

Runtime

@M了个J

<https://github.com/CoderMJLee>



实力IT教育 www.520it.com

- 讲一下 OC 的消息机制
 - OC中的方法调用其实都是转成了objc_msgSend函数的调用，给receiver（方法调用者）发送了一条消息（selector方法名）
 - objc_msgSend底层有3大阶段
 - ✓ 消息发送（当前类、父类中查找）、动态方法解析、消息转发

- 消息转发机制流程

- 什么是Runtime？平时项目中有用过么？
 - OC是一门动态性比较强的编程语言，允许很多操作推迟到程序运行时再进行
 - OC的动态性就是由Runtime来支撑和实现的，Runtime是一套C语言的API，封装了很多动态性相关的函数
 - 平时编写的OC代码，底层都是转换成了Runtime API进行调用

- 具体应用
 - ✓ 利用关联对象（AssociatedObject）给分类添加属性
 - ✓ 遍历类的所有成员变量（修改textfield的占位文字颜色、字典转模型、自动归档解档）
 - ✓ 交换方法实现（交换系统的方法）
 - ✓ 利用消息转发机制解决方法找不到的异常问题
 - ✓

- 打印结果分别是什么？

```
@interface MJStudent : MJPerson

@end
```

```
@interface MJPerson : NSObject

@end
```

```
@implementation MJStudent
- (instancetype)init
{
    if (self = [super init]) {
        NSLog(@"[self class] = %@", [self class]);
        NSLog(@"[super class] = %@", [super class]);
        NSLog(@"[self superclass] = %@", [self superclass]);
        NSLog(@"[super superclass] = %@", [super superclass]);
    }
    return self;
}
@end
```

```
BOOL res1 = [[NSObject class] isKindOfClass:[NSObject class]];
BOOL res2 = [[NSObject class] isKindOfClass:[NSObject class]];
BOOL res3 = [[MJPerson class] isKindOfClass:[MJPerson class]];
BOOL res4 = [[MJPerson class] isKindOfClass:[MJPerson class]];

NSLog(@"%d %d %d %d", res1, res2, res3, res4);
```

- 以下代码能不能执行成功？如果可以，打印结果是什么？

```
@interface MJPerson : NSObject
@property (nonatomic, copy) NSString *name;
- (void)print;
@end
```

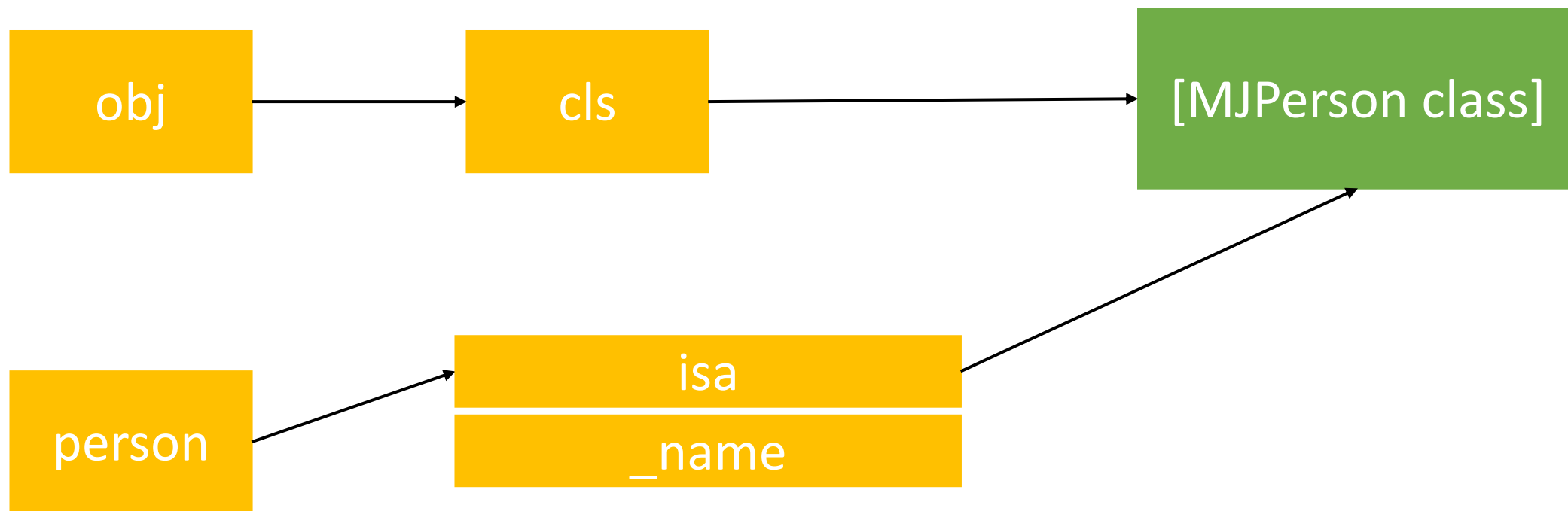
```
@implementation MJPerson
- (void)print {
    NSLog(@"my name's %@", self.name);
}
@end
```

```
@implementation ViewController

- (void)viewDidLoad {
    [super viewDidLoad];

    id cls = [MJPerson class];
    void *obj = &cls;
    [(__bridge id)obj print];
}

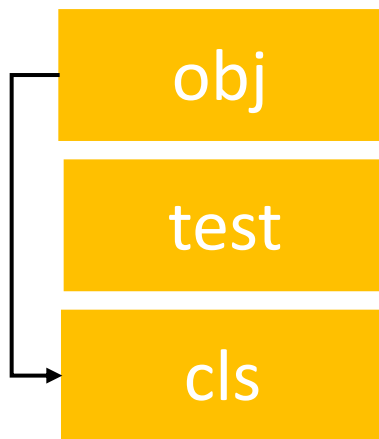
@end
```



低地址



高地址

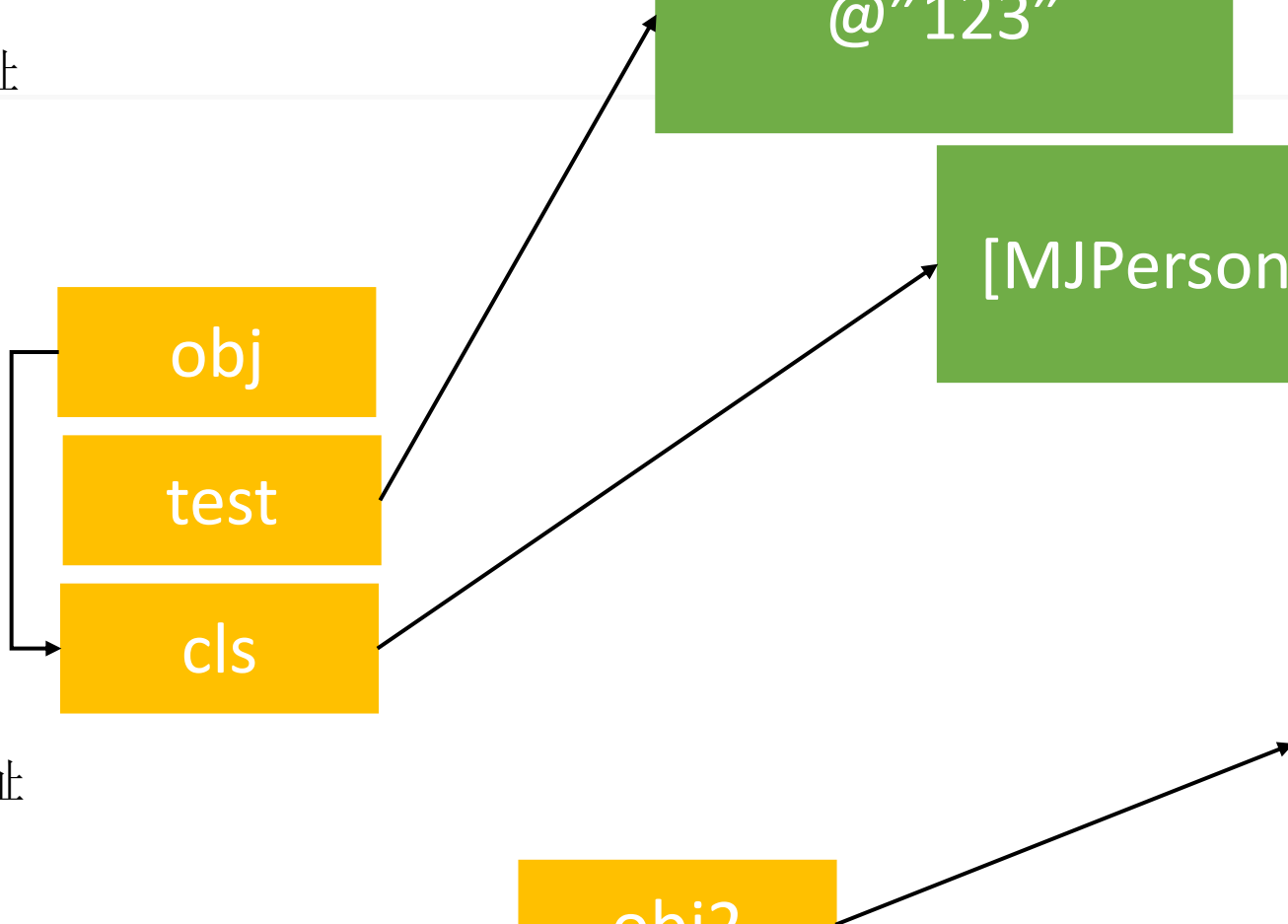


@"123"

[MJPerson class]

NSObject对象

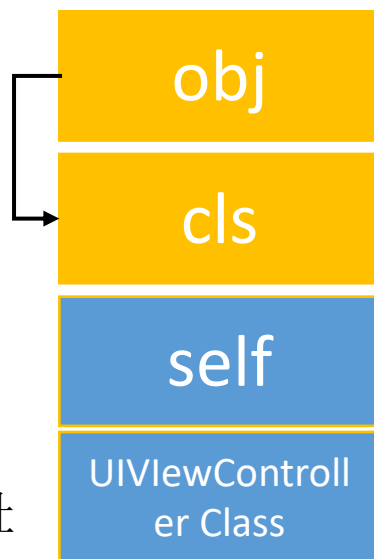
obj2



低地址



高地址



[MJPerson class]

- Objective-C是一门动态性比较强的编程语言，跟C、C++等语言有着很大的不同
- Objective-C的动态性是由Runtime API来支撑的
- Runtime API提供的接口基本都是C语言的，源码由C\C++\汇编语言编写

@selector(sendAction:to:forEvent:)

Method

SEL name

types

sendAction:to:forEvent:的实现

@selector(mj_sendAction:to:forEvent:)

Method

SEL name

types

mj_sendAction:to:forEvent:的实现

@selector(sendAction:to:forEvent:)

Method

SEL name

types

mj_sendAction:to:forEvent:的实现

@selector(mj_sendAction:to:forEvent:)

Method

SEL name

types

sendAction:to:forEvent:的实现

- 要想学习Runtime，首先要了解它底层的一些常用数据结构，比如isa指针
- 在arm64架构之前，isa就是一个普通的指针，存储着Class、Meta-Class对象的内存地址
- 从arm64架构开始，对isa进行了优化，变成了一个共用体（**union**）结构，还使用位域来存储更多的信息

```
union isa_t
{
    Class cls;
    uintptr_t bits;
    struct {
        uintptr_t nonpointer      : 1;
        uintptr_t has_assoc      : 1;
        uintptr_t has_cxx_dtor   : 1;
        uintptr_t shiftcls       : 33;
        uintptr_t magic          : 6;
        uintptr_t weakly_referenced : 1;
        uintptr_t deallocating    : 1;
        uintptr_t has_sidetable_rc : 1;
        uintptr_t extra_rc        : 19;
    };
};
```

■ nonpointer

- 0，代表普通的指针，存储着Class、Meta-Class对象的内存地址
- 1，代表优化过，使用位域存储更多的信息

■ has_assoc

- 是否有设置过关联对象，如果没有，释放时会更快

■ has_cxx_dtor

- 是否有C++的析构函数（.cxx_destruct），如果没有，释放时会更快

■ shiftcls

- 存储着Class、Meta-Class对象的内存地址信息

■ magic

- 用于在调试时分辨对象是否未完成初始化

■ weakly_referenced

- 是否有被弱引用指向过，如果没有，释放时会更快

■ deallocating

- 对象是否正在释放

■ extra_rc

- 里面存储的值是引用计数器减1

■ has_sidetable_rc

- 引用计数器是否过大无法存储在isa中
- 如果为1，那么引用计数会存储在一个叫SideTable的类的属性中

Class的结构

```
struct objc_class {  
    Class isa;  
    Class superclass;  
    cache_t cache; // 方法缓存  
    class_data_bits_t bits; // 用于获取具体的类信息  
};
```

& FAST_DATA_MASK

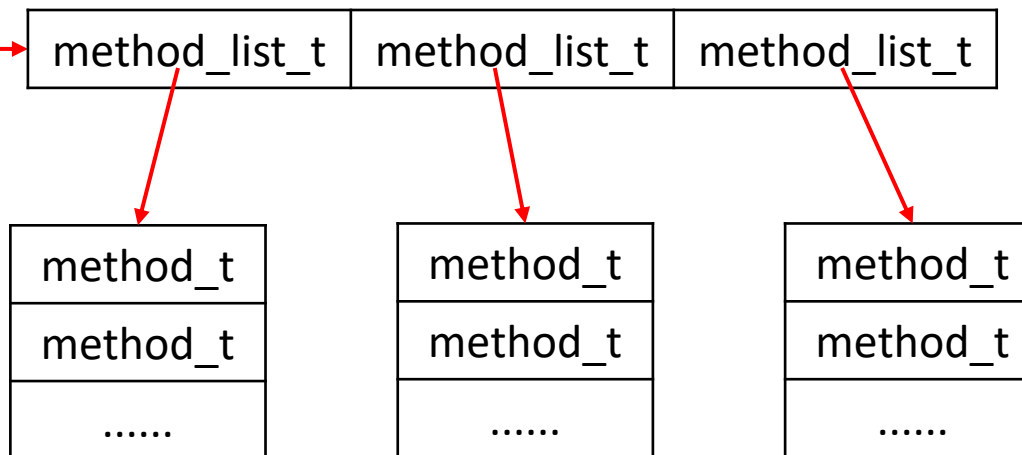
```
struct class_rw_t {  
    uint32_t flags;  
    uint32_t version;  
    const class_ro_t *ro; —————→  
    method_list_t *methods; // 方法列表  
    property_list_t *properties; // 属性列表  
    const protocol_list_t *protocols; // 协议列表  
    Class firstSubclass;  
    Class nextSiblingClass;  
    char *demangledName;  
};
```

```
struct class_ro_t {  
    uint32_t flags;  
    uint32_t instanceStart;  
    uint32_t instanceSize; // instance对象占用的内存空间  
#ifdef __LP64__  
    uint32_t reserved;  
#endif  
    const uint8_t *ivarLayout;  
    const char *name; // 类名  
    method_list_t *baseMethodList;  
    protocol_list_t *baseProtocols;  
    const ivar_list_t *ivars; // 成员变量列表  
    const uint8_t *weakIvarLayout;  
    property_list_t *baseProperties;  
};
```

class_rw_t

- class_rw_t里面的methods、properties、protocols是二维数组，是可读可写的，包含了类的初始内容、分类的内容

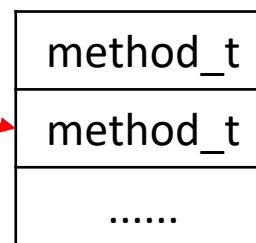
```
struct class_rw_t {  
    method_array_t methods;  
    property_array_t properties;  
    protocol_array_t protocols;  
};
```



class_ro_t

- class_ro_t里面的baseMethodList、baseProtocols、ivars、baseProperties是一维数组，是只读的，包含了类的初始内容

```
struct class_ro_t {  
    method_list_t * baseMethodList;  
    protocol_list_t * baseProtocols;  
    const ivar_list_t * ivars;  
    property_list_t * baseProperties;  
};
```



- method_t是对方法\函数的封装

```
struct method_t {  
    SEL name;    // 函数名  
    const char *types; // 编码 (返回值类型、参数类型)  
    IMP imp;     // 指向函数的指针 (函数地址)  
};
```

- IMP代表函数的具体实现

```
typedef id _Nullable (*IMP)(id _Nonnull, SEL _Nonnull, ...);
```

- SEL代表方法\函数名，一般叫做选择器，底层结构跟char *类似
- 可以通过@selector()和sel_registerName()获得
- 可以通过sel_getName()和NSStringFromSelector()转成字符串
- 不同类中相同名字的方法，所对应的方法选择器是相同的

```
typedef struct objc_selector *SEL;
```

- types包含了函数返回值、参数编码的字符串

返回值	参数1	参数2	参数n
-----	-----	-----	-------	-----

Type Encoding

- iOS中提供了一个叫做@encode的指令，可以将具体的类型表示成字符串编码

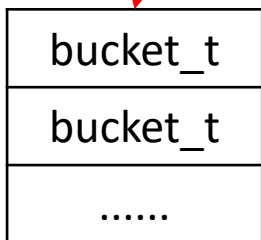
Code	Meaning
c	A char
i	An int
s	A short
l	A long l is treated as a 32-bit quantity on 64-bit programs.
q	A long long
C	An unsigned char
I	An unsigned int
S	An unsigned short
L	An unsigned long
Q	An unsigned long long
f	A float
d	A double
B	A C++ bool or a C99 _Bool

v	A void
*	A character string (char *)
@	An object (whether statically typed or typed id)
#	A class object (Class)
:	A method selector (SEL)
[array type]	An array
{name=type...}	A structure
(name=type...)	A union
bnum	A bit field of num bits
^type	A pointer to type
?	An unknown type (among other things, this code is used for function pointers)

- Class内部结构中有个方法缓存 (`cache_t`) , 用散列表 (哈希表) 来缓存曾经调用过的方法 , 可以提高方法的查找速度

```
struct cache_t {
    struct bucket_t *_buckets; // 散列表
    mask_t _mask; // 散列表的长度 - 1
    mask_t _occupied; // 已经缓存的方法数量
};
```

```
struct bucket_t {
    cache_key_t _key; // SEL作为key
    IMP _imp; // 函数的内存地址
};
```



- 缓存查找

- objc-cache.mm

- `bucket_t * cache_t::find(cache_key_t k, id receiver)`

@selector(studentTest) & _mask = 2

f(key) == index

@selector(personTest) & _mask = 2

@selector(goodStudentTest) & _mask = 7

空间换时间

0	NULL
1	NULL
2	bucket_t (_key = @selector(personTest), _imp)
3	NULL
4	NULL
5	NULL
.....

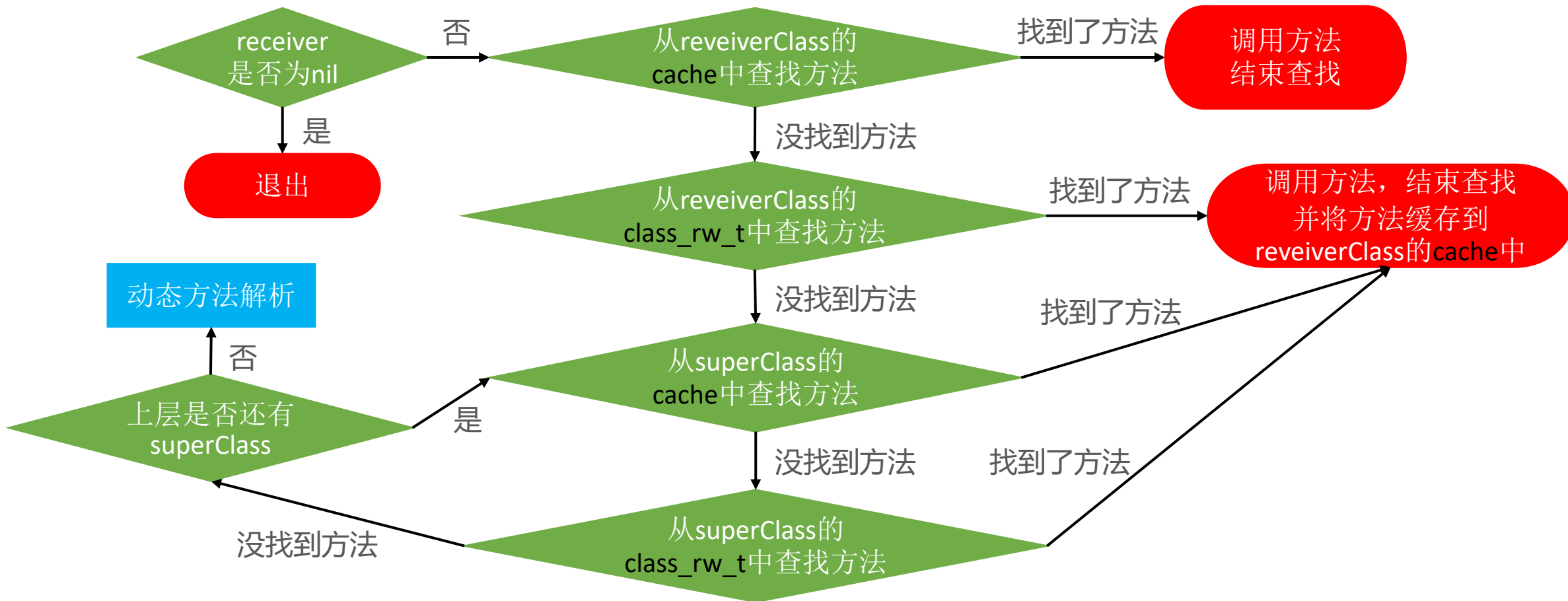
objc_msgSend执行流程

- OC中的方法调用，其实都是转换为objc_msgSend函数的调用
- objc_msgSend的执行流程可以分为3大阶段
 - 消息发送
 - 动态方法解析
 - 消息转发

objc_msgSend执行流程 – 源码跟读

- objc-msg-arm64.s
 - ENTRY _objc_msgSend
 - b.le LNilOrTagged
 - CacheLookup NORMAL
 - .macro CacheLookup
 - .macro CheckMiss
 - STATIC_ENTRY __objc_msgSend_uncached
 - .macro MethodTableLookup
 - __class_lookupMethodAndLoadCache3
- objc-runtime-new.mm
 - _class_lookupMethodAndLoadCache3
 - lookUpImpOrForward
 - getMethodNoSuper_nolock、search_method_list、log_and_fill_cache
 - cache_getImp、log_and_fill_cache、getMethodNoSuper_nolock、log_and_fill_cache
 - _class_resolveInstanceMethod
 - _objc_msgForward_imp_cache
- objc-msg-arm64.s
 - STATIC_ENTRY __objc_msgForward_imp_cache
 - ENTRY __objc_msgForward
- Core Foundation
 - __forwarding__ (不开源)

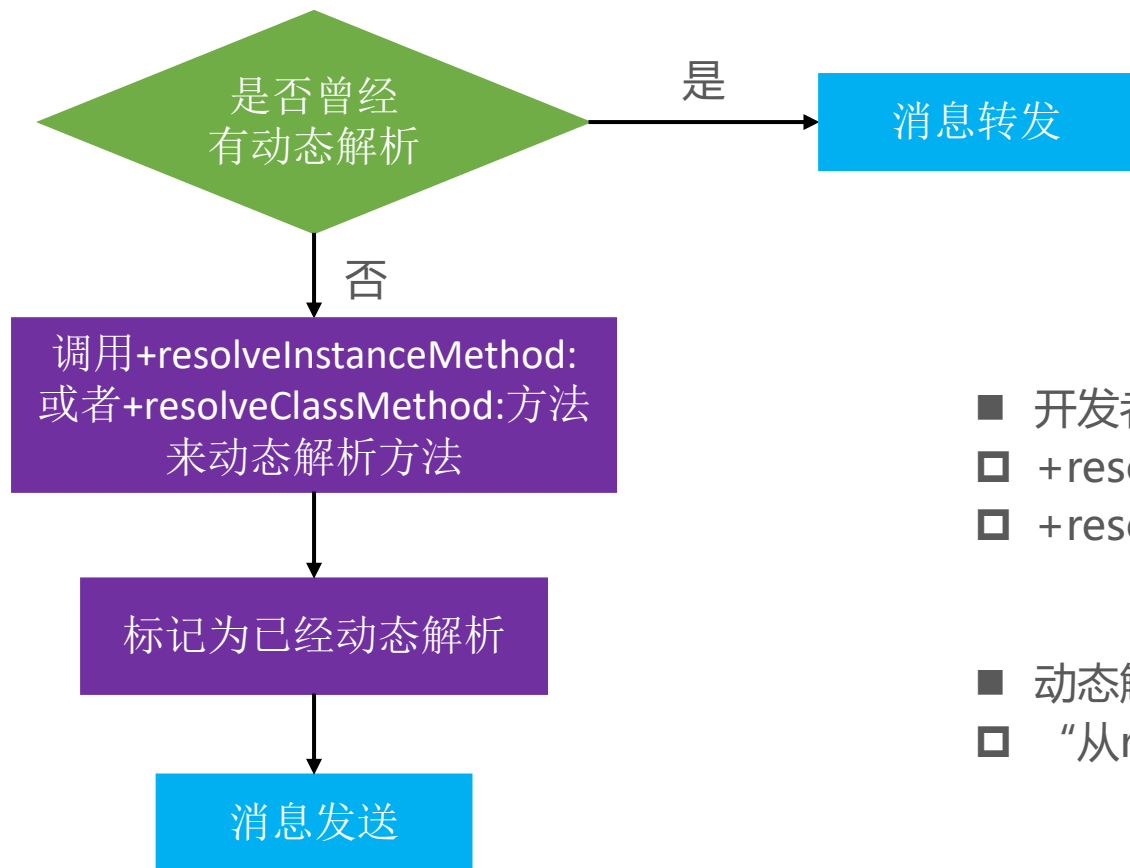
objc_msgSend执行流程01-消息发送



- 如果是从class_rw_t中查找方法
- 已经排序的，二分查找
- 没有排序的，遍历查找

- receiver通过isa指针找到receiverClass
- receiverClass通过superclass指针找到superClass

objc_msgSend执行流程02-动态方法解析



■ 开发者可以实现以下方法，来动态添加方法实现

□ +resolveInstanceMethod:

□ +resolveClassMethod:

■ 动态解析过后，会重新走“消息发送”的流程

□ “从receiverClass的cache中查找方法”这一步开始执行

动态添加方法

```
+ (BOOL)resolveInstanceMethod:(SEL)sel
{
    if (sel == @selector(test)) {
        Method method = class_getInstanceMethod(self, @selector(other));
        class_addMethod(self,
                        sel,
                        method_getImplementation(method),
                        method_getTypeEncoding(method));

        return YES;
    }
    return [super resolveInstanceMethod:sel];
}
```

```
void other(id self, SEL _cmd)
{
    NSLog(@"%@-%s--%s", self, sel_getName(_cmd), __func__);
}

+ (BOOL)resolveInstanceMethod:(SEL)sel
{
    if (sel == @selector(test)) {
        class_addMethod(self, sel, (IMP)other, "v@:");
        return YES;
    }
    return [super resolveInstanceMethod:sel];
}
```

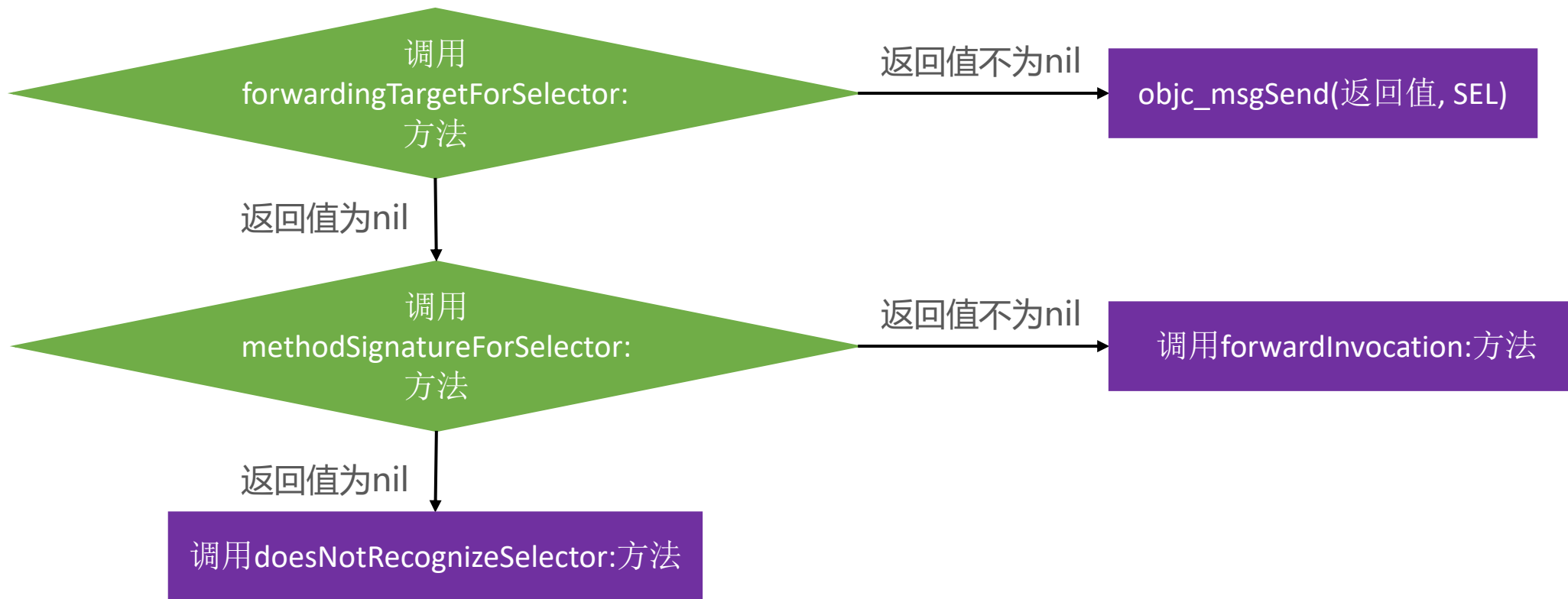
Method可以理解为等价于 `struct method_t *`

```
@interface MJPerson : NSObject
@property (assign, nonatomic) int age;
@end

@implementation MJPerson
@dynamic age;
@end
```

`@dynamic`是告诉编译器不用自动生成getter和setter的实现，等到运行时再添加方法实现

objc_msgSend的执行流程03-消息转发



- 开发者可以在forwardInvocation:方法中自定义任何逻辑
- 以上方法都有对象方法、类方法2个版本（前面可以是加号+，也可以是减号-）

生成NSMethodSignature

```
NSMethodSignature *signature = [NSMethodSignature signatureWithObjCTypes:"i@:i"];
```

```
NSMethodSignature *signature = [[[MJStudent alloc] init] methodSignatureForSelector:@selector(test:)];
```

super的本质

- super调用，底层会转换为objc_msgSendSuper2函数的调用，接收2个参数
- struct objc_super2
- SEL

```
struct objc_super2 {  
    id receiver;  
    Class current_class;  
};
```

- receiver是消息接收者
- current_class是receiver的Class对象

LLVM的中间代码（IR）

- Objective-C在变为机器代码之前，会被LLVM编译器转换为中间代码（Intermediate Representation）
- 可以使用以下命令行指令生成中间代码
 - `clang -emit-llvm -S main.m`
- 语法简介
 - @ - 全局变量
 - % - 局部变量
 - `alloca` - 在当前执行的函数的堆栈帧中分配内存，当该函数返回到其调用者时，将自动释放内存
 - `i32` - 32位4字节的整数
 - `align` - 对齐
 - `load` - 读出，`store` 写入
 - `icmp` - 两个整数值比较，返回布尔值
 - `br` - 选择分支，根据条件来转向label，不根据条件跳转的话类似 `goto`
 - `label` - 代码标签
 - `call` - 调用函数
- 具体可以参考官方文档：<https://llvm.org/docs/LangRef.html>

Runtime的应用01 – 查看私有成员变量

- 设置UITextField占位文字的颜色

```
self.textField.placeholder = @"请输入用户名";  
[self.textField setValue:[UIColor redColor] forKeyPath:@"_placeholderLabel.textColor"];
```

Runtime的应用02 – 字典转模型

- 利用Runtime遍历所有的属性或者成员变量
- 利用KVC设值



Runtime的应用02 – 替换方法实现

- `class_replaceMethod`
- `method_exchangeImplementations`

Runtime API01 – 类

- 动态创建一个类 (参数：父类，类名，额外的内存空间)

- `Class objc_allocateClassPair(Class superclass, const char *name, size_t extraBytes)`

- 注册一个类 (要在类注册之前添加成员变量)

- `void objc_registerClassPair(Class cls)`

- 销毁一个类

- `void objc_disposeClassPair(Class cls)`

- 获取isa指向的Class

- `Class object_getClass(id obj)`

- 设置isa指向的Class

- `Class object_setClass(id obj, Class cls)`

- 判断一个OC对象是否为Class

- `BOOL object_isClass(id obj)`

- 判断一个Class是否为元类

- `BOOL class_isMetaClass(Class cls)`

- 获取父类

- `Class class_getSuperclass(Class cls)`

- 获取一个实例变量信息

- `Ivar` `class_getInstanceVariable(Class cls, const char *name)`

- 拷贝实例变量列表（最后需要调用free释放）

- `Ivar *``class_copyIvarList(Class cls, unsigned int *outCount)`

- 设置和获取成员变量的值

- `void` `object_setIvar(id obj, Ivar ivar, id value)`

- `id` `object_getIvar(id obj, Ivar ivar)`

- 动态添加成员变量（已经注册的类是不能动态添加成员变量的）

- `BOOL` `class_addIvar(Class cls, const char * name, size_t size, uint8_t alignment, const char * types)`

- 获取成员变量的相关信息

- `const char *``ivar_getName(Ivar v)`

- `const char *``ivar_getTypeEncoding(Ivar v)`

Runtime API03 – 属性

- 获取一个属性

- `objc_property_t` `class_getProperty(Class cls, const char *name)`

- 拷贝属性列表 (最后需要调用free释放)

- `objc_property_t *``class_copyPropertyList(Class cls, unsigned int *outCount)`

- 动态添加属性

- `BOOL` `class_addProperty(Class cls, const char *name, const objc_property_attribute_t *attributes, unsigned int attributeCount)`

- 动态替换属性

- `void` `class_replaceProperty(Class cls, const char *name, const objc_property_attribute_t *attributes, unsigned int attributeCount)`

- 获取属性的一些信息

- `const char *``property_getName(objc_property_t property)`

- `const char *``property_getAttributes(objc_property_t property)`

Runtime API04 – 方法

■ 获得一个实例方法、类方法

■ `Method` `class_getInstanceMethod(Class cls, SEL name)`

□ `Method` `class_getClassMethod(Class cls, SEL name)`

■ 方法实现相关操作

□ `IMP` `class_getMethodImplementation(Class cls, SEL name)`

□ `IMP` `method_setImplementation(Method m, IMP imp)`

□ `void` `method_exchangeImplementations(Method m1, Method m2)`

■ 拷贝方法列表（最后需要调用free释放）

□ `Method *``class_copyMethodList(Class cls, unsigned int *outCount)`

■ 动态添加方法

□ `BOOL` `class_addMethod(Class cls, SEL name, IMP imp, const char *types)`

■ 动态替换方法

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

■ 获取方法的相关信息（带有copy的需要调用free去释放）

□ `SEL` `method_getName(Method m)`

□ `IMP` `method_getImplementation(Method m)`

□ `const char *``method_getTypeEncoding(Method m)`

□ `unsigned int` `method_getNumberOfArguments(Method m)`

□ `char *``method_copyReturnType(Method m)`

□ `char *``method_copyArgumentType(Method m, unsigned int index)`

Runtime API04 – 方法

■ 选择器相关

- `const char *sel_getName(SEL sel)`
- `SEL sel_registerName(const char *str)`

■ 用block作为方法实现

- `IMP imp_implementationWithBlock(id block)`
- `id imp_getBlock(IMP anImp)`
- `BOOL imp_removeBlock(IMP anImp)`