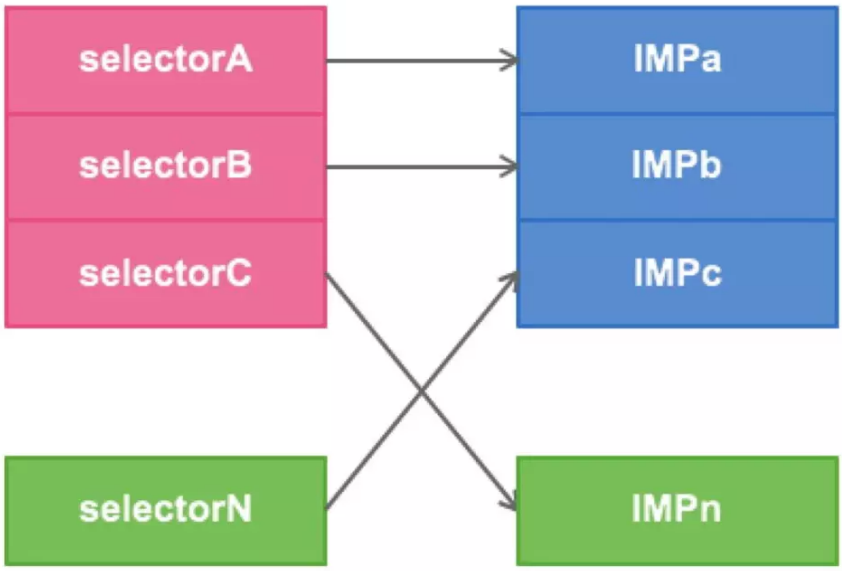
iOS开发·runtime原理与实践: 方法交换篇(Method Swizzling)(iOS“黑魔法”，埋点统计，禁止UI控件连续点击，防奔溃处理)

摘要：编程，只了解原理不行，必须实战才能知道应用场景。本系列尝试阐述runtime相关理论的同时介绍一些实战场景，而本文则是本系列的**方法交换**篇。**本文中**，第一节将介绍方法交换及注意点，第二节将总结一下方法交换相关的API，第三节将介绍方法交换几种的实战场景：统计VC加载次数并打印，防止UI控件短时间多次激活事件，防奔溃处理（数组越界问题）。

1. 原理与注意

原理

Method Swizzing是发生在运行时的，主要用于在运行时将两个Method进行交换，我们可以将Method Swizzling代码写到任何地方，但是只有在这段Method Swilzzling代码执行完毕之后互换才起作用。



用法

先给要替换的方法的类添加一个Category，然后在Category中的`+(void)load`方法中添加Method Swizzling方法，我们用来替换的方法也写在这个Category中。

由于load类方法是程序运行时这个类被加载到内存中就调用的一个方法，执行比较早，并且不需要我们手动调用。

注意要点

- Swizzling应该总在+load中执行
- Swizzling应该总是在dispatch_once中执行
- Swizzling在+load中执行时，不要调用[super load]。如果多次调用了[super load]，可能会出现“Swizzle无效”的假象。
- 为了避免Swizzling的代码被重复执行，我们可以通过GCD的dispatch_once函数来解决，利用dispatch_once函数内代码只会执行一次的特性。

2. Method Swizzling相关的API

- 方案1

```
class_getInstanceMethod(Class _Nullable cls, SEL _Nonnull name)
```

```
method_getImplementation(Method _Nonnull m)
```

```
class_addMethod(Class _Nullable cls, SEL _Nonnull name, IMP _Nonnull
imp,
                const char * _Nullable types)
```

```
class_replaceMethod(Class _Nullable cls, SEL _Nonnull name, IMP
_Nonnull imp,
                    const char * _Nullable types)
```

- 方案2

```
method_exchangeImplementations(Method _Nonnull m1, Method _Nonnull m2)
```

3. 应用场景与实践

3.1 统计VC加载次数并打印

- UIViewController+Logging.m

```
#import "UIViewController+Logging.h"
#import <objc/runtime.h>

@implementation UIViewController (Logging)

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

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

```
    // Logging
    NSLog(@"%@", NSStringFromClass([self class]));
}

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_getImplementation(swizzledMethod),
method_getTypeEncoding(swizzledMethod));

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

}
```

3.2 防止UI控件短时间多次激活事件

> 需求

当前项目写好的按钮，还没有全局地控制他们短时间内不可连续点击（也许有过零星地在某些网络请求接口之前做过一些控制）。现在来了新需求：本APP所有的按钮1秒内不可连续点击。你怎么做？一个个改？这种低效率低维护度肯定是不妥的。

> 方案

给按钮添加分类，并添加一个点击事件间隔的属性，执行点击事件的时候判断一下是否时间到了，如果时间不到，那么拦截点击事件。

怎么拦截点击事件呢？其实点击事件在runtime里面是发送消息，我们可以把要发送的消息的SEL 和自己写的SEL交换一下，然后在自己写的SEL里面判断是否执行点击事

件。

> 实践

UIButton是UIControl的子类，因而根据UIControl新建一个分类即可

- UIControl+Limit.m

```
#import "UIControl+Limit.h"
#import <objc/runtime.h>

static const char
*UIControl_acceptEventInterval="UIControl_acceptEventInterval";
static const char *UIControl_ignoreEvent="UIControl_ignoreEvent";

@implementation UIControl (Limit)

#pragma mark - acceptEventInterval
- (void)setAcceptEventInterval:(NSTimeInterval)acceptEventInterval
{
    objc_setAssociatedObject(self,UIControl_acceptEventInterval,
@(acceptEventInterval), OBJC_ASSOCIATION_RETAIN_NONATOMIC);
}

-(NSTimeInterval)acceptEventInterval {
    return [objc_getAssociatedObject(self,UIControl_acceptEventInterval)
doubleValue];
}

#pragma mark - ignoreEvent
-(void)setIgnoreEvent:(BOOL)ignoreEvent{
    objc_setAssociatedObject(self,UIControl_ignoreEvent, @(ignoreEvent),
OBJC_ASSOCIATION_ASSIGN);
}

-(BOOL)ignoreEvent{
    return [objc_getAssociatedObject(self,UIControl_ignoreEvent)
boolValue];
}

#pragma mark - Swizzling
+(void)load {
    Method a =
class_getInstanceMethod(self,@selector(sendAction:to:forEvent:));
    Method b =
class_getInstanceMethod(self,@selector(swizzled_sendAction:to:forEvent:));
    method_exchangeImplementations(a, b);//交换方法
}
```

```
- (void)swizzled_sendAction:(SEL)action to:(id)target forEvent:
(UIEvent*)event
{
    if(self.ignoreEvent){
        NSLog(@"btnAction is intercepted");
        return;}
    if(self.acceptEventInterval>0){
        self.ignoreEvent=YES;
        [self performSelector:@selector(setIgnoreEventWithNo)
withObject:nil afterDelay:self.acceptEventInterval];
    }
    [self swizzled_sendAction:action to:target forEvent:event];
}

-(void)setIgnoreEventWithNo{
    self.ignoreEvent=NO;
}

@end
```

- ViewController.m

```
-(void)setupSubViews{

    UIButton *btn = [UIButton new];
    btn =[[UIButton alloc]initWithFrame:CGRectMake(100,100,100,40)];
    [btn setTitle:@"btnTest"forState:UIControlStateNormal];
    [btn setTitleColor:[UIColor redColor]forState:UIControlStateNormal];
    btn.acceptEventInterval = 3;
    [self.view addSubview:btn];
    [btn addTarget:self
action:@selector(btnAction)forControlEvents:UIControlEventTouchUpInside];
}

- (void)btnAction{
    NSLog(@"btnAction is executed");
}
```

3.3 防奔溃处理：数组越界问题

> 需求

在实际工程中，可能在一些地方（比如取出网络响应数据）进行了数组NSArray取数据

的操作，而且以前的小哥们也没有进行防越界处理。测试方一不小心也没有测出数组越界情况下奔溃（因为返回的数据是动态的），结果以为没有问题了，其实还隐藏的生产事故的风险。

这时APP负责人说了，即使APP即使不能工作也不能Crash，这是最低的底线。那么这对数组越界的情况下的奔溃，你有没有办法拦截?

思路：对NSArray的`objectAtIndex:`方法进行Swizzling，替换一个有处理逻辑的方法。但是，这时候还是有个问题，就是类簇的Swizzling没有那么简单。

> 类簇

在iOS中NSNumber、NSArray、NSDictionary等这些类都是类簇(Class Clusters)，一个NSArray的实现可能由多个类组成。所以如果想对NSArray进行Swizzling，必须获取到其**“真身”**进行Swizzling，直接对NSArray进行操作是无效的。这是因为Method Swizzling对NSArray这些的类簇是不起作用的。

因为这些类簇类，其实是一种抽象工厂的设计模式。抽象工厂内部有很多其它继承自当前类的子类，抽象工厂类会根据不同情况，创建不同的抽象对象来进行使用。例如我们调用NSArray的`objectAtIndex:`方法，这个类会在方法内部判断，内部创建不同抽象类进行操作。

所以如果我们对NSArray类进行Swizzling操作其实只是对**父类**进行了操作，在NSArray内部会创建其他子类来执行操作，真正执行Swizzling操作的并不是NSArray自身，所以我们应该对其“真身”进行操作。

下面列举了NSArray和NSDictionary本类的类名，可以通过Runtime函数取出本类：

类名	真身
NSArray	__NSArrayI
NSMutableArray	__NSArrayM
NSDictionary	__NSDictionaryI
NSMutableDictionary	__NSDictionaryM

> 实践

好啦，新建一个分类，直接用代码实现，看看怎么取出真身的：

- NSArray+CrashHandle.m

```
@implementation NSArray (CrashHandle)

// Swizzling核心代码
// 需要注意的是，好多同学反馈下面代码不起作用，造成这个问题的原因大多都是其调用了super load方法。在下面的load方法中，不应该调用父类的load方法。
+ (void)load {
    Method fromMethod = class_getInstanceMethod(objc_getClass("__NSArrayI"), @selector(objectAtIndex:));
    Method toMethod = class_getInstanceMethod(objc_getClass("__NSArrayI"), @selector(lxz_objectAtIndex:));
    method_exchangeImplementations(fromMethod, toMethod);
}

// 为了避免和系统的方法冲突，我一般都会在swizzling方法前面加前缀
- (id)lxz_objectAtIndex:(NSUInteger)index {
    // 判断下标是否越界，如果越界就进入异常拦截
    if (self.count-1 < index) {
        @try {
            return [self lxz_objectAtIndex:index];
        }
        @catch (NSException *exception) {
            // 在崩溃后会打印崩溃信息。如果是线上，可以在这里将崩溃信息发送到服务器
            NSLog(@"---------- %s Crash Because Method %s  ----------\n", class_getName(self.class), __func__);
            NSLog(@"%@", [exception callStackSymbols]);
            return nil;
        }
        @finally {}
    } // 如果没有问题，则正常进行方法调用
    else {
        return [self lxz_objectAtIndex:index];
    }
}
```

这里面可能有个误会，`- (id)lxz_objectAtIndex:(NSUInteger)index {`里面调用了自身？这是递归吗？其实不是。这个时候方法替换已经有效了，`lxz_objectAtIndex`这个SEL指向的其实是原来系统的`objectAtIndex:`的IMP。因而不是递归。

- ViewController.m

```
- (void)viewDidLoad {
    [super viewDidLoad];
```

```
    // 测试代码
    NSArray *array = @[@0, @1, @2, @3];
    [array objectAtIndex:3];
    //本来要奔溃的
    [array objectAtIndex:4];
}
```

运行之后，发现没有崩溃，并打印了相关信息，如下所示。