iOS底层原理总结 - 探寻Runtime本质（四）

super的本质

首先来看一道面试题。 下列代码中Person继承自NSObject，Student继承自Person，写出下列代码输出内容。

```
#import "Student.h"
@implementation Student
- (instancetype)init
{
    if (self = [super init]) {
        NSLog(@"[self class] = %@", [self class]);
        NSLog(@"[self superclass] = %@", [self superclass]);
        NSLog(@"-----------------");
        NSLog(@"[super class] = %@", [super class]);
        NSLog(@"[super superclass] = %@", [super superclass]);

    }
    return self;
}
@end
```

直接来看一下打印内容

```
Runtime-super[6601:1536402] [self class] = Student
Runtime-super[6601:1536402] [self superclass] = Person
Runtime-super[6601:1536402] -----------------
Runtime-super[6601:1536402] [super class] = Student
Runtime-super[6601:1536402] [super superclass] = Person
```

上述代码中可以发现无论是 self还是super 调用 class或superclass 的结果 都是相同的。

我们通过一段代码来看一下super底层实现，为Person类提供run方法，Student类中重写run方法，方法内部调用[super run];，将Student.m转化为c++代码查看其底层实现。

```
- (void) run
```

```
{
    [super run];
    NSLog(@"Student...");
}
```

上述代码转化为c++代码:

```
static void _I_Student_run(Student * self, SEL _cmd) {

    ((void (*)(__rw_objc_super *, SEL))(void *)objc_msgSendSuper)
((__rw_objc_super){(id)self,
(id)class_getSuperclass(objc_getClass("Student"))},
sel_registerName("run"));


    NSLog((NSString
*)&__NSConstantStringImpl__var_folders_jm_dztwxsdn7bvbz__xj2vlp8980
000gn_T_Student_e677aa_mi_0);
}
```

通过上述源码可以发现:

> [super run];转化为底层源码内部其实调用的是objc_msgSendSuper函数。

> objc_msgSendSuper函数内传递了两个参数:__rw_objc_super结构体和sel_registerName("run")方法名。

> __rw_objc_super结构体内传入的参数是self和class_getSuperclass(objc_getClass("Student")),也就是Student的父类Person,

首先我们找到objc_msgSendSuper函数查看内部结构

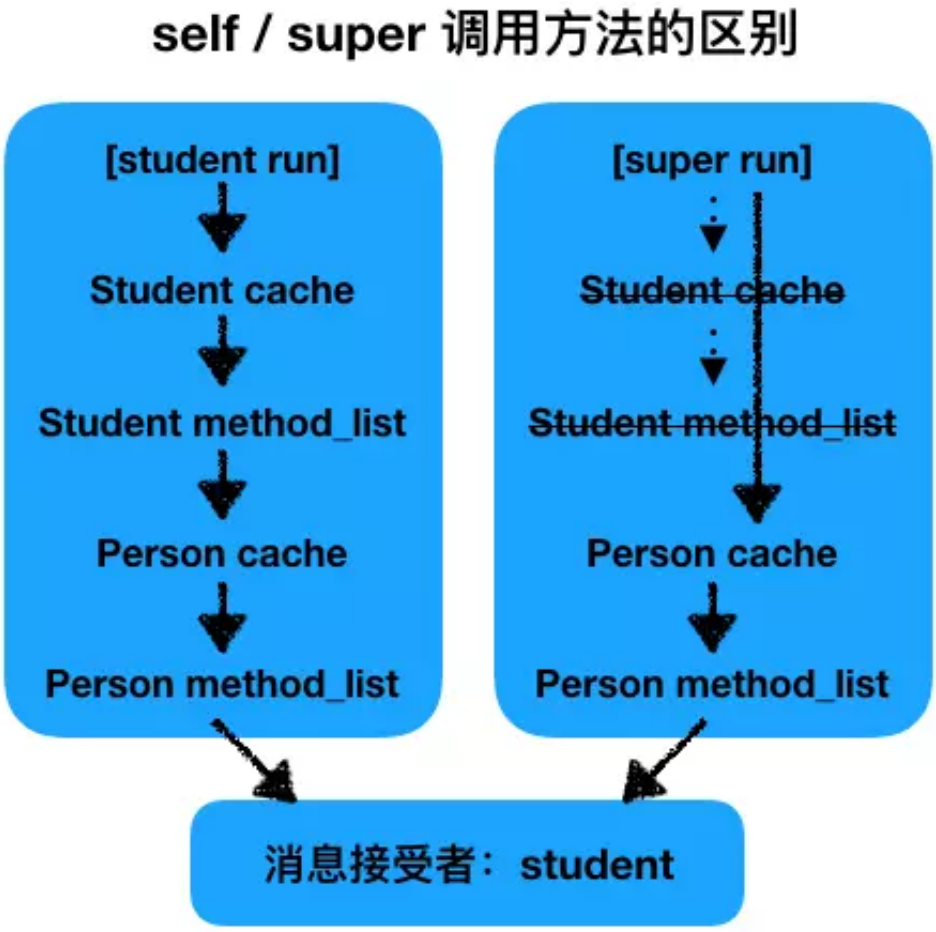
```
OBJC_EXPORT id _Nullable
objc_msgSendSuper(struct objc_super * _Nonnull super, SEL _Nonnull
op, ...)
    OBJC_AVAILABLE(10.0, 2.0, 9.0, 1.0, 2.0);
```

可以发现objc_msgSendSuper中传入的结构体是objc_super，我们来到objc_super内部查看其内部结构。 我们通过源码查找objc_super结构体查看其内部结构。

```
// 精简后的objc_super结构体
struct objc_super {
    __unsafe_unretained _Nonnull id receiver; // 消息接受者
    __unsafe_unretained _Nonnull Class super_class; // 消息接受者的父
类
    /* super_class is the first class to search */
    // 父类是第一个开始查找的类
};
```

从objc_super结构体中可以发现receiver消息接受者仍然为self，superclass仅仅是用来告知消息查找从哪一个类开始。从父类的类对象开始去查找。

我们通过一张图看一下其中的区别。



从上图中我们知道 **super调用方法的消息接受者receiver仍然是self，只是从父类的类对象开始去查找方法。**

那么此时重新回到面试题，我们知道class的底层实现如下面代码所示

```
+ (Class)class {
    return self;
}

- (Class)class {
    return object_getClass(self);
}
```

class内部实现是根据消息接受者返回其对应的类对象，最终会找到基类的方法列表中，而self和super的区别仅仅是self从本类类对象开始查找方法，super从父类类对象开始查找方法，因此最终得到的结果都是相同的。

另外我们在回到run方法内部，很明显可以发现，如果super不是从父类开始查找方法，从本类查找方法的话，就调用方法本身造成循环调用方法而crash。

同理superclass底层实现同class类似，其底层实现代码如下入所示:

```
+ (Class)superclass {
    return self->superclass;
}

- (Class)superclass {
    return [self class]->superclass;
}
```

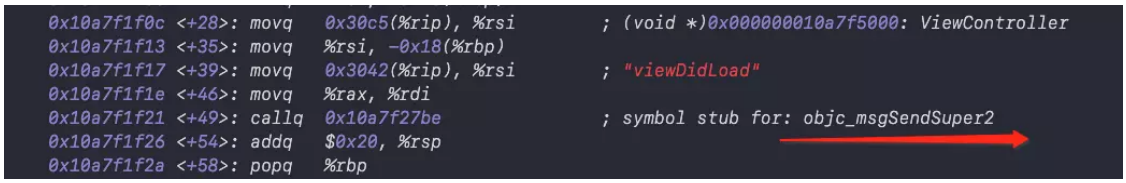
objc_msgSendSuper2函数

上述OC代码转化为c++代码并不能说明super底层调用函数就一定objc_msgSendSuper。

其实super底层真正调用的函数是objc_msgSendSuper2函数, 我们可以通过查看super调用方法转化为汇编代码来验证这一说法

```
- (void)viewDidLoad {
    [super viewDidLoad];
}
```

通过断点查看其汇编调用栈



上图中可以发现super底层其实调用的是objc_msgSendSuper2函数，我们来到源码中查找一下objc_msgSendSuper2函数的底层实现，我们可以在汇编文件中找到其相关底层实现。

```
ENTRY _objc_msgSendSuper2
UNWIND _objc_msgSendSuper2, NoFrame
MESSENGER_START

ldp x0, x16, [x0]       // x0 = real receiver, x16 = class
ldr x16, [x16, #SUPERCLASS] // x16 = class->superclass
CacheLookup NORMAL

END_ENTRY _objc_msgSendSuper2
```

通过上面汇编代码我们可以发现，其实底层是在函数内部调用的class->superclass获取父类，并不是我们上面分析的直接传入的就是父类对象。

其实_objc_msgSendSuper2内传入的结构体为objc_super2

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

我们可以发现objc_super2中除了消息接受者receiver，另一个成员变量current_class也就是当前类对象。 与我们上面分析的不同_objc_msgSendSuper2函数内其实传入的是当前类对象，然后在函数内部获取当前类对象的父类，并且从父类开始查找方法。 我们也可以通过代码验证上述结构体内成员变量究竟是当前类对象还是父类对象。下文中我们会通过另外一道面试题验证。

isKindOfClass 与 isMemberOfClass

首先看一下isKindOfClass isKindOfClass对象方法底层实现

```
- (BOOL)isMemberOfClass:(Class)cls {
    // 直接获取实例类对象并判断是否等于传入的类对象
    return [self class] == cls;
}

- (BOOL)isKindOfClass:(Class)cls {
    // 向上查询，如果找到父类对象等于传入的类对象则返回YES
    // 直到基类还不相等则返回NO
    for (Class tcls = [self class]; tcls; tcls = tcls->superclass)
{
        if (tcls == cls) return YES;
    }
    return NO;
}
```

isKindOfClass isKindOfClass类方法底层实现:

```
// 判断元类对象是否等于传入的元类元类对象
// 此时self是类对象 object_getClass((id)self)就是元类
+ (BOOL)isMemberOfClass:(Class)cls {
    return object_getClass((id)self) == cls;
}

// 向上查找，判断元类对象是否等于传入的元类对象
// 如果找到基类还不相等则返回NO
// 注意：这里会找到基类
+ (BOOL)isKindOfClass:(Class)cls {
    for (Class tcls = object_getClass((id)self); tcls; tcls = tcls-
>superclass) {
        if (tcls == cls) return YES;
    }
    return NO;
}
```

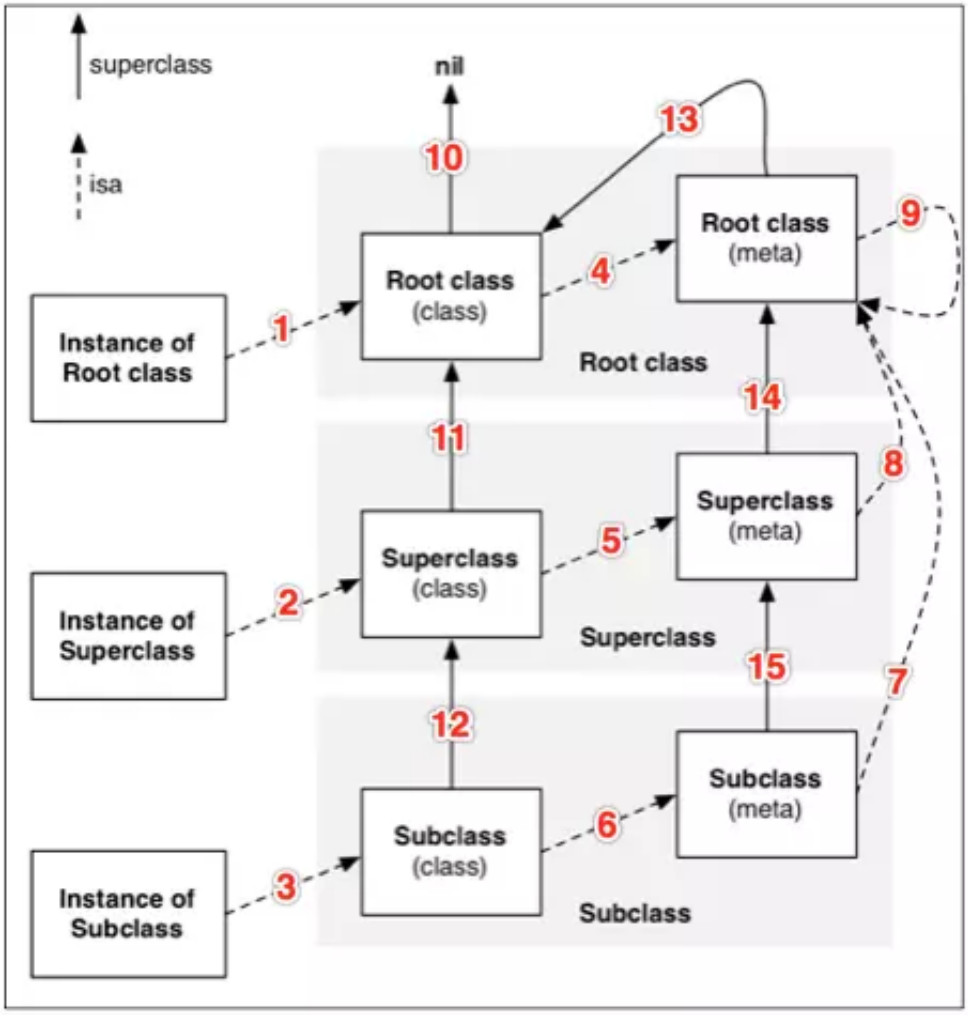
通过上述源码分析我们可以知道。 **isMemberOfClass 判断左边是否刚好等于右边类型。 isKindOfClass 判断左边或者左边类型的父类是否刚好等于右边类型。 注意：类方法内部是获取其元类对象进行比较**

我们查看以下代码

```
NSLog(@"%d",[Person isKindOfClass: [Person class]]);
NSLog(@"%d",[Person isKindOfClass: object_getClass([Person
```

```
class]]]);
NSLog(@"%d",[Person isKindOfClass: [NSObject class]]);

// 输出内容
Runtime-super[46993:5195901] 0
Runtime-super[46993:5195901] 1
Runtime-super[46993:5195901] 1
```

分析上述输出内容： 第一个 0：上面提到过类方法是获取self的元类对象与传入的参数进行比较，但是第一行我们传入的是类对象，因此返回NO。 第二个 1：同上，此时我们传入Person元类对象，此时返回YES。验证上述说法 第三个 1：我们发现此时传入的是NSObject类对象并不是元类对象，但是返回的值却是YES。 原因是基元类的superclass指针是指向基类对象的。如下图13号线



那么Person元类通过superclass指针一直找到基元类，还是不相等，此时再次通过superclass指针来到基类，那么此时发现相等就会返回YES了。

复习

通过一道面试题对之前学习的知识进行复习。 问：以下代码是否可以执行成功，如果可以，打印结果是什么。

```
// Person.h
#import <Foundation/Foundation.h>
@interface Person : NSObject
@property (nonatomic, strong) NSString *name;
- (void)test;
@end

// Person.m
#import "Person.h"
@implementation Person
- (void)test
{
    NSLog(@"test print name is : %@", self.name);
}
@end

// ViewController.m
@implementation ViewController
- (void)viewDidLoad {
    [super viewDidLoad];

    id cls = [Person class];
    void *obj = &cls;
    [(__bridge id)obj test];

    Person *person = [[Person alloc] init];
    [person test];
}
```

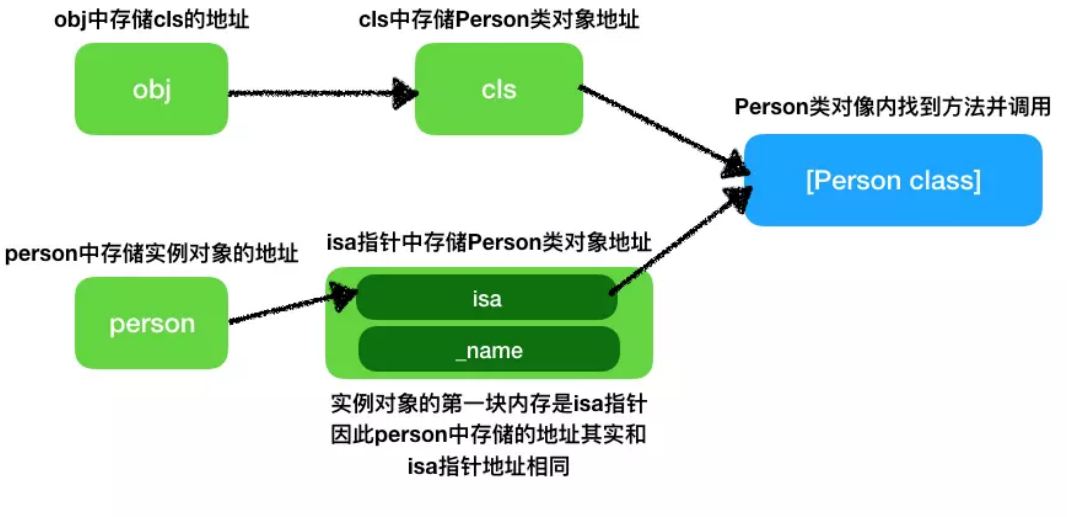
这道面试题确实很无厘头的一道题，日常工作中没有人这样写代码，但是需要解答这道题需要很完备的底层知识，我们通过这道题来复习一下，首先看一下打印结果。

```
Runtime面试题[15842:2579705] test print name is : <ViewController:
0x7f95514077a0>
Runtime面试题[15842:2579705] test print name is : (null)
```

通过上述打印结果我们可以看出，是可以正常运行并打印的，说明obj可以正常调用

test方法，但是我们发现打印self.name的内容却是<ViewController: 0x7f95514077a0>。下面person实例调用test不做过多解释了，主要用来和上面方法调用做对比。

为什么会是这样的结果呢？首先通过一张图看一下两种调用方法的内存信息。



通过上图我们可以发现两种方法调用方式很相近。那么obj为什么可以正常调用方法?

obj为什么可以正常调用方法

首先通过之前的学习我们知道，person调用方法时首先通过isa指针找到类对象进而查找方法并进行调用。

而person实例对象内实际上是取最前面8个字节空间也就是isa并通过计算得出类对象地址。

而通过上图我们可以发现，obj在调用test方法时，也会通过其内存地址找到cls，而cls中取出最前面8个字节空间其内部存储的刚好是Person类对象地址。因此obj是可以正常调用方法的。

为什么self.name打印内容为ViewController对象

问题出在[super viewDidLoad];这段代码中，通过上述对super本质的分析我们知道，super内部调用objc_msgSendSuper2函数。

我们知道objc_msgSendSuper2函数内部会传入两个参数，objc_super2结构体和

SEL，并且objc_super2结构体内有两个成员变量消息接受者和其父类。

```
struct objc_super2 {
    id receiver; // 消息接受者
    Class current_class; // 当前类
};
};
```

通过以上分析我们可以得知[super viewDidLoad];内部objc_super2结构体内存储如下所示

```
struct objc_super = {
    self,
    [ViewController Class]
};
```

那么objc_msgSendSuper2函数调用之前，会先创建局部变量objc_super2结构体用于为objc_msgSendSuper2函数传递的参数。

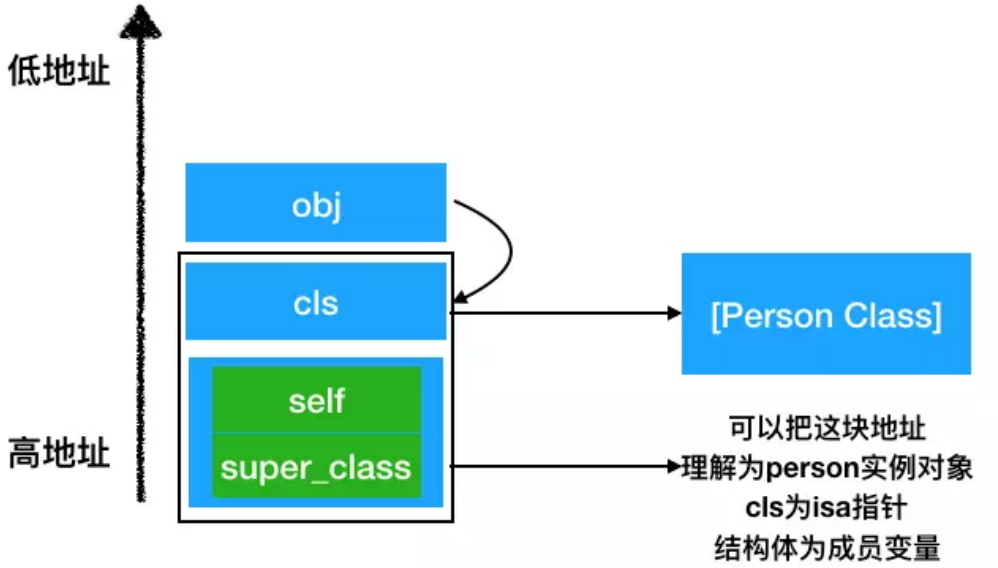
局部变量由高地址向低地址分配在栈空间

我们知道局部变量是存储在栈空间内的，并且是由高地址向低地址有序存储。 我们通过一段代码验证一下。

```
long long a = 1;
long long b = 2;
long long c = 3;
NSLog(@"%p %p %p", &a,&b,&c);
// 打印内容
0x7ffee9774958 0x7ffee9774950 0x7ffee9774948
```

通过上述代码打印内容，我们可以验证**局部变量在栈空间内是由高地址向低地址连续存储的。**

那么我们回到面试题中，通过上述分析我们知道，此时代码中包含局部变量以此为objc_super2 结构体、cls、obj。通过一张图展示一下这些局部变量存储结构。



上面我们知道当person实例对象调用方法的时候，

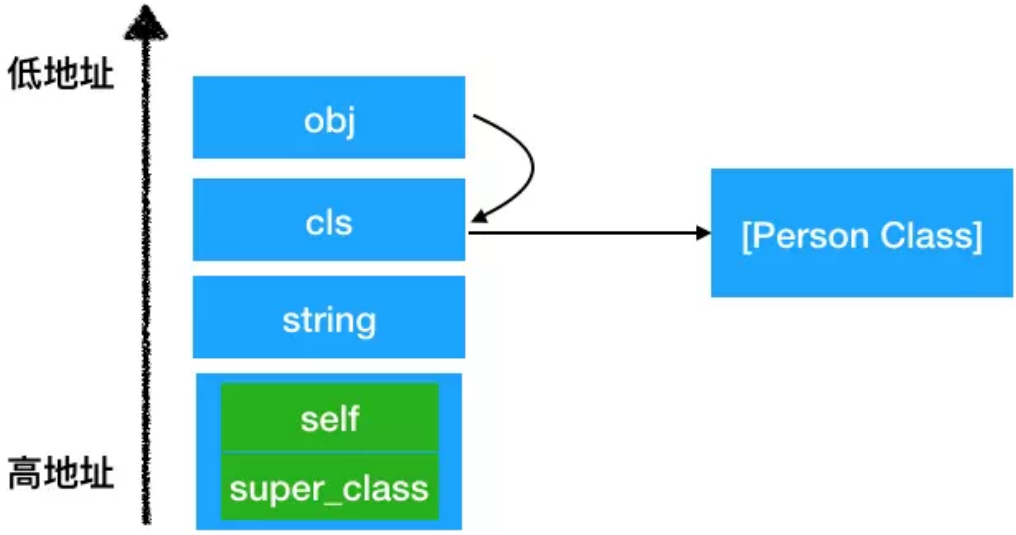
- 会取实例变量前8个字节空间也就是isa来找到类对象地址。那么当访问实例变量的时候，就跳过isa的8个字节空间往下面去找实例变量。
- 那么当obj在调用test方法的时候同样找到cls中取出前8个字节，也就是Person类对象的内存地址，那么当访问实例变量_name的时候，会继续向高地址内存空间查找，此时就会找到objc_super结构体，从中取出8个字节空间也就是self，因此此时访问到的self.name就是ViewController对象。
- 当访问成员变量_name的时候，test函数中的self也就是方法调用者其实是obj，那么self.name就是通过obj去找_name，跳过cls的8个指针，在取8个指针此时自然获取到ViewController对象。
- 因此上述代码中cls就相当于isa，isa下面的8个字节空间就相当于_name成员变量。因此成员变量_name的访问到的值就是cls地址后向高地址位取8个字节地址空间存储的值。

为了验证上述说法，我们做一个实验，在cls后高地址中添加一个string，那么此时cls下面的高地址位就是string。以下示例代码

```
- (void)viewDidLoad {
    [super viewDidLoad];

    NSString *string = @"string";
```

```
    id cls = [Person class];
    void *obj = &cls;
    [(__bridge id)obj test];

    Person *person = [[Person alloc] init];
    [person test];
}
```

此时的局部变量内存结构如下图所示



此时在访问_name成员变量的时候，越过cls内存往高地址找就会来到string，此时拿到的成员变量就是string了。 我们来看一下打印内容

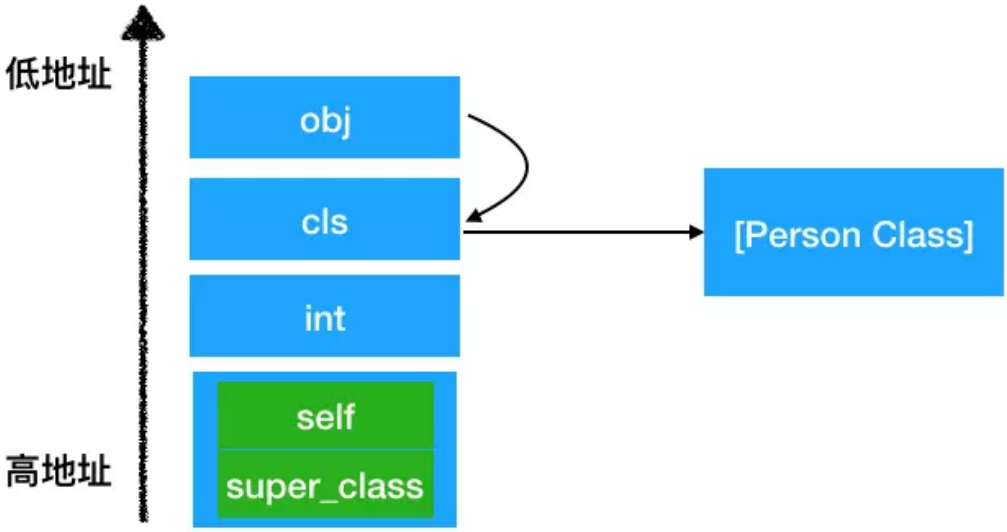
```
Runtime面试题[16887:2829028] test print name is : string
Runtime面试题[16887:2829028] test print name is : (null)
```

再通过一段代码使用int数据进行试验

```
- (void)viewDidLoad {
    [super viewDidLoad];

    int a = 3;

    id cls = [Person class];
```

```
    void *obj = &cls;
    [(__bridge id)obj test];

    Person *person = [[Person alloc] init];
    [person test];
}
// 程序crash，坏地址访问
```

我们发现程序因为坏地址访问而crash，此时局部变量内存结构如下图所示



当需要访问_name成员变量的时候，会在cls后高地址为查找8位的字节空间，而我们知道int占4位字节，那么此时8位的内存空间同时占据int数据及objc_super结构体内，因此就会造成坏地址访问而crash。

我们添加新的成员变量进行访问

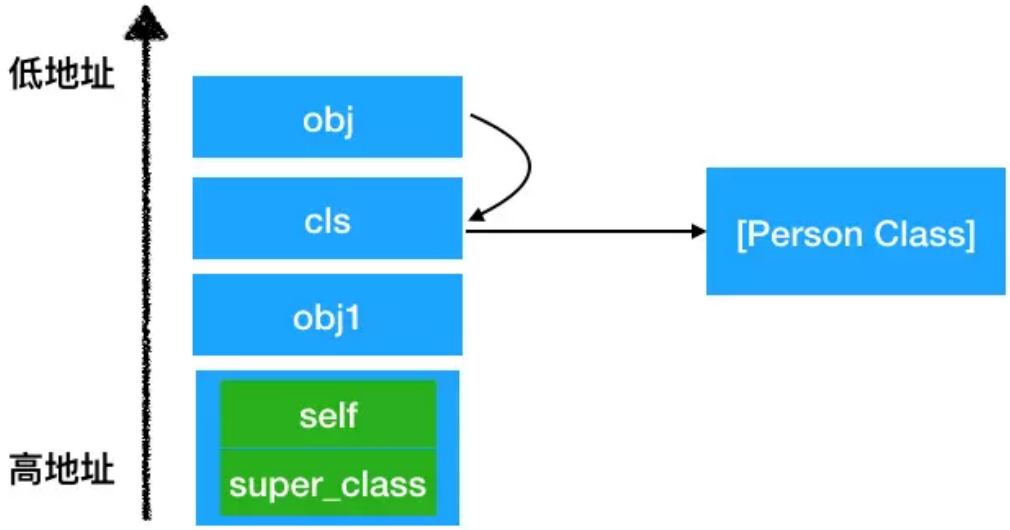
```
// Person.h
#import <Foundation/Foundation.h>
@interface Person : NSObject
@property (nonatomic, strong) NSString *name;
@property (nonatomic, strong) NSString *nickName;
- (void)test;
@end
------------
// Person.m
#import "Person.h"
@implementation Person
```

```
- (void)test
{
    NSLog(@"test print name is : %@", self.nickName);
}
@end
--------
//  ViewController.m
- (void)viewDidLoad {
    [super viewDidLoad];

    NSObject *obj1 = [[NSObject alloc] init];

    id cls = [Person class];
    void *obj = &cls;
    [(__bridge id)obj test];

    Person *person = [[Person alloc] init];
    [person test];
}
```

我们看一下打印内容

```
// 打印内容
// Runtime面试题[17272:2914887] test print name is :
<ViewController: 0x7ffc6010af50>
// Runtime面试题[17272:2914887] test print name is : (null)
```

可以发现此时打印的仍然是ViewController对象，我们先来看一下其局部变量内存结构



首先通过obj找到cls，cls找到类对象进行方法调用，此时在访问nickName时，obj查找成员变量，首先跳过8个字节的cls，之后跳过name所占的8个字节空间，最终再取8个字节空间取出其中的值作为成员变量的值，那么此时也就是self了。

总结：这道面试题虽然很无厘头，让人感觉无从下手但是考察的内容非常多。

1. super的底层本质为调用objc_msgSendSuper2函数，传入objc_super2结构体，结构体内部存储消息接受者和当前类，用来告知系统方法查找从父类开始。
2. 局部变量分配在栈空间，并且从高地址向低地址连续分配。先创建的局部变量分配在高地址，后续创建的局部变量连续分配在较低地址。
3. 方法调用的消息机制，通过isa指针找到类对象进行消息发送。
4. 指针存储的是实例变量的首字节地址，上述例子中person指针存储的其实就是实例变量内部的isa指针的地址。
5. 访问成员变量的本质，找到成员变量的地址，按照成员变量所占的字节数，取出地址中存储的成员变量的值。

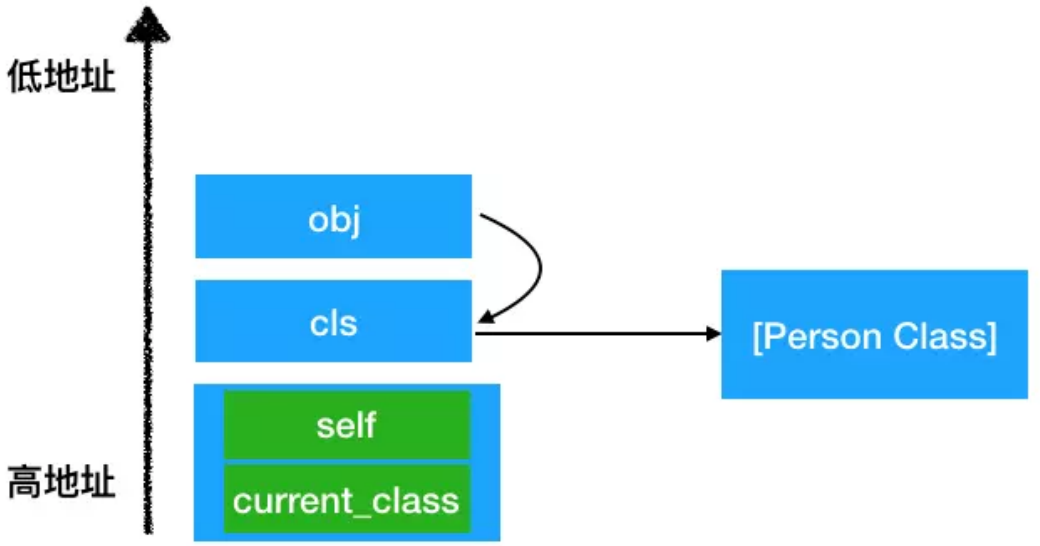
验证objc_msgSendSuper2内传入的结构体参数

我们使用以下代码来验证上文中遗留的问题

```
- (void)viewDidLoad {
```

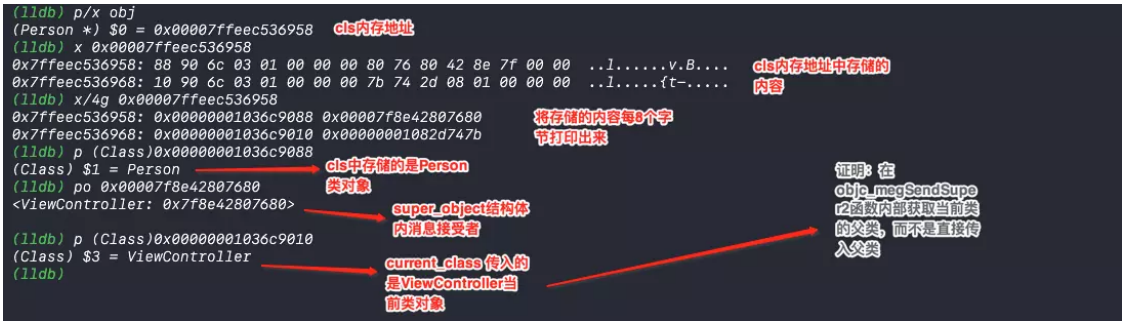
```
    [super viewDidLoad];
    id cls = [Person class];
    void *obj = &cls;
    [(__bridge id)obj test];
}
```

上述代码的局部变量内存结构我们之前已经分析过了，真正的内存结构应该如下图所示



通过上面对面试题的分析， 我们现在想要验证objc_msgSendSuper2函数内传入的结构体参数，只需要拿到cls的地址， 然后向后移8个地址就可以获取到objc_super结构体内的self， 再向后移8个地址就是current_class的内存地址。 通过打印current_class的内容，就可以知道传入objc_msgSendSuper2函数内部的是当前类对象还是父类对象了。

我们来证明他是UIViewController 还是ViewController即可



Runtime API

首先我们通过来看一段代码，后续Runtime API的使用均基于此代码。

```
// Person类继承自NSObject，包含run方法
@interface Person : NSObject
@property (nonatomic, strong) NSString *name;
- (void)run;
@end

#import "Person.h"
@implementation Person
- (void)run
{
    NSLog(@"%s",__func__);
}
@end

// Car类继承自NSObejct，包含run方法
#import "Car.h"
@implementation Car
- (void)run
{
    NSLog(@"%s",__func__);
}
@end
```

类相关API

```
1. 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)

2. 注册一个类（要在类注册之前添加成员变量）
void objc_registerClassPair(Class cls)

3. 销毁一个类
void objc_disposeClassPair(Class cls)

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

```objc
int main(int argc, const char * argv[]) {
    @autoreleasepool {
        // 创建类 superclass:继承自哪个类 name:类名 size_t:格外的大小，创建类是否需要扩充空间
        // 返回一个类对象
        Class newClass = objc_allocateClassPair([NSObject class], "Student", 0);

        // 添加成员变量
        // cls:添加成员变量的类 name:成员变量的名字 size:占据多少字节 alignment:内存对齐，最好写1 types:类型，int类型就是@encode(int) 也就是i
        class_addIvar(newClass, "_age", 4, 1, @encode(int));
        class_addIvar(newClass, "_height", 4, 1, @encode(float));

        // 添加方法
        class_addMethod(newClass, @selector(run), (IMP)run, "v@:");

        // 注册类
        objc_registerClassPair(newClass);

        // 创建实例对象
        id student = [[newClass alloc] init];

        // 通过KVC访问
        [student setValue:@10 forKey:@"_age"];
        [student setValue:@180.5 forKey:@"_height"];

        // 获取成员变量
        NSLog(@"_age = %@ , _height = %@",[student valueForKey:@"_age"], [student valueForKey:@"_height"]);

        // 获取类的占用空间
        NSLog(@"类对象占用空间%zd", class_getInstanceSize(newClass));

        // 调用动态添加的方法
        [student run];

    }
    return 0;
}

// 打印内容
// Runtime应用[25605:4723961] _age = 10 , _height = 180.5
// Runtime应用[25605:4723961] 类对象占用空间16
// Runtime应用[25605:4723961] <Student: 0x10072e420> - run
```

注意

类一旦注册完毕，就相当于类对象和元类对象里面的结构就已经创建好了。
因此必须在注册类之前，添加成员变量。方法可以在注册之后再添加，因为方法是可以动态添加的。
创建的类如果不需要使用了 ，需要释放类。

4. 获取isa指向的Class，如果将类对象传入获取的就是元类对象，如果是实例对象则为类对象
Class object_getClass(id obj)

```
int main(int argc, const char * argv[]) {
    @autoreleasepool {
        Person *person = [[Person alloc] init];
        NSLog(@"%p,%p,%p",object_getClass(person), [Person class],
              object_getClass([Person class]));
    }
    return 0;
}
// 打印内容
Runtime应用[21115:3807804] 0x100001298,0x100001298,0x100001270
```

5. 设置isa指向的Class，可以动态的修改类型。例如修改了person对象的类型，也就是说修改了person对象的isa指针的指向，中途让对象去调用其他类的同名方法。
Class object_setClass(id obj, Class cls)

```
int main(int argc, const char * argv[]) {
    @autoreleasepool {
        Person *person = [[Person alloc] init];
        [person run];

        object_setClass(person, [Car class]);
        [person run];
    }
    return 0;
}
// 打印内容
Runtime应用[21147:3815155] -[Person run]
Runtime应用[21147:3815155] -[Car run]
```

最终其实调用了car的**run**方法

6. 用于判断一个OC对象是否为Class
BOOL **object**_isClass(id obj)

```
// 判断OC对象是实例对象还是类对象
NSLog(@"%d",object_isClass(person)); // 0
NSLog(@"%d",object_isClass([person class])); // 1
NSLog(@"%d",object_isClass(object_getClass([person class]))); // 1
// 元类对象也是特殊的类对象
```

```
7. 判断一个Class是否为元类
BOOL class_isMetaClass(Class cls)
8. 获取类对象父类
Class class_getSuperclass(Class cls)
```

成员变量相关API

```
1. 获取一个实例变量信息，描述信息变量的名字，占用多少字节等
Ivar class_getInstanceVariable(Class cls, const char *name)

2. 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)

3. 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)

4. 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)

5. 获取成员变量的相关信息，传入成员变量信息，返回C语言字符串
const char *ivar_getName(Ivar v)
6. 获取成员变量的编码，types
const char *ivar_getTypeEncoding(Ivar v)

示例：
int main(int argc, const char * argv[]) {
    @autoreleasepool {
        // 获取成员变量的信息
        Ivar nameIvar = class_getInstanceVariable([Person class],
"_name");
        // 获取成员变量的名字和编码
        NSLog(@"%s, %s", ivar_getName(nameIvar),
ivar_getTypeEncoding(nameIvar));

        Person *person = [[Person alloc] init];
```

```
        // 设置和获取成员变量的值
        object_setIvar(person, nameIvar, @"xx_cc");
        // 获取成员变量的值
        object_getIvar(person, nameIvar);
        NSLog(@"%@", object_getIvar(person, nameIvar));
        NSLog(@"%@", person.name);

        // 拷贝实例变量列表
        unsigned int count ;
        Ivar *ivars = class_copyIvarList([Person class], &count);

        for (int i = 0; i < count; i ++) {
            // 取出成员变量
            Ivar ivar = ivars[i];
            NSLog(@"%s, %s", ivar_getName(ivar),
ivar_getTypeEncoding(ivar));
        }

        free(ivars);

    }
    return 0;
}

// 打印内容
// Runtime应用[25783:4778679] _name, @"NSString"
// Runtime应用[25783:4778679] xx_cc
// Runtime应用[25783:4778679] xx_cc
// Runtime应用[25783:4778679] _name, @"NSString"
```

属性相关API

```
1. 获取一个属性
objc_property_t class_getProperty(Class cls, const char *name)

2. 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)

3. 动态添加属性
BOOL class_addProperty(Class cls, const char *name, const
objc_property_attribute_t *attributes,
                  unsigned int attributeCount)

4. 动态替换属性
```

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

5. 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)
```

方法相关API

```
1. 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)

2. 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)

3. 拷贝方法列表（最后需要调用free释放）
Method *class_copyMethodList(Class cls, unsigned int *outCount)

4. 动态添加方法
BOOL class_addMethod(Class cls, SEL name, IMP imp, const char
*types)

5. 动态替换方法
IMP class_replaceMethod(Class cls, SEL name, IMP imp, const char
*types)

6. 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)

7. 选择器相关
const char *sel_getName(SEL sel)
SEL sel_registerName(const char *str)

8. 用block作为方法实现
IMP imp_implementationWithBlock(id block)
id imp_getBlock(IMP anImp)
```

```
BOOL imp_removeBlock(IMP anImp)
```