iOS开发之Runtime常用示例总结

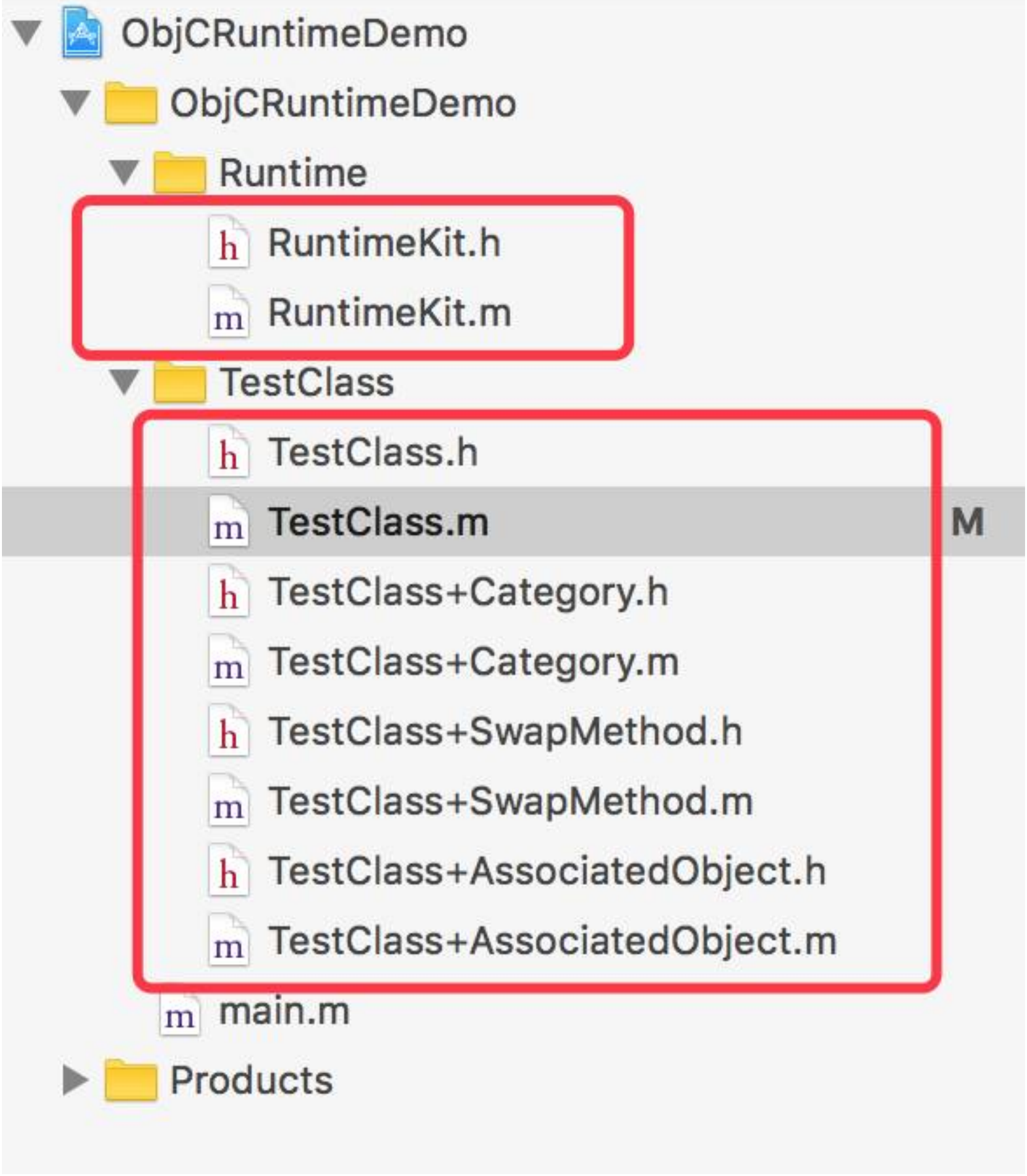
2017-02-28 青玉伏案 Cocoa开发者社区

经常有小伙伴私下在Q上问一些关于Runtime的东西，问我有没有Runtime的相关博客，之前还真没正儿八经的总结过。之前只是在解析第三方框架源码时，聊过一些用法，也就是这些第三方框架中用到的Runtime。比如属性关联，动态获取属性等等。本篇博客就针对Runtime这个主题来总结一些其常用的一些方法，当然“空谈误国”，今天博客中所聊的Runtime依然要依托于本篇博客所涉及的Demo。

本篇博客所聊的Runtime的内容大概有：动态获取类名、动态获取类的成员变量、动态获取类的属性列表、动态获取类的方法列表、动态获取类所遵循的协议列表、动态添加新的方法、类的实例方法实现的交换、动态属性关联、消息发送与消息转发机制等。当然，本篇博客总结的是运行时常用的功能，并不是所有Runtime的内容。

一、构建Runtime测试用例

本篇博客的内容是依托于实例的，所以我们在本篇博客中先构建我们的测试类，Runtime将会对该类进行相关的操作。下方就是本篇博客所涉及Demo的目录，上面的RuntimeKit类是讲Runtime常用的功能进行了简单的封装，而下方的TestClass以及相关的类目就是我们Runtime要操作的对象了。下方会对TestClass以及类目中的内容进行详细介绍。



下方这几个截图就是我们的测试类TestClass的主要部分，因为TestClass是专门用来测试的类，所以其涉及的内容要尽量的全面。TestClass遵循了NSCoding, NSCopying这两个协议，并且为其添加了公有属性、私有属性、私有成员变量、 公有实例方法、私有实例方法、类方法等。这些添加的内容，都将是我们Runtime的操作对象。下方那几个TestClass的类目稍后在使用Runtime时再进行介绍。

```objc
#import <Foundation/Foundation.h>

@interface TestClass : NSObject<NSCoding, NSCopying>
@property (nonatomic, strong) NSArray *publicProperty1;
@property (nonatomic, strong) NSString *publicProperty2;

+ (void)classMethod: (NSString *)value;
- (void)publicTestMethod1: (NSString *)value1 Second: (NSString *)value2;
- (void)publicTestMethod2;

- (void)method1;
@end
```

```objc
@interface TestClass(){
    NSInteger _var1;
    int _var2;
    BOOL _var3;
    double _var4;
    float _var5;
}
@property (nonatomic, strong) NSMutableArray *privateProperty1;
@property (nonatomic, strong) NSNumber *privateProperty2;
@property (nonatomic, strong) NSDictionary *privateProperty3;
@end
```

```objc
- (void)privateTestMethod1 {
    NSLog(@"privateTestMethod1");
}

- (void)privateTestMethod2 {
    NSLog(@"privateTestMethod2");
}

#pragma mark - 方法交换时使用
- (void)method1 {
    NSLog(@"我是Method1的实现");
}
```

二、RuntimeKit的封装

接下来我们就来看看RuntimeKit中的内容，其中对Runtime常用的方法进行了简单的封装。主要是动态的获取类的一些属性和方法的，以及动态方法添加和方法交换的。本部分的干货还是不

少的。

1、获取类名

动态的获取类名是比较简单的，使用class_getName(Class)就可以在运行时来获取类的名称。class_getName()函数返回的是一个char类型的指针，也就是C语言的字符串类型，所以我们要将其转换成NSString类型，然后再返回出去。下方的+fetchClassName:方法就是我们封装的获取类名的方法，如下所示：

```
/**
 获取类名

 @param class 相应类
 @return NSString: 类名
 */
+ (NSString *)fetchClassName:(Class)class {
    const char *className = class_getName(class);
    return [NSString stringWithUTF8String:className];
}
```

2、获取成员变量

下方这个+fetchIvarList:这个方法就是我们封装的获取类的成员变量的方法。当然我们在获取成员变量时，可以用ivar_getTypeEncoding()来获取相应成员变量的类型。使用ivar_getName()来获取相应成员变量的名称。下方就是对获取成员变量的功能的封装。返回的是一个数组，数组的元素是一个字典，而字典中存储的就是相应成员变量的名称和类型。

```
/**
 获取成员变量

 @param class Class
 @return NSArray
 */
+ (NSArray *)fetchIvarList:(Class)class {
    unsigned int count = 0;
    Ivar *ivarList = class_copyIvarList(class, &count);

    NSMutableArray *mutableList = [NSMutableArray arrayWithCapacity:count];
    for (unsigned int i = 0; i < count; i++ ) {
        NSMutableDictionary *dic = [NSMutableDictionary dictionaryWithCapacity:2];
        const char *ivarName = ivar_getName(ivarList[i]);
        const char *ivarType = ivar_getTypeEncoding(ivarList[i]);
        dic[@"type"] = [NSString stringWithUTF8String: ivarType];
        dic[@"ivarName"] = [NSString stringWithUTF8String: ivarName];

        [mutableList addObject:dic];
    }
    free(ivarList);
    return [NSArray arrayWithArray:mutableList];
}
```

下方就是调用上述方法获取的TestClass类的成员变量。当然在运行时就没有什么私有和公有之

分了，只要是成员变量就可以获取到。在OC中的给类添加成员属性其实就是添加了一个成员变量和getter以及setter方法。所以获取的成员列表中肯定带有成员属性，不过成员属性的名称前方添加了下划线来与成员属性进行区分。我们也可以获取成员变量的类型，下方的_var1是NSInteger类型，动态获取到的是q字母，其实是NSInteger的符号。而i就表示int类型，c表示Bool类型，d表示double类型，f则就表示float类型。当然这些基本类型都是由一个字母代替的，如果是引用类型的话，则直接就是一个字符串了，比如NSArray类型就是"@NSArray"。

```
获取TestClass的成员变量列表:(
        {
        ivarName = "_var1";
        type = q;
    },
        {
        ivarName = "_var2";
        type = i;
    },
        {
        ivarName = "_var3";
        type = c;
    },
        {
        ivarName = "_var4";
        type = d;
    },
        {
        ivarName = "_var5";
        type = f;
    },
        {
        ivarName = "_publicProperty1";
        type = "@\"NSArray\"";
    },
        {
        ivarName = "_publicProperty2";
        type = "@\"NSString\"";
    },
```

```
        ivarName = "_privateProperty1";
        type = "@\"NSMutableArray\"";
    },
        {
        ivarName = "_privateProperty2";
        type = "@\"NSNumber\"";
    },
        {
        ivarName = "_privateProperty3";
        type = "@\"NSDictionary\"";
    }
)
```

3.获取成员属性

上面获取的是类的成员变量，那么下方这个+fetchPropertyList:获取的就是成员属性。当然此刻获取的只包括成员属性，也就是那些有setter或者getter方法的成员变量。下方主要是使用了class_copyPropertyList(Class,&count)来获取的属性列表，然后通过for循环通过property_getName()来获取每个属性的名字。当然使用property_getName()获取到的名字依然是C语言的char类型的指针，所以我们还需要将其转换成NSString类型，然后放到数组中一并返回。如下所示：

```
/**
 获取类的属性列表，包括私有和公有属性，以及定义在延展中的属性

 @param class Class
 @return 属性列表数组
 */
+ (NSArray *)fetchPropertyList:(Class)class {
    unsigned int count = 0;
    objc_property_t *propertyList = class_copyPropertyList(class, &count);

    NSMutableArray *mutableList = [NSMutableArray arrayWithCapacity:count];
    for (unsigned int i = 0; i < count; i++ ) {
        const char *propertyName = property_getName(propertyList[i]);
        [mutableList addObject:[NSString stringWithUTF8String: propertyName]];
    }
    free(propertyList);
    return [NSArray arrayWithArray:mutableList];
}
```

下方这个截图就是调用上述方法获取的TestClass的所有的属性，当然dynamicAddProperty是我们使用Runtime动态给TestClass添加的，所以也是可以获取到的。当然我们获取到的属性的名称为了与其对应的成员变量进行区分，成员属性的名字前边是没有下划线的。

```
2017-01-18 11:25:22.834362 ObjCRuntimeDemo[28550:6087125]
获取TestClass的属性列表:(
    dynamicAddProperty,
    privateProperty1,
    privateProperty2,
    privateProperty3,
    publicProperty1,
    publicProperty2
)
```

4、获取类的实例方法

接下来我们就来封装一下获取类的实例方法列表的功能，下方这个+fetchMethodList:就是我们封装的获取类的实例方法列表的函数。在下方函数中，通过class_copyMethodList()方法获取类的实例方法列表，然后通过for循环使用method_getName()来获取每个方法的名称，然后将方法的名称转换成NSString类型，存储到数组中一并返回。具体代码如下所示：

```objc
/**
 获取类的实例方法列表: getter, setter, 对象方法等。但不能获取类方法

 @param class class description
 @return return value description
 */
+ (NSArray *)fetchMethodList:(Class)class {
    unsigned int count = 0;
    Method *methodList = class_copyMethodList(class, &count);

    NSMutableArray *mutableList = [NSMutableArray arrayWithCapacity:count];
    for (unsigned int i = 0; i < count; i++ ) {
        Method method = methodList[i];
        SEL methodName = method_getName(method);
        [mutableList addObject:NSStringFromSelector(methodName)];
    }
    free(methodList);
    return [NSArray arrayWithArray:mutableList];
}
```

下方这个截图就是上述方法在TestClass上运行的结果，其中打印了TestClass类的所有实例方法，当然其中也必须得包含成员属性的getter和setter方法。当然TestClass类目中的方法也是必须能获取到的。结果如下所示：

```
2017-01-18 11:25:22.834626 ObjCRuntimeDemo[28550:6087125]
获取TestClass的方法列表: (
    "dynamicAddMethod:",
    "publicTestMethod1:Second:",
    publicTestMethod2,
    privateTestMethod1,
    privateTestMethod2,
    method1,
    publicProperty1,
    "setPublicProperty1:",
    publicProperty2,
    "setPublicProperty2:",
    privateProperty1,
    "setPrivateProperty1:",
    privateProperty2,
    "setPrivateProperty2:",
    privateProperty3,
    "setPrivateProperty3:",
    dynamicAddProperty,
    "setDynamicAddProperty:",
    categoryMethod,
    swapMethod,
    method2,
    ".cxx_destruct",
    "methodSignatureForSelector:",
    "forwardInvocation:",
    "forwardingTargetForSelector:"
)
```

5、获取协议列表

下方是获取我们类所遵循协议列表的方法，主要使用了class_copyProtocolList()来获取列表，然后通过for循序使用protocol_getName()来获取协议的名称，最后将其转换成NSString类型放入数组中返回即可。

```objc
/**
 获取协议列表

 @param class class description
 @return return value description
 */
+ (NSArray *)fetchProtocolList:(Class)class {
    unsigned int count = 0;
    __unsafe_unretained Protocol **protocolList = class_copyProtocolList(class, &count);

    NSMutableArray *mutableList = [NSMutableArray arrayWithCapacity:count];
    for (unsigned int i = 0; i < count; i++ ) {
        Protocol *protocol = protocolList[i];
        const char *protocolName = protocol_getName(protocol);
        [mutableList addObject:[NSString stringWithUTF8String: protocolName]];
    }

    return [NSArray arrayWithArray:mutableList];
    return nil;
}
```

下方就是我们获取到的TestClass类所遵循的协议列表：

```
2017-01-18 11:25:22.834691 ObjCRuntimeDemo[28550:6087125]
获取TestClass的协议列表: (
    NSCoding,
    NSCopying
)
```

6、动态添加方法实现

下方就是动态的往相应类上添加方法以及实现。下方的+addMethod方法有三个参数，第一个参数是要添加方法的类，第二个参数是方法的SEL，第三个参数则是提供方法实现的SEL。稍后在消息发送和消息转发时会用到下方的方法。下方主要是使用class_getInstanceMethod()和method_getImplementation()这两个方法相结合获取相应SEL的方法实现。下方的IMP其实就是Implementation的方法缩写，获取到相应的方法实现后，然后再调用class_addMethod()方法将IMP与SEL进行绑定即可。具体做法如下所示。

```
/**
 往类上添加新的方法与其实现

 @param class 相应的类
 @param methodSel 方法的名
 @param methodSelImpl 对应方法实现的方法名
 */
+ (void)addMethod:(Class)class method:(SEL)methodSel method:(SEL)methodSelImpl {
    Method method = class_getInstanceMethod(class, methodSelImpl);
    IMP methodIMP = method_getImplementation(method);
    const char *types = method_getTypeEncoding(method);
    class_addMethod(class, methodSel, methodIMP, types);
}
```

7、方法实现交换

下方就是讲类的两个方法的实现进行交换。如果将MethodA与MethodB的方法实现进行交换的话，调用MethodA时就会执行MethodB的内容，反之亦然。

下方这段代码就是对上述方法的测试。下方是TestClass的一个类目，在该类目中将类目中的方法与TestClass中的方法进行了替换。也就是将method1与method2进行了替换，替换后在method2中调用的method2其实就是调用的method1。在第三方库中，经常会使用该特性，已达到AOP编程的目的。

三、属性关联

属性关联说白了就是在类目中动态的为我们的类添加相应的属性，如果看过之前发布的对Masonry框架源码解析的博客的话，对下方的属性关联并不陌生。在Masonry框架中就利用Runtime的属性关联在UIView的类目中给UIView添加了一个约束数组，用来记录添加在当前View上的所有约束。下方就是在TestClass的类目中通过objc_getAssociatedObject()和objc_setAssociatedObject()两个方法为TestClass类添加了一个dynamicAddProperty属性。上面我们获取到的属性列表中就含有该动态添加的成员属性。

下方就是属性关联的具体代码，如下所示。

四、消息处理与消息转发

在Runtime中不得不提的就是OC的消息处理和消息转发机制。当然网上也有不少相关资料，本篇博客为了完整性，还是要聊一下消息处理与消息转发的。当你调用一个类的方法时，先在本类中的方法缓存列表中进行查询，如果在缓存列表中找到了该方法的实现，就执行，如果找不到就在本类中的方列表中进行查找。在本类方列表中查找到相应的方法实现后就进行调用，如果没找到，就去父类中进行查找。如果在父类中的方法列表中找到了相应方法的实现，那么就执行，否则就执行下方的几步。

当调用一个方法在缓存列表，本类中的方法列表以及父类的方法列表找不到相应的实现时，到程序崩溃阶段中间还会有几步让你来挽救。接下来就来看看这几步该怎么走。

1.消息处理（Resolve Method）

当在相应的类以及父类中找不到类方法实现时会执行+resolveInstanceMethod:这个类方法。该方法如果在类中不被重写的话，默认返回NO。如果返回NO就表明不做任何处理，走下一步。如果返回YES的话，就说明在该方法中对这个找不到实现的方法进行了处理。在该方法中，我们可以为找不到实现的SEL动态的添加一个方法实现，添加完毕后，就会执行我们添加的方法实

现。这样，当一个类调用不存在的方法时，就不会崩溃了。具体做法如下所示：

2、消息快速转发

如果不对上述消息进行处理的话，也就是+resolveInstanceMethod:返回NO时，会走下一步消息转发，即-forwardingTargetForSelector:。该方法会返回一个类的对象，这个类的对象有SEL对应的实现，当调用这个找不到的方法时，就会被转发到SecondClass中去进行处理。这也就是所谓的消息转发。当该方法返回self或者nil, 说明不对相应的方法进行转发，那么就该走下一步了。

3.消息常规转发

如果不将消息转发给其他类的对象，那么就只能自己进行处理了。如果上述方法返回self的话，会执行-methodSignatureForSelector:方法来获取方法的参数以及返回数据类型，也就是说该方法获取的是方法的签名并返回。如果上述方法返回nil的话，那么消息转发就结束，程序崩溃，报出找不到相应的方法实现的崩溃信息。

在+resolveInstanceMethod:返回NO时就会执行下方的方法，下方也是讲该方法转发给SecondClass，如下所示：

今天的博客就先到这儿吧，当然还有其他一些Runtime的东西本篇博客并未涉及，如果以后解析那个开源库的源码时遇到了，我们在单独聊。依照惯例，本篇博客依附的Demo, 仍然会在Github上进行分享，下方是分享链接。

github源码分享链接：https://github.com/lizelu/ObjCRuntimeDemo

阅读原文