xx_cc Lv3
2018年04月07日 阅读 6035

关注

iOS底层原理总结 - 探寻OC对象的本质

iOS底层原理总结 - 探寻OC对象的本质

对小码哥底层班视频学习的总结与记录。面试题部分，通过对面试题的分析探索问题的本质内容。

> 面试题：一个NSObject对象占用多少内存?

探寻OC对象的本质，我们平时编写的Objective-C代码，底层实现其实都是C\C++代码。

OC的对象结构都是通过基础C\C++的结构体实现的。 我们通过创建OC文件及对象，并将OC文件转化为C++文件来探寻OC对象的本质

OC如下代码

复制代码

```
#import <Foundation/Foundation.h>

int main(int argc, const char * argv[]) {
    @autoreleasepool {
        NSObject *objc = [[NSObject alloc] init];

        NSLog(@"Hello, World!");
    }
    return 0;
}
```

我们通过命令行将OC的mian.m文件转化为c++文件。

复制代码

```
clang -rewrite-objc main.m -o main.cpp // 这种方式没有指定架构例如arm64架构 其中cpp代表（c plu
生成 main.cpp
```

我们可以指定架构模式的命令行，使用xcode工具 xcrun

复制代码

```
xcrun -sdk iphoneos clang -arch arm64 -rewrite-objc main.m -o main-arm64.cpp
生成 main-arm64.cpp
```

main-arm64.cpp 文件中搜索NSObjcet，可以找到NSObjcet_IMPL（IMPL代表 implementation 实现）

我们看一下NSObject_IMPL内部

复制代码

```
struct NSObject_IMPL {
        Class isa;
};
// 查看Class本质
typedef struct objc_class *Class;
我们发现Class其实就是一个指针，对象底层实现其实就是这个样子。
```

思考： 一个OC对象在内存中是如何布局的。 NSObjcet的底层实现，点击NSObjcet进入发现NSObject的内部实现

复制代码

```
@interface NSObject <NSObject> {
#pragma clang diagnostic push
#pragma clang diagnostic ignored "-Wobjc-interface-ivars"
    Class isa  OBJC_ISA_AVAILABILITY;
#pragma clang diagnostic pop
}
@end
```

转化为c语言其实就是一个结构体

```
struct NSObject_IMPL {
    Class isa;
};
```

> 那么这个结构体占多大的内存空间呢，我们发现这个结构体只有一个成员，isa指针，而指针在64位架构中占8个字节。也就是说一个NSObjec对象所占用的内存是8个字节。到这里我们已经可以基本解答第一个问题。但是我们发现NSObject对象中还有很多方法，那这些方法不占用内存空间吗？其实类的方法等也占用内存空间，但是这些方法所占用的存储空间并不在NSObject对象中。

为了探寻OC对象在内存中如何体现，我们来看下面一段代码

```
NSObject *objc = [[NSObject alloc] init];
```

上面一段代码在内存中如何体现的呢？上述一段代码中系统为NSObject对象分配8个字节的内存空间，用来存放一个成员isa指针。那么isa指针这个变量的地址就是结构体的地址，也就是NSObjcet对象的地址。 假设isa的地址为0x100400110，那么上述代码分配存储空间给NSObject对象，然后将存储空间的地址赋值给objc指针。objc存储的就是isa的地址。objc指向内存中NSObject对象地址，即指向内存中的结构体，也就是isa的位置。

自定义类的内部实现

```
@interface Student : NSObject{

    @public
    int _no;
    int _age;
}
@end
@implementation Student

int main(int argc, const char * argv[]) {
    @autoreleasepool {

        Student *stu = [[Student alloc] init];
        stu -> _no = 4;
        stu -> _age = 5;
```

```
        NSLog(@"%@",stu);
    }
    return 0;
}
@end
```

按照上述步骤同样生成c++文件。并查找Student，我们发现Student_IMPL

复制代码

```
struct Student_IMPL {
        struct NSObject_IMPL NSObject_IVARS;
        int _no;
        int _age;
};
```

发现第一个是 NSObject_IMPL的实现。而通过上面的实验我们知道NSObject_IMPL内部其实就是 Class isa 那么我们假设 struct NSObject_IMPL NSObject_IVARS; 等价于 Class isa;

可以将上述代码转化为

复制代码

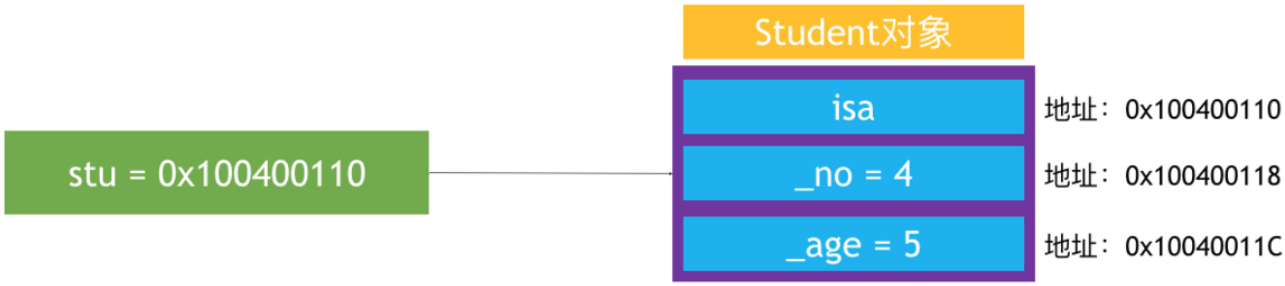
```
struct Student_IMPL {
        Class *isa;
        int _no;
        int _age;
};
```

因此此结构体占用多少存储空间，对象就占用多少存储空间。因此结构体占用的存储空间为，isa指针8个字节空间+int类型_no4个字节空间+int类型_age4个字节空间共16个字节空间

复制代码

```
Student *stu = [[Student alloc] init];
stu -> _no = 4;
stu -> _age = 5;
```

那么上述代码实际上在内存中的体现为，创建Student对象首先会分配16个字节，存储3个东西，isa指针8个字节，4个字节的_no ,4个字节的_age



sutdent对象的3个变量分别有自己的地址。而stu指向isa指针的地址。因此stu的地址为0x100400110，stu对象在内存中占用16个字节的空间。并且经过赋值，_no里面存储4，_age里面存储5

验证Student在内存中模样

复制代码

```
struct Student_IMPL {
    Class isa;
    int _no;
    int _age;
};

@interface Student : NSObject
{
    @public
    int _no;
    int _age;
}
@end

@implementation Student

int main(int argc, const char * argv[]) {
    @autoreleasepool {
            // 强制转化
            struct Student_IMPL *stuImpl = (__bridge struct Student_IMPL *)stu;
            NSLog(@"_no = %d, _age = %d", stuImpl->_no, stuImpl->_age); // 打印出 _no = 4
    }
    return 0;
}
```

上述代码将oc对象强转成Student_IMPL的结构体。也就是说把一个指向oc对象的指针，指向这种结构体。由于我们之前猜想，对象在内存中的布局与结构体在内存中的布局相同，那么如果可以转化成功，说明我们的猜想正确。由此说明stu这个对象指向的内存确实是一个结构体。

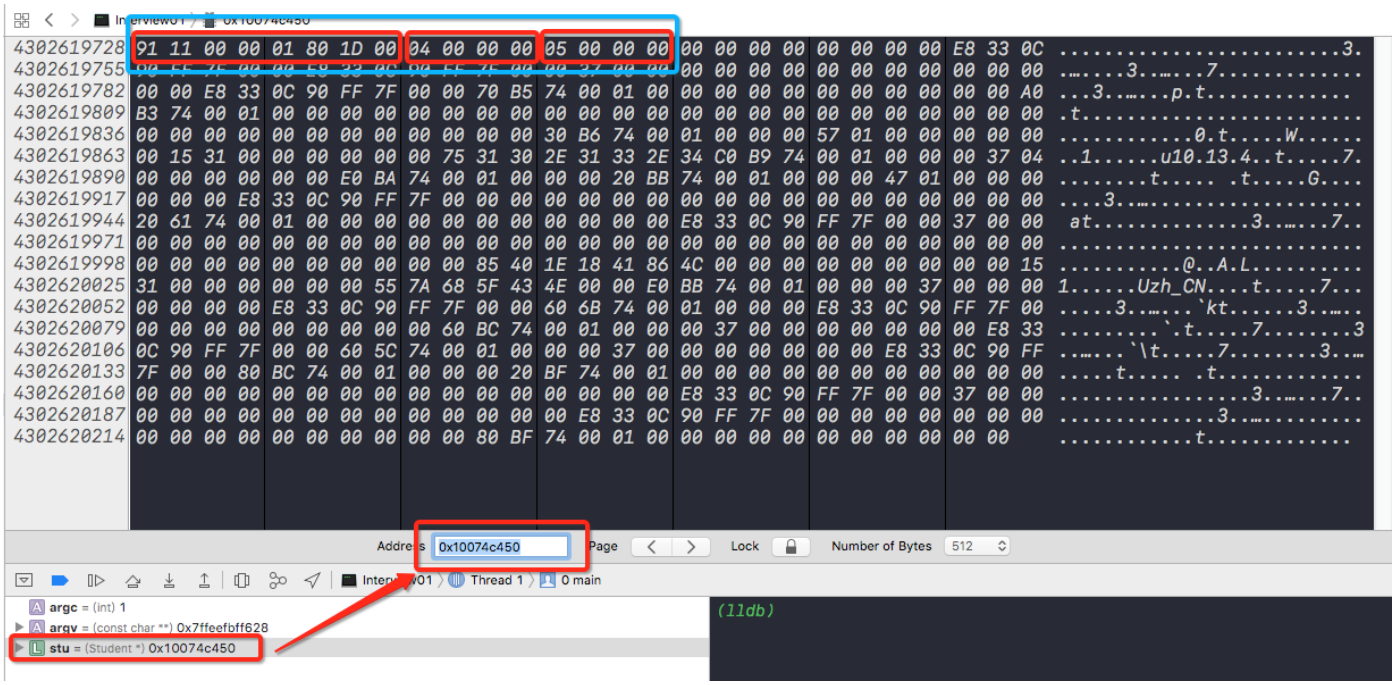
实际上想要获取对象占用内存的大小，可以通过更便捷的运行时方法来获取。

```
复制代码
class_getInstanceSize([Student class])
NSLog(@"%zd,%zd", class_getInstanceSize([NSObject class]) ,class_getInstanceSize([Studen
// 打印信息 8和16
```

窥探内存结构

实时查看内存数据

方式一：通过打断点。 Debug Workflow -> viewMemory address中输入stu的地址



从上图中，我们可以发现读取数据从高位数据开始读，查看前16位字节，每四个字节读出的数据为 16进制 0x0000004(4字节) 0x0000005(4字节) isa的地址为 00D1081000001119(8字节)

方式二：通过lldb指令xcode自带的调试器

```
复制代码
memory read 0x10074c450
// 简写  x 0x10074c450

// 增加读取条件
// memory read/数量格式字节数  内存地址
// 简写 x/数量格式字节数  内存地址
// 格式 x是16进制，f是浮点，d是10进制
```

```
// 字节大小    b：byte 1字节，h：half word 2字节，w：word 4字节，g：giant word 8字节

示例：x/4xw     //    /后面表示如何读取数据 w表示4个字节4个字节读取，x表示以16进制的方式读取数据，4则
```

同时也可以通过lldb修改内存中的值

复制代码

```
memory write 0x100400c68 6
将_no的值改为了6
```

```
(lldb) memory read 0x10074c450
0x10074c450: 91 11 00 00 01 80 1d 00 04 00 00 00 05 00 00 00  ................
0x10074c460: 00 00 00 00 00 00 00 00 e8 33 0c 90 ff 7f 00 00  .........3......
(lldb) x/4xw 0x10074c450
0x10074c450: 0x00001191 0x001d8001 0x00000004 0x00000005
(lldb) x/4dw 0x10074c450
0x10074c450: 4497
0x10074c454: 1933313
0x10074c458: 4
0x10074c45c: 5
(lldb) memory write 0x10074c458 6
(lldb) po stu->_no
6

(lldb)
```

> 那么一个NSObject对象占用多少内存？ **NSObjcet实际上是只有一个名为isa的指针的结构体，因此占用一个指针变量所占用的内存空间大小，如果64bit占用8个字节，如果32bit占用4个字节。**

更复杂的继承关系

> 面试题：在64bit环境下， 下面代码的输出内容?

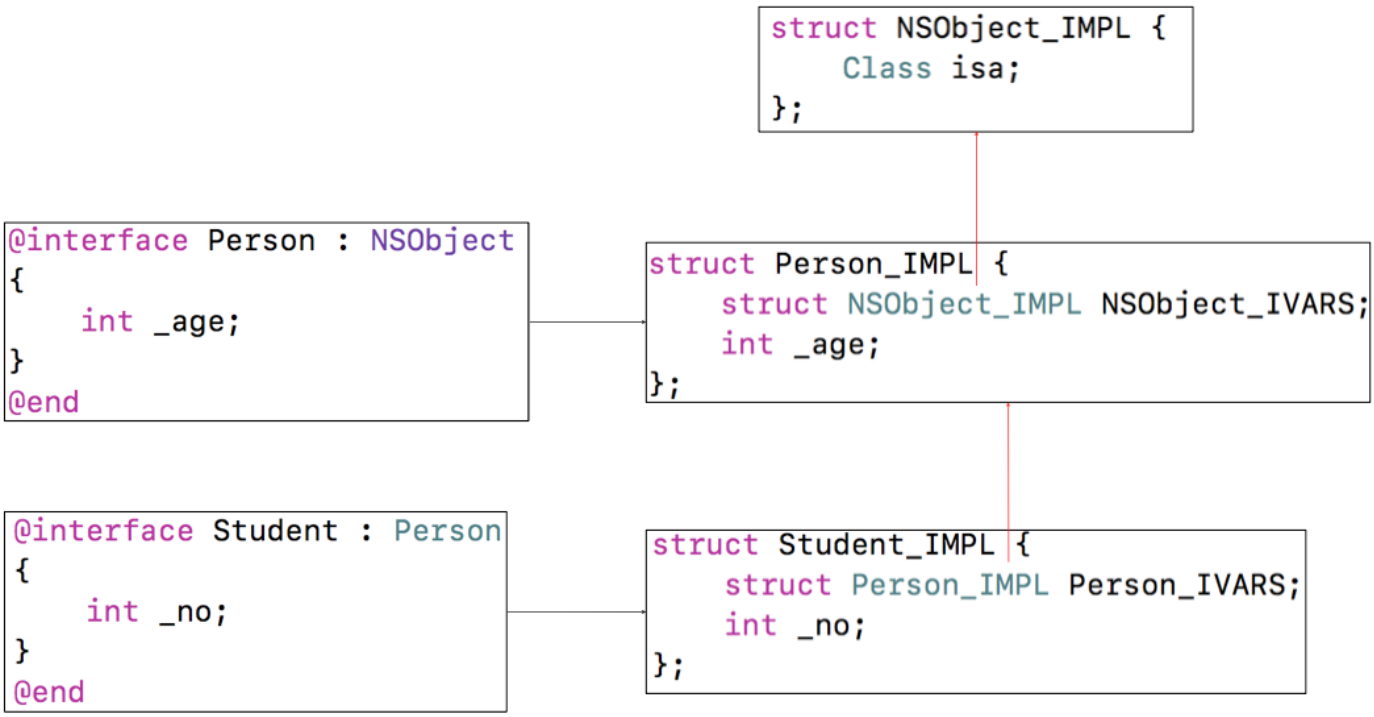
复制代码

```
/* Person */
@interface Person : NSObject
{
    int _age;
}
@end
```

```
@implementation Person
@end

/* Student */
@interface Student : Person
{
    int _no;
}
@end

@implementation Student
@end

int main(int argc, const char * argv[]) {
    @autoreleasepool {

        NSLog(@"%zd  %zd",
              class_getInstanceSize([Person class]),
              class_getInstanceSize([Student class])
              );
    }
    return 0;
}
```

> 这道面试题的实质是想问一个Person对象，一个Student对象分别占用多少内存空间?

我们依据上面的分析与发现，类对象实质上是以结构体的形式存储在内存中，画出真正的内存图例



我们发现只要是继承自NSObject的对象，那么底层结构体内一定有一个isa指针。 那么他们所占的内存空间是多少呢？单纯的将指针和成员变量所占的内存相加即可吗？上述代码实际打印的内容是16 16，也就是说，person对象和student对象所占用的内存空间都为16个字节。 其实实际上person对象确实只使用了12个字节。但是因为内存对齐的原因。使person对象也占用16个字节。

> 编译器在给结构体开辟空间时，首先找到结构体中最宽的基本数据类型，然后寻找内存地址能是该基本数据类型的整倍的位置，作为结构体的首地址。将这个最宽的基本数据类型的大小作为对齐模数。 为结构体的一个成员开辟空间之前，编译器首先检查预开辟空间的首地址相对于结构体首地址的偏移是否是本成员的整数倍，若是，则存放本成员，反之，则在本成员和上一个成员之间填充一定的字节，以达到整数倍的要求，也就是将预开辟空间的首地址后移几个字节。

> 我们可以总结内存对齐为两个原则： **原则 1. 前面的地址必须是后面的地址正数倍,不是就补齐。 原则 2. 整个Struct的地址必须是最大字节的整数倍。**

通过上述内存对齐的原则我们来看，person对象的第一个地址要存放isa指针需要8个字节，第二个地址要存放_age成员变量需要4个字节，根据原则一，8是4的整数倍，符合原则一，不需要补齐。然后检查原则2，目前person对象共占据12个字节的内存，不是最大字节数8个字节的整数倍，所以需要补齐4个字节，因此person对象就占用16个字节空间。

而对于student对象，我们知道sutdent对象中，包含person对象的结构体实现，和一个int类型的_no成员变量，同样isa指针8个字节，_age成员变量4个字节，_no成员变量4个字节，刚好满足原

则1和原则2，所以student对象占据的内存空间也是16个字节。

OC对象的分类

> 面试题：OC的类信息存放在哪里。 面试题：对象的isa指针指向哪里。

示例代码

复制代码

```
#import <Foundation/Foundation.h>
#import <objc/runtime.h>

/* Person */
@interface Person : NSObject <NSCopying>
{
    @public
    int _age;
}
@property (nonatomic, assign) int height;
- (void)personMethod;
+ (void)personClassMethod;
@end

@implementation Person
- (void)personMethod {}
+ (void)personClassMethod {}
@end

/* Student */
@interface Student : Person <NSCoding>
{
    @public
    int _no;
}
@property (nonatomic, assign) int score;
- (void)studentMethod;
+ (void)studentClassMethod;
@end

@implementation Student
- (void)studentMethod {}
+ (void)studentClassMethod {}
@end
```

```
int main(int argc, const char * argv[]) {
    @autoreleasepool {
        NSObject *object1 = [[NSObject alloc] init];
        NSObject *object2 = [[NSObject alloc] init];

        Student *stu = [[Student alloc] init];
        [Student load];

        Person *p1 = [[Person alloc] init];
        p1->_age = 10;
        [p1 personMethod];
        [Person personClassMethod];
        Person *p2 = [[Person alloc] init];
        p2->_age = 20;
    }
    return 0;
}
```

OC的类信息存放在哪里 OC对象主要可以分为三种

1. instance对象（实例对象）
2. class对象（类对象）
3. meta-class对象（元类对象）

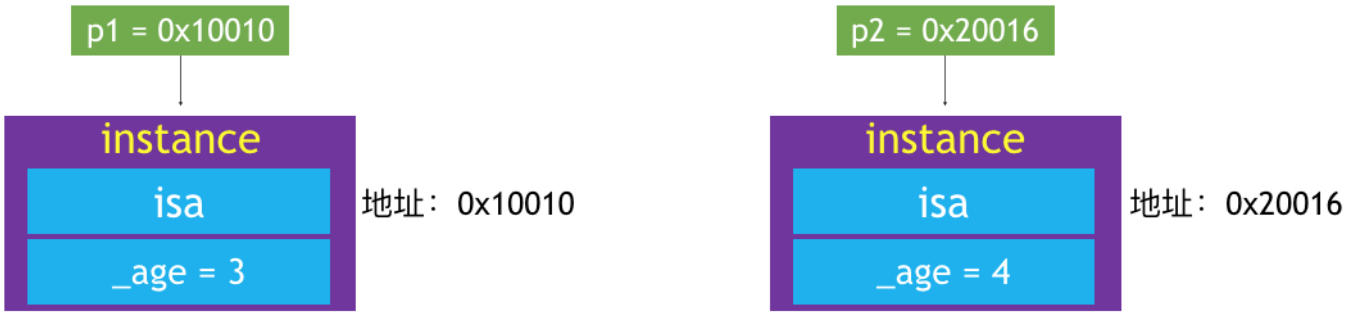
instance对象就是通过类alloc出来的对象，每次调用alloc都会产生新的instance对象

复制代码

```
NSObjcet *object1 = [[NSObjcet alloc] init];
NSObjcet *object2 = [[NSObjcet alloc] init];
```

object1和object2都是NSObject的instace对象（实例对象），但他们是不同的两个对象，并且分别占据着两块不同的内存。 instance对象在内存中存储的信息包括

1. isa指针
2. 其他成员变量



> 衍生问题：在上图实例对象中根本没有看到方法，那么实例对象的方法的代码放在什么地方呢？那么类的方法的信息，协议的信息，属性的信息都存放在什么地方呢？

class对象 我们通过class方法或runtime方法得到一个class对象。class对象也就是类对象

复制代码

```
Class objectClass1 = [object1 class];
Class objectClass2 = [object2 class];
Class objectClass3 = [NSObject class];

// runtime
Class objectClass4 = object_getClass(object1);
Class objectClass5 = object_getClass(object2);
NSLog(@"%p %p %p %p %p", objectClass1, objectClass2, objectClass3, objectClass4, objectC
```

每一个类在内存中有且只有一个class对象。可以通过打印内存地址证明

class对象在内存中存储的信息主要包括

1. isa指针
2. superclass指针
3. 类的属性信息（@property），类的成员变量信息（ivar）
4. 类的对象方法信息（instance method），类的协议信息（protocol）

成员变量的值时存储在实例对象中的，因为只有当我们创建实例对象的时候才为成员变赋值。但是成员变量叫什么名字，是什么类型，只需要有一份就可以了。所以存储在class对象中。

类方法放在那里？ **元类对象 meta-class**

```
复制代码
//runtime中传入类对象此时得到的就是元类对象
Class objectMetaClass = object_getClass([NSObject class]);
// 而调用类对象的class方法时得到还是类对象，无论调用多少次都是类对象
Class cls = [[NSObject class] class];
Class objectClass3 = [NSObject class];
class_isMetaClass(objectMetaClass) // 判断该对象是否为元类对象
NSLog(@"%p %p %p", objectMetaClass, objectClass3, cls); // 后面两个地址相同，说明多次调用clas
```

每个类在内存中有且只有一个meta-class对象。 meta-class对象和class对象的内存结构是一样的，但是用途不一样，在内存中存储的信息主要包括

1. isa指针
2. superclass指针
3. 类的类方法的信息（class method）

meta-class对象和class对象的内存结构是一样的，所以meta-class中也有类的属性信息，类的对象方法信息等成员变量，但是其中的值可能是空的。

对象的isa指针指向哪里

1. 当对象调用实例方法的时候，我们上面讲到，实例方法信息是存储在class类对象中的，那么要想找到实例方法，就必须找到class类对象，那么此时isa的作用就来了。

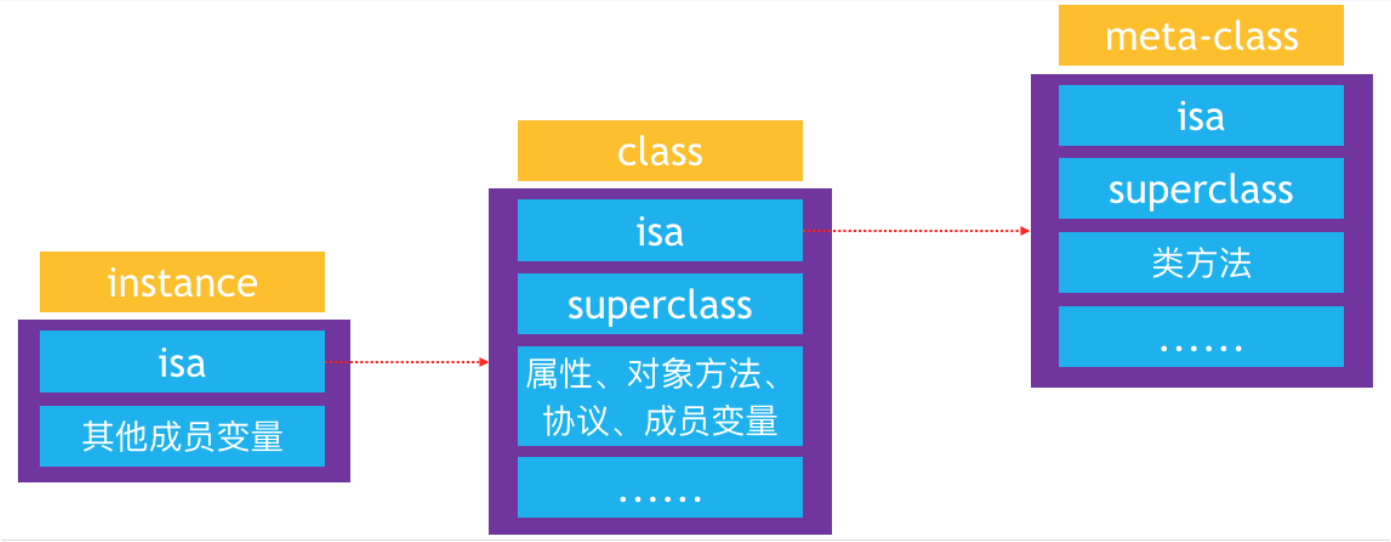
```
[stu studentMethod];
```

instance的isa指向class，当调用对象方法时，通过instance的isa找到class，最后找到对象方法的实现进行调用。

2. 当类对象调用类方法的时候，同上，类方法是存储在meta-class元类对象中的。那么要找到类方法，就需要找到meta-class元类对象，而class类对象的isa指针就指向元类对象

```
[Student studentClassMethod];
```

class的isa指向meta-class 当调用类方法时，通过class的isa找到meta-class，最后找到类方法的实现进行调用

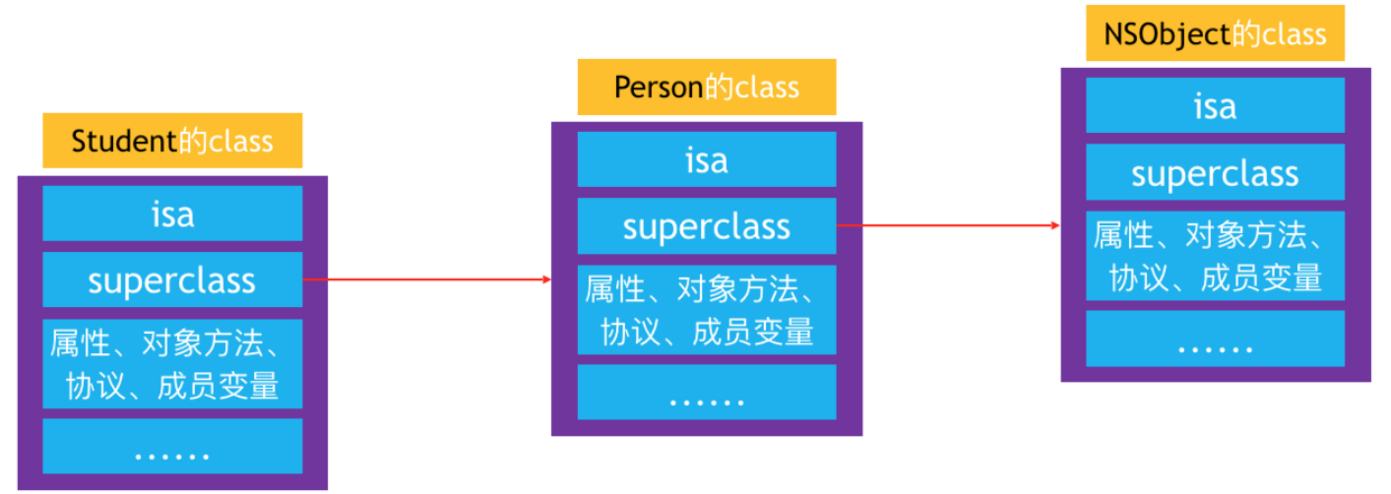


3. 当对象调用其父类对象方法的时候，又是怎么找到父类对象方法的呢？，此时就需要使用到class类对象superclass指针。

```
[stu personMethod];
[stu init];
```

复制代码

当Student的instance对象要调用Person的对象方法时，会先通过isa找到Student的class，然后通过superclass找到Person的class，最后找到对象方法的实现进行调用，同样如果Person发现自己没有响应的对象方法，又会通过Person的superclass指针找到NSObject的class对象，去寻找响应的方法

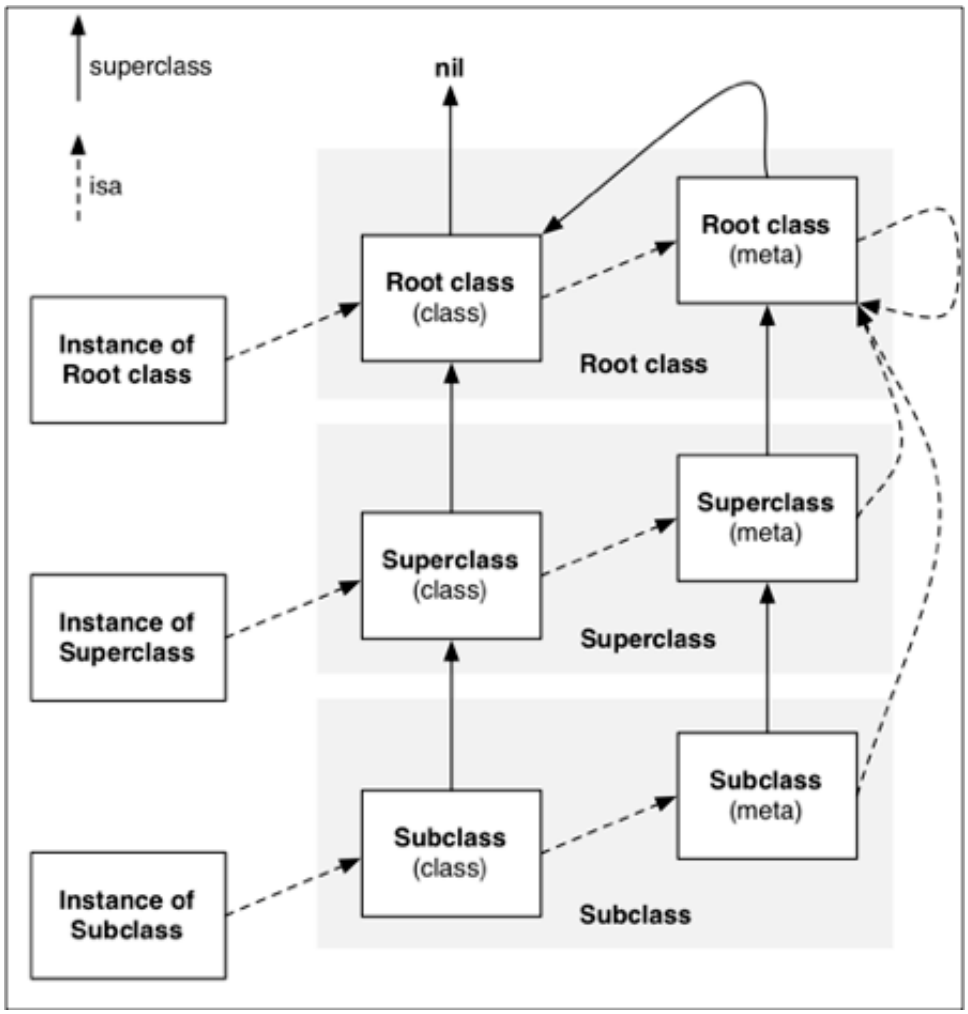


4. 当类对象调用父类的类方法时，就需要先通过isa指针找到meta-class，然后通过superclass去寻找响应的方法

```
[Student personClassMethod];
[Student load];
```

当Student的class要调用Person的类方法时，会先通过isa找到Student的meta-class，然后通过superclass找到Person的meta-class，最后找到类方法的实现进行调用

最后又是这张静定的isa指向图，经过上面的分析我们在来看这张图，就显得清晰明了很多。



> **对isa、superclass总结**
>
> 1. instance的isa指向class
> 2. class的isa指向meta-class
> 3. meta-class的isa指向基类的meta-class，基类的isa指向自己
> 4. class的superclass指向父类的class，如果没有父类，superclass指针为nil
> 5. meta-class的superclass指向父类的meta-class，基类的meta-class的superclass指向基类的class
> 6. instance调用对象方法的轨迹，isa找到class，方法不存在，就通过superclass找父类

> 7. class调用类方法的轨迹，isa找meta-class，方法不存在，就通过superclass找父类

如何证明isa指针的指向真的如上面所说?

我们通过如下代码证明：

```
复制代码
NSObject *object = [[NSObject alloc] init];
Class objectClass = [NSObject class];
Class objectMetaClass = object_getClass([NSObject class]);

NSLog(@"%p %p %p", object, objectClass, objectMetaClass);
```

打断点并通过控制台打印相应对象的isa指针

```
(lldb) p/x object->isa
(Class) $2 = 0x001dffff96537141 NSObject
(lldb) p/x objectClass
(Class) $3 = 0x00007fff96537140 NSObject
(lldb)
```

我们发现object->isa与objectClass的地址不同，这是因为从64bit开始，isa需要进行一次位运算，才能计算出真实地址。而位运算的值我们可以通过下载objc源代码找到。

```
# if __arm64__
#   define ISA_MASK        0x0000000ffffffff8ULL
# elif __x86_64__
#   define ISA_MASK        0x00007ffffffffff8ULL
# endif
```

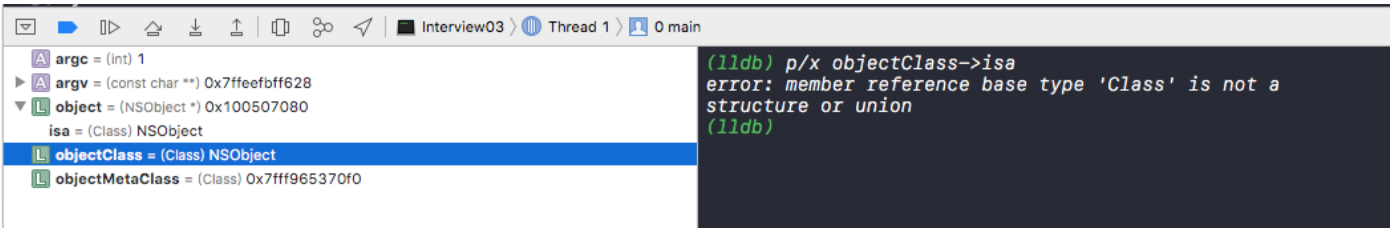
我们通过位运算进行验证。

```
(lldb) p/x object->isa
(Class) $2 = 0x001dffff96537141 NSObject
(lldb) p/x objectClass
(Class) $3 = 0x00007fff96537140 NSObject
(lldb) p/x 0x00007ffffffffff8 & 0x001dffff96537141
(long) $4 = 0x00007fff96537140
(lldb)
```

我们发现，object-isa指针地址0x001dffff96537141经过同0x00007ffffffffff8位运算，得出

objectClass的地址0x00007fff96537140

接着我们来验证class对象的isa指针是否同样需要位运算计算出meta-class对象的地址。 当我们以同样的方式打印objectClass->isa指针时，发现无法打印



同时也发现左边objectClass对象中并没有isa指针。我们来到Class内部看一下

复制代码

```
typedef struct objc_class *Class;

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;
    struct objc_method_list * _Nullable * _Nullable methodLists                    OBJC2
    struct objc_cache * _Nonnull cache                          OBJC2_UNAVAILABLE;
    struct objc_protocol_list * _Nullable protocols             OBJC2_UNAVAILABLE;
#endif

} OBJC2_UNAVAILABLE;
/* Use `Class` instead of `struct objc_class *` */
```

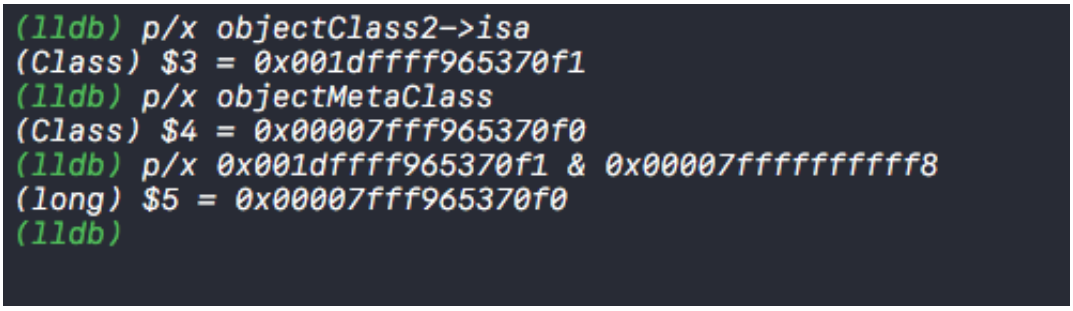
相信了解过isa指针的同学对objc_class结构体内的内容很熟悉了，今天这里不深入研究，我们只看第一个对象是一个isa指针，为了拿到isa指针的地址，我们自己创建一个同样的结构体并通过强制转化拿到isa指针。

复制代码

```
struct xx_cc_objc_class{
    Class isa;
};
```

```
Class objectClass = [NSObject class];
struct xx_cc_objc_class *objectClass2 = (__bridge struct xx_cc_objc_class *)(objectClass
```

此时我们重新验证一下



确实，objectClass2的isa指针经过位运算之后的地址是meta-class的地址。

本文面试题总结：

1. 一个NSObject对象占用多少内存？ 答：一个指针变量所占用的大小（64bit占8个字节，32bit占4个字节）

2. 对象的isa指针指向哪里？ 答：instance对象的isa指针指向class对象，class对象的isa指针指向meta-class对象，meta-class对象的isa指针指向基类的meta-class对象，基类自己的isa指针也指向自己。

3. OC的类信息存放在哪里？ 答：成员变量的具体值存放在instance对象。对象方法，协议，属性，成员变量信息存放在class对象。类方法信息存放在meta-class对象。

文中如果有不对的地方欢迎指出。我是xx_cc，一只长大很久但还没有二够的家伙。需要视频一起探讨学习的coder可以加我Q：2336684744

> **关注下面的标签，发现更多相似文章**

iOS 命令行 C++ Xcode

xx_cc Lv3 iOS 开发 @ 无
发布了 12 篇专栏 · 获得点赞 2,165 · 获得阅读 58,839

关注

评论

您需要绑定手机号后才可在掘金社区内发布内容。

kevislin

你好，我有个疑问是关于id类型的，id按照定义是指向objc_obejct的指针，
那么 id objc =[[UIButton alloc] init] 这句话等式右边返回的是一个对象结构体的地址，但是objc指向的是另外一个不同大小的结构体，请问这是怎样兼容的?

16天前 回复

RunTitan Lv2 iOS开发工程师

你好, 相关视频方便分享一下吗?感谢

1月前 回复

经天纬地 iOS 开发

然后oc的对象本质是什么，一句话总结。

7月前 回复

哆来

p/x object->isa
p/x objectClass
xcode9.4 xcode10 已经不用位运算了，亲测

8月前 回复

Yaanco iOS

回复 哆来: 我打印的跟作者的是一样的呀，xcode 10.1

5月前 回复