Cocos2d-x + Lua接入iOS原生SDK的实现方案

相信很多朋友在使用Cocos2d-x+Lua开发游戏时都遇到过接入iOS原生SDK的问题，比如常见的接应用内支付SDK，广告SDK或是一些社交平台SDK等等，我也没少接过这类SDK。这篇文章主要是对我做过项目中接入iOS原生SDK实现方案的一个总结，在这里分享给大家，希望对自己和大家的开发工作都有帮助。

在展开正文之前，先做几点说明：

1.我这里说的iOS原生SDK是指那些完全用Objective-C语言开发，为原生iOS程序设计的SDK。swift很好很强大，不过我还没用过，惭愧，不过语言终归只是表达方式而已，解决问题的思路都是一样的；

2.这里假设游戏的主要逻辑使用lua实现，对于主要逻辑使用C++实现的，用本文的思路一样可行，并且设计上更简单，接着往下看就知道了.

3.本文以Quick-Cocos2d-x 2.1版本为例进行讲解，主要因为这个是我之前做项目用得最多的一个版本，-x新版本变动比较大，但是，还是那句话，**解决问题的思路是相同的**。

----------正式开始的分割线----------

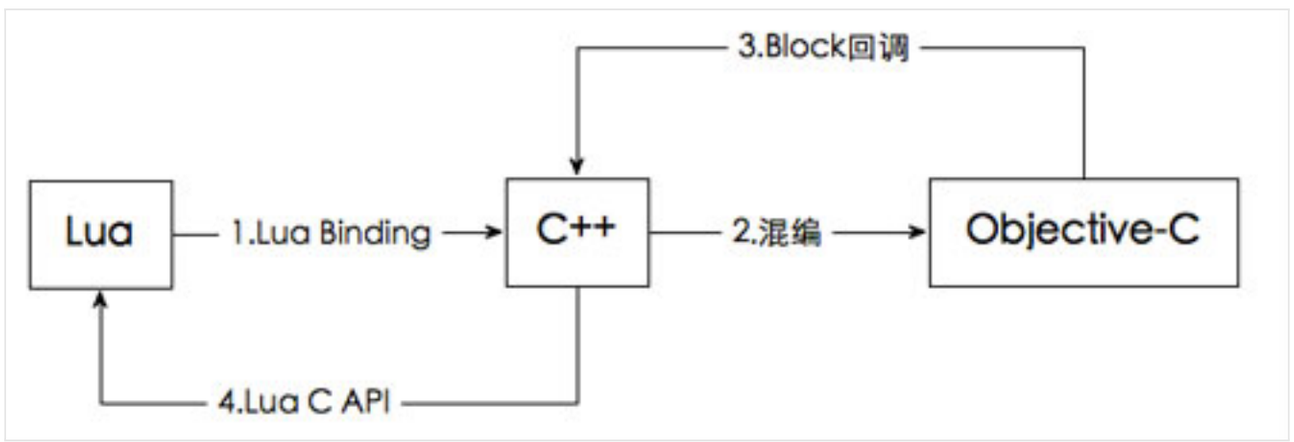
好了，我们正式开始。开门见山！

解决这种接入问题，实际上最主要是解决不同语言交互的问题，一旦跨过语言交互的障碍，剩下的事情就so easy!

由于涉及到Lua、C++、Objective-C三种语言，这个问题表面上看起来错综复杂，但其实只要冷静地（=。=）梳理，我们便可以得到正确的思路。

因为我们游戏的逻辑主要是用Lua实现的（前面已经做过假设），而SDK是用Objective-C实现，所以这里我们需要解决Lua与Objective-C的交互问题，即最终希望达到的目标是，在Lua层面“调用”Objective-C的代码（注意这里的调用是加引号的，间接的调用），而当Objective-C层面收到SDK的回调，再通知Lua。我们知道，Lua并没有简单的方法直接和Objective-C交流，但是Lua可以通过Lua Binding和C/C++交流，而我们又知道，C++和Objective-C可以混编，即C++可以直接调用（这里调用没引号，是真的直接调用）Objective-C的代码。想到这里，思路就很明显了，我们可以使用C++为Lua和Objective-C的交互充当桥梁，进而实现Lua到Objective-C的交互。

根据上面的分析，我们可以用如下图表达我们的思路，我们这里将语言交互的过程分成了4个小部分：



整个语言交互的过程可以总结为：Lua调用Lua Binding的C++接口，C++接口调用混编的Objective-C接口，而Objective-C通过block形式的回调，将结果通知给C++，C++通过Lua的C API将最终结果返回给Lua。这样一趟下来，就完成了Lua与Objective-C的整个交互过程。

简单的说一下这4部分：

1.Lua Binding

将C/C++接口导出给Lua调用的方法，由于篇幅的原因这里就不展开了，具体可以参考Lua的文档，以及网上其他地方的文章。

2.混编

Objective-C的一大优点就是可以和C与C++混编使用，就像同一个语言一样共存在一个实现文件里面。具体混编规则也不说了，这里只提两个小细节：

1）在XCode下混编的实现文件后缀是.mm，而不能是.cpp或者是.m；

2）混编的实现文件引用头文件的地方，C++或者C的用#include，而Objective-C用#import，相互没有影响。

3.Block回调

Block是Objective-C一个非常棒的特性，更棒的是在Block里面还可以直接写C++代码：）具体想了解的可以看苹果官方文档。

其实在最初，我曾经尝试过使用发送通知的方式来实现Objective-C对C++的回调，即Objective-C收到SDK回调，给C++部分发送附带回调信息的通知，虽然cocos2d-x中有现成的NotificationCenter来帮助实现，但这种方式的一个显而易见的弊端是大大增加了C++代码和Objective-C代码的耦合度，Objective-C部分也要混编C++调用C++的NotificationCenter发通知，C++部分也要混编Objective-C代码，调用C++的NotificationCenter收通知，这种

结构实在是有够烦躁的。

相比之下，使用Block回调就干净利落太多，Objective-C这边一切都是纯粹的，它并不需要知道自己要被C++调用还是Objective-C调用，也不需要花很多精力在返回回调上，只需要干好自己的本职工作，然后在适当的时候调用Block就一切搞定。

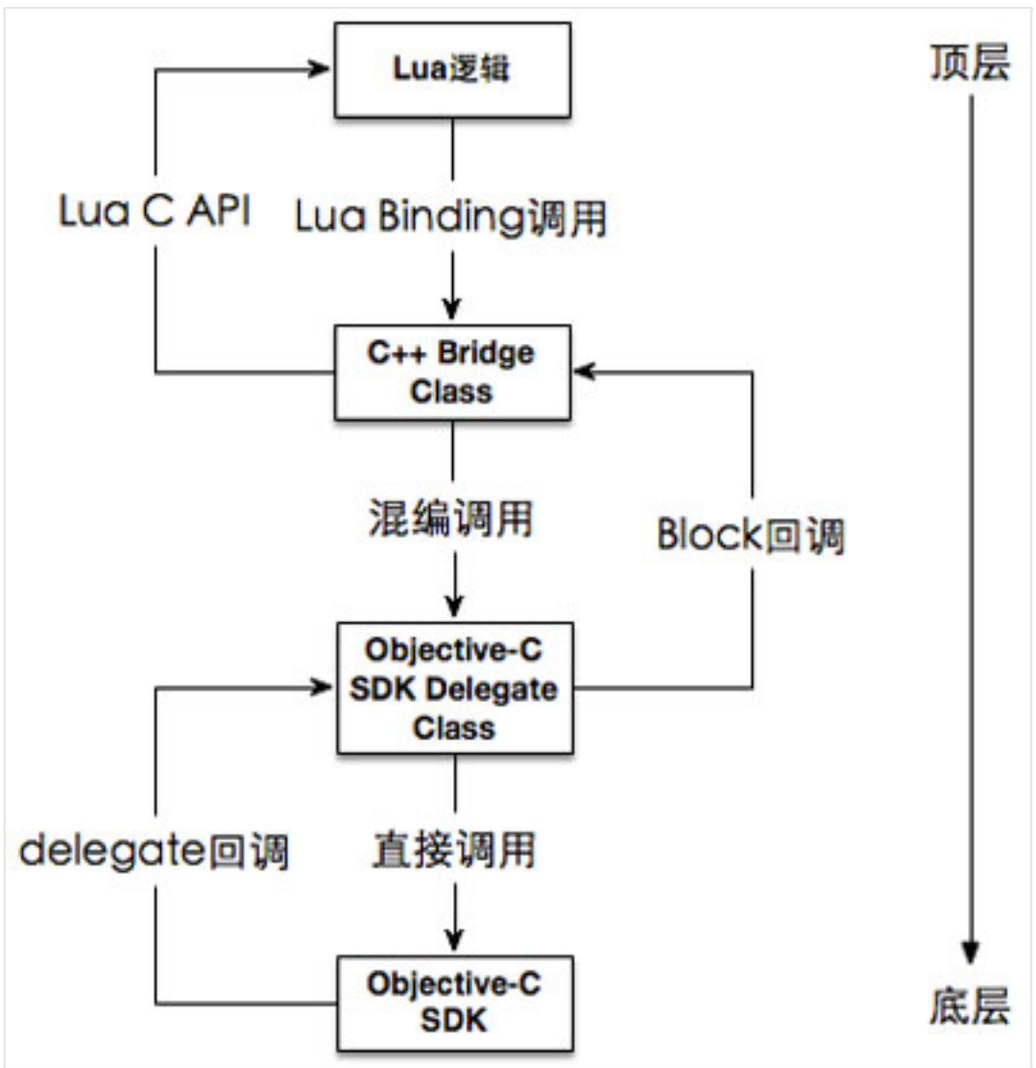
4.Lua C API

Lua C API用于C/C++与Lua的交互，在Cocos2d-x中这些C API已经被封装成了更加易用的C++ Class API。这里要提到的是，在用这套API调用Lua函数的时候，为了传参，需要参数入栈的操作，这个入栈的顺序影响到了Lua函数接受到参数的顺序，不过好在规则很简单：先入栈的参数排在前或者说是入栈顺序和实参顺序相同。举例，如果C++这边调用Lua函数func时，入栈的顺序是A，B，C，那么就是调用函数func(A,B,C)

----------渐入佳境的分割线----------

在整个语言交互的过程中，如果认为Lua是顶层，Objective-C是底层，那么在实际游戏中交互的过程就是一个自顶向下的过程，然而我们在实现各层级代码的时候，需要自底向上完成，因为在顶层Lua代码中的逻辑，是由底层Objective-C SDK的接口与功能决定的。即**我们需要先根据SDK中原始的Objective-C的接口，做适合我们游戏Objective-C封装代理类，然后根据封装结果实现C++的bridge接口，最后再实现Lua的对应逻辑。**

根据以上分析，从层级的角度，设计如下：



除过Lua的逻辑，我们最重要需要实现的两部分内容：C++ Bridge Class 和 Objective-C SDK Delegate Class。前者是起桥梁作用的接口类，原则上不做任何与游戏逻辑相关的数据处理，而后者负责封装原始的SDK接口，接收以及初步处理SDK回调数据。前者的实现依赖于后者的实现，而后者的实现又依赖于SDK。SDK取得的数据最终通过层层传递，交给Lua逻辑处理，最终保证对数据处理的游戏逻辑尽可能多的放到Lua层中。

这样设计的好处有很多，一方面，顶层的游戏逻辑变动，不影响下层多语言交互代码，另一方面，底层的SDK变动，如版本更新甚至更换，不影响上层游戏逻辑，多层次结构有效地降低了复杂度，隔离了变化，对于频繁的需求变更，这种结构也可以保证扩展的便利。

----------总结的分割线----------

综上所述，解决接入iOS原生SDK的问题，主要需要4步：

1.根据SDK接口与功能实现Objective-C SDK Delegate Class；

2.根据Objective-C SDK Delegate Class实现对应的C++ Bridge Class；

3.根据C++ Bridge Class生成对应的Lua Binding代码；

4.写Lua层逻辑。

----------最后的分割线----------

好了，最后，又到了激动人心的上代码的环节了。**下面就以某iOS第三方计费SDK为例，来说明下实现接入的步骤。**

这个SDK只有一个头文件GameBilling.h，主要使用到的方法和Protocol如下：（为了避免篇幅过长等原因，把注释和不必要的代码都删掉了）。我用代码注释的方式说明了各方法的用途。

```
1
```

```
+ (GameBilling *)initializeGameBilling;

-  (void)setDialogOrientationMask:(UIInterfaceOrientationMask)orientationMask;

- (void)doBillingWithUIAndBillingIndex:(NSString *)billingIndex isRepeated:(BOOL)isRepeated cpParam:
(NSString*)cpParam;

@protocol GameBillingDelegate<NSObject>

@required

- (void)onBillingResult:(BillingResultType)resultCode billingIndex:(NSString *)index message:
(NSString *)message;

@end
```

以上前两个方法用于初始化SDK，并且和游戏的逻辑没什么太大关系，所以我们把对他们的调用放在程序开始的位置，不必导出给Lua。第三个方法在用户确认付费时使用，需要导出给Lua，当用户在游戏界面做相应操作时候调用。最后的delegate的回调，我们用前面提到的Objective-C SDK Delegate Class来接收，并作初步处理，再用Block传给C++ Bridge Class.

好的，那我们先来完成Objective-C SDK Delegate Class。这里这个Objective-C做成了个简单的单例来使用，实际可能不需要这么做。

先完成头文件，这里命名为CMGCIAPiOS.h，如下：

```
#import "GameBilling.h"

typedef void (^BillingResultCallback)(BOOL success, NSString *index,NSString *message);

@interface CMGCIAPiOS : NSObject<GameBillingDelegate>

{

    GameBilling *_sdk;

    NSString *_billingIndex;

    BillingResultCallback _callback;

}

+(id)sharedInstance;

-(void)setDialogOrientationMask:(UIInterfaceOrientationMask)orientationMask;

-(void)doBillingWithUIAndBillingIndex:(NSString *)billingIndex

                           isRepeated:(BOOL)isRepeated

                              cpParam:(NSString*)cpParam

                       resultCallback:(BillingResultCallback)callback;

@end
```

应该很清楚，就不多做说明了。

下面是实现文件CMGCIAPiOS.m，如下：

```
#import "CMGCIAPiOS.h"

@implementation CMGCIAPiOS

static CMGCIAPiOS *_sharedInstance = nil;

+ (id)sharedInstance

{

    @synchronized(self)

    {

        if (_sharedInstance == nil)

        {

            _sharedInstance = [[CMGCIAPiOS alloc] init];

        }

    }

    return _sharedInstance;
```

```
}
-(id) init
{
    if( (self = [super init]) )
    {
        _sdk = [GameBilling initializeGameBilling];
        _sdk.delegate = self;
    }
    return self;
}
-(void)setDialogOrientationMask:(UIInterfaceOrientationMask)orientationMask
{
    [_sdk setDialogOrientationMask:orientationMask];
}
- (void)doBillingWithUIAndBillingIndex:(NSString *)billingIndex
                            isRepeated:(BOOL)isRepeated
                               cpParam:(NSString*)cpParam
                        resultCallback:(BillingResultCallback)callback
{
    if(_callback != nil)
    {
        [_callback release];
        _callback = nil;
    }
    _callback = [callback copy];
    [_sdk doBillingWithUIAndBillingIndex:billingIndex
                              isRepeated:isRepeated
                                 cpParam:cpParam];
}
#pragma mark - GameBillingDelegate
- (void)onBillingResult:(BillingResultType)resultCode
           billingIndex:(NSString *)index
                message:(NSString *)message
{
    BOOL b = (resultCode == BillingResultType_PaySuccess || resultCode == BillingResultType_PaySuccess_Activated);
    NSLog(@"billing = %@ %@ %@", b ? @"yes":@"no", index, message);

    if(_callback != nil)
    {
        _callback(b,index,message);

        [_callback release];
        _callback = nil;
    }
}
@end
```

可以看到对提到的几个方法都做了封装，并且接收了回调。

下面是C++ Bridge Class部分，头文件CMGCIAP.h:

```
#include <iostream>
class CMGCIAP
{
public:
    CMGCIAP();
    ~CMGCIAP();

public:
    static CMGCIAP *sharedInstance();

    bool init();

    void setDoBillingCallbackScriptHandler(int scriptHandler);

    void doBillingWithUI(const char* billingIndex,
                         bool isRepeated,
                         const char* cpParam);

private:

    int m_doBillingCallbackScriptHandler;

};
```

由于用Cocos2d-x的tolua工具做Lua Binding的原因，我把设置Lua回调的方法单独提出来了，如下：

```
void setDoBillingCallbackScriptHandler(int scriptHandler);
```

更好的做法是把这个scriptHandler放到下面这个函数中，这样接口就可以和Objective的保持一致了。

```
void doBillingWithUI(const char* billingIndex,
                     bool isRepeated,
                     const char* cpParam);
```

不过也没关系，独立设置Lua回调函数也有更灵活的优点。

注意，C++ Bridge Class头文件一定保持“纯洁性”，做纯粹的C++文件，不能出现Objective-C的任何代码，否则就破坏了上面讲到的层次结构。

下面是实现文件CMGCIAP.mm:

```
#include "CMGCIAP.h"
#include "cocos2d.h"
#include "script_support/CCScriptSupport.h"
#import "CMGCIAPiOS.h"
#import <Foundation/Foundation.h>
#import <UIKit/UIKit.h>
USING_NS_CC;
static CMGCIAP* s_sharedInstance = NULL;
CMGCIAP::CMGCIAP()
{
    m_doBillingCallbackScriptHandler = 0;
```

```
}
CMGCIAP::~CMGCIAP()
{

}
CMGCIAP *CMGCIAP::sharedInstance()
{
    if (s_sharedInstance == NULL)
    {
        s_sharedInstance = new CMGCIAP();
    }
    return s_sharedInstance;
}
bool CMGCIAP::init()
{

    [[CMGCIAPiOS sharedInstance] setDialogOrientationMask:UIInterfaceOrientationMaskPortrait];

    return true;
}
void CMGCIAP::setDoBillingCallbackScriptHandler(int scriptHandler)
{
    m_doBillingCallbackScriptHandler = scriptHandler;
}
void CMGCIAP::doBillingWithUI(const char* billingIndex,
                              bool isRepeated,
                              const char* cpParam)
{

    NSString *billingIndexString = [NSString stringWithUTF8String:billingIndex];

    NSString *cpParamString = [NSString stringWithUTF8String:cpParam];

    [[CMGCIAPiOS sharedInstance] doBillingWithUIAndBillingIndex:billingIndexString
                                                     isRepeated:isRepeated
                                                        cpParam:cpParamString
                                                  resultCallback:^(BOOL success, NSString *index,NSString *message)
{


        CCLuaStack *stack = CCLuaEngine::defaultEngine()->getLuaStack();
        stack->clean();
        stack->pushBoolean(success);
        stack->pushString([index UTF8String]);
        stack->pushString([message UTF8String]);
        stack->executeFunctionByHandler(m_doBillingCallbackScriptHandler, 3);

```

```
60      }];
61  }
```

好了，接下来只需要对C++ Bridge Class做Lua Binding，生成绑定文件，如果用tolua做绑定，绑定配置文件如下：

```
1   class CMGCIAP
2   {
3       static CMGCIAP *sharedInstance();
4
5       void setDoBillingCallbackScriptHandler(LUA_FUNCTION nHandler);
6
7       void doBillingWithUI(const char* billingIndex,
8                            bool isRepeated,
9                            const char* cpParam);
10  }
```

OK，到这里主要的编码工作就完成了，记得要在程序的适当位置做好Lua Binding初始化工作。

如果一切顺利，在以上工作完成后，在Lua里面已经可以直接调用SDK的接口了，接下来的事情就靠你们了。

文章到此也差不多了，整个思路和方案就是这些，如果有什么地方不理解或者不明白，欢迎在原文留言讨论。

来源网址：http://www.cnblogs.com/flyFreeZn/p/4152881.html