

34-iOS黑魔法RuntimeMethodSwizzling背后的原理

你好，我是戴铭。

提到Object-C中的Runtime，你可能一下就想到了iOS的黑魔法Method Swizzling。毕竟，这个黑魔法可以帮助我们运行时进行方法交换，或者在原方法执行之前插入自定义方法，以保证在业务面向对象编程方式不被改变的情况下，进行切面功能的开发。但是，运行时进行方法交换同时也会带来一定的风险。所以，今天我就来和你详细聊聊Runtime Method Swizzling 的原理。

Runtime Method Swizzling 编程方式，也可以叫作AOP（Aspect-Oriented Programming，面向切面编程）。

AOP 是一种编程范式，也可以说是一种编程思想，使用 AOP 可以解决 OOP（Object Oriented Programming，面向对象编程）由于切面需求导致单一职责被破坏的问题。通过 AOP 可以不侵入 OOP 开发，非常方便地插入切面需求功能。

比如，我在专栏[第9篇文章](#)中介绍无侵入埋点方案时，就提到了通过 AOP 在不侵入原有功能代码的情况下插入收集埋点的功能。

除此之外，还有一些主业务无关的逻辑功能，也可以通过 AOP 来完成，这样主业务逻辑就能够满足 OOP 单一职责的要求。而如果没有使用 AOP，鉴于OOP的局限性，这些与主业务无关的代码就会到处都是，增大了工作量不说，还会加大维护成本。

但是我们也知道，iOS 在运行时进行 AOP 开发会有风险，不能简单地使用 Runtime 进行方法交换来实现 AOP 开发。因此，我今天就来跟你说下直接使用 Runtime 方法交换开发的风险有哪些，而安全的方法交换原理又是怎样的？

直接使用 Runtime 方法交换开发的风险有哪些？

Objective-C 是门动态语言，可以在运行时做任何它能做的事情。这其中的功劳离不开 Runtime 这个库。正因为如此，Runtime 成为了 iOS 开发中 Objective-C 和 C 的分水岭。

Runtime 不光能够进行方法交换，还能够在运行时处理 Objective-C 特性相关（比如类、成员函数、继承）的增删改操作。

苹果公司已经开源了Runtime，在 GitHub 上有[可编译的 Runtime 开源版本](#)。你可以通过于德志 (@halfrost) 博客的三篇 Runtime 文章，即[isa和Class](#)、[消息发送与转发](#)，以及[如何正确使用Runtime](#)，来一边学习一边调试。

直接使用 Runtime 进行方法交换非常简单，代码如下：

```
#import "SMHook.h"
#import <objc/runtime.h>

@implementation SMHook

+ (void)hookClass:(Class)classObject fromSelector:(SEL)fromSelector toSelector:(SEL)toSelector {
    Class class = classObject;
```

```
// 得到被交换类的实例方法
Method fromMethod = class_getInstanceMethod(class, fromSelector);
// 得到交换类的实例方法
Method toMethod = class_getInstanceMethod(class, toSelector);

// class_addMethod() 函数返回成功表示被交换的方法没实现，然后通过 class_addMethod() 函数先实现；返回失败则表示被交
if(class_addMethod(class, fromSelector, method_getImplementation(toMethod), method_getTypeEncoding(toMe
    // 进行方法的交换
    class_replaceMethod(class, toSelector, method_getImplementation(fromMethod), method_getTypeEncoding
} else {
    // 交换 IMP 指针
    method_exchangeImplementations(fromMethod, toMethod);
}
}
@end
```

如代码所示：通过 `class_getInstanceMethod()` 函数可以得到被交换类的实例方法和交换类的实例方法。使用 `class_addMethod()` 函数来添加方法，返回成功表示被交换的方法没被实现，然后通过 `class_addMethod()` 函数实现；返回失败则表示被交换方法已存在，可以通过 `method_exchangeImplementations()` 函数直接进行 IMP 指针交换以实现方法交换。

但是，像上面这段代码一样，直接使用 Runtime 的方法进行方法交换会有很多风险，[RSSwizzle](#) 库里指出了四个典型的直接使用 Runtime 方法进行方法交换的风险。我稍作整理，以方便你查看，并便于你理解后续的内容。

第一个风险是，需要在 `+load` 方法中进行方法交换。因为如果在其他时候进行方法交换，难以保证另外一个线程中不会同时调用被交换的方法，从而导致程序不能按预期执行。

第二个风险是，被交换的方法必须是当前类的方法，不能是父类的方法，直接把父类的实现拷贝过来不会起作用。父类的方法必须在调用的时候使用，而不是方法交换时使用。

第三个风险是，交换的方法如果依赖了 `cmd`，那么交换后，如果 `cmd` 发生了变化，就会出现各种奇怪问题，而且这些问题还很难排查。特别是交换了系统方法，你无法保证系统方法内部是否依赖了 `cmd`。

第四个风险是，方法交换命名冲突。如果出现冲突，可能会导致方法交换失败。

更多关于运行时方法交换的风险，你可以查看 Stackoverflow 上的问题讨论 “[What are the Dangers of Method Swizzling in Objective C?](#)”。

可以看到，直接使用 Runtime 进行方法交换的风险非常大，那么安全的方法交换是怎样的呢？接下来，我就来跟你介绍一个更安全的运行时方法交换库 [Aspects](#)。

更安全的方法交换库Aspects

Aspects 是一个通过 Runtime 消息转发机制来实现方法交换的库。它将所有的方法调用都指到 `_objc_msgForward` 函数调用上，按照自己的方式实现了消息转发，自己处理参数列表，处理返回值，最后通过 `NSInvocation` 调用来实现方法交换。同时，Aspects 还考虑了一些方法交换可能会引发的风险，并进行了处理。

通过学习Aspects 的源码，你能够从中学习到如何处理这些风险。比如，热修复框架 [JSPatch](#)就是学习了Aspects 的实现方式。因此，接下来我会展开Aspects的源码，带你一起看看它是如何解决这些问题的。这样，你再遇到类似问题时，或借鉴其中的解决思路，或经过实践、思考后形成自己的更优雅解决方法。

虽然 Aspects 对于一些风险进行了规避，但是在使用不当的情况下依然会有风险，比如 hook 已经被 hook 过的方法，那么之前的 hook 会失效，而且新的 hook 也会出错。所以，即使是 Aspects，在工程中也不能滥用。

现在，我们先一起看一段**如何使用 Aspects 的示例代码**：

```
[UIViewController aspect_hookSelector:@selector(viewWillAppear:) withOptions:AspectPositionAfter usingBlock
    NSLog(@"View Controller %@ will appear animated: %tu", aspectInfo.instance, animated);
} error:NULL];
```

上面这段代码是 Aspects 通过运行时方法交换，按照 AOP 方式添加埋点的实现。代码简单，可读性高，接口使用 Block 也非常易用。按照这种方式，直接使用Aspects即可。

接下来，我就跟你说下 **Aspect 实现方法交换的原理**。

Aspects 的整体流程是，先判断是否可进行方法交换。这一步会进行安全问题的判断处理。如果没有风险的话，再针对要交换的是类对象还是实例对象分别进行处理。

- 对于类对象的方法交换，会先修改类的 forwardInvocation，将类的实现转成自己的。然后，重新生成一个方法用来交换。最后，交换方法的 IMP，方法调用时就会直接对交换方法进行消息转发。
- 对于实例对象的方法交换，会先创建一个新的类，并将当前实例对象的 isa 指针指向新创建的类，然后再修改类的方法。

整个流程的入口是 aspect_add() 方法，这个方法里包含了 Aspects 的两个核心方法，第一个是进行安全判断的 aspect_isSelectorAllowedAndTrack 方法，第二个是执行类对象和实例对象方法交换的 aspect_prepareClassAndHookSelector 方法。

aspect_isSelectorAllowedAndTrack 方法，会对一些方法比如 retain、release、autorelease、forwardInvocation 进行过滤，并对 dealloc 方法交换做了限制，要求只能使用 AspectPositionBefore 选项。同时，它还会过滤没有响应的方法，直接返回 NO。

安全判断执行完，就开始执行方法交换的 aspect_prepareClassAndHookSelector 方法，其实现代码如下：

```
static void aspect_prepareClassAndHookSelector(NSObject *self, SEL selector, NSError **error) {
    NSCParameterAssert(selector);
    Class klass = aspect_hookClass(self, error);
    Method targetMethod = class_getInstanceMethod(klass, selector);
    IMP targetMethodIMP = method_getImplementation(targetMethod);
    if (!aspect_isMsgForwardIMP(targetMethodIMP)) {
        // 创建方法别名
        const char *typeEncoding = method_getTypeEncoding(targetMethod);
```

```
SEL aliasSelector = aspect_aliasForSelector(selector);
if (![klass instancesRespondToSelector:aliasSelector]) {
    __unused BOOL addedAlias = class_addMethod(klass, aliasSelector, method_getImplementation(target),
        NSCAssert(addedAlias, @"Original implementation for %@ is already copied to %@ on %@", NSStringFromClass(klass), NSStringFromClass(target), NSStringFromClass(klass)));
}

// 使用 forwardInvocation 进行方法交换。
class_replaceMethod(klass, selector, aspect_getMsgForwardIMP(self, selector), typeEncoding);
AspectLog(@"Aspects: Installed hook for -[%@ %@].", klass, NSStringFromSelector(selector));
}
}
```

可以看到，通过 `aspect_hookClass()` 函数可以判断出 class 的 selector 是实例方法还是类方法，如果是实例方法，会通过 `class_addMethod` 方法生成一个交换方法，这样在 `forwardInvocation` 时就能够直接执行交换方法。`aspect_hookClass` 还会对类对象、元类、KVO 子类化的实例对象、class 和 isa 指向不同的情况进行处理，使用 `aspect_swizzleClassInPlace` 混写 baseClass。

小结

在今天这篇文章中，我和你梳理了直接使用 Runtime 进行方法交换会有哪些问题，进而为了解决这些问题，我又和你分享了一个更安全的方法交换库 Aspects。

在文章最后，我想你说的是，对于运行时进行方法交换，有的开发者在碰到了几次问题之后，就敬而远之了，但其实很多问题在你了解了原因后就不那么可怕了。比如说，了解更多运行时原理和优秀方法交换库的实现细节，能够增强你使用运行时方法交换的信心，从而这个技术能够更好地为你提供服务，去帮助你更加高效地去解决某一类问题。

课后作业

你是怎么使用方法交换的？用的什么库？和 Aspects 比，这些库好在哪儿？

感谢你的收听，欢迎你在评论区给我留言分享你的观点，也欢迎把它分享给更多的朋友一起阅读。

iOS 开发高手课

从原理到实战，带你解决 80% 的开发难题

戴 铭

前滴滴出行技术专家



新版升级：点击「👤请朋友读」，20位好友免费读，邀请订阅更有**现金**奖励。

精选留言：

- Usama Bin Laden 2019-05-28 08:47:05
方法交换，都没用过库，都是直接写的。。。