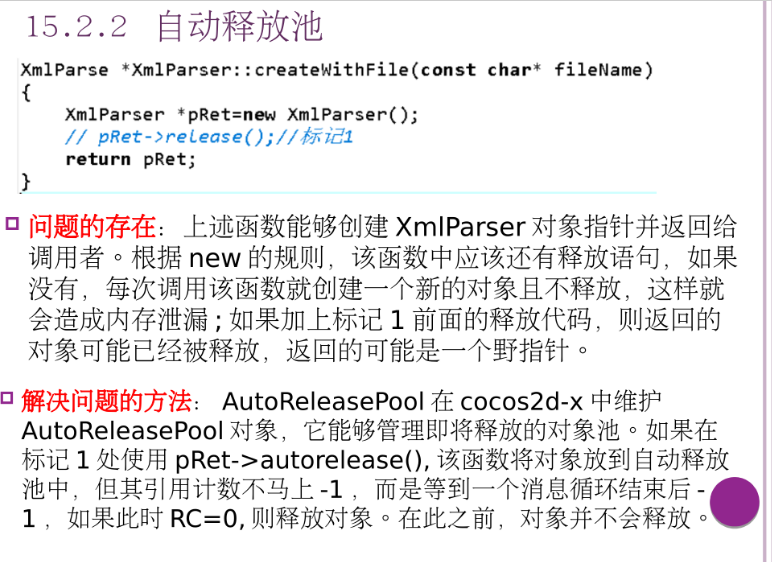




新建一个Ref对象时，其引用计数为1，当引用计数为0时它就会被释放。

#### 问题：什么时候检测Ref对象引用计数为0然后进行释放。

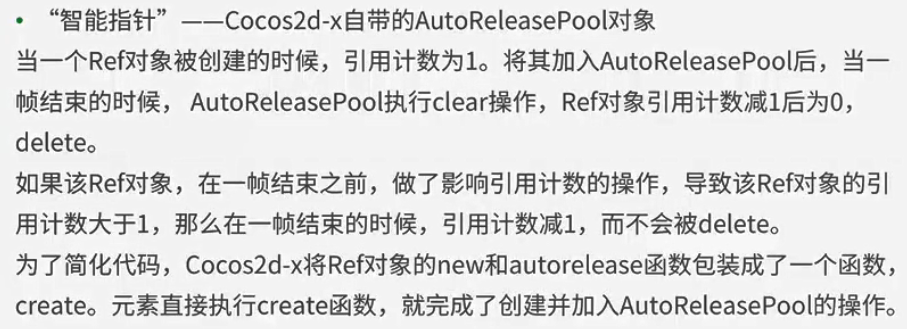
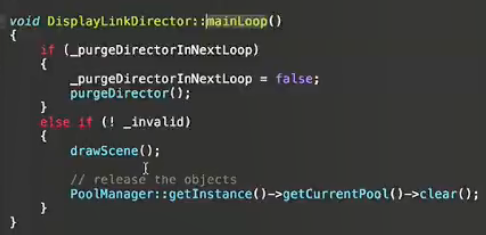
### 自动释放池autoReleasePool



可以这样理解：使用new之后如果没有用delete就会造成内存泄漏，但是为了防止后面忘了delete,可以使用autoRelease方法，将该对象延迟释放，即在一个消息循环结束后释放。

### PoolManager

引擎每一帧都会调用场景绘制函数drawScene, 然后清空autoReleasePool

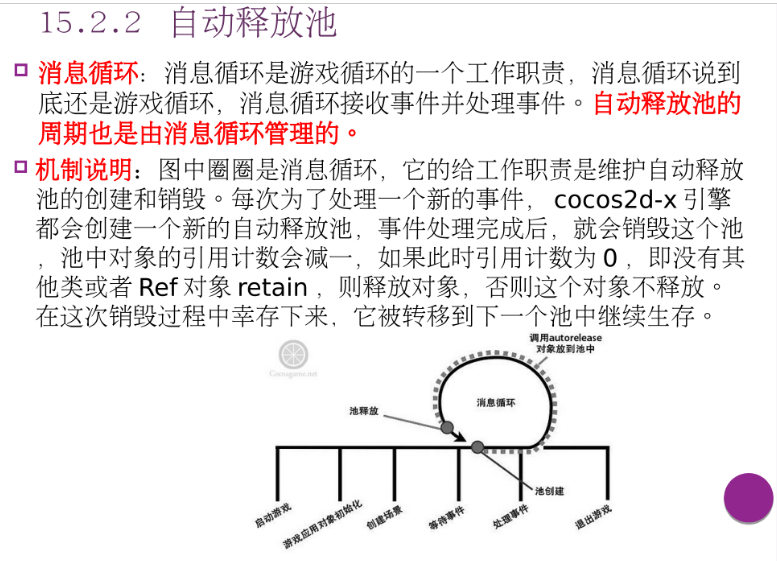


假设一帧中需要调用100个函数，而每个函数都创建一百个动态变量，这些动态变量仅在当前函数中被使用到。如果我们将这些动态变量加入到autoReleasePool中，这些动态变量只有在每一帧结束时才会被清理，最多会有一万个动态变量没有被删除，造成大量的内存浪费。

这样运行下来一帧最多有一百个动态白变量在内存中。

注意：是创建一个AutoRelaeasePool的局部变量，而不是new一个动态对象。

### 消息循环event circle



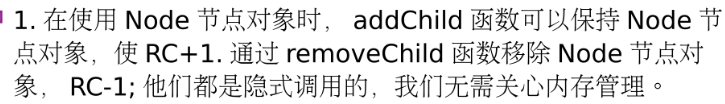
自动释放池：Ref对象调用其autoRelease方法后，会被加入到自动释放池，一个消息循环结束后，自动释放池会被销毁，销毁前池中的对象的引用计数都会减一，这样就做到了**延迟释放**。

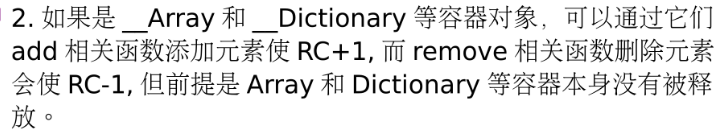
### retain和release

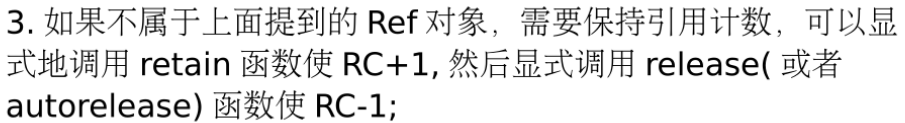
addChild方法会对子对象进行retain，因为其被父对象引用了一次。



removeChild方法会对其子对象进行autoRelease, 因为父对象对子对象的引用解除。





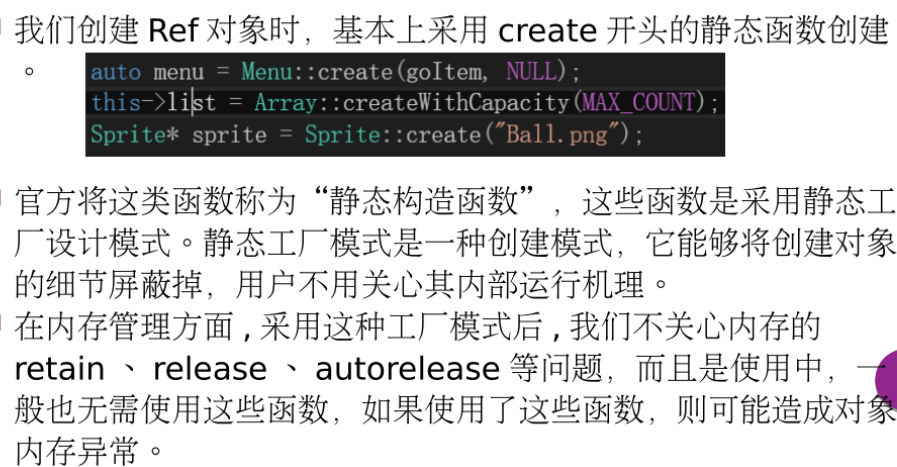


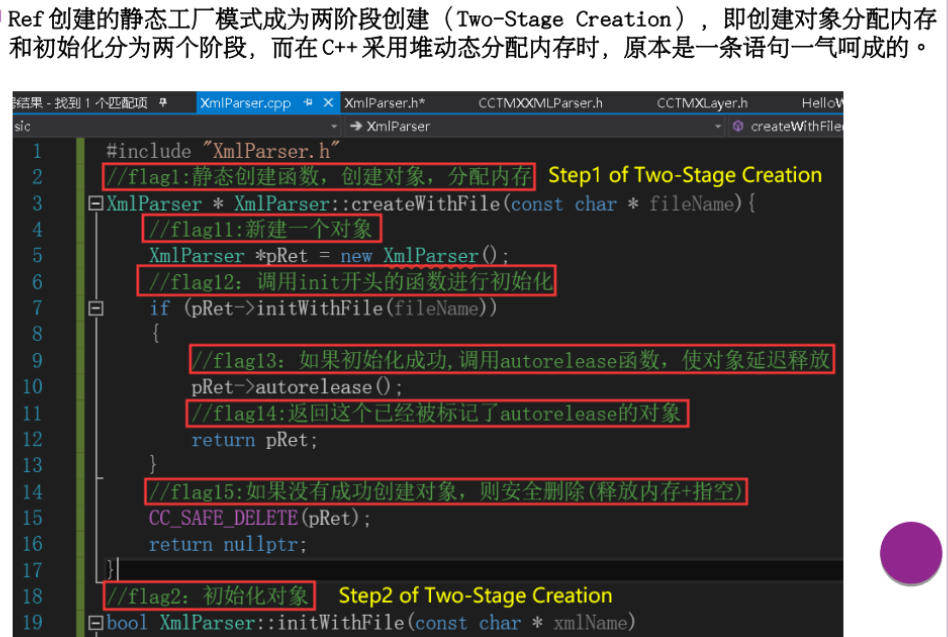


有些函数是隐式的调用了retain或release, 比如create静态方法, 这时注意不要显示的多加上retain或release。

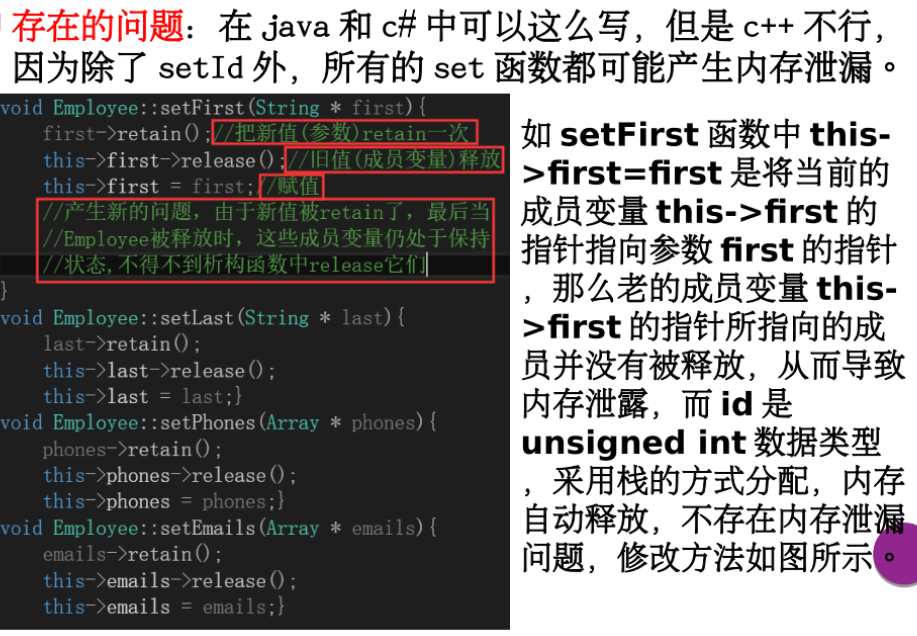
## 相关的设计模式

### 静态工厂：静态构造函数create





### 访问器



对象只有一个，但是可以有多个指针指向它。

#### 问题：为什么要对新值进行retain

**因为引用新值的指针多了一个，引用计数需要加一。**

构造函数中肯定对first进行了初始化操作，类似于下面的代码：

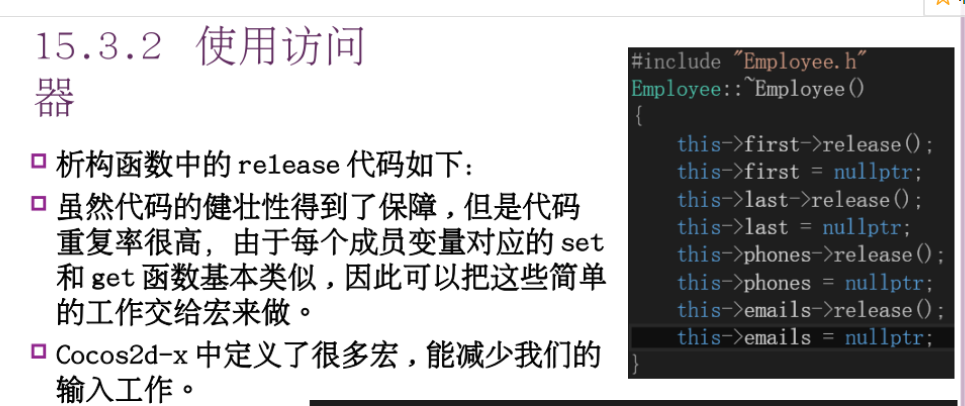
this->first = new String();//first指向空对象s0。

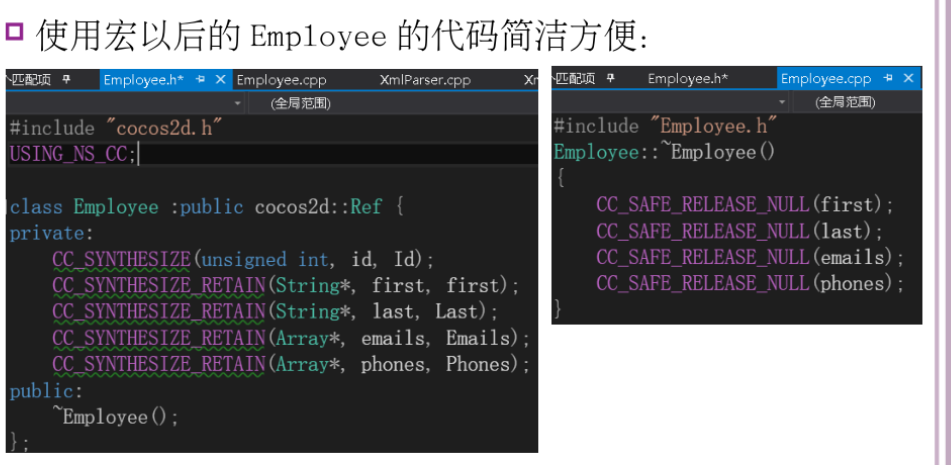
假如后面对成员变量String对象first进行了两次set操作，

setFirst(s1);//s1被retain了一次，s0被release,引用计数归0，后面会被引擎自动释放，first指向s1。s1是在setFirst函数外面被创建的，这时this->first也指向了s1, 所以其引用计数应该加一，需要对其release。形参first可以不用管，因为在函数生命周期结束后，first的生命周期也随之结束。

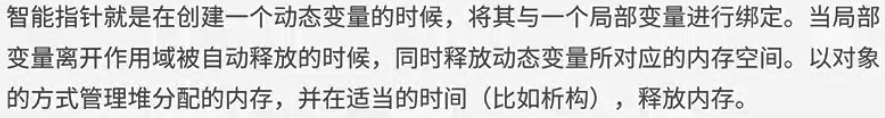
直白的说，this->first引用了谁，就需要对谁的引用计数加一，如果不引用了就将其引用计数减一。

所以后面析构函数中需要进行release, 以和构造函数中新建对象（引用计数初值为一）相对应。





## C++智能指针



一个动态变量可以同时关联多个局部变量（比如在不同函数中），只有当所有局部变量都释放掉时，动态变量才会释放掉。

### **问题：**它是怎么同时关联多个局部变量的呢，一般情况下不能同时运行两个函数吧？

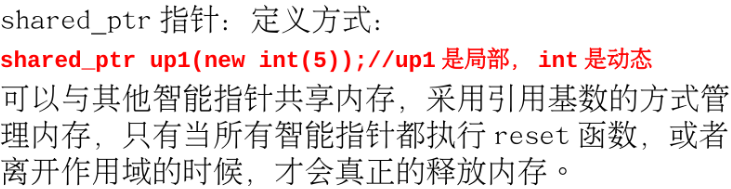
所以常在多线程的情况下使用。

### 例子：

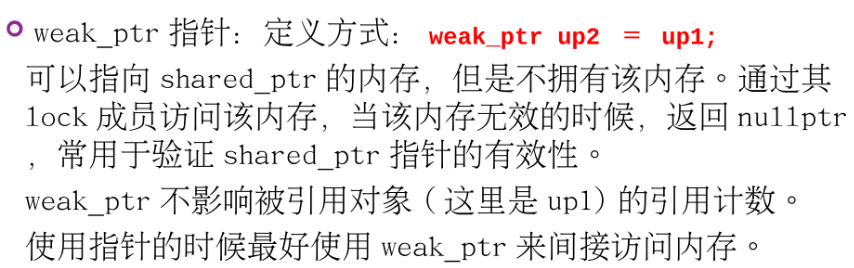
比如在一个函数中创建一个局部对象，创建时会掉用它的构造函数，类的构造函数中会new一个动态变量。函数结束时会释放该局部对象，同时调用其起析构函数释放动态变量。

### 分类：

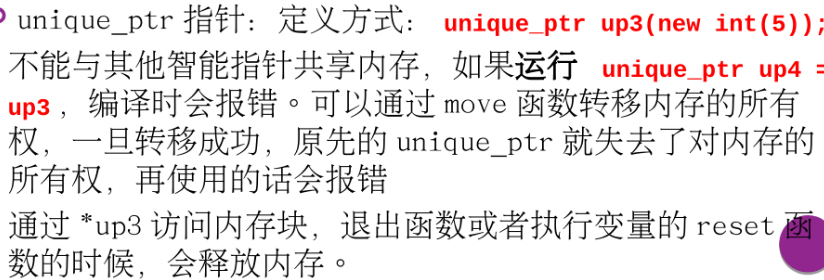
#### shared\_ptr



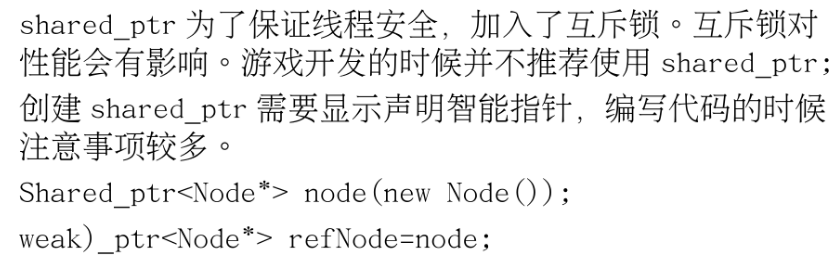
#### weak\_ptr



#### unique\_ptr



### 缺点



# 实验心得Part2：

## 纹理缓存

一个精灵做一个图片会耗费比较多的IO时间，所以一般是若干精灵做成一个大图，一次性读入内存（创建纹理缓存)，然后裁剪出来,这种方法内存会耗费多一些。

把大图读入计算机内存中，然后利用纹理缓存反复使用，读入需要的部分存入大图的时候，一般有一个loading的过程。这个时候可以用logo来分散注意力在此过程中会启动一个子线程，进行加载和预处理，后面就比较流畅。

## 精灵帧缓存

单例设计模式，整个程序运行过程中只有一个精灵帧缓存。

Step1创建精灵帧缓存(texturePacker生成的大图)：将每个frame(精灵帧，小图)创建出来并放入cache中，以便后续可以反复使用。

Stpe2用精灵帧创建精灵

由于精灵帧缓存的加载过程是游戏启动时，因此该步骤会比较快。

纹理缓存和精灵帧缓存的博客https://blog.csdn.net/Meloor/article/details/101479021#2\_10

## 帧动画

用单个文件创建精灵帧，然后组成帧动画(animation)。

从精灵帧缓存中读取精灵帧，然后组成动画（animation）。

一个精灵做一个图片会耗费比较多的IO时间，所以使用精灵帧缓存时间上更优 。

帧动画博客https://blog.csdn.net/Meloor/article/details/103117176

## 粒子系统

用帧动画实现一些不规则的动态场景如篝火，会消耗大量内存。

如果改用粒子系统，粒子种类不多，因其参数不同表现不同，大量的粒子重复出现，数量较大，进行重复渲染，占用内存不大。

## 优化策略

1. 把游戏中需要使用的图片（精灵）都做成精灵帧大图（使用TexturePacker），包括帧动画中所要用到的精灵帧。这样可以减少读取图片时的IO时间，此外，由于精灵帧缓存的加载过程是游戏启动时，所以在游戏中生成精灵时会更快。
2. 有些帧动画可以用粒子系统来代替，这样可以节约大量的内存。