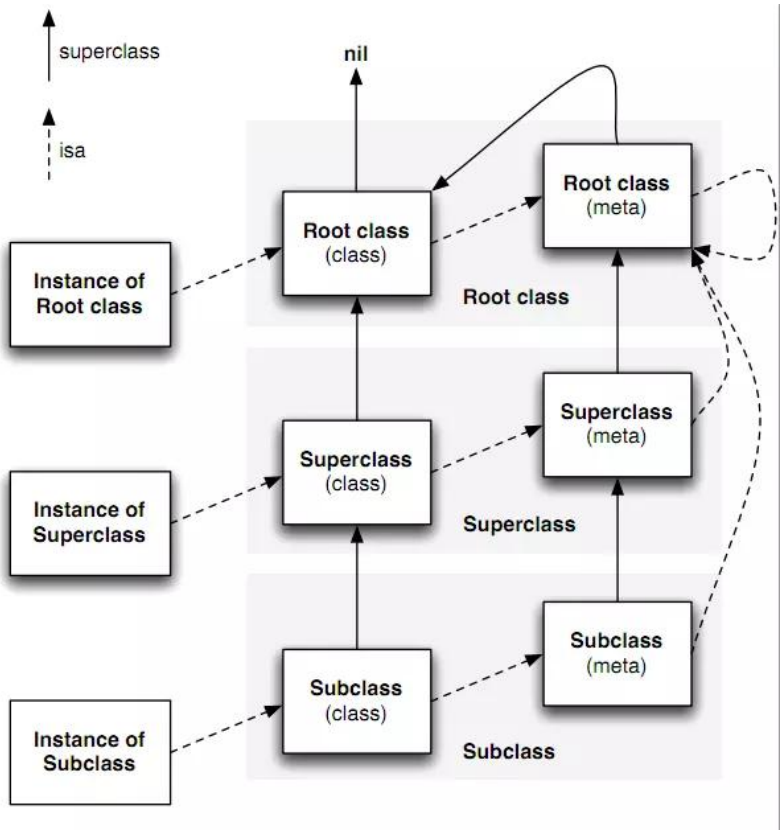
Runtime 面试题

一、objc 对象的 isa 的指针指向什么？有什么作用？

指向他的类对象,从而可以找到对象上的方法

详解：下图很好的描述了对象，类，元类之间的关系：



图中实线是 super_class 指针，虚线是 isa 指针。

1.Root class (class)其实就是 NSObject，NSObject 是没有超类的，所以 Root class(class)的 superclass 指向 nil。
2.每个 Class 都有一个 isa 指针指向唯一的 Meta class
3.Root class(meta)的 superclass 指向 Root class(class)，也就是 NSObject，形成一个回路。
4.每个 Meta class 的 isa 指针都指向 Root class (meta)。

二、一个 NSObject 对象占用多少内存空间?

受限于内存分配的机制，一个 NSObject 对象都会分配 16byte 的内存空间。

但是实际上在 64 位 下，只使用了 8byte;
在 32 位下，只使用了 4byte

一个 NSObject 实例对象成员变量所占的大小，实际上是 8 字节

```
#import <Objc/Runtime>
Class_getInstanceSize([NSObject Class])
```

本质是

```
size_t class_getInstanceSize(Class cls)
{
    if (!cls) return 0;
    return cls->alignedInstanceSize();
}
```

获取 Obj-C 指针所指向的内存的大小，实际上是 16 字节

```
#import <malloc/malloc.h>
malloc_size((__bridge const void *)obj);
```

对象在分配内存空间时，会进行内存对齐，所以在 iOS 中，分配内存空间都是 16 字节 的倍数。

可以通过以下网址 ： openSource.apple.com/tarballs 来查看源代码。

逻辑iOS技术号

三、说一下对 class_rw_t 的理解？

rw 代表可读可写。

ObjC 类中的属性、方法还有遵循的协议等信息都保存在 class_rw_t 中：

```
// 类结构
struct class_rw_t {
    // Be warned that Symbolication knows the layout of this structure.
    uint32_t flags;
    uint32_t version;

    const class_ro_t *ro; // 指向只读的结构体,存放类初始信息

    /*
     这三个都是二位数组，是可读可写的，包含了类的初始内容、分类的内容。
     methods中，存储 method_list_t ----> method_t
     二维数组，method_list_t --> method_t
     这三个二位数组中的数据有一部分是从class_ro_t中合并过来的。
     */
    method_array_t methods; // 方法列表（类对象存放对象方法，元类对象存放类方法）
    property_array_t properties; // 属性列表
    protocol_array_t protocols; //协议列表

    Class firstSubclass;
    Class nextSiblingClass;

    //...
    }
```

四、说一下对 class_ro_t 的理解？

存储了当前类在编译期就已经确定的属性、方法以及遵循的协议。

```
struct class_ro_t {
    uint32_t flags;
    uint32_t instanceStart;
    uint32_t instanceSize;
    uint32_t reserved;

    const uint8_t * ivarLayout;

    const char * name;
    method_list_t * baseMethodList;
    protocol_list_t * baseProtocols;
    const ivar_list_t * ivars;

    const uint8_t * weakIvarLayout;
    property_list_t *baseProperties;
};
```

五、说一下对 isa 指针的理解

说一下对 isa 指针的理解， 对象的 isa 指针指向哪里？isa 指针有哪两种类型？

isa 等价于 is kind of

- 实例对象 isa 指向类对象
- 类对象指 isa 向元类对象
- 元类对象的 isa 指向元类的基类

isa 有两种类型

- 纯指针，指向内存地址
- NON_POINTER_ISA，除了内存地址，还存有一些其他信息

isa 源码分析

在 Runtime 源码查看 isa_t 是共用体。简化结构如下：

```
union isa_t
{
    Class cls;
    uintptr_t bits;
    # if __arm64__ // arm64架构
#   define ISA_MASK        0x0000000ffffffff8ULL //用来取出33位内存地址使用（&）操作
#   define ISA_MAGIC_MASK  0x000003f000000001ULL
#   define ISA_MAGIC_VALUE 0x000001a000000001ULL
    struct {
        uintptr_t nonpointer        : 1; //0：代表普通指针，1：表示优化过的，可以存储更多信息。
        uintptr_t has_assoc         : 1; //是否设置过关联对象。如果没设置过，释放会更快
        uintptr_t has_cxx_dtor      : 1; //是否有C++的析构函数
        uintptr_t shiftcls          : 33; // MACH_VM_MAX_ADDRESS 0x1000000000 内存地址值
        uintptr_t magic             : 6; //用于在调试时分辨对象是否未完成初始化
        uintptr_t weakly_referenced : 1; //是否有被弱引用指向过
        uintptr_t deallocating      : 1; //是否正在释放
        uintptr_t has_sidetable_rc  : 1; //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做SideTable的类的
        uintptr_t extra_rc          : 19; //里面存储的值是引用计数器减1

#       define RC_ONE   (1ULL<<45)
#       define RC_HALF  (1ULL<<18)
    };

# elif __x86_64__ // arm86架构,模拟器是arm86
#   define ISA_MASK        0x00007ffffffffff8ULL
#   define ISA_MAGIC_MASK  0x001f800000000001ULL
#   define ISA_MAGIC_VALUE 0x001d800000000001ULL
    struct {
        uintptr_t nonpointer        : 1;
        uintptr_t has_assoc         : 1;
        uintptr_t has_cxx_dtor      : 1;
        uintptr_t shiftcls          : 44; // MACH_VM_MAX_ADDRESS 0x7fffffe00000
        uintptr_t magic             : 6;
        uintptr_t weakly_referenced : 1;
        uintptr_t deallocating      : 1;
        uintptr_t has_sidetable_rc  : 1;
        uintptr_t extra_rc          : 8;
#       define RC_ONE   (1ULL<<56)
#       define RC_HALF  (1ULL<<7)
    };

# else
# error unknown architecture for packed isa
# endif

};
```

六、说一下 Runtime 的方法缓存？存储的形式、数据结构以及查找的过程？

cache_t 增量扩展的哈希表结构。哈希表内部存储的 bucket_t。

bucket_t 中存储的是 SEL 和 IMP 的键值对。

- 如果是有序方法列表，采用二分查找
- 如果是无序方法列表，直接遍历查找

cache_t 结构体

```
// 缓存曾经调用过的方法，提高查找速率
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，散列表的长度使大于已经缓存的数量的。
    //...
}
```

```
struct bucket_t {
    cache_key_t _key; //SEL作为Key @selector()
    IMP _imp; // 函数的内存地址
    //...
}
```

散列表查找过程，在 objc-cache.mm 文件中

```
// 查询散列表，k
bucket_t * cache_t::find(cache_key_t k, id receiver)
{
    assert(k != 0); // 断言

    bucket_t *b = buckets(); // 获取散列表
    mask_t m = mask(); // 散列表长度 - 1
    mask_t begin = cache_hash(k, m); // & 操作
    mask_t i = begin; // 索引值
    do {
        if (b[i].key() == 0  ||  b[i].key() == k) {
            return &b[i];
        }
    } while ((i = cache_next(i, m)) != begin);
    // i 的值最大等于mask,最小等于0。

    // hack
    Class cls = (Class)((uintptr_t)this - offsetof(objc_class, cache));
    cache_t::bad_cache(receiver, (SEL)k, cls);
}
```

上面是查询散列表函数，其中 cache_hash(k, m)是静态内联方法，将传入的 key 和 mask 进行&操作返回 uint32_t 索引值。do-while 循环查找过程，当发生冲突 cache_next 方法将索引值减 1。

七、使用 runtime Associate 方法关联的对象，需要在主对象 dealloc 的时候释放么？

无论在 MRC 下还是 ARC 下均不需要，被关联的对象在生命周期内要比对象本身释放的晚很多，它们会在被 NSObject -dealloc 调用的 object_dispose()方法中释放。

详解：

```
1、调用 -release ：引用计数变为零
对象正在被销毁，生命周期即将结束.
不能再有新的 __weak 弱引用，否则将指向 nil.
调用 [self dealloc]

2、 父类调用 -dealloc
继承关系中最直接继承的父类再调用 -dealloc
如果是 MRC 代码 则会手动释放实例变量们（iVars）
继承关系中每一层的父类 都再调用 -dealloc

>3、NSObject 调 -dealloc
只做一件事：调用 Objective-C runtime 中object_dispose() 方法

>4. 调用 object_dispose()
为 C++ 的实例变量们（iVars）调用 destructors
为 ARC 状态下的 实例变量们（iVars） 调用 -release
解除所有使用 runtime Associate方法关联的对象
解除所有 __weak 引用
调用 free()
```

八、实例对象的数据结构？

具体可以参看 Runtime 源代码，在文件 objc-private.h 的第 127-232 行。

```
struct objc_object {
    isa_t isa;
    //...
}
```

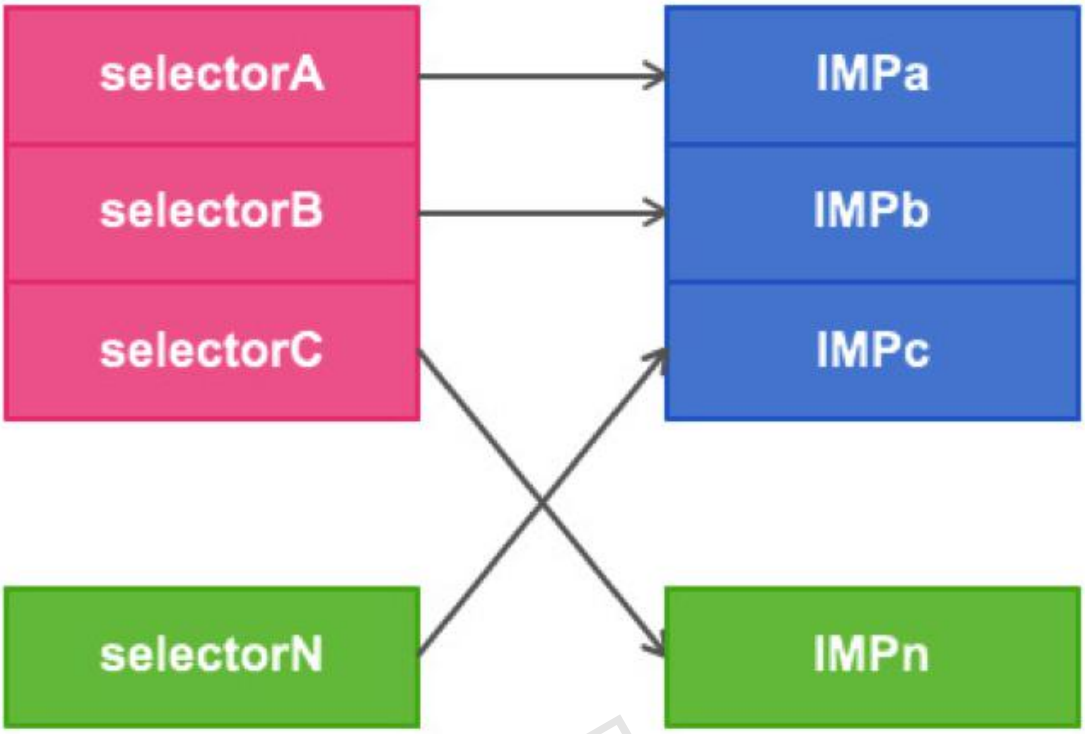
本质上 objc_object 的私有属性只有一个 isa 指针。指向 类对象 的内存地址。

九、什么是 method swizzling（俗称黑魔法）

简单说就是进行方法交换

在 Objective-C 中调用一个方法，其实是向一个对象发送消息，查找消息的唯一依据是 selector 的名字。利用 Objective-C 的动态特性，可以实现在运行时偷换 selector 对应的方法实现，达到给方法挂钩的目的。每个类都有一个方法列表，存放着方法的名字和方法实现的映射关系，selector 的本质其实就是方法名，IMP 有点类似函数指针，指向具体的 Method 实现，通过 selector 就可以找到对应的 IMP。
换方法的几种实现方式

- 利用 method_exchangeImplementations 交换两个方法的实现
- 利用 class_replaceMethod 替换方法的实现
- 利用 method_setImplementation 来直接设置某个方法的 IMP



十、什么时候会报 unrecognized selector 的异常？

objc 在向一个对象发送消息时，runtime 库会根据对象的 isa 指针找到该对象实际所属的类，然后在该类中的方法列表以及其父类方法列表中寻找方法运行，如果，在最顶层的父类中依然找不到相应的方法时，会进入消息转发阶段，如果消息三次转发流程仍未实现，则程序在运行时会挂掉并抛出异常 unrecognized selector sent to XXX 。

十一、如何给 Category 添加属性？关联对象以什么形式进行存储？

查看的是 关联对象 的知识点。

详细的说一下 关联对象。

关联对象 以哈希表的格式，存储在一个全局的单例中。

```
@interface NSObject (Extension)

@property (nonatomic,copy  ) NSString *name;

@end


@implementation NSObject (Extension)

- (void)setName:(NSString *)name {

    objc_setAssociatedObject(self, @selector(name), name, OBJC_ASSOCIATION_COPY_NONATOMIC);
}


- (NSString *)name {

    return objc_getAssociatedObject(self,@selector(name));
}

@end
```

十二、能否向编译后得到的类中增加实例变量？能否向运行时创建的类中添加实例变量？为什么？

不能向编译后得到的类中增加实例变量；

能向运行时创建的类中添加实例变量；

1.因为编译后的类已经注册在 runtime 中,类结构体中的 objc_ivar_list 实例变量的链表和 instance_size 实例变量的内存大小已经确定，同时 runtime 会调用 class_setvarlayout 或 class_setWeakIvarLayout 来处理 strong weak 引用.所以不能向存在的类中添加实例变量。
2.运行时创建的类是可以添加实例变量，调用 class_addIvar 函数．但是的在调用 objc_allocateClassPair 之后，objc_registerClassPair 之前,原因同上.

十三、类对象的数据结构？

具体可以参看 Runtime 源代码。

类对象就是 objc_class。

```
struct objc_class : objc_object {
    // Class ISA;
    Class superclass; //父类指针
    cache_t cache;             // formerly cache pointer and vtable 方法缓存
    class_data_bits_t bits;    // class_rw_t * plus custom rr/alloc flags 用于获取地址

    class_rw_t *data() {
        return bits.data(); // &FAST_DATA_MASK 获取地址值
    }
```

它的结构相对丰富一些。继承自 `objc_object` 结构体，所以包含 `isa` 指针

- `isa`：指向元类
- `superClass`：指向父类
- `Cache`：方法的缓存列表
- `data`：顾名思义，就是数据。是一个被封装好的 `class_rw_t` 。

十四、runtime 如何通过 selector 找到对应的 IMP 地址？

每一个类对象中都一个方法列表,方法列表中记录着方法的名称,方法实现,以及参数类型,其实 selector 本质就是方法名称,通过这个方法名称就可以在方法列表中找到对应的方法实现.

逻辑iOS技术号

十五、runtime 如何实现 weak 变量的自动置 nil？知道 SideTable 吗？

runtime 对注册的类会进行布局，对于 weak 修饰的对象会放入一个 hash 表中。 用 weak 指向的对象内存地址作为 key，当此对象的引用计数为 0 的时候会 dealloc，假如 weak 指向的对象内存地址是 a，那么就会以 a 为键， 在这个 weak 表中搜索，找到所有以 a 为键的 weak 对象，从而设置为 nil。

更细一点的回答：

1.初始化时：runtime 会调用 objc_initWeak 函数，初始化一个新的 weak 指针指向对象的地址。
2.添加引用时：objc_initWeak 函数会调用 objc_storeWeak() 函数， objc_storeWeak()的作用是更新指针指向，创建对应的弱引用表。
3.释放时,调用 clearDeallocating 函数。clearDeallocating 函数首先根据对象地址获取所有 weak 指针地址的数组，然后遍历这个数组把其中的数据设为 nil，最后把这个 entry 从 weak 表中删除，最后清理对象的记录。

SideTable 结构体是负责管理类的引用计数表和 weak 表，

详解：参考自《Objective-C 高级编程》一书
1.初始化时：runtime 会调用 objc_initWeak 函数，初始化一个新的 weak 指针指向对象的地址。

```
{
    NSObject *obj = [[NSObject alloc] init];
    id __weak obj1 = obj;
}
```

当我们初始化一个 weak 变量时，runtime 会调用 NSObject.mm 中的 objc_initWeak 函数。

```
// 编译器的模拟代码
 id obj1;
 objc_initWeak(&obj1, obj);
/*obj引用计数变为0，变量作用域结束*/
 objc_destroyWeak(&obj1);
```

通过 objc_initWeak 函数初始化“附有 weak 修饰符的变量（obj1）”，在变量作用域结束时通过 objc_destoryWeak 函数释放该变量（obj1）。

2. 添加引用时：objc_initWeak 函数会调用 objc_storeWeak() 函数， objc_storeWeak()的作用是更新指针指向，创建对应的弱引用表。

objc_initWeak 函数将“附有 weak 修饰符的变量（obj1）”初始化为 0（nil）后，会将“赋值对象”（obj）作为参数，调用 objc_storeWeak 函数。

```
obj1 = 0;
obj_storeWeak(&obj1, obj);
```

也就是说：

weak 修饰的指针默认值是 nil （在 Objective-C 中向 nil 发送消息是安全的）

然后 obj_destroyWeak 函数将 0（nil）作为参数，调用 objc_storeWeak 函数。

```
objc_storeWeak(&obj1, 0);
```

前面的源代码与下列源代码相同。

```
// 编译器的模拟代码
id obj1;
obj1 = 0;
objc_storeWeak(&obj1, obj);
/* ... obj的引用计数变为0，被置nil ... */
objc_storeWeak(&obj1, 0);
```

objc_storeWeak 函数把第二个参数的赋值对象（obj）的内存地址作为键值，将第一个参数__weak 修饰的属性变量（obj1）的内存地址注册到 weak 表中。如果第二个参数（obj）为 0（nil），那么把变量（obj1）的地址从 weak 表中删除。

由于一个对象可同时赋值给多个附有__weak 修饰符的变量中，所以对于一个键值，可注册多个变量的地址。

可以把 objc_storeWeak(&a, b)理解为：objc_storeWeak(value, key)，并且当 key 变 nil，将 value 置 nil。在 b 非 nil 时，a 和 b 指向同一个内存地址，在 b 变 nil 时，a 变 nil。此时向 a 发送消息不会崩溃：在 Objective-C 中向 nil 发送消息是安全的。

3. 释放时,调用 clearDeallocating 函数。clearDeallocating 函数首先根据对象地址获取所有 weak 指针地址的数组，然后遍历这个数组把其中的数据设为 nil，最后把这个 entry 从 weak 表中删除，最后清理对象的记录。

当 weak 引用指向的对象被释放时，又是如何去处理 weak 指针的呢？当释放对象时，其基本流程如下：

1.调用 objc_release
2.因为对象的引用计数为 0，所以执行 dealloc
3.在 dealloc 中，调用了_objc_rootDealloc 函数
4.在_objc_rootDealloc 中，调用了 object_dispose 函数
5.调用 objc_destructInstance
6.最后调用 objc_clear_deallocating

对象被释放时调用的 objc_clear_deallocating 函数:

1.从 weak 表中获取废弃对象的地址为键值的记录
2.将包含在记录中的所有附有 weak 修饰符变量的地址，赋值为 nil

3.将 weak 表中该记录删除
4.从引用计数表中删除废弃对象的地址为键值的记录

总结:

其实 Weak 表是一个 hash（哈希）表，Key 是 weak 所指对象的地址，Value 是 weak 指针的地址（这个地址的值是所指对象指针的地址）数组。

十六、objc 中向一个 nil 对象发送消息将会发生什么？

如果向一个 nil 对象发送消息，首先在寻找对象的 isa 指针时就是 0 地址返回了，所以不会出现任何错误。也不会崩溃。

详解:

如果一个方法返回值是一个对象，那么发送给 nil 的消息将返回 0(nil)；

如果方法返回值为指针类型，其指针大小为小于或者等于 sizeof(void*) ，float，double，long double 或者 long long 的整型标量，发送给 nil 的消息将返回 0；

如果方法返回值为结构体,发送给 nil 的消息将返回 0。结构体中各个字段的值将都是 0；

如果方法的返回值不是上述提到的几种情况，那么发送给 nil 的消息的返回值将是未定义的。

十七、objc 在向一个对象发送消息时，发生了什么？

objc 在向一个对象发送消息时，runtime 会根据对象的 isa 指针找到该对象实际所属的类，然后在该类中的方法列表以及其父类方法列表中寻找方法运行，如果一直到根类还没找到，转向拦截调用，走消息转发机制，一旦找到 ，就去执行它的实现 IMP 。

逻辑iOS技术号

十八、isKindOfClass 与 isMemberOfClass

下面代码输出什么?

```
@interface Sark : NSObject
@end
@implementation Sark
@end
int main(int argc, const char * argv[]) {
    @autoreleasepool {
        BOOL res1 = [(id)[NSObject class] isKindOfClass:[NSObject class]];
        BOOL res2 = [(id)[NSObject class] isMemberOfClass:[NSObject class]];
        BOOL res3 = [(id)[Sark class] isKindOfClass:[Sark class]];
        BOOL res4 = [(id)[Sark class] isMemberOfClass:[Sark class]];
        NSLog(@"%d %d %d %d", res1, res2, res3, res4);
    }
    return 0;
}
```

答案：1000

详解:

在 isKindOfClass 中有一个循环，先判断 class 是否等于 meta class，不等就继续循环判断是否等于 meta class 的 super class，不等再继续取 super class，如此循环下去。

[NSObject class]执行完之后调用 isKindOfClass, 第一次判断先判断 NSObject 和 NSObject 的 meta class 是否相等，之前讲到 meta class 的时候放了一张很详细的图，从图上我们也可以看出，NSObject 的 meta class 与本身不等。接着第二次循环判断 NSObject 与 meta class 的 superclass 是否相等。还是从那张图上面我们可以看到：Root class(meta) 的 superclass 就是 Root class(class)，也就是 NSObject 本身。所以第二次循环相等，于是第一行 res1 输出应该为 YES。

同理，[Sark class]执行完之后调用 isKindOfClass，第一次 for 循环，Sark 的 Meta Class 与[Sark class] 不等，第二次 for 循环，Sark Meta Class 的 super class 指向的是 NSObject Meta Class， 和 Sark Class 不相等。第三次 for 循环，NSObject Meta Class 的 super class 指向的是 NSObject Class，和 Sark Class 不相等。第四次循环，NSObject Class 的 super class 指向 nil， 和 Sark Class 不相等。第四次循环之后，退出循环，所以第三行的 res3 输出为 NO。

isMemberOfClass 的源码实现是拿到自己的 isa 指针和自己比较，是否相等。

第二行 isa 指向 NSObject 的 Meta Class，所以和 NSObject Class 不相等。第四行，isa 指向 Sark 的 Meta Class，和 Sark Class 也不等，所以第二行 res2 和第四行 res4 都输出 NO。

十九、Category 在编译过后，是在什么时机与原有的类合并到一起的？

1. 程序启动后，通过编译之后，Runtime 会进行初始化，调用 _objc_init。
2. 然后会 map_images。
3. 接下来调用 map_images_nolock。
4. 再然后就是 read_images，这个方法会读取所有的类的相关信息。
5. 最后是调用 reMethodizeClass:，这个方法是重新方法化的意思。
6. 在 reMethodizeClass: 方法内部会调用 attachCategories:，这个方法会传入 Class 和 Category，会将方法列表，协议列表等与原有的类合并。最后加入到 class_rw_t 结构体中。

二十、Category 有哪些用途？

- 给系统类添加方法、属性（需要关联对象）。
- 对某个类大量的方法，可以实现按照不同的名称归类。

二十一、Category 的实现原理？

被添加在了 class_rw_t 的对应结构里。

Category 实际上是 Category_t 的结构体，在运行时，新添加的方法，都被以倒序插入到原有方法列表的最前面，所以不同的 Category，添加了同一个方法，执行的实际上是最后一个。

拿方法列表举例，实际上是一个二维的数组。

Category 如果翻看源码的话就会知道实际上是一个 _catrgory_t 的结构体。

--
例如我们在程序中写了一个 Nsobject+Tools 的分类，那么被编译为 C++ 之后，实际上是：

```
static struct _catrgory_t _OBJC_$_CATEGORY_NSObject_$_Tools __attribute__ ((used,section),("__DATA,__objc__const"))
{
    // name
    // class
    // instance method list
    // class method list
    // protocol list
    // properties
}
```

Category 在刚刚编译完的时候，和原来的类是分开的，只有在程序运行起来后，通过 Runtime，Category 和原来的类才会合并到一起。

mememove，memcpy：这俩方法是位移、复制，简单理解就是原有的方法移动到最后，根根新开辟的控件，把前面的位置留给分类，然后分类中的方法，按照倒序依次插入，可以得出的结论就就是，越晚参与编译的分类，里面的方法才是生效的那个。

二十二、_objc_msgForward 函数是做什么的，直接调用它将会发生什么？

_objc_msgForward 是 IMP 类型，用于消息转发的：当向一个对象发送一条消息，但它并没有实现的时候，_objc_msgForward 会尝试做消息转发。

详解：_objc_msgForward 在进行消息转发的过程中会涉及以下这几个方法：

1. List itemresolveInstanceMethod:方法 (或 resolveClassMethod:)。
2. List itemforwardingTargetForSelector:方法
3. List itemmethodSignatureForSelector:方法
4. List itemforwardInvocation:方法
5. List itemdoesNotRecognizeSelector: 方法

具体请见：请看 Runtime 在工作中的运用 第三章 Runtime 方法调用流程；

二十三、[self class] 与 [super class]

下面的代码输出什么？

```
@implementation Son : Father
- (id)init
{
    self = [super init];
    if (self) {
        NSLog(@"%@", NSStringFromClass([self class]));
        NSLog(@"%@", NSStringFromClass([super class]));
    }
    return self;
}
@end
```

NSStringFromClass([self class]) = Son
NSStringFromClass([super class]) = Son

详解：这个题目主要是考察关于 Objective-C 中对 self 和 super 的理解。

self 是类的隐藏参数，指向当前调用方法的这个类的实例；

super 本质是一个编译器标示符，和 self 是指向的同一个消息接受者。不同点在于：super 会告诉编译器，当调用方法时，去调用父类的方法，而不是本类中的方法。

当使用 self 调用方法时，会从当前类的方法列表中开始找，如果没有，就从父类中再找；而当使用 super 时，则从父类的方法列表中开始找。然后调用父类的这个方法。

在调用[super class]的时候，runtime 会去调用 objc_msgSendSuper 方法，而不是 objc_msgSend；

```
OBJC_EXPORT void objc_msgSendSuper(void /* struct objc_super *super, SEL op, ... */ )

/// Specifies the superclass of an instance.
struct objc_super {
    /// Specifies an instance of a class.
    __unsafe_unretained id receiver;

    /// Specifies the particular superclass of the instance to message.
#if !defined(__cplusplus)  &&  !__OBJC2__
    /* For compatibility with old objc-runtime.h header */
    __unsafe_unretained Class class;
#else
    __unsafe_unretained Class super_class;
#endif
    /* super_class is the first class to search */
};
```

在 objc_msgSendSuper 方法中，第一个参数是一个 objc_super 的结构体，这个结构体里面有两个变量，一个是接收消息的 receiver，一个是当前类的父类 super_class。

objc_msgSendSuper 的工作原理应该是这样的:
从 objc_super 结构体指向的 superClass 父类的方法列表开始查找 selector，找到后以 objc->receiver 去调用父类的这个 selector。注意，最后的调用者是 objc->receiver，而不是 super_class!

那么 objc_msgSendSuper 最后就转变成:

```
// 注意这里是从父类开始msgSend，而不是从本类开始
objc_msgSend(objc_super->receiver, @selector(class))

/// Specifies an instance of a class.  这是类的一个实例
    __unsafe_unretained id receiver;


// 由于是实例调用，所以是减号方法
- (Class)class {
    return object_getClass(self);
}
```

由于找到了父类 NSObject 里面的 class 方法的 IMP，又因为传入的入参 objc_super->receiver = self。self 就是 son，调用 class，所以父类的方法 class 执行 IMP 之后，输出还是 son，最后输出两个都一样，都是输出 son。

逻辑iOS技术号