方法调用的本质

本文我们探寻方法调用的本质，首先通过一段代码，将方法调用代码转为c++代码查看方法调用的本质是什么样的。 **xcrun -sdk iphoneos clang -arch arm64 -rewrite-objc main.m**

```
[person test];
//  ---------- c++底层代码
((void (*)(id, SEL))(void *)objc_msgSend)((id)person,
sel_registerName("test"));
```

通过上述源码可以看出c++底层代码中方法调用其实都是转化为 objc_msgSend函数，OC的方法调用也叫消息机制，表示给方法调用者发送消息。 拿上述代码举例，上述代码中实际为给person实例对象发送一条test消息。 消息接受者：person 消息名称：test 在方法调用的过程中可以分为三个阶段。

消息发送阶段：负责从类及父类的缓存列表及方法列表查找方法。 动态解析阶段：如果消息发送阶段没有找到方法，则会进入动态解析阶段，负责动态的添加方法实现。 消息转发阶段：如果也没有实现动态解析方法，则会进行消息转发阶段，将消息转发给可以处理消息的接受者来处理。

如果消息转发也没有实现，就会报方法找不到的错误，无法识别消息，unrecognzied selector sent to instance

接下来我们通过源码探寻消息发送者三个阶段分别是如何实现的。

消息发送

在runtime源码中搜索_objc_msgSend查看其内部实现，在objc-msg-arm64.s汇编文件可以知道_objc_msgSend函数的实现

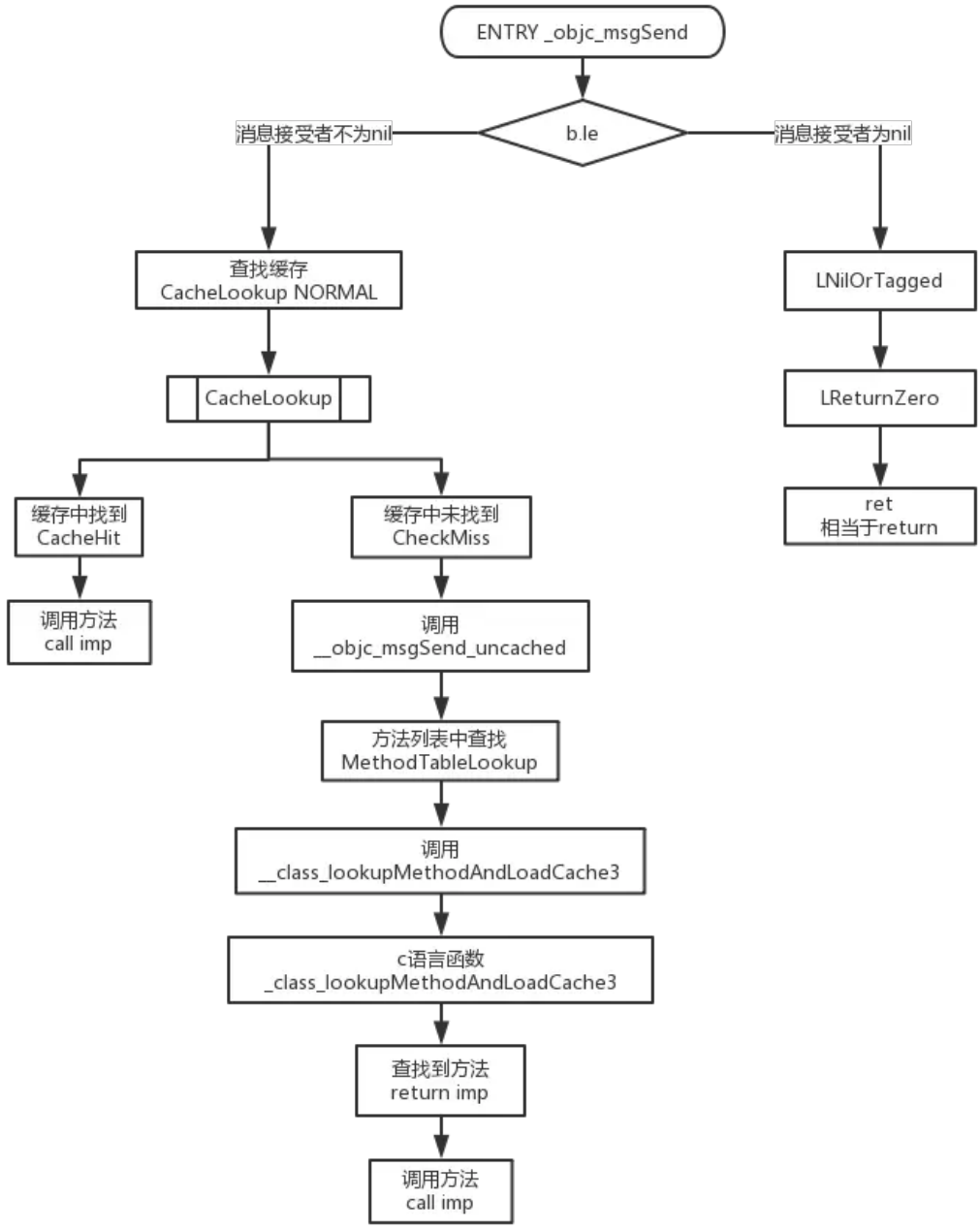
```
    ENTRY _objc_msgSend
    UNWIND _objc_msgSend, NoFrame
    MESSENGER_START

    cmp x0, #0          // nil check and tagged pointer check
    b.le    LNilOrTagged        //  (MSB tagged pointer looks
negative)
```

```
    ldr x13, [x0]        // x13 = isa
    and x16, x13, #ISA_MASK // x16 = class
LGetIsaDone:
    CacheLookup NORMAL      // calls imp or objc_msgSend_uncached
```

上述汇编源码中会首先判断消息接受者reveiver的值。 如果传入的消息接受者为nil则会执行LNilOrTagged，LNilOrTagged内部会执行LReturnZero，而LReturnZero内部则直接return0。 如果传入的消息接受者不为nill则执行CacheLookup，内部对方法缓存列表进行查找，如果找到则执行CacheHit，进而调用方法。否则执行CheckMiss，CheckMiss内部调用__objc_msgSend_uncached。__objc_msgSend_uncached内会执行MethodTableLookup也就是方法列表查找，MethodTableLookup内部的核心代码__class_lookupMethodAndLoadCache3也就是c语言函数_class_lookupMethodAndLoadCache3

c语言_class_lookupMethodAndLoadCache3函数内部则是对方法查找的核心源代码。

首先通过一张图看一下汇编语言中_objc_msgSend的运行流程。



方法查找的核心函数就是_class_lookupMethodAndLoadCache3函数，接下来重点分析_class_lookupMethodAndLoadCache3函数内的源码。

_class_lookupMethodAndLoadCache3 函数

```
IMP _class_lookupMethodAndLoadCache3(id obj, SEL sel, Class cls)
{
```

```
    return lookUpImpOrForward(cls, sel, obj,
                              YES/*initialize*/, NO/*cache*/,
YES/*resolver*/);
}
```

lookUpImpOrForward 函数

```
IMP lookUpImpOrForward(Class cls, SEL sel, id inst,
                       bool initialize, bool cache, bool resolver)
{
    // initialize = YES , cache = NO , resolver = YES
    IMP imp = nil;
    bool triedResolver = NO;
    runtimeLock.assertUnlocked();

    // 缓存查找，因为cache传入的为NO，这里不会进行缓存查找，因为在汇编语言中
CacheLookup已经查找过
    if (cache) {
        imp = cache_getImp(cls, sel);
        if (imp) return imp;
    }

    runtimeLock.read();
    if (!cls->isRealized()) {
        runtimeLock.unlockRead();
        runtimeLock.write();
        realizeClass(cls);
        runtimeLock.unlockWrite();
        runtimeLock.read();
    }
    if (initialize  &&  !cls->isInitialized()) {
        runtimeLock.unlockRead();
        _class_initialize (_class_getNonMetaClass(cls, inst));
        runtimeLock.read();
    }

 retry:
    runtimeLock.assertReading();

    // 防止动态添加方法，缓存会变化，再次查找缓存。
    imp = cache_getImp(cls, sel);
    // 如果查找到imp，直接调用done，返回方法地址
    if (imp) goto done;
```

```
    // 查找方法列表，传入类对象和方法名
    {
        // 根据sel去类对象里面查找方法
        Method meth = getMethodNoSuper_nolock(cls, sel);
        if (meth) {
            // 如果方法存在，则缓存方法，
            // 内部调用的就是 cache_fill 上文中已经详细讲解过这个方法，这里
不在赘述了。
            log_and_fill_cache(cls, meth->imp, sel, inst, cls);
            // 方法缓存之后，取出imp，调用done返回imp
            imp = meth->imp;
            goto done;
        }
    }

    // 如果类方法列表中没有找到，则去父类的缓存中或方法列表中查找方法
    {
        unsigned attempts = unreasonableClassCount();
        // 如果父类缓存列表及方法列表均找不到方法，则去父类的父类去查找。
        for (Class curClass = cls->superclass;
             curClass != nil;
             curClass = curClass->superclass)
        {
            // Halt if there is a cycle in the superclass chain.
            if (--attempts == 0) {
                _objc_fatal("Memory corruption in class list.");
            }

            // 查找父类的缓存
            imp = cache_getImp(curClass, sel);
            if (imp) {
                if (imp != (IMP)_objc_msgForward_impcache) {
                    // 在父类中找到方法，在本类中缓存方法，注意这里传入的是
cls，将方法缓存在本类缓存列表中，而非父类中
                    log_and_fill_cache(cls, imp, sel, inst,
curClass);
                    // 执行done，返回imp
                    goto done;
                }
                else {
                    // 跳出循环，停止搜索
                    break;
                }
            }

            // 查找父类的方法列表
            Method meth = getMethodNoSuper_nolock(curClass, sel);
```

```
            if (meth) {
                // 同样拿到方法，在本类进行缓存
                log_and_fill_cache(cls, meth->imp, sel, inst,
curClass);
                imp = meth->imp;
                // 执行done，返回imp
                goto done;
            }
        }
    }

    // ---------------- 消息发送阶段完成 ---------------------

    // ---------------- 进入动态解析阶段 ---------------------
    // 上述列表中都没有找到方法实现，则尝试解析方法
    if (resolver  &&  !triedResolver) {
        runtimeLock.unlockRead();
        _class_resolveMethod(cls, sel, inst);
        runtimeLock.read();
        triedResolver = YES;
        goto retry;
    }

    // ---------------- 动态解析阶段完成 ---------------------

    // ---------------- 进入消息转发阶段 ---------------------
    imp = (IMP)_objc_msgForward_impcache;
    cache_fill(cls, sel, imp, inst);

 done:
    runtimeLock.unlockRead();
    // 返回方法地址
    return imp;
}
```

getMethodNoSuper_nolock 函数

方法列表中查找方法

```
getMethodNoSuper_nolock(Class cls, SEL sel)
{
    runtimeLock.assertLocked();
    assert(cls->isRealized());
    // cls->data() 得到的是 class_rw_t
```

```
    // class_rw_t->methods 得到的是methods二维数组
    for (auto mlists = cls->data()->methods.beginLists(),
              end = cls->data()->methods.endLists();
         mlists != end;
         ++mlists)
    {
         // mlists 为 method_list_t
        method_t *m = search_method_list(*mlists, sel);
        if (m) return m;
    }
    return nil;
}
```

上述源码中getMethodNoSuper_nolock函数中通过遍历方法列表拿到method_list_t最终通过search_method_list函数查找方法

search_method_list函数

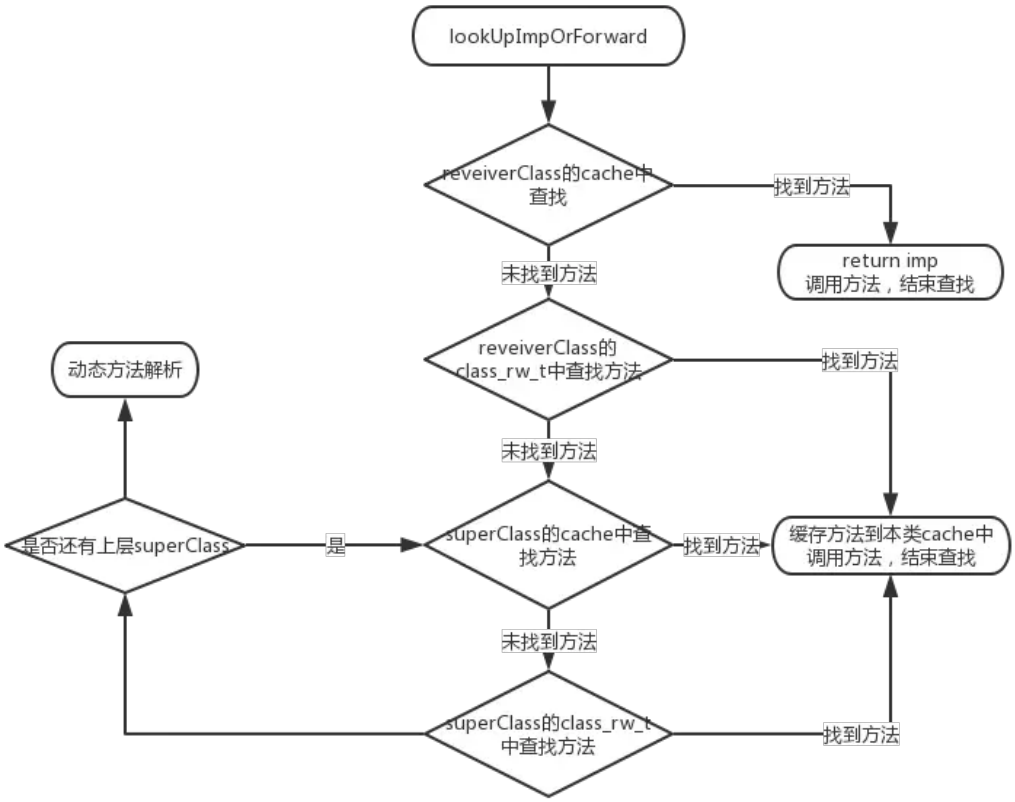
```
static method_t *search_method_list(const method_list_t *mlist, SEL
sel)
{
    int methodListIsFixedUp = mlist->isFixedUp();
    int methodListHasExpectedSize = mlist->entsize() ==
sizeof(method_t);
    // 如果方法列表是有序的，则使用二分法查找方法，节省时间
    if (__builtin_expect(methodListIsFixedUp &&
methodListHasExpectedSize, 1)) {
        return findMethodInSortedMethodList(sel, mlist);
    } else {
        // 否则则遍历列表查找
        for (auto& meth : *mlist) {
            if (meth.name == sel) return &meth;
        }
    }
    return nil;
}
```

findMethodInSortedMethodList函数内二分查找实现原理

```
static method_t *findMethodInSortedMethodList(SEL key, const
method_list_t *list)
```

```
{
    assert(list);

    const method_t * const first = &list->first;
    const method_t *base = first;
    const method_t *probe;
    uintptr_t keyValue = (uintptr_t)key;
    uint32_t count;
    // >>1 表示将变量n的各个二进制位顺序右移1位，最高位补二进制0。
    // count >>= 1 如果count为偶数则值变为(count / 2)。如果count为奇数则
值变为(count-1) / 2
    for (count = list->count; count != 0; count >>= 1) {
        // probe 指向数组中间的值
        probe = base + (count >> 1);
        // 取出中间method_t的name，也就是SEL
        uintptr_t probeValue = (uintptr_t)probe->name;
        if (keyValue == probeValue) {
            // 取出 probe
            while (probe > first && keyValue ==
(uintptr_t)probe[-1].name) {
                probe--;
            }
           // 返回方法
            return (method_t *)probe;
        }
        // 如果keyValue > probeValue 则折半向后查询
        if (keyValue > probeValue) {
            base = probe + 1;
            count--;
        }
    }

    return nil;
}
```

至此为止，消息发送阶段已经完成。 我们通过一站图来看一下_class_lookupMethodAndLoadCache3函数内部消息发送的整个流程



如果消息发送阶段没有找到方法，就会进入动态解析方法阶段。

动态解析阶段

当本类包括父类cache包括class_rw_t中都找不到方法时，就会进入动态方法解析阶段。我们来看一下动态解析阶段源码。

动态解析的方法

```
    if (resolver  &&  !triedResolver) {
        runtimeLock.unlockRead();
        _class_resolveMethod(cls, sel, inst);
        runtimeLock.read();
        // Don't cache the result; we don't hold the lock so it may
have
        // changed already. Re-do the search from scratch instead.
        triedResolver = YES;
        goto retry;
    }
```

_class_resolveMethod函数内部，根据类对象或元类对象做不同的操作

```
void _class_resolveMethod(Class cls, SEL sel, id inst)
{
    if (! cls->isMetaClass()) {
        // try [cls resolveInstanceMethod:sel]
        _class_resolveInstanceMethod(cls, sel, inst);
    }
    else {
        // try [nonMetaClass resolveClassMethod:sel]
        // and [cls resolveInstanceMethod:sel]
        _class_resolveClassMethod(cls, sel, inst);
        if (!lookUpImpOrNil(cls, sel, inst,
                            NO/*initialize*/, YES/*cache*/,
NO/*resolver*/))
        {
            _class_resolveInstanceMethod(cls, sel, inst);
        }
    }
}
```

上述代码中可以发现，动态解析方法之后，会将triedResolver = YES;那么下次就不会在进行动态解析阶段了，之后会重新执行retry，会重新对方法查找一遍。也就是说无论我们是否实现动态解析方法，无论动态解析方法是否成功，retry之后都不会在进行动态的解析方法了。

如何动态解析方法

动态解析对象方法时，会调用+(BOOL)resolveInstanceMethod:(SEL)sel方法。 动态解析类方法时，会调用+(BOOL)resolveClassMethod:(SEL)sel方法。 这里以实例对象为例通过代码来看一下动态解析的过程

```
@implementation Person
- (void) other {
    NSLog(@"%s", __func__);
}

+ (BOOL)resolveInstanceMethod:(SEL)sel
{
    // 动态的添加方法实现
```

```
    if (sel == @selector(test)) {
        // 获取其他方法 指向method_t的指针
        Method otherMethod = class_getInstanceMethod(self,
@selector(other));

        // 动态添加test方法的实现
        class_addMethod(self, sel,
method_getImplementation(otherMethod),
method_getTypeEncoding(otherMethod));

        // 返回YES表示有动态添加方法
        return YES;
    }

    NSLog(@"%s", __func__);
    return [super resolveInstanceMethod:sel];
}

@end
```

```
int main(int argc, const char * argv[]) {
    @autoreleasepool {
        Person *person = [[Person alloc] init];
        [person test];
    }
    return 0;
}
// 打印结果
// -[Person other]
```

上述代码中可以看出，person在调用test方法时经过动态解析成功调用了other方法。

通过上面对消息发送的分析我们知道，当本类和父类cache和class_rw_t中都找不到方法时，就会进行动态解析的方法，也就是说会自动调用类的resolveInstanceMethod:方法进行动态查找。因此我们可以在resolveInstanceMethod:方法内部使用class_addMethod动态的添加方法实现。

这里需要注意class_addMethod用来向具有给定名称和实现的类添加新方法，class_addMethod将添加一个方法实现的覆盖，但是不会替换已有的实现。也就是说如果上述代码中已经实现了-(void)test方法，则不会再动态添加方法，这点在上

述源码中也可以体现，因为一旦找到方法实现就直接return imp并调用方法了，不会再执行动态解析方法了。

class_addMethod 函数

我们来看一下class_addMethod函数的参数分别代表什么。

```
    /**
     第一个参数：  cls:给哪个类添加方法
     第二个参数：  SEL name:添加方法的名称
     第三个参数：  IMP imp: 方法的实现，函数入口，函数名可与方法名不同（建议与方法名相同）
     第四个参数：  types :方法类型，需要用特定符号，参考API
     */
class_addMethod(__unsafe_unretained Class cls, SEL name, IMP imp,
const char *types)
```

上述参数上文中已经详细讲解过，这里不再赘述。

需要注意的是我们在上述代码中通过class_getInstanceMethod获取Method的方法

```
// 获取其他方法 指向method_t的指针
Method otherMethod = class_getInstanceMethod(self,
@selector(other));
```

其实Method是objc_method类型结构体，可以理解为其内部结构同method_t结构体相同，上文中提到过method_t是代表方法的结构体，其内部包含SEL、type、IMP，我们通过自定义method_t结构体，将objc_method强转为method_t查看方法是否能够动态添加成功。

```
struct method_t {
    SEL sel;
    char *types;
    IMP imp;
};

- (void) other {
    NSLog(@"%s", __func__);
}
```

```
+ (BOOL)resolveInstanceMethod:(SEL)sel
{
    // 动态的添加方法实现
    if (sel == @selector(test)) {
        // Method强转为method_t
        struct method_t *method = (struct method_t
*)class_getInstanceMethod(self, @selector(other));

        NSLog(@"%s,%p,%s",method->sel,method->imp,method->types);

        // 动态添加test方法的实现
        class_addMethod(self, sel, method->imp, method->types);

        // 返回YES表示有动态添加方法
        return YES;
    }

    NSLog(@"%s", __func__);
    return [super resolveInstanceMethod:sel];
}
```

查看打印内容

```
动态解析方法[3246:1433553] other,0x100000d00,v16@0:8
动态解析方法[3246:1433553] -[Person other]
```

可以看出确实可以打印出相关信息，那么我们就可以理解为objc_method内部结构同method_t结构体相同，可以代表类定义中的方法。

另外上述代码中我们通过method_getImplementation函数和method_getTypeEncoding函数获取方法的imp和type。当然我们也可以通过自己写的方式来调用，这里以动态添加有参数的方法为例。

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

```
void cook(id self ,SEL _cmd,id Num)
{
    // 实现内容
    NSLog(@"%@的%@方法动态实现了,参数
为%@",self,NSStringFromSelector(_cmd),Num);
}
```

上述代码中当调用eat:方法时，动态添加了cook函数作为其实现并添加id类型的参数。

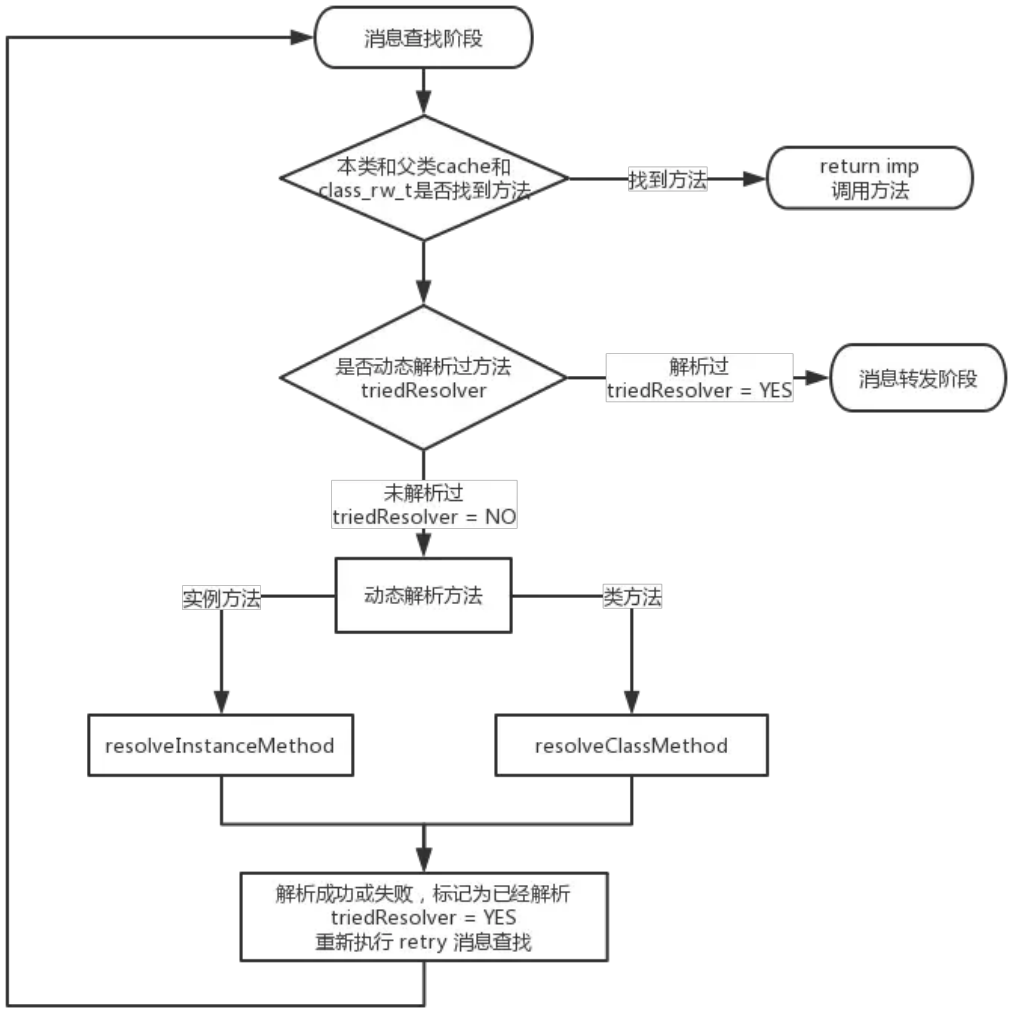
动态解析类方法

当动态解析类方法的时候，就会调用+(BOOL)resolveClassMethod:(SEL)sel函数，而我们知道类方法是存储在元类对象里面的，因此cls第一个对象需要传入元类对象以下代码为例

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

+ (BOOL)resolveClassMethod:(SEL)sel
{
    if (sel == @selector(test)) {
        // 第一个参数是object_getClass(self)，传入元类对象。
        class_addMethod(object_getClass(self), sel, (IMP)other,
"v16@0:8");
        return YES;
    }
    return [super resolveClassMethod:sel];
}
```

我们在上述源码的分析中提到过，无论我们是否实现了动态解析的方法，系统内部都会执行retry对方法再次进行查找，那么如果我们实现了动态解析方法，此时就会顺利查找到方法，进而返回imp对方法进行调用。如果我们没有实现动态解析方法。就会进行消息转发。

接下来看一下动态解析方法流程图示



消息转发

如果我们自己也没有对方法进行动态的解析，那么就会进行消息转发

```
imp = (IMP)_objc_msgForward_impcache;
cache_fill(cls, sel, imp, inst);
```

自己没有能力处理这个消息的时候，就会进行消息转发阶段，会调用_objc_msgForward_impcache函数。

通过搜索可以在汇编中找到__objc_msgForward_impcache函数实现，__objc_msgForward_impcache函数中调用__objc_msgForward进而找到

__objc_forward_handler。

```
objc_defaultForwardHandler(id self, SEL sel)
{
    _objc_fatal("%c[%s %s]: unrecognized selector sent to instance
%p "
                "(no message forward handler is installed)",
                class_isMetaClass(object_getClass(self)) ? '+' : '-
',
                object_getClassName(self), sel_getName(sel), self);
}
void *_objc_forward_handler = (void*)objc_defaultForwardHandler;
```

我们发现这仅仅是一个错误信息的输出。 其实消息转发机制是不开源的，但是我们可以猜测其中可能拿返回的对象调用了objc_msgSend，重走了一遍消息发送，动态解析，消息转发的过程。最终找到方法进行调用。

我们通过代码来看一下，首先创建Car类继承自NSObject，并且Car有一个- (void)driving方法，当Person类实例对象失去了驾车的能力，并且没有在开车过程中动态的学会驾车，那么此时就会将开车这条信息转发给Car，由Car实例对象来帮助person对象驾车。

```
#import "Car.h"
@implementation Car
- (void) driving
{
    NSLog(@"car driving");
}
@end

--------------

#import "Person.h"
#import <objc/runtime.h>
#import "Car.h"
@implementation Person
- (id)forwardingTargetForSelector:(SEL)aSelector
{
    // 返回能够处理消息的对象
    if (aSelector == @selector(driving)) {
        return [[Car alloc] init];
    }
    return [super forwardingTargetForSelector:aSelector];
```

```
}
@end

--------------

#import<Foundation/Foundation.h>
#import "Person.h"
int main(int argc, const char * argv[]) {
    @autoreleasepool {

        Person *person = [[Person alloc] init];
        [person driving];
    }
    return 0;
}

// 打印内容
// 消息转发[3452:1639178] car driving
```

由上述代码可以看出，当本类没有实现方法，并且没有动态解析方法，就会调用forwardingTargetForSelector函数，进行消息转发，我们可以实现forwardingTargetForSelector函数，在其内部将消息转发给可以实现此方法的对象。

如果forwardingTargetForSelector函数返回为nil或者没有实现的话，就会调用methodSignatureForSelector方法，用来返回一个方法签名，这也是我们正确跳转方法的最后机会。

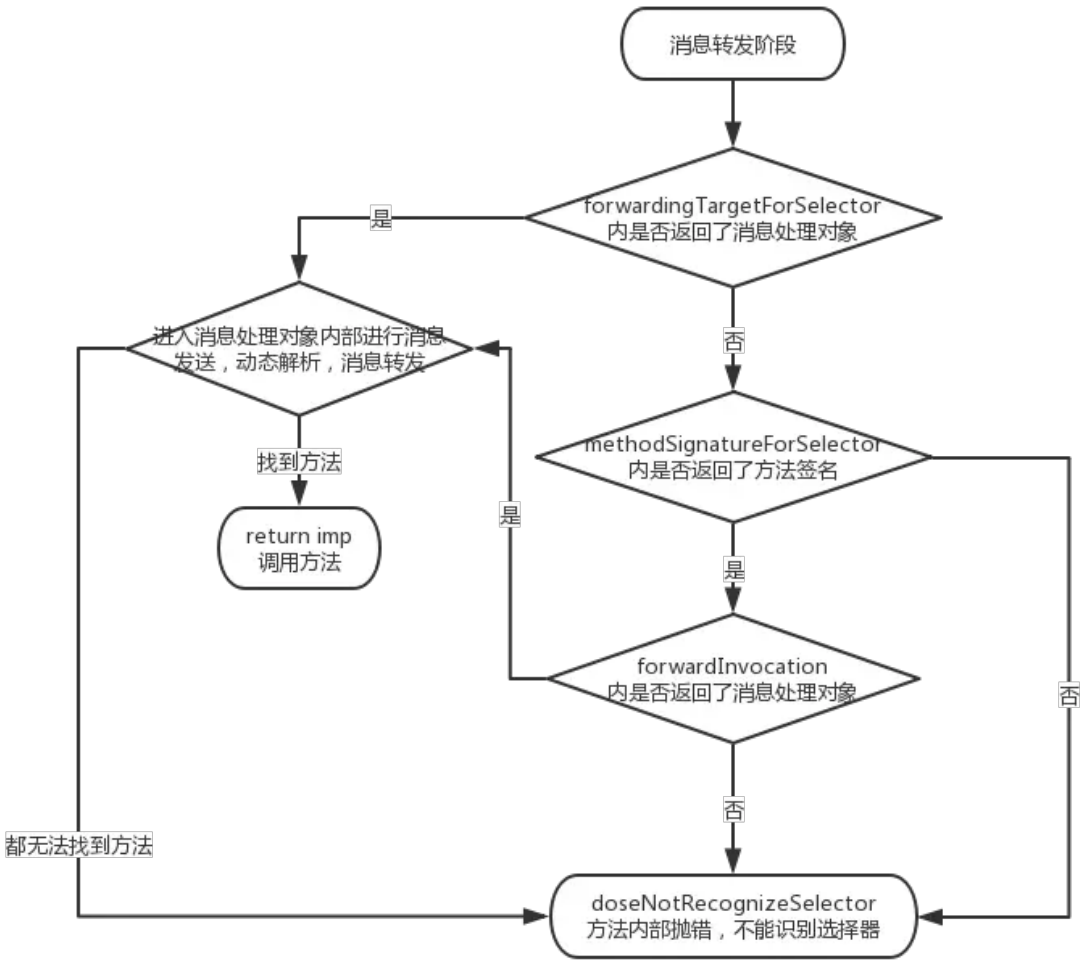
如果methodSignatureForSelector方法返回正确的方法签名就会调用forwardInvocation方法，forwardInvocation方法内提供一个NSInvocation类型的参数，NSInvocation封装了一个方法的调用，包括方法的调用者，方法名，以及方法的参数。在forwardInvocation函数内修改方法调用对象即可。

如果methodSignatureForSelector返回的为nil，就会来到doseNotRecognizeSelector:方法内部，程序crash提示无法识别选择器unrecognized selector sent to instance。 我们通过以下代码进行验证

```
- (id)forwardingTargetForSelector:(SEL)aSelector
{
    // 返回能够处理消息的对象
    if (aSelector == @selector(driving)) {
        // 返回nil则会调用methodSignatureForSelector方法
```

```
        return nil;
        // return [[Car alloc] init];
    }
    return [super forwardingTargetForSelector:aSelector];
}

// 方法签名：返回值类型、参数类型
- (NSMethodSignature *)methodSignatureForSelector:(SEL)aSelector
{
    if (aSelector == @selector(driving)) {
       // return [NSMethodSignature signatureWithObjCTypes: "v@:"];
       // return [NSMethodSignature signatureWithObjCTypes:
"v16@0:8"];
       // 也可以通过调用Car的methodSignatureForSelector方法得到方法签名，
这种方式需要car对象有aSelector方法
        return [[[Car alloc] init] methodSignatureForSelector:
aSelector];

    }
    return [super methodSignatureForSelector:aSelector];
}

//NSInvocation 封装了一个方法调用，包括：方法调用者，方法，方法的参数
//    anInvocation.target 方法调用者
//    anInvocation.selector 方法名
//    [anInvocation getArgument: NULL atIndex: 0]; 获得参数
- (void)forwardInvocation:(NSInvocation *)anInvocation
{
//   anInvocation中封装了methodSignatureForSelector函数中返回的方法。
//   此时anInvocation.target 还是person对象，我们需要修改target为可以执行
方法的方法调用者。
//   anInvocation.target = [[Car alloc] init];
//   [anInvocation invoke];
    [anInvocation invokeWithTarget: [[Car alloc] init]];
}

// 打印内容
// 消息转发[5781:2164454] car driving
```

上述代码中可以发现方法可以正常调用。接下来我们来看一下消息转发阶段的流程图



NSInvocation

methodSignatureForSelector方法中返回的方法签名，在forwardInvocation中被包装成NSInvocation对象，NSInvocation提供了获取和修改方法名、参数、返回值等方法，也就是说，在forwardInvocation函数中我们可以对方法进行最后的修改。

同样上述代码，我们为driving方法添加返回值和参数，并在forwardInvocation方法中修改方法的返回值及参数。

```
#import "Car.h"
@implementation Car
- (int) driving:(int)time
{
    NSLog(@"car driving %d",time);
    return time * 2;
```

```
}
@end

#import "Person.h"
#import <objc/runtime.h>
#import "Car.h"

@implementation Person
- (id)forwardingTargetForSelector:(SEL)aSelector
{
    // 返回能够处理消息的对象
    if (aSelector == @selector(driving)) {
        return nil;
    }
    return [super forwardingTargetForSelector:aSelector];
}

// 方法签名：返回值类型、参数类型
- (NSMethodSignature *)methodSignatureForSelector:(SEL)aSelector
{
    if (aSelector == @selector(driving:)) {
         // 添加一个int参数及int返回值type为 i@:i
         return [NSMethodSignature signatureWithObjCTypes: "i@:i"];
    }
    return [super methodSignatureForSelector:aSelector];
}


//NSInvocation 封装了一个方法调用，包括：方法调用者，方法，方法的参数
- (void)forwardInvocation:(NSInvocation *)anInvocation
{
    int time;
    // 获取方法的参数，方法默认还有self和cmd两个参数，因此新添加的参数下标为2
    [anInvocation getArgument: &time atIndex: 2];
    NSLog(@"修改前参数的值 = %d",time);
    time = time + 10; // time = 110
    NSLog(@"修改前参数的值 = %d",time);
    // 设置方法的参数 此时将参数设置为110
    [anInvocation setArgument: &time atIndex:2];

    // 将tagert设置为Car实例对象
    [anInvocation invokeWithTarget: [[Car alloc] init]];

    // 获取方法的返回值
    int result;
    [anInvocation getReturnValue: &result];
    NSLog(@"获取方法的返回值 = %d",result); // result = 220,说明参数修改
```

```
成功

    result = 99;
    // 设置方法的返回值 重新将返回值设置为99
    [anInvocation setReturnValue: &result];

    // 获取方法的返回值
    [anInvocation getReturnValue: &result];
    NSLog(@"修改方法的返回值为 = %d",result);    // result = 99
}

#import<Foundation/Foundation.h>
#import "Person.h"
int main(int argc, const char * argv[]) {
    @autoreleasepool {
        Person *person = [[Person alloc] init];
        // 传入100，并打印返回值
        NSLog(@"[person driving: 100] = %d",[person driving: 100]);
    }
    return 0;
}
```

```
消息转发[6415:2290423] 修改前参数的值 = 100
消息转发[6415:2290423] 修改前参数的值 = 110
消息转发[6415:2290423] car driving 110
消息转发[6415:2290423] 获取方法的返回值 = 220
消息转发[6415:2290423] 修改方法的返回值为 = 99
消息转发[6415:2290423] [person driving: 100] = 99
```

从上述打印结果可以看出forwardInvocation方法中可以对方法的参数及返回值进行修改。

并且我们可以发现，在设置tagert为Car实例对象时，就已经对方法进行了调用，而在forwardInvocation方法结束之后才输出返回值。

通过上述验证我们可以知道只要来到forwardInvocation方法中，我们便对方法调用有了绝对的掌控权，可以选择是否调用方法，以及修改方法的参数返回值等等。

类方法的消息转发

类方法消息转发同对象方法一样，同样需要经过消息发送，动态方法解析之后才会

进行消息转发机制。我们知道类方法是存储在元类对象中的，元类对象本来也是一种特殊的类对象。需要注意的是，类方法的消息接受者变为类对象。

当类对象进行消息转发时，对调用相应的+号的forwardingTargetForSelector、methodSignatureForSelector、forwardInvocation方法，需要注意的是+号方法仅仅没有提示，而不是系统不会对类方法进行消息转发。

下面通过一段代码查看类方法的消息转发机制。

```
int main(int argc, const char * argv[]) {
    @autoreleasepool {
        [Person driving];
    }
    return 0;
}

#import "Car.h"
@implementation Car
+ (void) driving;
{
    NSLog(@"car driving");
}
@end

#import "Person.h"
#import <objc/runtime.h>
#import "Car.h"

@implementation Person

+ (id)forwardingTargetForSelector:(SEL)aSelector
{
    // 返回能够处理消息的对象
    if (aSelector == @selector(driving)) {
        // 这里需要返回类对象
        return [Car class];
    }
    return [super forwardingTargetForSelector:aSelector];
}
// 如果forwardInvocation函数中返回nil 则执行下列代码
// 方法签名：返回值类型、参数类型
+ (NSMethodSignature *)methodSignatureForSelector:(SEL)aSelector
{
    if (aSelector == @selector(driving)) {
        return [NSMethodSignature signatureWithObjCTypes: "v@:"];
    }
```

```
    return [super methodSignatureForSelector:aSelector];
}

+ (void)forwardInvocation:(NSInvocation *)anInvocation
{
    [anInvocation invokeWithTarget: [Car class]];
}

// 打印结果
// 消息转发[6935:2415131] car driving
```

上述代码中同样可以对类对象方法进行消息转发。需要注意的是类方法的接受者为类对象。其他同对象方法消息转发模式相同。

总结

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）。方法调用过程中也就是objc_msgSend底层实现分为三个阶段：消息发送、动态方法解析、消息转发。本文主要对这三个阶段相互之间的关系以及流程进行的探索。上文中已经讲解的很详细，这里不再赘述。