

jackyshan_ Lv3

2018年04月01日 阅读 22949

[关注](#)

iOS Runtime详解



Runtime的特性主要是消息(方法)传递，如果消息(方法)在对象中找不到，就进行转发，具体怎么实现的呢。我们从下面几个方面探寻Runtime的实现机制。

- Runtime介绍
- Runtime消息传递

[首页](#) ▾

Runtime介绍

Objective-C 扩展了 C 语言，并加入了面向对象特性和 Smalltalk 式的消息传递机制。而这个扩展的核心是一个用 C 和 编译语言 写的 Runtime 库。它是 Objective-C 面向对象和动态机制的基石。

Objective-C 是一个动态语言，这意味着它不仅需要一个编译器，也需要一个运行时系统来动态得创建类和对象、进行消息传递和转发。理解 Objective-C 的 Runtime 机制可以帮我们更好的了解这个语言，适当的时候还能对语言进行扩展，从系统层面解决项目中的一些设计或技术问题。了解 Runtime，要先了解它的核心 - 消息传递（Messaging）。

Runtime 其实有两个版本：“modern”和“legacy”。我们现在用的 Objective-C 2.0 采用的是现行（Modern）版的 Runtime 系统，只能运行在 iOS 和 macOS 10.5 之后的 64 位程序中。而 macOS 较老的 32 位程序仍采用 Objective-C 1 中的（早期）Legacy 版本的 Runtime 系统。这两个版本最大的区别在于当你更改一个类的实例变量的布局时，在早期版本中你需要重新编译它的子类，而现行版就不需要。

Runtime 基本是用 C 和 汇编 写的，可见苹果为了动态系统的高效而作出的努力。你可以在[这里](#)下到苹果维护的开源代码。苹果和GNU各自维护一个开源的 runtime 版本，这两个版本之间都在努力的保持一致。

平时的业务中主要是使用[官方Api](#)，解决我们框架性的需求。

高级编程语言想要成为可执行文件需要先编译为汇编语言再汇编为机器语言，机器语言也是计算机能够识别的唯一语言，但是 OC 并不能直接编译为汇编语言，而是要先转写为纯 C 语言再进行编译和汇编的操作，从 OC 到 C 语言的过渡就是由runtime来实现的。然而我们使用 OC 进行面向对象开发，而 C 语言更多的是面向过程开发，这就需要将面向对象的类转变为面向过程的结构体。

Runtime消息传递

一个对象的方法像这样 [obj foo]，编译器转成消息发送 objc_msgSend(obj, foo)，Runtime 时执行的流程是这样的：

- 首先，通过 obj 的 isa 指针找到它的 class；
- 在 class 的 method list 找 foo；
- 如果 class 中没到 foo，继续往它的 superclass 中找；
- 一旦找到 foo 这个函数，就去执行它的实现 IMP。

[首页](#)[探索掘金](#)

但这种实现有个问题，效率低。但一个 `class` 往往只有 20% 的函数会被经常调用，可能占总调用次数的 80%。每个消息都需要遍历一次 `objc_method_list` 并不合理。如果把经常被调用的函数缓存下来，那可以大大提高函数查询的效率。这也就是 `objc_class` 中另一个重要成员 `objc_cache` 做的事情 - 再找到 `foo` 之后，把 `foo` 的 `method_name` 作为 `key`，`method_imp` 作为 `value` 给存起来。当再次收到 `foo` 消息的时候，可以直接在 `cache` 里找到，避免去遍历 `objc_method_list`。从前面的源代码可以看到 `objc_cache` 是存在 `objc_class` 结构体中的。

`objc_msgSend`的方法定义如下：

复制代码

```
OBJC_EXPORT id objc_msgSend(id self, SEL op, ...)
```

那消息传递是怎么实现的呢？我们看看对象(object)，类(class)，方法(method)这几个的结构体：

复制代码

```
//对象
struct objc_object {
    Class isa OBJC_ISA_AVAILABILITY;
};

//类
struct objc_class {
    Class isa OBJC_ISA_AVAILABILITY;
#ifdef __OBJC2__
    Class super_class OBJC2_UNAVAILABLE;
    const char *name OBJC2_UNAVAILABLE;
    long version OBJC2_UNAVAILABLE;
    long info OBJC2_UNAVAILABLE;
    long instance_size OBJC2_UNAVAILABLE;
    struct objc_ivar_list *ivars OBJC2_UNAVAILABLE;
    struct objc_method_list **methodLists OBJC2_UNAVAILABLE;
    struct objc_cache *cache OBJC2_UNAVAILABLE;
    struct objc_protocol_list *protocols OBJC2_UNAVAILABLE;
#endif
} OBJC2_UNAVAILABLE;

//方法列表
struct objc_method_list {
    struct objc_method_list *obsolete OBJC2_UNAVAILABLE;
    int method_count OBJC2_UNAVAILABLE;
#ifdef __LP64__
    int space OBJC2_UNAVAILABLE;
#endif
    /* variable length structure */
    struct objc_method method_list[1] OBJC2_UNAVAILABLE;
} OBJC2_UNAVAILABLE;

//方法
```



首页 ▾

探索掘金



```

    char *method_types      OBJC2_UNAVAILABLE;
    IMP method_imp          OBJC2_UNAVAILABLE;
}

```

1. 系统首先找到消息的接收对象，然后通过对象的 `isa` 找到它的类。
2. 在它的类中查找 `method_list`，是否有 `selector` 方法。
3. 没有则查找父类的 `method_list`。
4. 找到对应的 `method`，执行它的 `IMP`。
5. 转发 `IMP` 的 `return` 值。

下面讲讲消息传递用到的一些概念：

- 类对象(objc_class)
- 实例(objc_object)
- 元类(Meta Class)
- Method(objc_method)
- SEL(objc_selector)
- IMP
- 类缓存(objc_cache)
- Category(objc_category)

类对象(objc_class)

Objective-C 类是由 `Class` 类型来表示的，它实际上是一个指向 `objc_class` 结构体的指针。

复制代码

```
typedef struct objc_class *Class;
```

查看 `objc/runtime.h` 中 `objc_class` 结构体的定义如下：

复制代码

```

struct objc_class {
    Class _Nonnull isa  OBJC_ISA_AVAILABILITY;

    #if !__OBJC2__
        Class _Nullable super_class      OBJC2_UNAVAILABLE;
        const char * _Nonnull name      OBJC2_UNAVAILABLE;
        long version                    OBJC2_UNAVAILABLE;
        long info                      OBJC2_UNAVAILABLE;
        long instance_size              OBJC2_UNAVAILABLE;
        struct objc_ivar_list * _Nullable ivars      OBJC2_UNAVAILABLE;
    #endif
}

```



首页 ▼

探索掘金



```
#endif
```

```
} OBJC2_UNAVAILABLE;
```

`struct objc_class` 结构体定义了很多变量，通过命名不难发现，结构体里保存了指向父类的指针、类的名字、版本、实例大小、实例变量列表、方法列表、缓存、遵守的协议列表等，一个类包含的信息也不就正是这些吗？没错，类对象就是一个结构体 `struct objc_class`，这个结构体存放的数据称为元数据(`metadata`)，该结构体的第一个成员变量也是 `isa` 指针，这就说明了 `Class` 本身其实也是一个对象，因此我们称之为类对象，类对象在编译期产生用于创建实例对象，是单例。

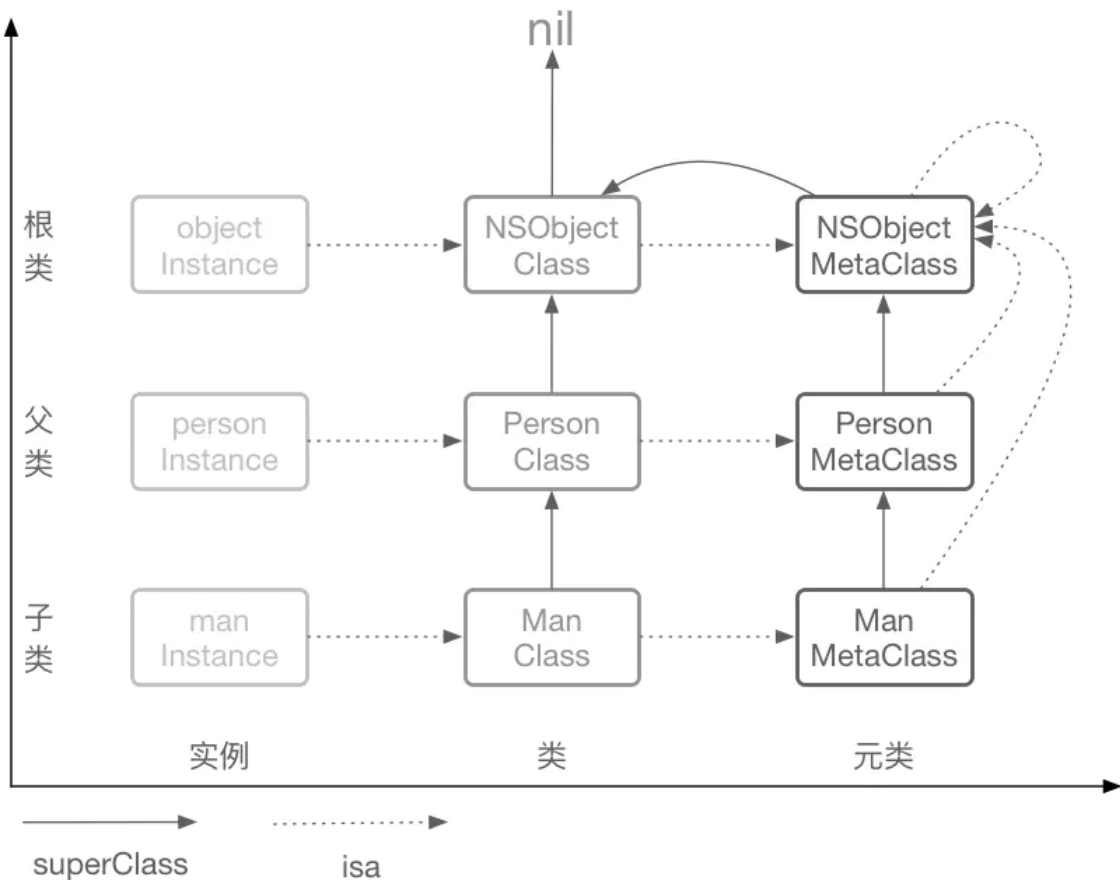
实例(objc_object)

[复制代码](#)

```
/// Represents an instance of a class.
struct objc_object {
    Class isa OBJC_ISA_AVAILABILITY;
};

/// A pointer to an instance of a class.
typedef struct objc_object *id;
```

类对象中的元数据存储的都是如何创建一个实例的相关信息，那么类对象和类方法应该从哪里创建呢？就是从 `isa` 指针指向的结构体创建，类对象的 `isa` 指针指向的我们称之为元类(`metaclass`)，元类中保存了创建类对象以及类方法所需的所有信息，因此整个结构应该如下图所示：



元类(Meta Class)

通过上图我们可以看出整个体系构成了一个自闭环，`struct objc_object` 结构体 实例 它的 `isa` 指针指向类对象，类对象的 `isa` 指针指向了元类，`super_class` 指针指向了父类的类对象，而元类的 `super_class` 指针指向了父类的元类，那元类的 `isa` 指针又指向了自己。

元类(Meta Class)是一个类对象的类。在上面我们提到，所有的类自身也是一个对象，我们可以向这个对象发送消息(即调用类方法)。为了调用类方法，这个类的 `isa` 指针必须指向一个包含这些类方法的一个 `objc_class` 结构体。这就引出了 `meta-class` 的概念，元类中保存了创建类对象以及类方法所需的所有信息。任何 `NSObject` 继承体系下的 `meta-class` 都使用 `NSObject` 的 `meta-class` 作为自己的所属类，而基类的 `meta-class` 的 `isa` 指针是指向它自己。

Method(objc_method)

先看下定义

复制代码

```
runtime.h
/// An opaque type that represents a method in a class definition. 代表类定义中一个方法的不透明类!
typedef struct objc_method *Method;
```



首页 ▾

探索掘金



```
char *method_types
IMP method_imp
```

```
OBJC2_UNAVAILABLE;
OBJC2_UNAVAILABLE;
```

Method 和我们平时理解的函数是一致的，就是表示能够独立完成一个功能的一段代码，比如：

[复制代码](#)

```
- (void)logName
{
    NSLog(@"name");
}
```

这段代码，就是一个函数。

我们来看下 **objc_method** 这个结构体的内容：

- SEL method_name 方法名
- char *method_types 方法类型
- IMP method_imp 方法实现

在这个结构体中，我们已经看到了 **SEL** 和 **IMP**，说明 **SEL** 和 **IMP** 其实都是 **Method** 的属性。

我们接着来看 **SEL**。

SEL(objc_selector)

先看下定义

[复制代码](#)

```
Objc.h
/// An opaque type that represents a method selector. 代表一个方法的不透明类型
typedef struct objc_selector *SEL;
```

objc_msgSend 函数第二个参数类型为 **SEL**，它是 **selector** 在 **Objective-C** 中的表示类型（**Swift** 中是 **Selector** 类）。**selector** 是方法选择器，可以理解为区分方法的 **ID**，而这个 **ID** 的数据结构是 **SEL**：

[复制代码](#)

```
@property SEL selector;
```

可以看到 **selector** 是 **SEL** 的一个实例。

其实 `selector` 就是个映射到方法的 C 字符串，你可以用 `Objective-C` 编译器命令 `@selector()` 或者 `Runtime` 系统的 `sel_registerName` 函数来获得一个 `SEL` 类型的方法选择器。

`selector` 既然是一个 `string`，我觉得应该是类似 `className+method` 的组合，命名规则有两条：

- 同一个类，`selector`不能重复
- 不同的类，`selector`可以重复

这也带来了一个弊端，我们在写 C 代码的时候，经常会用到函数重载，就是函数名相同，参数不同，但是这在 `Objective-C` 中是行不通的，因为 `selector` 只记了 `method` 的 `name`，没有参数，所以没法区分不同的 `method`。

比如：

```
- (void)caculate(NSInteger)num;  
- (void)caculate(CGFloat)num;
```

复制代码

是会报错的。

我们只能通过命名来区别：

```
- (void)caculateWithInt(NSInteger)num;  
- (void)caculateWithFloat(CGFloat)num;
```

复制代码

在不同类中相同名字的方法所对应的方法选择器是相同的，即使方法名字相同而变量类型不同也会导致它们具有相同的方法选择器。

IMP

看下 `IMP` 的定义

```
/// A pointer to the function of a method implementation. 指向一个方法实现的指针  
typedef id (*IMP)(id, SEL, ...);  
#endif
```

复制代码

就是指向最终实现程序的内存地址的指针。



首页 ▾

探索掘金



在 iOS 的 **Runtime** 中，**Method** 通过 **selector** 和 **IMP** 两个属性，实现了快速查询方法及实现，相对提高了性能，又保持了灵活性。

类缓存(objc_cache)

当 **Objective-C** 运行时通过跟踪它的 **isa** 指针检查对象时，它可以找到一个实现许多方法的对象。然而，你可能只调用它们的一小部分，并且每次查找时，搜索所有选择器的类分派表没有意义。所以类实现一个缓存，每当你搜索一个类分派表，并找到相应的选择器，它把它放入它的缓存。所以当 **objc_msgSend** 查找一个类的选择器，它首先搜索类缓存。这是基于这样的理论：如果你在类上调用一个消息，你可能以后再次调用该消息。

为了加速消息分发，系统会对方法和对应的地址进行缓存，就放在上述的 **objc_cache**，所以在实际运行中，大部分常用的方法都是会被缓存起来的，**Runtime** 系统实际上非常快，接近直接执行内存地址的程序速度。

Category(objc_category)

Category 是表示一个指向分类的结构体的指针，其定义如下：

[复制代码](#)

```
struct category_t {
    const char *name;
    classref_t cls;
    struct method_list_t *instanceMethods;
    struct method_list_t *classMethods;
    struct protocol_list_t *protocols;
    struct property_list_t *instanceProperties;
};
```

[复制代码](#)

name: 是指 class_name 而不是 category_name。

cls: 要扩展的类对象，编译期间是不会定义的，而是在Runtime阶段通过name对应到对应的类对象。

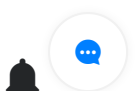
instanceMethods: category中所有给类添加的实例方法的列表。

classMethods: category中所有添加的类方法的列表。

protocols: category实现的所有协议的列表。

instanceProperties: 表示Category里所有的properties，这就是我们可以通过objc_setAssociatedObject和ot

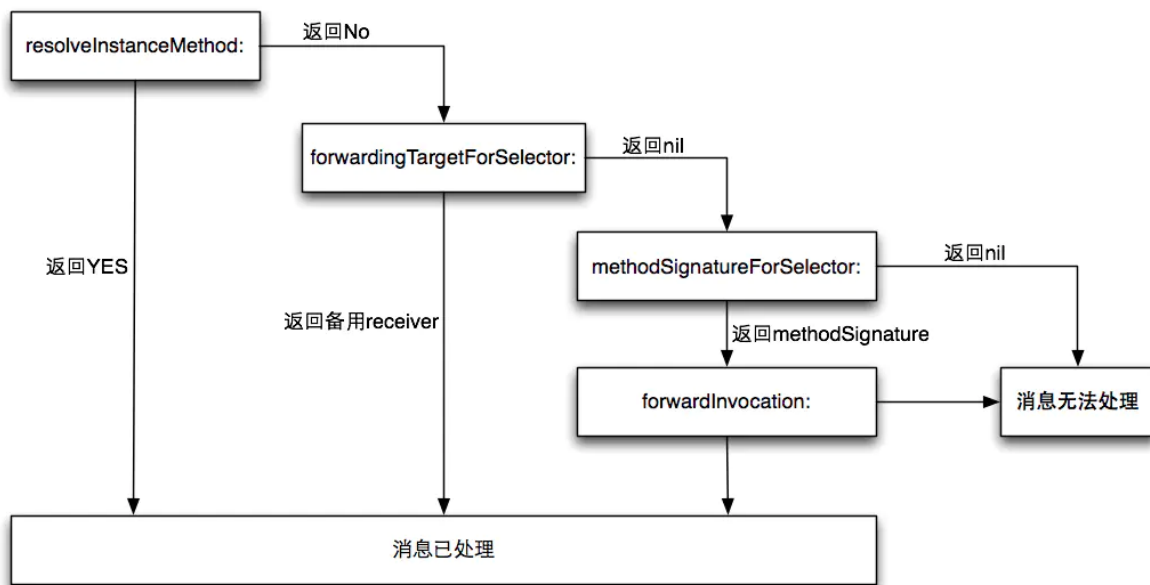
从上面的 **category_t** 的结构体中可以看出，分类中可以添加实例方法，类方法，甚至可以实现协议，添加属性，不可以添加成员变量。



前文介绍了进行一次发送消息会在相关的类对象中搜索方法列表，如果找不到则会沿着继承树向上一直搜索知道继承树根部（通常为 `NSObject` ），如果还是找不到并且消息转发都失败了就回执行 `doesNotRecognizeSelector:` 方法报 `unrecognized selector` 错。那么消息转发到底是什么呢？接下来将会逐一介绍最后的三次机会。

- 动态方法解析
- 备用接收者
- 完整消息转发

消息转发流程简图：



动态方法解析

首先，`Objective-C` 运行时调用 `+resolveInstanceMethod:` 或者 `+resolveClassMethod:`，让你有机会提供一个函数实现。如果你添加了函数并返回 `YES`，那运行时系统就会重新启动一次消息发送的过程。

实现一个动态方法解析的例子如下：

复制代码

```
- (void)viewDidLoad {
    [super viewDidLoad];
    // Do any additional setup after loading the view, typically from a nib.
    //执行foo函数
    [self performSelector:@selector(foo:)];
}

+ (BOOL)resolveInstanceMethod:(SEL)sel {
    if (sel == @selector(foo:)) { //如果是执行foo函数，就动态解析，指定新的IMP
```



首页 ▾

探索掘金



```

    }
    return [super resolveInstanceMethod:sel];
}

void fooMethod(id obj, SEL _cmd) {
    NSLog(@"Doing foo");//新的foo函数
}

```

打印结果： 2018-04-01 12:23:35.952670+0800 ocram[87546:23235469] Doing foo

可以看到虽然没有实现 `foo:` 这个函数，但是我们通过 `class_addMethod` 动态添加 `fooMethod` 函数，并执行 `fooMethod` 这个函数的 `IMP`。从打印结果看，成功实现了。

如果 `resolve` 方法返回 `NO`，运行时就会移到下一步：`forwardingTargetForSelector`。

备用接收者

如果目标对象实现了 `-forwardingTargetForSelector:`，`Runtime` 这时就会调用这个方法，给你把这个消息转发给其他对象的机会。

实现一个备用接收者的例子如下：

复制代码

```

#import "ViewController.h"
#import "objc/runtime.h"

@interface Person: NSObject

@end

@implementation Person

- (void)foo {
    NSLog(@"Doing foo");//Person的foo函数
}

@end

@interface ViewController ()

@end

@implementation ViewController

```



首页 ▼

探索掘金



```

[super viewDidLoad];
// Do any additional setup after loading the view, typically from a nib.
//执行foo函数
[self performSelector:@selector(foo)];
}

+ (BOOL)resolveInstanceMethod:(SEL)sel {
    return YES;//返回YES, 进入下一步转发
}

- (id)forwardingTargetForSelector:(SEL)aSelector {
    if (aSelector == @selector(foo)) {
        return [Person new];//返回Person对象, 让Person对象接收这个消息
    }

    return [super forwardingTargetForSelector:aSelector];
}

@end

```

打印结果: 2018-04-01 12:45:04.757929+0800 ocram[88023:23260346] Doing foo

可以看到我们通过 `forwardingTargetForSelector` 把当前 `ViewController` 的方法转发给了 `Person` 去执行了。打印结果也证明我们成功实现了转发。

完整消息转发

如果在上一步还不能处理未知消息, 则唯一能做的就是启用完整的消息转发机制了。首先它会发送 `-methodSignatureForSelector:` 消息获得函数的参数和返回值类型。如果 `-methodSignatureForSelector:` 返回 `nil`, `Runtime` 则会发出 `-doesNotRecognizeSelector:` 消息, 程序这时也就挂掉了。如果返回了一个函数签名, `Runtime` 就会创建一个 `NSInvocation` 对象并发送 `-forwardInvocation:` 消息给目标对象。

实现一个完整转发的例子如下:

复制代码

```

#import "ViewController.h"
#import "objc/runtime.h"

@interface Person: NSObject

@end

```

```
- (void)foo {
    NSLog(@"Doing foo");//Person的foo函数
}

@end

@interface ViewController ()

@end

@implementation ViewController

- (void)viewDidLoad {
    [super viewDidLoad];
    // Do any additional setup after loading the view, typically from a nib.
    //执行foo函数
    [self performSelector:@selector(foo)];
}

+ (BOOL)resolveInstanceMethod:(SEL)sel {
    return YES;//返回YES, 进入下一步转发
}

- (id)forwardingTargetForSelector:(SEL)aSelector {
    return nil;//返回nil, 进入下一步转发
}

- (NSMethodSignature *)methodSignatureForSelector:(SEL)aSelector {
    if ([NSStringFromSelector(aSelector) isEqualToString:@"foo"]) {
        return [NSMethodSignature signatureWithObjCTypes:"v@:"];//签名, 进入forwardInvocation
    }

    return [super methodSignatureForSelector:aSelector];
}

- (void)forwardInvocation:(NSInvocation *)anInvocation {
    SEL sel = anInvocation.selector;

    Person *p = [Person new];
    if([p respondsToSelector:sel]) {
        [anInvocation invokeWithTarget:p];
    }
    else {
        [self doesNotRecognizeSelector:sel];
    }
}

}
```



打印结果： 2018-04-01 13:00:45.423385+0800 ocram[88353:23279961] Doing foo

从打印结果来看，我们实现了完整的转发。通过签名，`Runtime` 生成了一个对象 `anInvocation`，发送给了 `forwardInvocation`，我们在 `forwardInvocation` 方法里面让 `Person` 对象去执行了 `foo` 函数。签名参数 `v@:` 怎么解释呢，这里苹果文档[Type Encodings](#)有详细的解释。

以上就是 `Runtime` 的三次转发流程。下面我们讲讲 `Runtime` 的实际应用。

Runtime应用

`Runtime` 简直就是做大型框架的利器。它的应用场景非常多，下面就介绍一些常见的应用场景。

- 关联对象(Objective-C Associated Objects)给分类增加属性
- 方法魔法(Method Swizzling)方法添加和替换和KVO实现
- 消息转发(热更新)解决Bug(JSPatch)
- 实现NSCoding的自动归档和自动解档
- 实现字典和模型的自动转换(MJExtension)

关联对象(Objective-C Associated Objects)给分类增加属性

我们都是知道分类是不能自定义属性和变量的。下面通过关联对象实现给分类添加属性。

关联对象Runtime提供了下面几个接口：

复制代码

```
//关联对象
void objc_setAssociatedObject(id object, const void *key, id value, objc_AssociationPolicy policy);
//获取关联的对象
id objc_getAssociatedObject(id object, const void *key)
//移除关联的对象
void objc_removeAssociatedObjects(id object)
```

参数解释

复制代码

`id object`: 被关联的对象
`const void *key`: 关联的key, 要求唯一
`id value`: 关联的对象
`objc_AssociationPolicy policy`: 内存管理的策略

```
typedef OBJC_ENUM(uintptr_t, objc_AssociationPolicy) {
    OBJC_ASSOCIATION_ASSIGN = 0,           /**< Specifies a weak reference to the associated object.
    OBJC_ASSOCIATION_RETAIN_NONATOMIC = 1, /**< Specifies a strong reference to the associated object.
        * The association is not made atomically. */
    OBJC_ASSOCIATION_COPY_NONATOMIC = 3,   /**< Specifies that the associated object is copied.
        * The association is not made atomically. */
    OBJC_ASSOCIATION_RETAIN = 01401,       /**< Specifies a strong reference to the associated object.
        * The association is made atomically. */
    OBJC_ASSOCIATION_COPY = 01403         /**< Specifies that the associated object is copied.
        * The association is made atomically. */
};
```

下面实现一个 `UIView` 的 `Category` 添加自定义属性 `defaultColor`。

```
#import "ViewController.h"
#import "objc/runtime.h"

@interface UIView (DefaultColor)

@property (nonatomic, strong) UIColor *defaultColor;

@end

@implementation UIView (DefaultColor)

@dynamic defaultColor;

static char kDefaultColorKey;

- (void)setDefaultColor:(UIColor *)defaultColor {
    objc_setAssociatedObject(self, &kDefaultColorKey, defaultColor, OBJC_ASSOCIATION_RETAIN_NONATOMIC);
}

- (id)defaultColor {
    return objc_getAssociatedObject(self, &kDefaultColorKey);
}

@end

@interface ViewController ()

@end

@implementation ViewController
```

```
// Do any additional setup after loading the view, typically from a nib.

UIView *test = [UIView new];
test.defaultColor = [UIColor blackColor];
NSLog(@"%@", test.defaultColor);
}

@end
```

打印结果： 2018-04-01 15:41:44.977732+0800 ocram[2053:63739]
UIExtendedGrayColorSpace 0 1

打印结果来看，我们成功在分类上添加了一个属性，实现了它的 **setter** 和 **getter** 方法。通过关联对象实现的属性的内存管理也是有 **ARC** 管理的，所以我们只需要给定适当的内存策略就行了，不需要操心对象的释放。

我们看看内存测量对于的属性修饰。

内存策略	属性修饰	描述
OBJC_ASSOCIATION_ASSIGN	@property (assign) 或 @property (unsafe_unretained)	指定一个关联对象的弱引用。
OBJC_ASSOCIATION_RETAIN_NONATOMIC	@property (nonatomic, strong)	@property (nonatomic, strong) 指定一个关联对象的强引用，不能被原子化使用。
OBJC_ASSOCIATION_COPY_NONATOMIC	@property (nonatomic, copy)	指定一个关联对象的copy引用，不能被原子化使用。
OBJC_ASSOCIATION_RETAIN	@property (atomic, strong)	指定一个关联对象的强引用，能被原子化使用。
OBJC_ASSOCIATION_COPY	@property (atomic, copy)	指定一个关联对象的copy引用，能被原子化使用。

方法魔法(Method Swizzling)方法添加和替换和KVO实现

实际上添加方法刚才在讲消息转发的时候，动态方法解析的时候就提到了。

复制代码

```
//class_addMethod(Class _Nullable __unsafe_unretained cls, SEL _Nonnull name, IMP _Nonnull  
class_addMethod([self class], sel, (IMP)fooMethod, "v@:");
```

- cls 被添加方法的类
- name 添加的方法的名称的SEL
- imp 方法的实现。该函数必须至少要有两个参数，self,cmd
- 类型编码

方法替换

下面实现一个替换 `ViewController` 的 `viewDidLoad` 方法的例子。

复制代码

```
@implementation ViewController

+ (void)load {
    static dispatch_once_t onceToken;
    dispatch_once(&onceToken, ^{
        Class class = [self class];
        SEL originalSelector = @selector(viewDidLoad);
        SEL swizzledSelector = @selector(jkviewDidLoad);

        Method originalMethod = class_getInstanceMethod(class,originalSelector);
        Method swizzledMethod = class_getInstanceMethod(class,swizzledSelector);

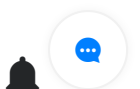
        //judge the method named swizzledMethod is already existed.
        BOOL didAddMethod = class_addMethod(class, originalSelector, method_getImplementation(originalMethod), method_getTypeEncoding(originalMethod));
        // if swizzledMethod is already existed.
        if (didAddMethod) {
            class_replaceMethod(class, swizzledSelector, method_getImplementation(originalMethod), method_getTypeEncoding(originalMethod));
        }
        else {
            method_exchangeImplementations(originalMethod, swizzledMethod);
        }
    });
}

- (void)jkviewDidLoad {
    NSLog(@"替换的方法");

    [self viewDidLoad];
}
```


[首页](#) ▼

探索掘金



```
NSLog(@"自带的方法");

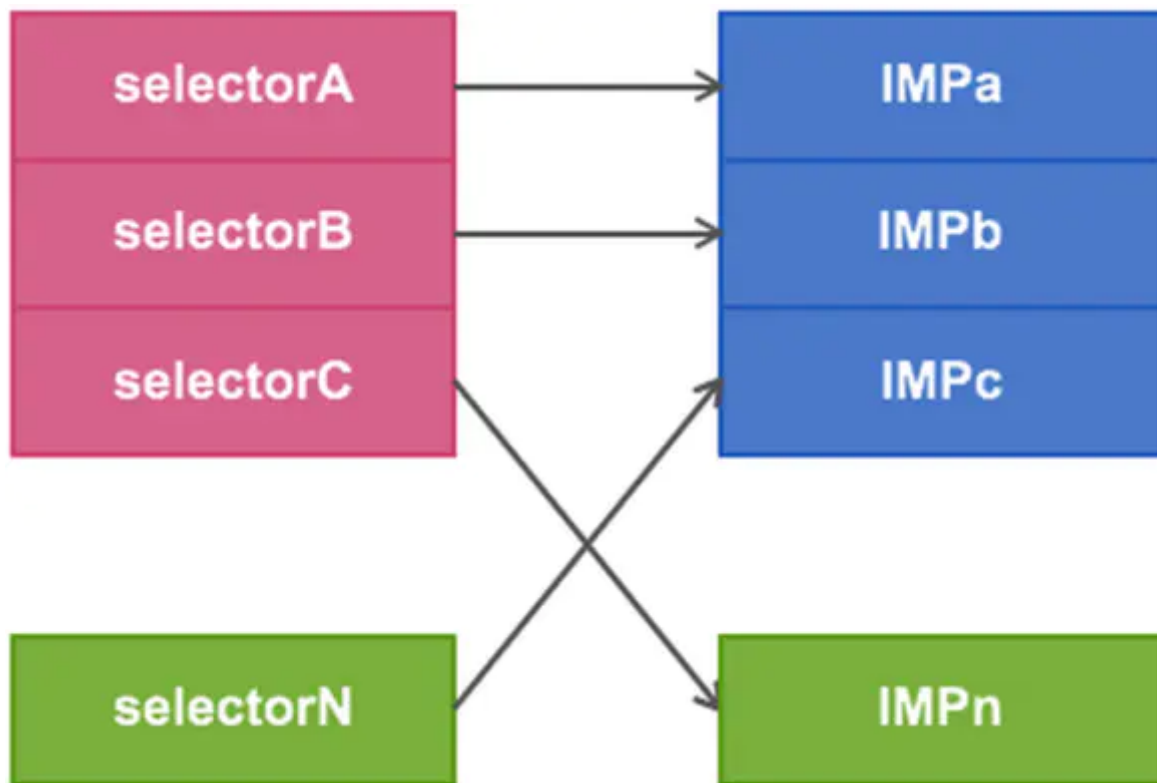
[super viewDidLoad];
}

@end
```

swizzling 应该只在 **+load** 中完成。在 **Objective-C** 的运行时中，每个类有两个方法都会自动调用。**+load** 是在一个类被初始装载时调用，**+initialize** 是在应用第一次调用该类的类方法或实例方法前调用的。两个方法都是可选的，并且只有在方法被实现的情况下才会被调用。

swizzling 应该只在 **dispatch_once** 中完成,由于 **swizzling** 改变了全局的状态，所以我们需要确保每个预防措施在运行时都是可用的。原子操作就是这样一个用于确保代码只会被执行一次的预防措施，就算是在不同的线程中也能确保代码只执行一次。**Grand Central Dispatch** 的 **dispatch_once** 满足了所需要的需求，并且应该被当做使用 **swizzling** 的初始化单例方法的标准。

实现图解如下图。



从图中可以看出，我们通过swizzling特性，将selectorC的方法实现IMPc与selectorN的方法实现IMPn交换了，当我们调用selectorC，也就是给对象发送selectorC消息时，所查找到的对应的方法实现就是IMPn而不是IMPc了。

全称是Key-value observing，翻译成键值观察。提供了一种当其它对象属性被修改的时候能通知当前对象的机制。再MVC大行其道的Cocoa中，KVO机制很适合实现model和controller类之间的通讯。

KVO 的实现依赖于 Objective-C 强大的 Runtime，当观察某对象 A 时，KVO 机制动态创建一个对象 A 当前类的子类，并为这个新的子类重写了被观察属性 keyPath 的 setter 方法。setter 方法随后负责通知观察对象属性的改变状况。

Apple 使用了 isa-swizzling 来实现 KVO。当观察对象 A 时，KVO 机制动态创建一个新的名为：NSKVONotifying_A 的新类，该类继承自对象A的本类，且 KVO 为 NSKVONotifying_A 重写观察属性的 setter 方法，setter 方法会负责在调用原 setter 方法之前和之后，通知所有观察对象属性值的更改情况。

- NSKVONotifying_A 类剖析

[复制代码](#)

```
NSLog(@"self->isa:%@",self->isa);
NSLog(@"self class:%@",[self class]);
```

在建立KVO监听前，打印结果为：

[复制代码](#)

```
self->isa:A
self class:A
```

在建立KVO监听之后，打印结果为：

[复制代码](#)

```
self->isa:NSKVONotifying_A
self class:A
```

在这个过程，被观察对象的 isa 指针从指向原来的 A 类，被 KVO 机制修改为指向系统新创建的子类 NSKVONotifying_A 类，来实现当前类属性值改变的监听；所以当我们从应用层面上看来，完全没有意识到有新的类出现，这是系统“隐瞒”了对 KVO 的底层实现过程，让我们误以为还是原来的类。但是此时如果我们创建一个新的名为“NSKVONotifying_A”的类，就会发现系统运行到注册 KVO 的那段代码时程序就崩溃，因为系统在注册监听的时候动态创建了名为 NSKVONotifying_A 的中间类，并指向这个中间类了。

- 子类setter方法剖析

willChangeValueForKey: 被调用, 通知系统该 **keyPath** 的属性值即将变更; 当改变发生后,
didChangeValueForKey: 被调用, 通知系统该 **keyPath** 的属性值已经变更; 之后,
observeValueForKey:ofObject:change:context: 也会被调用。且重写观察属性的 **setter** 方法这种继承方式的注入是在运行时而不是编译时实现的。

KVO 为子类的观察者属性重写调用存取方法的工作原理在代码中相当于:

[复制代码](#)

```
- (void)setName:(NSString *)newName {
    [self willChangeValueForKey:@"name"];    //KVO 在调用存取方法之前总调用
    [super setValue:newName forKey:@"name"]; //调用父类的存取方法
    [self didChangeValueForKey:@"name"];    //KVO 在调用存取方法之后总调用
}
```

消息转发(热更新)解决Bug(JSPatch)

[JSPatch](#) 是一个 iOS 动态更新框架, 只需在项目中引入极小的引擎, 就可以使用 JavaScript 调用任何 Objective-C 原生接口, 获得脚本语言的优势: 为项目动态添加模块, 或替换项目原生代码动态修复 bug。

关于消息转发, 前面已经讲到过了, 消息转发分为三级, 我们可以在每级实现替换功能, 实现消息转发, 从而不会造成崩溃。[JSPatch](#)不仅能够实现消息转发, 还可以实现方法添加、替换能一系列功能。

实现NSCoding的自动归档和自动解档

原理描述: 用 **runtime** 提供的函数遍历 **Model** 自身所有属性, 并对属性进行 **encode** 和 **decode** 操作。核心方法: 在 **Model** 的基类中重写方法:

[复制代码](#)

```
- (id)initWithCoder:(NSCoder *)aDecoder {
    if (self = [super init]) {
        unsigned int outCount;
        Ivar * ivars = class_copyIvarList([self class], &outCount);
        for (int i = 0; i < outCount; i++) {
            Ivar ivar = ivars[i];
            NSString * key = [NSString stringWithUTF8String:ivar_getName(ivar)];
            [self setValue:[aDecoder decodeObjectForKey:key] forKey:key];
        }
    }
    return self;
}
```

[首页](#) ▼

```

unsigned int outCount;
Ivar * ivars = class_copyIvarList([self class], &outCount);
for (int i = 0; i < outCount; i++) {
    Ivar ivar = ivars[i];
    NSString * key = [NSString stringWithUTF8String:ivar_getName(ivar)];
    [aCoder encodeObject:[self valueForKey:key] forKey:key];
}
}

```

实现字典和模型的自动转换(MJExtension)

原理描述：用 `runtime` 提供的函数遍历 `Model` 自身所有属性，如果属性在 `json` 中有对应的值，则将其赋值。核心方法：在 `NSObject` 的分类中添加方法

[复制代码](#)

```

- (instancetype)initWithDict:(NSDictionary *)dict {

    if (self = [self init]) {
        // (1) 获取类的属性及属性对应的类型
        NSMutableArray * keys = [NSMutableArray array];
        NSMutableArray * attributes = [NSMutableArray array];
        /*
         * 例子
         * name = value3 attribute = T@"NSString",C,N,V_value3
         * name = value4 attribute = T^i,N,V_value4
         */
        unsigned int outCount;
        objc_property_t * properties = class_copyPropertyList([self class], &outCount);
        for (int i = 0; i < outCount; i++) {
            objc_property_t property = properties[i];
            // 通过property_getName函数获得属性的名字
            NSString * propertyName = [NSString stringWithCString:property_getName(property)
                                     encoding:NSUTF8StringEncoding];
            [keys addObject:propertyName];
            // 通过property_getAttributes函数可以获得属性的名字和@encode编码
            NSString * propertyAttribute = [NSString stringWithCString:property_getAttributes(property)
                                         encoding:NSUTF8StringEncoding];
            [attributes addObject:propertyAttribute];
        }
        // 立即释放properties指向的内存
        free(properties);

        // (2) 根据类型给属性赋值
        for (NSString * key in keys) {
            if ([dict valueForKey:key] == nil) continue;
            [self setValue:[dict valueForKey:key] forKey:key];
        }
    }
}

```


[首页](#)



```
}
```

以上就是 **Runtime** 应用的一些场景，本文到此结束了。

关注我

欢迎关注公众号：jackyshan，技术干货首发微信，第一时间推送。



关注下面的标签，发现更多相似文章

[Objective-C](#)[Apple](#)[JSPatch](#)

jackyshan_ Lv3 iOS @ 广州
获得点赞 2,170 · 获得阅读 139,656

[关注](#)

安装掘金浏览器插件

打开新标签页发现好内容，掘金、GitHub、Dribbble、ProductHunt 等站点内容轻松获取。快来安装掘金浏览器插件获取高质量内容吧！

蜀黍n

[首页](#) ▼

}

作者 我想问一下 在消息进入第二次转发的时候这个是不是应该返回NO，这样才能进入第二次转发，小白求指点

7月前



回复

最好的我a

首先，通过obj的isa指针找到它的 class；
我觉得作者这句话说的不够严谨

7月前



回复

东厂胡一刀 Lv1 iOS软件开发

感谢作者。

1年前



回复

LeeG_Z 代码玩家

看你开篇介绍了runtime现在有2个版本的概述 因为你会参考新版本讲,为啥用一些已经废弃的版本讲呢?? 真搞不懂为啥都千篇一律都是这样的

1年前

1

回复

我爱野指针 Lv1 iOS开发

全是互相借鉴的，OBJC2_UNAVAILABLE 这么大的字。。

1年前

DesmondDAI Software Engineer @ ...

想问下怎么知道类对象是单例？

2年前



回复

ChakkeiLam iOS开发工程师

我想问问备用接收者那个的地方，应该是返回NO进入下一步转发，可我看你都是返回了YES。这篇文章质量很高，算是看到的比较好的关于runtime的解析文章了。

2年前

1

回复

minjing_lin Lv2 iOS、Flutter、搬...

我测试的是：YES和NO都可以，不实现也可以。

1年前

纯路人 摸鱼铲屎

这一步，如果没有添加新方法的话，返回yes和no都可以

1年前

Kimball iOS开发工程师



首页 ▾

探索掘金



pro648 Lv1



回复 minjing_lin: resolveInstanceMethod: 返回值只用于内部打印, 所以返回YES、NO都会进入下一阶段。具体的查看这里 github.com

8月前

巴里小短腿 Lv1 iOS开发/Flutter开发

看过的最清晰最简洁的一篇runtime总结 赞赞赞!!!

2年前

 1  回复

🌸🌸🌸🌸🌸81069 iOS @ 测试

是啊 别的博主长篇大论 看着头疼

1月前

李树龙

```
BOOL didAddMethod = class_addMethod(class, originalSelector,
method_getImplementation(swizzledMethod), method_getTypeEncoding(swizzledMethod));
```

这一行的代码写错了吧, 因该是swizzledSelector 判断添加方法的 selector 是否存在吧

2年前

  回复

crassusxst

很好, 认真的学习了一遍总算理解了

2年前

  回复海的原滋味 Lv2 iOS开发 @ 来自火星的...

优秀

2年前

  回复

慕栩 iOS

为什么消息转发那里示意图是resolveInstanceMethod返回YES进入消息已处理, 返回NO的时候才进行下一步forwardingTargetForSelector, 但是下面的代码又是说

```
+ (BOOL)resolveInstanceMethod:(SEL)sel {
    return YES;//返回YES, 进入下一步转发
}, 求解释
```

2年前

  回复

繁华落尽0124 程序猿 @ Elink

应该是笔误。返回NO时, 才会进入下一步转发。

2年前

草栩 iOS



首页 ▾

探索掘金




```
但是我拿上面的代码实测+ (BOOL)resolveInstanceMethod:(SEL)sel {  
    return YES;  
}  
}返回YES或者NO都进行了消息转发，执行了Person的foo函数
```

2年前

慕栩 iOS

回复 [繁华落尽0124](#): 但是我拿上面的代码实测+ (BOOL)resolveInstanceMethod:(SEL)sel {
 return YES;
}
}返回YES或者NO都进行了消息转发，执行了Person的foo函数

2年前

繁华落尽0124 程序猿 @ Elink

回复 [慕栩](#): 这里直接使用 `[super resolveInstanceMethod:sel]`。虽然返回的是`YES`，但是并没有进行动态方法解析，消息还是没被处理，还是会进行转发。

2年前

jackyshan_ [Lv3](#) (作者) iOS @ 广州

回复 [繁华落尽0124](#): 消息还是没被处理，还是会进行转发。正解。

2年前

相关推荐

Mco · 19小时前 · Objective-C

最新 Category 加载

👍 2

💬

茶底世界之下 · 3天前 · Objective-C

iOS Button连接处理防止按钮连续点击

👍 4

💬 7

确认过眼神啊 · 1天前 · Objective-C

ios之C语言第二篇:标识符

👍 4

💬

来钓鱼啊 · 3天前 · Objective-C

nil, Nil, NULL, NSNull, kCFNull 区别

👍 5

💬



首页 ▼

探索掘金





Hello_Kid · 5天前 · Objective-C

学会位运算，助力开发高性能



确认过眼神啊 · 2天前 · Objective-C

ios之C语言第一篇:关键字



茶底世界之下 · 4天前 · Objective-C

iOS UISlider自定义渐变色滑杆



Hello_Kid · 2天前 · Objective-C

手撕 iOS 底层03 -- NSObject的alloc分析



网易云音乐大前端团队 · 1月前 · Swift / Objective-C

网易云音乐 iOS 14 小组件实战手册

