Method Swizzling 和 AOP 实践

顾 鹏 07 January 2015

上一篇介绍了 Objective-C Messaging。利用 Objective-C 的 Runtime 特性，我们可以给语言做扩展，帮助解决项目开发中的一些设计和技术问题。这一篇，我们来探索一些利用 Objective-C Runtime 的黑色技巧。这些技巧中最具争议的或许就是 Method Swizzling 。

介绍一个技巧，最好的方式就是提出具体的需求，然后用它跟其他的解决方法做比较。

所以，先来看看我们的需求：*对 App 的用户行为进行追踪和分析*。简单说，就是当用户看到某个 `View` 或者点击某个 `Button` 的时候，就把这个事件记下来。

手动添加

最直接粗暴的方式就是在每个 `viewDidAppear` 里添加记录事件的代码。

```
@implementation MyViewController ()

- (void)viewDidAppear:(BOOL)animated
{
    [super viewDidAppear:animated];

    // Custom code

    // Logging
    [Logging logWithEventName:@“my view did appear”];
}


- (void)myButtonClicked:(id)sender
{
    // Custom code

    // Logging
    [Logging logWithEventName:@“my button clicked”];
}
```

这种方式的缺点也很明显：它破坏了代码的干净整洁。因为 Logging 的代码本身并不属于 ViewController 里的主要逻辑。随着项目扩大、代码量增加，你的 ViewController 里会到处散布着 Logging 的代码。这时，要找到一段事件记录的代码会变得困难，也很容易忘记添加事件记录的代码。

你可能会想到用继承或类别，在重写的方法里添加事件记录的代码。代码可以是长的这个样子：

```
@implementation UIViewController ()

- (void)myViewDidAppear:(BOOL)animated
{
    [super viewDidAppear:animated];

    // Custom code

    // Logging
    [Logging logWithEventName:NSStringFromClass([self class])];
}
```

```
- (void)myButtonClicked:(id)sender
{
    // Custom code

    // Logging
    NSString *name = [NSString stringWithFormat:@“my button in %@ is clicke
    [Logging logWithEventName:name];
}
```

Logging 的代码都很相似，通过继承或类别重写相关方法是可以把它从主要逻辑中剥离出来。但同时也带来新的问题：

1. 你需要继承 UIViewController, UITableViewController, UICollectionViewController 所有这些 ViewController ，或者给他们添加类别；
2. 每个 ViewController 里的 ButtonClick 方法命名不可能都一样；
3. 你不能控制别人如何去实例化你的子类；
4. 对于类别，你没办法调用到原来的方法实现。大多时候，我们重写一个方法只是为了添加一些代码，而不是完全取代它。

5. 如果有两个类别都实现了相同的方法，运行时没法保证哪一个类别的方法会给调用。

Method Swizzling

Method Swizzling 利用 Runtime 特性把一个方法的实现与另一个方法的实现进行替换。

上一篇文章 有讲到每个类里都有一个 Dispatch Table ，将方法的名字（SEL）跟方法的实现（IMP，指向 C 函数的指针）一一对应。Swizzle 一个方法其实就是在程序运行时在 Dispatch Table 里做点改动，让这个方法的名字（SEL）对应到另个 IMP 。

首先定义一个类别，添加将要 Swizzled 的方法：

```
@implementation UIViewController (Logging)

- (void)swizzled_viewDidAppear:(BOOL)animated
{
    // call original implementation
    [self swizzled_viewDidAppear:animated];

    // Logging
    [Logging logWithEventName:NSStringFromClass([self class])];
}
```

代码看起来可能有点奇怪，像递归不是么。当然不会是递归，因为在 runtime 的时候，函数实现已经被交换了。调用 `viewDidAppear:` 会调用你实现的 `swizzled_viewDidAppear:`，而在 `swizzled_viewDidAppear:` 里调用 `swizzled_viewDidAppear:` 实际上调用的是原来的 `viewDidAppear:` 。

接下来实现 swizzle 的方法：

```
@implementation UIViewController (Logging)

void swizzleMethod(Class class, SEL originalSelector, SEL swizzledSelector)
{
    // the method might not exist in the class, but in its superclass
    Method originalMethod = class_getInstanceMethod(class, originalSelector
    Method swizzledMethod = class_getInstanceMethod(class, swizzledSelector
```

```
    // class_addMethod will fail if original method already exists
    BOOL didAddMethod = class_addMethod(class, originalSelector, method_get

    // the method doesn't exist and we just added one
    if (didAddMethod) {
        class_replaceMethod(class, swizzledSelector, method_getImplementati
    }
    else {
        method_exchangeImplementations(originalMethod, swizzledMethod);
    }
}
```

这里唯一可能需要解释的是 `class_addMethod` 。要先尝试添加原 selector 是为了做一层保护，因为如果这个类没有实现 `originalSelector` ，但其父类实现了，那 `class_getInstanceMethod` 会返回父类的方法。这样 `method_exchangeImplementations` 替换的是父类的那个方法，这当然不是你想要的。所以我们先尝试添加 `orginalSelector` ，如果已经存在，再用 `method_exchangeImplementations` 把原方法的实现跟新的方法实现给交换掉。

最后，我们只需要确保在程序启动的时候调用 `swizzleMethod` 方法。比如，我们可以在之前 `UIViewController` 的 Logging 类别里添加 `+load:` 方法，然后在 `+load:` 里把 `viewDidAppear` 给替换掉：

```
@implementation UIViewController (Logging)

+ (void)load
{
    swizzleMethod([self class], @selector(viewDidAppear:), @selector(swizzl
}
```

一般情况下，类别里的方法会重写掉主类里相同命名的方法。如果有两个类别实现了相同命名的方法，只有一个方法会被调用。但 `+load:` 是个特例，当一个类被读到内存的时候， runtime 会给这个类及它的每一个类别都发送一个 `+load:` 消息。

其实，这里还可以更简化点：直接用新的 IMP 取代原 IMP ，而不是替换。只需要有全局的函数指针指向原 IMP 就可以。

```
void (gOriginalViewDidAppear)(id, SEL, BOOL);

void newViewDidAppear(UIViewController *self, SEL _cmd, BOOL animated)
{
    // call original implementation
    gOriginalViewDidAppear(self, _cmd, animated);

    // Logging
    [Logging logWithEventName:NSStringFromClass([self class])];
}

+ (void)load
{
    Method originalMethod = class_getInstanceMethod(self, @selector(viewDid
    gOriginalViewDidAppear = (void *)method_getImplementation(originalMetho

    if(!class_addMethod(self, @selector(viewDidAppear:), (IMP) newViewDidAp
        method_setImplementation(originalMethod, (IMP) newViewDidAppear);
    }
}
```

通过 Method Swizzling，我们成功把逻辑代码跟处理事件记录的代码解耦。

当然除了 Logging，还有很多类似的事务，如 Authentication 和 Caching。这些事务琐碎，跟主要业务逻辑无关，在很多地方都有，又很难抽象出来单独的模块。这种程序设计问题，业界也给了他们一个名字 - Cross Cutting Concerns。

而像上面例子用 Method Swizzling 动态给指定的方法添加代码，以解决 Cross Cutting Concerns 的编程方式叫：Aspect Oriented Programming

Aspect Oriented Programming （面向切面编程）

Wikipedia 里对 AOP 是这么介绍的:

> *An aspect can alter the behavior of the base code by applying advice (additional behavior) at various join points (points in a program) specified in a quantification or query called a pointcut (that detects whether a given join point matches).*

在 Objective-C 的世界里，这句话意思就是利用 Runtime 特性给指定的方法添加自定义代码。有很多方式可以实现 AOP，Method Swizzling 就是其中之

一。而且幸运的是，目前已经有一些第三方库可以让你不需要了解 Runtime ，就能直接开始使用 AOP 。

Aspects 就是一个不错的 AOP 库，封装了 Runtime ， Method Swizzling 这些黑色技巧，只提供两个简单的API：

```
+ (id<AspectToken>)aspect_hookSelector:(SEL)selector
                           withOptions:(AspectOptions)options
                            usingBlock:(id)block
                                 error:(NSError **)error;
- (id<AspectToken>)aspect_hookSelector:(SEL)selector
                          withOptions:(AspectOptions)options
                           usingBlock:(id)block
                                error:(NSError **)error;
```

使用 Aspects 提供的 API，我们之前的例子会进化成这个样子：

```
@implementation UIViewController (Logging)

+ (void)load
{
    [UIViewController aspect_hookSelector:@selector(viewDidAppear:)
                              withOptions:AspectPositionAfter
                               usingBlock:^(id<AspectInfo> aspectInfo) {
        NSString *className = NSStringFromClass([[aspectInfo instance] clas
        [Logging logWithEventName:className];
                               } error:NULL];
}
```

你可以用同样的方式在任何你感兴趣的方法里添加自定义代码，比如 IBAction 的方法里。更好的方式，你提供一个 Logging 的配置文件作为唯一处理事件记录的地方：

```
@implementation AppDelegate (Logging)

+ (void)setupLogging
{
    NSDictionary *config = @{
        @"MainViewController": @{
            GLLoggingPageImpression: @"page imp - main page",
            GLLoggingTrackedEvents: @[
                @{
```

```
                    GLLoggingEventName: @"button one clicked",
                    GLLoggingEventSelectorName: @"buttonOneClicked:",
                    GLLoggingEventHandlerBlock: ^(id<AspectInfo> aspectInfo
                        [Logging logWithEventName:@"button one clicked"];
                    },
                },
                @{
                    GLLoggingEventName: @"button two clicked",
                    GLLoggingEventSelectorName: @"buttonTwoClicked:",
                    GLLoggingEventHandlerBlock: ^(id<AspectInfo> aspectInfo
                        [Logging logWithEventName:@"button two clicked"];
                    },
                },
            ],
        },

        @"DetailViewController": @{
            GLLoggingPageImpression: @"page imp - detail page",
        }
    };

    [AppDelegate setupWithConfiguration:config];
}

+ (void)setupWithConfiguration:(NSDictionary *)configs
{
    // Hook Page Impression
    [UIViewController aspect_hookSelector:@selector(viewDidAppear:)
                              withOptions:AspectPositionAfter
                               usingBlock:^(id<AspectInfo> aspectInfo) {
                                       NSString *className = NSStringFromCl
                                    [Logging logWithEventName:className];
                               } error:NULL];

    // Hook Events
    for (NSString *className in configs) {
        Class clazz = NSClassFromString(className);
        NSDictionary *config = configs[className];

        if (config[GLLoggingTrackedEvents]) {
            for (NSDictionary *event in config[GLLoggingTrackedEvents]) {
                SEL selekor = NSSelectorFromString(event[GLLoggingEventSele
                AspectHandlerBlock block = event[GLLoggingEventHandlerBlock

                [clazz aspect_hookSelector:selekor
                                   withOptions:AspectPositionAfter
                                    usingBlock:^(id<AspectInfo> aspectInfo) {
                                        block(aspectInfo);
```

```
                                    } error:NULL];

            }
        }
    }
}
```

然后在 `-application:didFinishLaunchingWithOptions:` 里调用 `setupLogging`:

```
- (BOOL)application:(UIApplication *)application didFinishLaunchingWithOpti
    // Override point for customization after application launch.

    [self setupLogging];
    return YES;
}
```

最后的话

利用 objective-C Runtime 特性和 Aspect Oriented Programming，我们可以把琐碎事务的逻辑从主逻辑中分离出来，作为单独的模块。它是对面向对象编程模式的一个补充。Logging 是个经典的应用，这里做个抛砖引玉，发挥想象力，可以做出其他有趣的应用。

使用 Aspects 完整的例子可以从这里获得：AspectsDemo。

如果你有什么问题和想法，欢迎留言或者发邮件给我 peng@glowing.com 进行讨论。

Reference

- method-swizzling
- method replacement for fun and profit
- Aspects

Live with less, share with more. weibo: @no-computer

如何自己动手实现 KVO

本文是 Objective-C Runtime 系列文章的第三篇。如果你对 Objective-C

Runtime 还不是很了解，可以先去看看前两篇文章： Objective-C Runtime Method Swizzling 和 AOP 实践 本篇会探究 KVO (Key-Value…