Hexo

2017-11-27

一种基于KVO的页面加载，渲染耗时监控方法

打广告：有兴趣加入阿里巴巴手淘基础架构平台移动高可用团队的请微博联系我@盗版五子棋

和同事zb一起维护了一个ARM64的专栏iOS调试进阶，有兴趣的可以读读

在介绍本文之前，请先允许我提出一个问题，如果你要无痕监控任意一个页面（UIViewController及其子类）的加载或者渲染时间，你会怎么做。

很多人都会想到说用AOP啊，利用 Method Swizzling 来进行方法替换从而获得方法调用耗时。比如我们有一个 ViewController ，如果其实现了一个 viewDidLoad 方法进行睡眠5秒，如下所示：

```
@implementation ViewController

- (void)viewDidLoad
{
    [super viewDidLoad];
    sleep(5);
}

@end
```

相信很多人的第一直觉会是如下AOP代码（我们省略Method Swizzling相关的代码）：

```
@implementation UIViewController (TestCase)

+ (void)load
{
    static dispatch_once_t onceToken;
    dispatch_once(&onceToken, ^{
        wzq_swizzleMethod([UIViewController class], @selector(viewDidLoad), @se
    });
}

- (void)wzq_viewDidLoad
{
    NSDate *date = [NSDate date];
    [self wzq_viewDidLoad];

    NSTimeInterval duration = [[NSDate date] timeIntervalSinceDate:date];
    NSLog(@"Page %@ cost %g in viewDidLoad", [self class], duration);
}

@end
```

但是，如果你自己尝试了你会发现，你测算的时间压根不是**5**秒。

为什么呢？其原因在于我们 Method Swizzling 的时候，因为采用了对基类 UIViewController 进行替换，获取到的 viewDidLoad 对应的IMP是属于基类 UIViewController 的，而并不是 ViewController 自身覆写的，所以我们监控的其实从子类 ViewController 调用 [super viewDidLoad] 的时候调用基类IMP的耗时。

好，看到这，有人就想了对应的方法，把 -[ViewController viewDidLoad] 的IMP替换掉就行了。方法很多种，比如创建一个 ViewController 的**Category**进行替换。**但是这种方法你好像没办法任意对某个页面进行替换。**

> 有人说你可以runtime遍历所有类判断是不是 UIViewController 的子类，然后动态替换。理论是可行的，效率嘛，是比较低的。

方案

根据上述我们所知的缺陷，我们需要有一个兼顾动态性和性能的方案，能够直接获取到子类的IMP，这样才能达到我们对于页面加载渲染时间（ viewDidLoad，viewDidAppear 和 viewWillAppear ）监控的需求。

基于这个需求，我很快想到了**基于KVO**的方案（如果你对KVO不了解，可以参考我的文章：KVO在不同的二进制中多个符号并存的Crash问题）。我们知道，在对于任意对象进行KVO监控的时候，iOS底层实际上帮你动态创建了一个隐蔽的类，同时帮了做了大量的 `setter`，`getter` 函数的override，并调用原来类对应函数实现，从而让你神不知鬼不觉的以为你还在用原来的类进行操作。

那我们该怎么做呢?

1. 对我们需要监听的类的实例进行KVO，随便监听一个不存在的KeyPath。我们压根不需要KVO的任何回调，我们只是需要它能帮我们**创建子类**而已。
2. 对KVO创建出来的子类添加我们需要Swizzle的方法对应的SEL及其IMP。因为本质上KVO只是对setter和getter方法进行了**override**，**如果我们不提供我们自己的实现，还是会调用到原来的类的IMP。**
3. 在实例销毁的时候，将KVO监听移除，不然会导致**KVO still registering when deallocated这样的Crash。**

总体来说，我们需要做的就是三件事。

1. 对实例进行KVO

KVO方法只能在对象实例上进行操作，我们首先要获取到的就是 `UIViewController` 及其子类的实例。

遍历头文件，发现UIViewController的初始化方法比较少，归纳为如下三种：

```
init
initWithCoder:
initWithNibName:bundle:
```

我们先Swizzle这几个方法：

```
 wzq_swizzleMethod([UIViewController class], @selector(initWithNibName:bundle:)
wzq_swizzleMethod([UIViewController class], @selector(initWithCoder:), @selecto
wzq_swizzleMethod([UIViewController class], @selector(init), @selector(wzq_init
```

这几个方法调用的时候，实例对象对应的内存已经分配出来了，无非就是构造函数还没赋值，但是我们也能进行KVO了。KVO的代码如下所示：

```
NSString *identifier = [NSString stringWithFormat:@"wzq_%@", [[NSProcessInfo pr
[vc addObserver:[NSObject new] forKeyPath:identifier options:NSKeyValueObservin
```

2. 添加我们想要的方法

我们刚刚已经对页面实例进行了KVO操作，此时对于原先类别为 ViewController 的 vc 对象来说，内部其实已经变成**NSKVONotifying_ViewController类型了。**。如果我们想对其所在的类型添加方法的话，不能直接用 [vc class]，因为这个方法已经被内部override成了 ViewController 。我们需要使用 object_getClass 这个类进行真正的类型获取，如下所示：

```
 // NSKVONotifying_ViewController
Class kvoCls = object_getClass(vc);
// ViewController
Class originCls = class_getSuperclass(kvoCls);

// 获取原来实现的encoding
const char *originViewDidLoadEncoding = method_getTypeEncoding(class_getInstanc
const char *originViewDidAppearEncoding = method_getTypeEncoding(class_getInsta
const char *originViewWillAppearEncoding = method_getTypeEncoding(class_getIns

// 重点，添加方法。
class_addMethod(kvoCls, @selector(viewDidLoad), (IMP)wzq_viewDidLoad, originVi
class_addMethod(kvoCls, @selector(viewDidAppear:), (IMP)wzq_viewDidAppear, ori
class_addMethod(kvoCls, @selector(viewWillAppear:), (IMP)wzq_viewWillAppear, o
```

上述代码非常通俗易懂，不再赘述，替换完的方法如下，我们以 wzq_viewDidLoad 举例：

```
static void wzq_viewDidLoad(UIViewController *kvo_self, SEL _sel)
{
    Class kvo_cls = object_getClass(kvo_self);
    Class origin_cls = class_getSuperclass(kvo_cls);

    // 注意点
    IMP origin_imp = method_getImplementation(class_getInstanceMethod(origin_c
    assert(origin_imp != NULL);

    void(*func)(UIViewController *, SEL) =  (void(*)(UIViewController *, SEL))

    NSDate *date = [NSDate date];

    func(kvo_self, _sel);

    NSTimeInterval duration = [[NSDate date] timeIntervalSinceDate:date];
    NSLog(@"Class %@ cost %g in viewDidLoad", [kvo_self class], duration);
}
```

重点关注下上述代码中的**注意点**，之前我们在KVO生成的类中对应添加了原本没有的实现，因此 -[ViewController viewDidLoad] 会走到我们的 wzq_viewDidLoad 方法中，但是我们怎么才能调用到原来的 viewDidLoad 的呢？我们之前并没有保存对应的IMP呀。

这里还是利用了KVO的特殊性：内部生成的NSKVONotifying_ViewController实际上是继承自ViewController的

因此， `Class origin_cls = class_getSuperclass(kvo_cls);` 实际上获取到了 `ViewController` 类，我们从中取出对应的IMP，进行直接调用即可。

3. 移除KVO

我们利用Associate Object去移除就好了。一个对象释放的时候会自动去清除其所在的 `assoicate object` 。

基于这个原理，我们可以实现如下代码：

我们构建一个桩，把所有无用的KVO监听都设置给这个桩，如下所示：

```
[vc addObserver:[WZQKVOObserverStub stub] forKeyPath:identifier options:NSKeyV
```

然后我们构建一个移除器，这个移除器弱引用保存了vc的实例和对应的keypath，如下：

```
WZQKVORemover *remover = [WZQKVORemover new];
remover.obj = vc;
remover.keyPath = identifier.copy;
```

然后我们把这个移除器利用 `associate object` 设置给对应的vc。

```
objc_setAssociatedObject(vc, &wzq_associateRemoveKey, remover, OBJC_ASSOCIATIO
```

而在对应的移除器的 `dealloc` 方法里，我们把kvo监听给移除就可以了。

```
- (void)dealloc
{
#ifdef DEBUG
    NSLog(@"WZQKVORemover called");
#endif
    if (_obj) {
        [_obj removeObserver:[WZQKVOObserverStub stub] forKeyPath:_keyPath];
    }
}
```

额外

利用 associate object 移除KVO的正确性是有保障的，具体见runtime中 associate object 的源码：

```
void objc_removeAssociatedObjects(id object)
{
    if (object && object->hasAssociatedObjects()) {
        _object_remove_assocations(object);
    }
}

void _object_remove_assocations(id object) {
    vector< ObjcAssociation,ObjcAllocator<ObjcAssociation> > elements;
    {
        AssociationsManager manager;
        AssociationsHashMap &associations(manager.associations());
        if (associations.size() == 0) return;
        disguised_ptr_t disguised_object = DISGUISE(object);
        AssociationsHashMap::iterator i = associations.find(disguised_object);
        if (i != associations.end()) {
            // copy all of the associations that need to be removed.
            ObjectAssociationMap *refs = i->second;
            for (ObjectAssociationMap::iterator j = refs->begin(), end = refs->
                elements.push_back(j->second);
            }
            // remove the secondary table.
            delete refs;
            associations.erase(i);
        }
    }
    // the calls to releaseValue() happen outside of the lock.
    for_each(elements.begin(), elements.end(), ReleaseValue());
}
```

代码

本文的工程代码托管在Github上，包含了测试用例（默认带干扰测试），还没来得及搞成cocoapods，等我有时间了搞一下。但是你直接拖拽源码使用是一点问题都没有。

#iOS

Comments Share

OLDER
注意系统库的坑之load函数调用多次

TAGS

iOS (32)
shell (1)

TAG CLOUD

Android C Growth JavaScript Math Performance R Reverse Engineering Swift XNU iOS shell

ARCHIVES

November 2017 (2)
October 2017 (1)
September 2017 (2)
August 2017 (1)
July 2017 (1)
June 2017 (3)
April 2017 (2)
January 2017 (2)
October 2016 (1)
September 2016 (1)
July 2016 (1)
May 2016 (1)
April 2016 (2)
March 2016 (4)
February 2016 (5)
December 2015 (6)
November 2015 (5)
October 2015 (4)
September 2015 (4)

RECENTS

一种基于KVO的页面加载，渲染耗时监控方法
注意系统库的坑之load函数调用多次
iOS内存abort(Jetsam) 原理探究
基于桥的全量方法Hook方案 - 探究苹果主线程检查实现
KVO在不同的二进制中多个符号并存的Crash问题