lianxu.me

I make Mac software

10个迷惑新手的Cocoa&Objective-c开发问题

1. language background
2. runtime
3. thread
4. runloop
5. protocol, delegate
6. event responder
7. memory management
8. class heritage, category and extensions
9. drawing issue
10. design pattern

首先请谅解我可能使用很多英文，毕竟英文资料将来会是你的主要资料来源。

这篇教程将描述一些我见到的众多Cocoa开发新手遇到的问题和障碍。并不会手把手教你:“这个函数什么意思，哪个函数如何使用”，而是站在一定高度，统观各种技术所处的角色，让你不会迷失在各种技术细节中。在你继续深入学习MacOS编程之前，请停下脚步看清楚这些问题。如果你是新手，这个教程不要希望一次能看的非常透彻，学一定阶段反回来再看看又会有新的体会的。

1. language background

首先c语言背景，必须。 很多人问：“没有任何语言基础，我不想学c直接学objective-c。是否可以？”这里我简单说几句，objc是c的超集，也就是说大部分objc代码其实是c、而且众多传统开源项目都是c写成的。你不学好c在unix世界里只能是个二流开发者！也许说得过于严厉，不过自己斟酌把。c++呢大概了解一下即可，因为它太庞杂了。

接着English,必须。 苹果不会把它们文档都写成中文的。“什么，有人翻译？”等有人闲着翻译出来了的时候，大家都已经卖了很多软件了。你也是跟着人家屁股后面做开发。

2. runtime（运行时）

Objective-c是动态语言, 很多新手或者开发人员常常被runtime这个东西所迷惑。而恰恰这是一个非常重要的概念。我可以这么问：“如果让你（设计）实现一个计算机语言，你要如何下手？”很少程序员这么思考过。但是这么一问，就会强迫你从更高层次思考1以前的问题了。 注意我这句话‘设计’括起来了，稍微次要点，关键是实现。

我把实现分成3种不同的层次:

第一种是传统的面向过程的语言开发，例如c语言。实现c语言编译器很简单，只要按照语法规则实现一个LALR语法分析器就可以了，编译器优化是非常难的topic，不在这里讨论范围内，忽略。 这里我们实现了编译器其中最最基础和原始的目标之一就是把一份代码里的函数名称，转化成一个相对内存地址，把调用这个函数的语句转换成一个jmp跳转指令。在程序开始运行时候，调用语句可以正确跳转到对应的函数地址。这样很好，也很直白，但是太死板了。Everything is predetermined.

我们希望语言更加灵活，于是有了第二种改进，开发面向对象的语言，例如c++。 c++在c的基础上增加了类的部分。但这到底意味着什么呢？我们再写它的编译器要如何考虑呢？其实，就是让编译器多绕个弯，在严格的c编译器上增加一层类处理的机制，把一个函数限制在它处在的class环境里，每次请求一个函数调用，先找到它的对象, 其类型,返回值，参数等等，确定了这些后再jmp跳转到需要的函数。这样很多程序增加了灵活性同样一个函数调用会根据请求参数和类的环境返回完全不同的结果。增加类机制后，就模拟了现实世界的抽象模式，不同的对象有不同的属性和方法。同样的方法，不同的类有不同的行为！ 这里大家就可以看到作为一个编译器开发者都做了哪些进一步的思考。虽然面相对象语言有所改进，但还是死板, 我们仍然叫c++是static language.

希望更加灵活！于是我们完全把上面哪个类的实现部分抽象出来，做成一套完整运行阶段的检测环境，形成第三种，动态语言。这次再写编译器甚至保留部分代码里的sytax名称，名称错误检测，runtime环境注册所以全局的类，函数，变量等等信息等等，我们可以无限的为这个层增加必要的功能。调用函数时候，会先从这个运行时环境里检测所以可能的参数再做jmp跳转。这，就是runtime。编译器开发起来比上面更加弯弯绕。但是这个层极大增加了程序的灵活性。 例如当调用一个函数时候，前2种语言，很有可能一个jmp到了一个非法地址导致程序crash, 但是在这个层次里面，runtime就过滤掉了这些可能性。 这就是为什么dynamic langauge更加强壮。 因为编译器和runtime环境开发人员已经帮你处理了这些问题。

好了上面说着这么多，我们再返回来看objective-c的这些语句：

```
1  id obj=self;
2  if ([obj respondsToSelector:@selector(function1:)) {
3  }
4  if ([obj isKindOfClass:[NSArray class]] ) {
5  }
6  if ([obj conformsToProtocol:@protocol(myProtocol)]) {
7  }
8  if ([[obj class] isSubclassOfClass:[NSArray class]]) {
9  }
10 [obj someNonExistFunction];
```

看似很简单的语句，但是为了让语言实现这个能力，语言开发者要付出很多努力实现runtime环境。这里运行时环境处理了弱类型、函数存在检查工作。runtime会检测注册列表里是否存在对应的函数，类型是否正

确，最后确定下来正确的函数地址，再进行保存寄存器状态，压栈，函数调用等等实际的操作。

```
id knife=[Knife grateKnife];
NSArray *monsterList=[NSArray array];
[monsterList makeObjectsPerformSelector:@selector(killMonster:) withObject:knife];
```

用c,c++完成这个功能还是比较非常麻烦的，但是动态语言处理却非常简单并且这些语句让objc语言更加intuitive。

在Objc中针对对象的函数调用将不再是普通的函数调用，`[obj function1With:var1];`这样的函数调用将被运行时环境转换成`objc_msgSend(target,@selector(function1With:),var1);`。Objc的runtime环境是开源的，所以我们可以拿出一下实现做简单介绍，可以看到objc_msgSend由汇编语言实现，我们甚至不必阅读代码，只需查看注释就可以了解，运行时环境在函数调用前做了比较全面的安全检查，已确保动态语言函数调用不会导致程序crash。对于希望深入学习的朋友可以自行下载Objc-runtime源代码来阅读，这里就不再深入讲解。

```
/********************************************************************
 * id       objc_msgSend(id self,
 *          SEL op,
 *          ...)
 *
 * On entry: a1 is the message receiver,
 *           a2 is the selector
 ********************************************************************/

    ENTRY objc_msgSend
# check whether receiver is nil
    teq     a1, #0
    moveq   a2, #0
    bxeq    lr

# save registers and load receiver's class for CacheLookup
    stmfd   sp!, {a4,v1-v3}
    ldr     v1, [a1, #ISA]

# receiver is non-nil: search the cache
    CacheLookup a2, LMsgSendCacheMiss

# cache hit (imp in ip) - prep for forwarding, restore registers and call
    teq v1, v1      /* set nonstret (eq) */
    ldmfd   sp!, {a4,v1-v3}
    bx      ip

# cache miss: go search the method lists
LMsgSendCacheMiss:
    ldmfd   sp!, {a4,v1-v3}
    b   _objc_msgSend_uncached

LMsgSendExit:
    END_ENTRY objc_msgSend


    .text
    .align 2
_objc_msgSend_uncached:

# Push stack frame
    stmfd   sp!, {a1-a4,r7,lr}
    add     r7, sp, #16
```

```
44      SAVE_VFP
45
46  # Load class and selector
47      ldr a1, [a1, #ISA]      /* class = receiver->isa  */
48      # MOVE  a2, a2          /* selector already in a2 */
49
50  # Do the lookup
51      MI_CALL_EXTERNAL(__class_lookupMethodAndLoadCache)
52      MOVE    ip, a1
53
54  # Prep for forwarding, Pop stack frame and call imp
55      teq v1, v1      /* set nonstret (eq) */
56      RESTORE_VFP
57      ldmfd   sp!, {a1-a4,r7,lr}
58      bx  ip
```

现在说一下runtime的负面影响: 1. 关于执行效率问题。“静态语言执行效率要比动态语言高”，这句没错。因为一部分cpu计算损耗在了runtime过程中，而从上面的汇编代码也可以看出，大概损耗在哪些地方。而静态语言生成的机器指令更简洁。正因为知道这个原因，所以开发语言的人付出很大一部分努力为了保持runtime小巧上。所以objecitve-c是c的超集+一个小巧的runtime环境。 但是，换句话说，从算法角度考虑，这点复杂度不算差别的，Big O notation结果不会有差别。(It's not log(n) vs n^2) 2. 另外一个就是安全性。动态语言由于运行时环境的需求，会保留一些源码级别的程序结构。这样就给破解带来的方便之门。一个现成的说明就是，java,大家都知道java运行在jre上面。这就是典型的runtime例子。它的执行文件.class全部可以反编译回近似源代码。所以这里的额外提示就是如果你需要写和安全有关的代码，离objc远点，直接用c去。

简单理解：“Runtime is everything between your each function call.”

但是大家要明白，第二点我提到runtime并不只是因为它带来了这些简便的语言特性。而是这些简单的语言特性,在实际运用中需要你从完全不同的角度考虑和解决问题。只是计算1+1，很多语言都是一样的，但是随着问题的复杂，项目的增长，静态语言和动态语言就会演化出完全不同的风景。

3. thread

“thread synchronization another notorious trouble!”

记得上学时候学操作系统这门课，里面都会有专门一章介绍任务调度和生产者消费者的问题。 这就是为今后使用进程、线程开发打基础。概念很简单，但难点在synchronization(同步)，因为死锁检测算法不是100%有效，否则就根本没有死锁这个说法了。另一个原因是往往这类错误很隐晦，静态分析很难找到。同时多线程开发抽象度较高需要经验去把握。

总体来说，我见到的在这方面的问题可以分为一下几点：

1. 对系统整体结构认识模糊

不知道多线程开发的几个基点，看别人代码越看越糊涂的。一会NSThread,一会Grand Central Dispatch、

block，一会又看到了pthread等等。Apple封装了很多线程的API, 多线程开发的基本结构入下图:

Mac OS Thread Architecture

可以看到在多线程开发中你可以选择这上面这4种不同的方式。

Mach是核心的操作系统部分。其实这个我也不是非常熟悉，至少我还没有读到过直接使用mach做多线程的代码。

pthread（POSIX Threads）是传统的多线程标准，灵活、轻巧，但是需要理论基础，开发复杂。需要注意一点，根据apple文档提示，在Cocoa下使用pthread需要先启动至少一个NSThread，确定进入多线程环境后才可以。

NSThread是Mac OS 10.0后发布的多线程API较为高层，但是缺乏灵活性，而且和pthread相比效率低下。

Grand Central Dispatch 是10.6后引入的开源多线程库，它介于pthread和NSThread之间。比NSThread更灵活、小巧，并且不需要像pthread一样考虑很多lock的问题。同时objective-c 2.0发布的新语法特性之一blocks，也正是根据Grand Central Dispatch需求推出的。

所以在你写多线程代码或者阅读多线程代码时候，心理要先明确使用哪种技术。

2. thread和Reference Counting内存管理造成的问题。

线程里面的方法都要放到NSAutoreleasePool里面吗?

这类问题很常见，迷惑的原因是不明白 NSAutoreleasePool 到底是干什么用的。NSAutoreleasePool跟

thread其实关系并不显著，它提供一个临时内存管理空间，好比一个沙箱，确保不会有不当的内存分配泄露出来，在这个空间内新分配的对象要向这个pool做一下注册告诉：“pool，我新分配一块空间了”。当pool drain掉或者release，它里面分配过的内存同样释放掉。可见和thread没有很大关系。但是，我们阅读代码的时候经常会看到，新开线程的函数内总是以NSAutoreleasePool开始结束。这又是为什么呢！？因为thread内恰好是最适合需要它的地方！ 线程函数应该计算量大，时间长(supposed to be heavy)。在线程里面可能会有大量对象生成，这时使用autoreleasepool管理更简洁。所以这里的答案是，不一定非要在线程里放NSAutoreleasePool，相对的在cocoa环境下任意地方都可以使用NSAutoreleasePool。如果你在线程内不使用NSAutoreleasePool，要记得在内部alloc和relase配对出现保证没有内存泄露。

3. 线程安全

每个程序都有一个主线程(main thread),它负责处理事件响应，和UI更新。

更新UI问题。很多新手会因为这个问题，导致程序崩溃或出现各种问题。而且逛论坛会看到所以人都会这么告诉你：“不要在后台线程更新你的UI”。其实这个说法不严密，在多线程环境里处理这个问题需要谨慎，而且要了解线程安全特性。

首先我们需要把“UI更新”这个词做一个说明，它可以有2个层次理解，首先是绘制，其次是显示。这里绘制是可以在任何线程里进行，但是要向屏幕显示出来就需要在主线程操作了。我举个例子说明一下，例如现在我们有一个NSImageView，里面设置了一个NSImage, 这时我想给NSImage加个变色滤镜，这个过程就可以理解为绘制。那么我完全可以再另外一个线程做这个比较费时的操作，滤镜增加完毕再通知NSImageView显示一下。另一个例子就是，Twitter客户端会把每一条微博显示成一个cell，但是速度很快，这就是因为它先对cell做了offscreen的渲染，然后再拿出来显示。

所以通过这点我们也可以得到进一步的认识，合理设计view的更新是非常重要的部分。很多新手写得代码片段没错，只是因为放错了地方就导致整个程序出现各种问题。

根据苹果线程安全摘要说明，再其它线程更新view需要使用lockFocusIfCanDraw和unlockFocus锁定，确保不会出现安全问题。

另外还要知道常用容器的线程安全情况。immutable的容器是线程安全的，而mutable容器则不是。例如NSArray和NSMutableArray。

4. Asynchronous（异步） vs. Synchronous（同步）

> *我在一个view要显示多张web图片，我想问一下，我是应该采用异步一个一个下载的方式，还是应该采用多线程同时下载的方式，还是2个都用，那种方式好呢?*

实际上单独用这 2 个方法都不好。并不是简单的用了更多线程就提高速度。这个问题同时涉及客户端和服务器的情况。

处理这种类型的程序，比较好的结构应该是：非主线程建立一个队列(相当于Asynchronous任务)，队列里同时启动n个下载任务(相当于Synchronous任务)。这里的n在2~8左右就够了。这个结构下可以认为队列里面每n个任务之间是异步关系，但是这n个任务之间又是同步关系，每次同时下载2~8张图片。这样处理基本可以满足速度要求和各类服务器限制。

5. thread和runloop

runloop是线程里的一部分，但我觉得有必要单独拿出来写，是因为它涉及的东西比较容易误解，而说明它的文章又不多。

4. runloop

thread和runloop在以前，开发者根本不太当成一个问题。因为没有静态语言里runloop就是固定的线程执行loop。而现在Cocoa新手搞不明白的太多了，因为没有从动态角度看它，首先回想一下第2点介绍的runtime概念，接着出一个思考题。

现在有一个程序片段如下：

```
- (void)myThread:(id)sender
{
    NSAutoreleasePool *pool=[[NSAutoreleasePool alloc] init];
    while (TRUE) {
        //do some jobs
        //break in some condition
        usleep(10000);
        [pool drain];
    }
    [pool release];
}
```

现在要求，做某些设计，使得当这个线程运行的同时，还可以从其它线程里往它里面随意*增加*或*去掉*不同的计算任务。 这，就是NSRunloop的最原始的开发初衷。让一个线程的计算任务更加灵活。 这个功能在c, c++里也许可以做到但是非常难，最主要的是因为语言能力的限制，以前的程序员很少这么去思考。

好，现在我们对上面代码做一个非常简单的进化：

```
NSMutableArray *targetQueue;
NSMutableArray *actionQueue;
- (void)myThread:(id)sender
{
    NSAutoreleasePool *pool=[[NSAutoreleasePool alloc] init];
    while (TRUE) {

        //do some jobs
        //break in some condition
        int n=[targetQueue count];
        assert(n==[actionQueue count]);
        for(int i=0;i&lt;n;i++){
            id target=[targetQueue objectAtIndex:i];
            SEL action=NSSelectorFromString([actionQueue objectAtIndex:i]);
            if ([target respondsToSelector:action]) {
                [target performSelector:action withObject:nil];
```

```
            }
        }

        usleep(10000);

        [pool drain];
    }

    [pool release];
}
```

注意，这里没有做线程安全处理，记住Mutable container is not thread safe. 这个简单的扩展，让我们看到了如何利用runtime能力让线程灵活起来。当我们从另外线程向targetQueue和actionQueue同时加入对象和方法时候，这个线程函数就有了执行一个额外代码的能力。

有人会问,哪里有runloop? 那个是nsrunloop? 看不出来啊。

```
while (TRUE) {
    //break in some condition
}
```

这个结构就叫线程的runloop, 它和NSRunloop这个类虽然名字很像，但完全不是一个东西。以前在使用静态语言开始时候，程序员没有什么迷惑，因为没有NSRunloop这个东西。 我接着来说，这个NSRunloop是如何来得。

第二段扩展代码里面确实没有NSRunloop这个玩意儿，我们接着做第3次改进。 这次我们的目的是把其中动态部分抽象出来。

```
@interface MyNSTimer : NSObject
{
    id target;
    SEL action;
    float interval;
    CFAbsoluteTime lasttime;
}
- (void)invoke;
@end

@implementation MyNSTimer
- (void)invoke;
{
    if ([target respondsToSelector:action]) {
        [target performSelector:action withObject:nil];
    }
}
@end


#!objc
@interface MyNSRunloop : NSObject
{
    NSMutableArray *timerQueue;
}
- (void)addTimer:(MyNSTimer*)t;
- (void)executeOnce;
@end

@implementation MyNSRunloop
```

```
- (void)addTimer:(MyNSTimer*)t;
{
    @synchronized(timerQueue){
        [timerQueue addObject:t];
    }
}
- (void)executeOnce;
{
    CFAbsoluteTime currentTime=CFAbsoluteTimeGetCurrent();
    @synchronized(timerQueue){
        for(MyNSTimer *t in timerQueue){
            if(currentTime-t.lasttime&gt;t.interval){
                t.lasttime=currentTime;
                [t invoke];
            }
        }
    }
}
@end


#!objc
@interface MyNSThread : NSObject
{
    MyNSRunloop *runloop;
}
- (void)main:(id)sender;
@end

@implementation MyNSThread
- (void)main:(id)sender
{
    NSAutoreleasePool *pool=[[NSAutoreleasePool alloc] init];
    while (TRUE) {
        //do some jobs
        //break in some condition
        [runloop executeOnce];
        usleep(10000);
        [pool drain];
    }
    [pool release];
}
@end
```

走到这里，我们就算是基本把Runloop结构抽象出来了。例如我有一个MyNSThread实例，myThread1。我可以给这个实例的线程添加需要的任务，而myThread1内部的MyNSRunloop对象会管理好这些任务。

```
MyNSTimer *timer1=[MyNSTimer scheduledTimerWithTimeInterval:1 target:obj1 selector:@
[myThread1.runloop addTimer:timer1];

MyNSTimer *timer2=[MyNSTimer scheduledTimerWithTimeInterval:2 target:obj2 selector:@
[myThread1.runloop addTimer:timer2];
```

当你看懂了上面的代码也许会感叹，‘原来是这么回事啊！为什么把这么简单的功能搞这么复杂呢？’其实就是这么回事，把Runloop抽象出来可以使得线程任务管理更加loose coupling，给设计模式提供更大的空间。这样第三方开发者不需要过深入的涉及线程内部代码而轻松管理线程任务。另外请注意，这里MyNSRunloop, MyNSTimer等类是我写得一个模拟情况，真实的NSRunloop实现肯定不是这么简单。这里为了说明一个思想。这种思想贯穿整个cocoa framework，从界面更新到event管理。

5. protocol and delegate

这个会列出来因为，我感觉问它的数量仅此于内存管理部分，它们用得很频繁，并且这些是设计模式的重要组成部分。

先从字面解释一下。protocol 是名词协议。delegate 有2个词性，名词是代表，动词是委托。这2个东西是虚、实的对应体。打个现实的比方，protocol 就好比法律，delegate 就好比法院。法律是虚的，是概念，只能通过法院来执行和体现。法院是法律的具体体现，法院要按照法律规定的来操作，绝对不能自己随意发挥。

有了上面的说明后，很多东西就顺理成章了。下面先写一个 protocol:

```
1 @protocol XYZPieChartViewDataSource
2 - (NSUInteger)numberOfSegments;
3 - (CGFloat)sizeOfSegmentAtIndex:(NSUInteger)segmentIndex;
4 - (NSString *)titleForSegmentAtIndex:(NSUInteger)segmentIndex;
5 @end
```

这段代码定义了一个叫 XYZPieChartViewDataSource 的协议。里面包含 3 个方法声明。

我见到过一些学习编程的人，他们总是潜意识的认为程序代码就是用来计算的，那么这个 protocol 怎么计算呢？其实，protocol 的实际计算作用为 0，主要是工程作用。也就是说，它不做任何计算，主要是给程序员阅读用的，就像编辑器左边显示的行数差不多。是一个标记。下面我们来看如何使用这个标记。

```
1 @interface XYZPieChartView : NSView
2 @property (weak) id  dataSource;
3 ...
4 @end
```

这里的类 XYZPieChartView 里包含一个属性 dataSource 它的类型定义是 `id <XYZPieChartViewDataSource>` 这个类型翻译成人类语言就是，dataSource 是一个遵循 XYZPieChartViewDataSource 协议的对象。至于“遵循 XYZPieChartViewDataSource 协议” 到底意味着什么，看看协议定义就可以了。dataSource 这个对象内部必定需要实现

```
1 - (NSUInteger)numberOfSegments;
2 - (CGFloat)sizeOfSegmentAtIndex:(NSUInteger)segmentIndex;
3 - (NSString *)titleForSegmentAtIndex:(NSUInteger)segmentIndex;
```

这 3 个方法。

一个遵循协议的对象，其规定的方法实现有 2 种可能性。第一种，是 Cocoa 框架内部帮你实现了，2 是 Apple 只定义了协议，你必须自己实现这些方法。下面列出 2 个实际开发种的例子给大家对比学习：

NSTableView and NSTableViewDataSource Protocol

表格是最常见的控件。苹果的 NSTableView 就是最典型的例子。玩过 NSTableView 的朋友肯定见过 NSTableViewDataSource 协议。这就是苹果定义协议，你需要自己实现代码的例子。NSTableView 里有

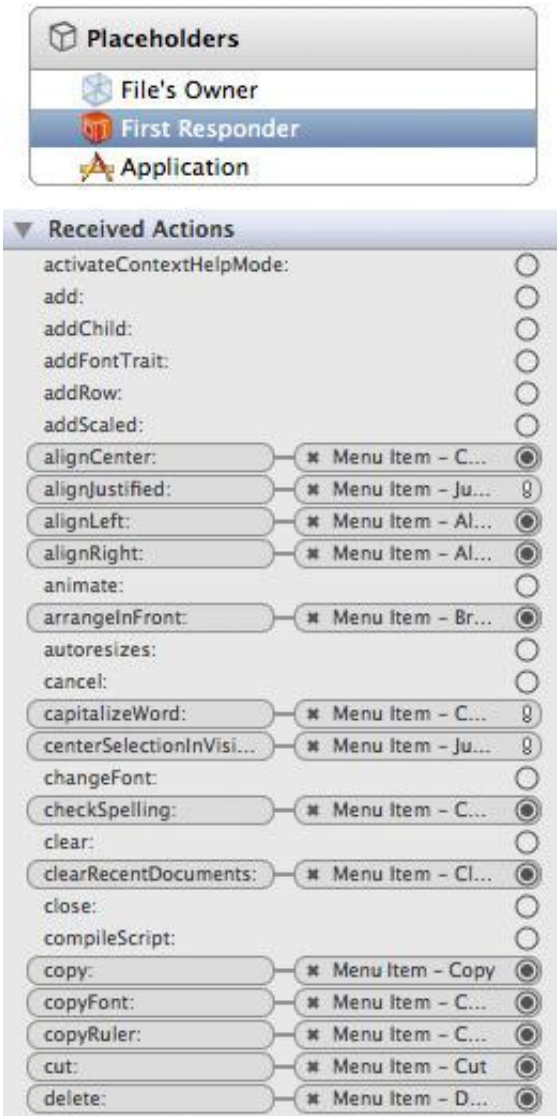
一个属性是

IBOutlet id<NSTableViewDataSource> datasource 当你把一个对象赋值给 tableview.datasource 时，你必须给这个对象内部实现 NSTableViewDataSource 协议规定的方法。因为 tableview 在显示数据的时候，要通过你提高的对象获取数据，而获取数据的方法是由这个 NSTableViewDataSource 协议定义下来的。

NSCoding Protocol

待写...

6. event responder



Interface Builder First Responder

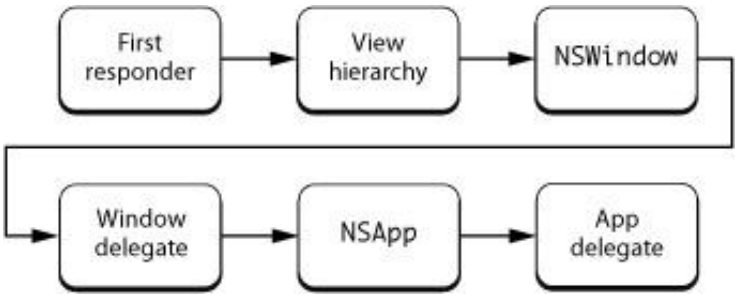
使用过Xcode的开发者都知道Interface Builder这个开发组件，在Xcode4版本以后该组件已经和xcode整合到一起。它是苹果软件开发中非常重要的部分。ib为开发者减轻了很大一部分界面设计工作。但是其中有一

个东西让新接触ib的开发者一头雾水，那就是First Responder, 它是什么东西，为何它会有那么多Actions。这节我会详细介绍如何理解Responder和Cocoa下的事件响应链。

First Responder在IB属性为Placeholders，这意味着它属于一个虚拟实例。就好比TextField里面的string placeholder一样，只是临时显示一下。真正的first responder会被其它对象代替。实际上，任何派生自NSResponder类的对象都可以替代First Responder。而First Responder里面的所有Actions就是NSResponder提供的接口函数，当然你也可以定义自己的响应函数。

MacOS在系统内部会维护一个称为"The Responder Chain"的链表。该列表内容为responder对象实例，它们会对各种系统事件做出响应。最上面的哪个对象就叫做first responder，它是最先接收到系统事件的对象。如果该对象不处理该事件，系统会将这个事件向下传递，直到找到响应事件的对象，我们可以理解为该事件被该这个对象截取了。

The Responder Chain基本结构如下图所示：



The Responder Chain

在理解了上面的概念之后，我希望使用一个例子让大家对responder有更加具体的认识。大家都知道NSTextField这个控件，它是最常见的控件之一。它最基本功能是显示一个字符串，如果启用可选，那么用户可以选中文本，拷贝文本，如果开启编辑选项，还可以运行用户编辑文本，等等基本操作。

下面展示给大家的例子是创建一个我们自己创建的简单textfield叫LXTextField。它不属于NSTextField而是派生自NSView，具有功能显示字符串，全选字符串，响应用户cmd+c的拷贝操作，三个基本功能。注意NSView派生自NSResponder。

```
1  //
2  //  LXTextField.h
3  //  lxtextfield
4  //
5  //  Created by xu lian on 12-03-09.
6  //  Copyright (c) 2012 Beyondcow. All rights reserved.
7  //
8
9  #import &lt;AppKit/AppKit.h&gt;
10
11 @interface LXTextField : NSView
12 {
13     NSString *stringValue;
14     BOOL selectAll;
15 }
16 @property(retain,nonatomic) NSString *stringValue;
```

```
17
18  @end
19
20
21
22  #!objc
23  //
24  //  LXTextField.m
25  //  lxtextfield
26  //
27  //  Created by xu lian on 12-03-09.
28  //  Copyright (c) 2012 Beyondcow. All rights reserved.
29  //
30
31  #import "LXTextField.h"
32
33  @implementation LXTextField
34  @synthesize stringValue;
35
36  - (void)awakeFromNib
37  {
38      selectAll = NO;
39  }
40
41  - (id)initWithFrame:(NSRect)frameRect
42  {
43      if( self = [super initWithFrame:frameRect] ){
44          selectAll = NO;
45      }
46      return self;
47  }
48
49  - (BOOL)acceptsFirstResponder
50  {
51      return YES;
52  }
53
54  - (BOOL)becomeFirstResponder
55  {
56      return YES;
57  }
58
59  - (BOOL)resignFirstResponder
60  {
61      selectAll=NO;
62      [self setNeedsDisplay:YES];
63      return YES;
64  }
65
66
67  - (void)setStringValue:(NSString *)string{
68      stringValue = string;
69      [self setNeedsDisplay:YES];
70  }
71
72  - (void)drawRect:(NSRect)dirtyRect
73  {
74      if (selectAll) {
75          NSRect r = NSZeroRect;
76          r.size = [stringValue sizeWithAttributes:nil];
77          [[NSColor selectedControlColor] set];
78          NSRectFill(r);
79      }
80      [stringValue drawAtPoint:NSZeroPoint withAttributes:nil];
81  }
```

```
82
83  - (IBAction)selectAll:(id)sender;
84  {
85      selectAll=YES;
86      [self setNeedsDisplay:YES];
87  }
88
89  - (IBAction)copy:(id)sender;
90  {
91      NSPasteboard *pasteBoard = [NSPasteboard generalPasteboard];
92      [pasteBoard declareTypes:[NSArray arrayWithObjects:NSStringPboardType, nil] ow
93      [pasteBoard setString:stringValue forType:NSStringPboardType];
94  }
95
96  - (void)mouseDown:(NSEvent *)theEvent
97  {
98      if ([theEvent clickCount]&gt;=2) {
99          selectAll=YES;
100     }
101     [self setNeedsDisplay:YES];
102 }
103
104 - (void)keyDown:(NSEvent *)theEvent
105 {
106 }
107
108 @end
```

运行实例，可以看到随着LXTextField收到系统发送的becomeFirstResponder消息，LXTextField变成responder chain中的frist responder, 这时候可以理解为IB里的哪个First Responder虚拟实例被该LXTextField取代。这时候mainMenu上哪些菜单项,例如：全选(cmd+a), 拷贝(cmd+a)等事件都会最先发给当前这个LXTextField。一旦你的LXTextField实现了NSResponder的哪些默认函数，那么该对象就会截取系统事件。当然这些事件具体如何实现还是需要你自己写代码实现。例如这里的 – (IBAction)copy:(id)sender; 显然我手动实现了textfield的copy能力。

注意上述代码中我实现了一个空函数- (void)keyDown:(NSEvent *)theEvent 这意味着我们希望LXTextField截取键盘事件而不再传递给responder chain后续对象。当然，如果我们希望LXTextField响应特定键盘事件，而其他事件继续传给其他响应对象，我们可以编写如下代码。

```
1  - (void)keyDown:(NSEvent *)theEvent
2  {
3      if(condition){
4          do something;
5      }else{
6          [super keyDown:theEvent];
7      }
8  }
```

待写...

7. memory management

内存管理问题，也许是问得最多的问题了吧。

1. 内存管理规则 Cocoa下程序开发内存管理使用reference counting(RC)机制。从10.7以后apple开始推荐automatic reference counting(ARC)机制。大家是否知道从旧时代的RC到ARC机制到底意味着什么呢？为什么ARC从开发速度，到执行速度和稳定性都要优于rc?

开发速度不言而喻，你少写很多release代码，甚至很少去操心这部分。

执行速度呢？这个还要从runtime说起，还记得我在第 2 点说得一句话么：“Runtime is everything between your each function call.”

RC有一个古老的内存管理哲学：**谁分配谁释放**。 通过counting来确定一个资源有几个使用者。道理很简单，但是往往简单的东西人却会犯错。从来没有一个程序员可以充满信心的说，我写得代码从来没有过内存泄露。这样来看，我们就更需要让程序可以自己处理这个管理机制，这就需要把这个机制放到runtime里。

所以RC->ARC就是把内存管理部分从普通开发者的函数中移到了函数外的runtime中。因为runtime的开发原型简单，逻辑层次更高，所以做这个开发和管理出错的概率更小。实际上编译器开发人员对这部分经过无数次测试，所以可以说用arc几乎不会出错。另外由于编译的额外优化，使得这个部分比程序员自己写得代码要快速很多。而且对于一些通用的开发模式，例如autorelease对象，arc有更优秀的算法保证autoreleasepool里的对象更少。

1. RC规则 首先说一下rc是什么，r-Reference参照，引用 c-counting计数, rc就是引用计数。俗话说就是记录使用者的数量。　例如现在我有一个房间空着，大家可以进去随意使用，但是你进门前，需要给门口的计数牌子+1, 出门时候-1。　这时候这个门口的牌子就是该房间里的人数。一但这个牌子变为 0 我就可以把房间关闭。

这个规则可以让NSObject决定是不是要释放内存。当一个对象alloc时候，系统分配其一块内存并且object自动计数retainCount=1 这时候每当[object retain]一次retainCount+1（这里虽然简写也是rc不过是巧合或者当时开发人员故意选的retain这个词吧）每次[object release]时候retainCount-1　当retainCount==0时候object就真正把这快内存还给系统。

1. 常用container的Reference Counting特性 这个规则很简单把。但是这块确实让新手最头疼的地方。问题出在，新手总想去验证rc规则，又总是发现和自己的期望不符合。 无数次看到有人写下如下句子 NSLog(@”%d”,[object retainCount]);while([object retainCount]>0){ [object release]; }

当然了，我也做过类似的动作，那种希望一切尽在掌握中的心态。但是你会看到其他人告诉这么做完全没有意义，RC规则并不是这么用的。

首先，这个数字也许并不是你心目中的哪个。因为很难跟踪到底哪些地方引用的该资源。你建立的资源不光只有你的代码才会用到，你调用的各种Framework，Framework调用的Framework，都有可能改变这个资源的retainCount。

其次，这里每个数字意味着有其它对象引用该资源，这样的暴力释放很容易导致程序崩溃。就好比，其它人也许可以翻牌子把门口哪个牌子上的数字改变，但是这会出现问题。还有很多人在里面，把牌子变成0房

间锁了结果谁也出不来。又或者，减少牌子上的数字，人进的过多房间变得过于拥挤。

所以去验证rc规则，或者单纯的改变retainCount并不是明智之举。你能做的就是理解规则，使用规则，读文档了解container的引用特性。或者干脆移到 Automatic Reference Counting (ARC) 上面。

> *我有一个NSMutableArray里面保存了1000个NSString对象，我在release的时候需要循环释放1000个string么？还是只需要release NSMutableArray。*

就像上面提到的，如果你了解container的引用特性，这个问题自然就解决了。“NSMutableArray在添加、插入objects时会做retain操作。” 通过这一句话就分析出，用户不否需要帮助NSMutableArray释放1000个string。回忆上面提到的管理哲学，“谁分配谁释放” 编写NSMutableArray的程序员非常熟悉这个规则，NSMutableArray内部retain了，NSMutableArray自然要负责release。但是NSMutableArray才不会管你在外面什么地方引用了这1000个string，它只管理好内部的rc就够了。所以如果你在NSMutableArray外面对1000个string retain了，你自然需要release。相应的，你作为创建这个NSMutableArray的程序员，你只管release这个NSMutableArray就可以了。

最后说一下不用arc的情况。目前情况来看，有不少第三方的库并未支持arc，所以如果你的旧项目使用了这些库，请检查是否作者发布了新版本，或者你需要自己修正支持arc。

8. class heritage, category and extensions

Objective-C 的 OOP 特性提供 subclass 和 category 这2个非常重要的部分。subclass 应该反复被各种编程书籍介绍过。它是 OOP 继承特性的关键语法，它给类添加了延续并且多样化自己的方法。可以说没有继承就没有 OOP 这玩意。而 category 相对于 subclass 就不那么出名了。其实 category 思想出世于 smalltalk，所以它不能算是一个新生事物。

先说一下这2个特性最主要的区别。简单可以这么理解，subclass 体现了类的上下级关系，而 category 是类间的平级关系。

Subclass and Category

如上图所示，左侧是subclass，可以看到class, subclass1, subclass2是递进关系。同时下面的子类完全继承父类的方法，并且可以覆盖父类的方法。subclass2拥有function1,function2,function3三个函数方法。function1的执行代码来自subclass1, function2的执行代码来自于subclass2。

右侧是category。可以看到，无论如何扩展类的category，最终就只有一个类class。category可以说是类的不同方法的小集合，它把一个类的方法划分成不同的区块。请注意观察，每个category块内的方法名称都没有重复的。这正是category的重要要求。

经过上面简单解释了解了这2点的基本区别，现在深入说一下category。

在Objective-c语言设计之初一个主要的哲学观点就是尽量让一个程序员维护庞大的代码集。(对于庞大的项目'原则'和'协议'是非常重要的东西。甚至编写良好的文件名都是非常重要的开发技巧)根据结构化程序设计的经验出发，把一个大块代码划分成一些小块的代码更便于程序员管理。于是objc借用了smalltalk的categories概念。允许程序员把一系列功能相近的方法组织到一个单独的文件内，使得这些代码更容易识别。

更进一步的，和c,c++这种静态语言相比。objc把class categories功能集成到了run-time里面。因此，objc的categories允许程序员为已经存在的类添加新的方法而不需要重新编译旧的类。一旦一个category加入，它可以访问该类所有方法和实例变量，包括私有变量。

category不仅可以为原有class添加方法，而且如果category方法与类内某个方法具有同样的method signature，那么category里的方法将会替换类的原有方法。~~这是category的替换特性。利用这个特性，category还可以用来修复一些bugs。例如已经发布的Framework出现漏洞，如果不便于重新发布新版本，可以使用category替换特性修复漏洞。另外，~~由于category有run-time级别的集成度，所以使得cocoa程序安全性有所下降。许多黑客就是利用 category、posing2、Method Swizzling 等方法破解软件，或者为软件增加新功能。一个很典型的例子就是，我原来发布的QQ表情管理器（目前已经不再维护）。

值得注意的一点是，由于一个类的categories之间是平级关系。所以如果不同categories拥有相同的方法，这个调用结果是未知的：

> *Category methods should not override existing methods (class or instance). Two different categories implementing the same method results in undefined behaviour.*

（因为 Posing、Method Swizzling 这个话题有些深入，本文里我就不介绍了。有兴趣自行Google）

Objc中Categories有其局限的部分，就是你不能为原有的class添加变量，只能添加方法。当然方法里可以添加局部变量。在这个局限基础上就有其它语言做了进一步改进，例如TOM语言就为Categories增加了添加类变量的能力。

自从Objc 2.0以后，语言引入了一个新的特性叫做 Class Extensions, 它可以看做是一类特殊的 category，可以给原有类增加新的属性和方法。

通过 http://developer.apple.com/library/ios/documentation/cocoa/conceptual/ProgrammingWithObjectiveC/CustomizingExistingClasses/CustomizingExistingClasses.html介绍我们可以看出，如果 categories 是为类增加外部方法的话，那么 extensions 就是用做类的内部拓展。

Class extensions 的外观很简单，就是一个 Category 后面括号内的名字为空：

```
1 @interface ClassName ()
2 @end
```

接下来，你就可以给你的类里添加属性，方法了：

```
1 @interface XYZPerson (){
2     id _someCustomInstanceVariable;
3 }
4 @property NSObject *extraProperty;
5 - (void)assignUniqueIdentifier;
6 @end
```

Class extensions 常用来定义类的私有变量和方法。

总上所属，如果你开发时候遇到无论如何都需要为类添加变量的情况，最好的选择就是subclass。相反如果你只希望增加一些函数簇。Categories是最好的选择。而类内部需要用到的私有变量和方法则最好写在 Class extensions 里。

Categories关注的重心是代码设计，把不同功能的方法分离开。在Objc里因为Categories是runtimes级别的特性，所以这种分离不仅体现在源码结构上，同时体现在运行时过程中。这意味着一个category里的方法在程序运行中如果没有被调用，那么它就不会被加载到内存中。所以合理的使用categories会减少你的程序内存消耗。

所以我个人给大家的建议是，每个Cocoa程序员都应该收集整理自己的一套NS类函数的Categories扩展库。这对你今后程序开发效率和掌控情况都有很大提高。

9. Drawing Issues

大家知道，MacOS 是一个非常注重UI的系统。所以在 MacOS 编程里绘制是一个非常重要的部分。第9部分，我会介绍 MacOS 下绘制编程。

从绘制技术分类上看，Cocoa程序员能接触的几种绘制技术列表如下：

1. Cocoa Drawing(NS-prefix)

2. Core Graphics(CG-prefix, called Quartz 2D)
3. Core Animation
4. Core Image
5. OpenGL

在这里我不打算给大家介绍如何绘制具体的按钮或者表格。只是介绍一下，它们的代码风格，优势和限制。

Cocoa Drawing

Cocoa Drawing应该是学习Cocoa程序开发最先接触的绘制技术。也是目前大多数MacOS程序所使用的绘制技术，其底层使用Quartz 2D(Core Graphics)。苹果对应文档为 Cocoa Drawing Guide。Cocoa Drawing并没有统一的绘制函数，所有绘制函数分散在几个主要的NS类的下面。例如, NSImage, NSBezierPath, NSString, NSAttributedString, NSColor, NSShadow，NSGradient ...

所以很简单，当你看到如下代码就可以判断，使用的是Cocoa Drawing方法

```
1 [anImage drawInRect:rect fromRect:NSZeroRect operation:NSCompositeSourceOver fractio
2
3 [@"some text" drawAtPoint:NSZeroPoint withAttributes:attrs];
4
5 NSBezierPath *p=[NSBezierPath bezierPathWithRect:rect];
6 [[NSColor redColor] set];
7 [p fill];
```

这种代码多出现在NSView的drawRect函数内。Cocoa Drawing 的渲染上下文是 NSGraphicsContext，我不断的看到很多新手把 NSGraphicsContext 和 CoreGraphics 的 CGContextRef 搞混。虽然它们很像并且也确实是有关系的，不过如果你不了解当绘制时候的 render context 很多时候将得到一个空白页面的结果。

Core Graphics

Core Graphics 是 Cocoa Drawing layer 的底层技术，在 iOS 开发中非常普遍，因为 iOS 系统中并不存在 Cocoa layer 所以网上可以找到的多是 Core Graphics 绘制代码段子，这给那些不了解 Mac 开发的新手来说造成了很大困扰。Cocoa 是 Mac OS 下的 application framework 而 iOS 下的 application framework 则是 UIKit.framework又叫 Cocoa Touch，它们分享部分代码基础但又不完全一样。例如，Cocoa Touch 下的 UIView 的渲染上下文会使用 UIGraphicsGetCurrentContext() 取得，它得到的是一个 CGContextRef 指针，而在 NSView 里多用 [NSGraphicsContext currentContext] 取得渲染上下文。它得到的是一个 NSGraphicsContext 对象。当然 NSView 里也可以通过 CGContextRef ctx = [[NSGraphicsContext currentContext] graphicsPort]; 来取得一个 Core Graphics 渲染上下文。 可见 Mac OS 下的开发更为灵活一些。因为 iOS 中的 UIKit 开发初期就瞄准了显卡硬件加速，所有 UIView 都是默认 layer-backed 的。iOS 开发者必须使用 Core Graphics 和 Core Animation 这几个相对底层的绘制技术。

请看下面等价代码，作用是绘制一个白色矩形。但是分别使用 Core Graphics 和 Cocoa Drawing:

```
const CGFloat white[]={ 1.0, 1.0, 1.0, 1.0 };
CGContextSetFillColor(cgContextRef, white);
CGContextSetBlendMode(cgContextRef, kCGBlendModeNormal);
CGContextFillRect(cgContextRef, CGRectMake(0, 0, width, height));

[[NSColor whiteColor] set];
NSRectFillUsingOperation(NSMakeRect(0, 0, width, height), NSCompositeSourceOver);
```

可以看出，这是2种*风格完全不同*的绘制技术。Cocoa Drawing 是分散式的绘制函数，而 Core Graphics 是传统的类似 OpenGL 的集成式的绘制方式。其实 Cocoa Drawing 下层是 Core Graphics， Core Graphics 的下层是 OpenGL。

在 OSX 下 NSGraphicsContext 和 CGContextRef 大部分时候是可以相互转换的。

NSGraphicsContext 到 CGContextRef:

```
CGContextRef ctx = [[NSGraphicsContext currentContext] graphicsPort];
CGContextSaveGState(ctx);
//Core Graphic drawing code here
CGContextRestoreGState(ctx);
```

CGContextRef 到 NSGraphicsContext:

```
NSGraphicsContext *ctx = [NSGraphicsContext graphicsContextWithGraphicsPort:cgContex
[NSGraphicsContext saveGraphicsState];
[NSGraphicsContext setCurrentContext:ctx];
//cocoa drawing code here
[NSGraphicsContext restoreGraphicsState];
```

大部分时候使用 Cocoa Drawing 可以绘制出需要的效果，但是某些特殊时候需要 Core Graphic 绘制，例如一些特殊的阴影，clip效果，自定义pattern phase，blending style等等。

Core Animation

如果说 Core Graphics 和 Cocoa Drawing 是通用的 UI 绘制框架的话，那么 CA 显然是界面动画绘制的高级技术。 Core Animation 的对应 Cocoa Animation 部分应该是 NSAnimation 和 NSViewAnimation，但这2个差距比较大。NSAnimation 出现与 OS X 10.4，Core Animation 是 10.5 后出现的。NSViewAnimation 功能和使用相对简单。

简单来说，Core Animation 的作用对象是 CALayer, NSAnimation 的作用对象是 NSView。了解你的程序界面是在处理那种对象很重要。

Core Image

对于这个绘制技术，这篇文章给了我很多启示大家也可以看看。Notes on Rendering 2D Graphics on a Mac 虽然是一篇 note 但是，记录了很多实际应用中的经历，可以对个各种绘制技术有一个比较全面的解析。

根据此文的介绍。Core Image 适合处理小量大图，而非常不适合处理大量小图。因为 CI 利用 GPU 运算，

而数据到 GPU 的round-trip 时间数量级在 millisecond。这就意味着，1000 张小图分别再 GPU 运算，时间至少再 1000*1 ms。此文作者尝试绘制3000张小图片，利用 Cocoa Drawing 原本耗时 750ms，但是改用CI后耗时猛增到3秒。

所以，这就是CI在osx绘制技术里所处的宏观角色：单图做实时处理。

openGL

待写...

10. design pattern

待写...

1:这里其实很有意思，为何我用"更高层次思考"，而不是"更底层次"。作为一个编译器和语言开发人员，面对的问题确实更底层没错，但是他们思考的维度更高，更抽象，这样子。一个不算恰当的比方就好像一个三维世界的人处理二维世界的一条线的问题。

2:Posing技术在10.5以后deprecated，并且64bit run-time也不再支持

最近事情比较忙。 我只能忙里偷闲过来接着写这篇教程。大家莫怪。

This entry was posted in Tutorial and tagged Cocoa, Objective-C on November 14, 2012 [http://lianxu.me/2012/11/10-cocoa-objc-newbie-problems/] by keefo.

杨萧玉 • 2个月前

posting应该改为posing吧，还有Quazrtz 2D应该是Quartz 2D嘿嘿

回复 • 分享 ›

keefo 管理员 ➦ 杨萧玉 • 2个月前

谢谢指正，已经修改。欢迎你这样热心的朋友。

回复 • 分享 ›

杨萧玉 ➦ keefo • 2个月前

:)

回复 • 分享 ›

X • 4个月前

博主好牛，大牛!

回复 • 分享 ›

Esay • 8个月前

好文!

回复 • 分享 ›

jack • 1年前

非常好

回复 • 分享 ›

dopcn • 1年前

期待delegate部分

回复 • 分享 ›

byszhao • 2年前

非常不错，通俗易懂

回复 • 分享 ›

teddy • 2年前

好文章 很感谢!

回复 • 分享 ›

嘘嘘 • 2年前

好帅。。。很有意思，通熟易懂

回复 • 分享 ›

mxmcc • 2年前

runloop 那块写的好棒,一看就懂

回复 • 分享 ›

riven • 2年前

话说用Category给原来的类添加属性可以用objc_setAssociatedObject和objc_getAssociatedObject呀。

回复 · 分享 ›

wzxit1314 · 2年前

Objc中Categories有其局限的部分，就是你不能为原有的class添加变量，只能添加方法。
--> 使用关联(Associated)可以为categoriets添加变量.

回复 · 分享 ›

keefo 管理员 ➚ wzxit1314 · 2年前

那个叫做Objective-C 2.0 Class Extensions而不是Associated。文档是:
http://developer.apple.com/lib...

回复 · 分享 ›

leolovenet · 2年前

读完表示感谢

1 回复 · 分享 ›

6david9 · 2年前

runloop那块介绍的不错，但是category那块说分类可以替换原有类的方法，我保留意见。尽量避免在开发中这么写，分类只是用来扩展，而不是改错。替换方法可以用class_replaceMethod.

回复 · 分享 ›

余猛 唐 ➚ 6david9 · 2年前

可以使用Method Swizzling。替换原有的方法。不过使用category也可以实现。但多个category的相同方法中的调用顺序没有保证。

回复 · 分享 ›

Sasori · 2年前

arc不是runtime管理的，是编译器优化的吧

1 回复 · 分享 ›

keefo 管理员 ➚ Sasori · 2年前

谢谢指出, 已经改正.

回复 · 分享 ›

riven ➚ keefo · 2年前

确实是编译器行为，和Runtime没关系

回复 · 分享 ›

刘志 · 2年前

你那个yy播放器的代码能给看看吗？公司最近要开发os音频播放，可惜以前做的是ios，这方面没经验，369590327@qq.com。

3 ^ | v • 回复 • 分享 ›

xpjian • 2年前

长见识了。

^ | v • 回复 • 分享 ›

327beckham • 2年前

新手看过之后，表示很好。虽然有的看懂了，有的没看懂。很喜欢有前人像这样对知识的梳理。让我这样的新人能更快的学习。谢谢

^ | v • 回复 • 分享 ›