

# OC语法

@M了个J

<https://github.com/CoderMJLee>



实力IT教育 [www.520it.com](http://www.520it.com)

- 一个NSObject对象占用多少内存？
  - 系统分配了16个字节给NSObject对象（通过malloc\_size函数获得）
  - 但NSObject对象内部只使用了8个字节的空间（64bit环境下，可以通过class\_getInstanceSize函数获得）
  
- 对象的isa指针指向哪里？
  - instance对象的isa指向class对象
  - class对象的isa指向meta-class对象
  - meta-class对象的isa指向基类的meta-class对象
  
- OC的类信息存放在哪里？
  - 对象方法、属性、成员变量、协议信息，存放在class对象中
  - 类方法，存放在meta-class对象中
  - 成员变量的具体值，存放在instance对象

- iOS用什么方式实现对一个对象的KVO? (KVO的本质是什么?)
- 利用RuntimeAPI动态生成一个子类, 并且让instance对象的isa指向这个全新的子类
- 当修改instance对象的属性时, 会调用Foundation的\_NSSetXXXValueAndNotify函数
- ✓ willChangeValueForKey:
- ✓ 父类原来的setter
- ✓ didChangeValueForKey:
- 内部会触发监听器 (Observer) 的监听方法( observeValueForKeyPath:ofObject:change:context:)
  
- 如何手动触发KVO?
- 手动调用willChangeValueForKey:和didChangeValueForKey:
  
- 直接修改成员变量会触发KVO么?
- 不会触发KVO

- 通过KVC修改属性会触发KVO么？
  - 会触发KVO
  
- KVC的赋值和取值过程是怎样的？原理是什么？

- Category的使用场合是什么？
- Category的实现原理
  - Category编译之后的底层结构是struct category\_t，里面存储着分类的对象方法、类方法、属性、协议信息
  - 在程序运行的时候，runtime会将Category的数据，合并到类信息中（类对象、元类对象中）
- Category和Class Extension的区别是什么？
  - Class Extension在编译的时候，它的数据就已经包含在类信息中
  - Category是在运行时，才会将数据合并到类信息中

# 面试题 - Category

- Category中有load方法吗？load方法是什么时候调用的？load 方法能继承吗？
  - 有load方法
  - load方法在runtime加载类、分类的时候调用
  - load方法可以继承，但是一般情况下不会主动去调用load方法，都是让系统自动调用
  
- load、initialize方法的区别什么？它们在category中的调用的顺序？以及出现继承时他们之间的调用过程？
  
- Category能否添加成员变量？如果可以，如何给Category添加成员变量？
  - 不能直接给Category添加成员变量，但是可以间接实现Category有成员变量的效果

- block的原理是怎样的？本质是什么？
  - 封装了函数调用以及调用环境的OC对象
  
- \_\_block的作用是什么？有什么使用注意点？
  
- block的属性修饰词为什么是copy？使用block有哪些使用注意？
  - block一旦没有进行copy操作，就不会在堆上
  - 使用注意：循环引用问题
  
- block在修改NSMutableArray，需不需要添加\_\_block？

# Objective-C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



- 所以Objective-C的面向对象都是基于C\C++的数据结构实现的
- 思考：Objective-C的对象、类主要是基于C\C++的什么数据结构实现的？
  - 结构体
- 将Objective-C代码转换为C\C++代码
  - ▣ `xcrun -sdk iphoneos clang -arch arm64 -rewrite-objc` OC源文件 `-o` 输出的CPP文件
  - 如果需要链接其他框架，使用`-framework`参数。比如`-framework UIKit`



- 思考：一个OC对象在内存中是如何布局的？
- NSObject的底层实现

```
@interface NSObject {  
    Class isa;  
}  
@end
```

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

```
typedef struct objc_class *Class;
```

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

obj = 0x100400110

NSObject对象

isa

地址： 0x100400110

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

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

stu = 0x100400110

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

Student对象

isa

地址: 0x100400110

\_no = 4

地址: 0x100400118


\_age = 5

地址: 0x10040011C

```
struct Student_IMPL *stu2 = (__bridge struct Student_IMPL *)stu;  
NSLog(@"%d, %d", stu2->_no, stu2->_age);
```

# OC对象的本质

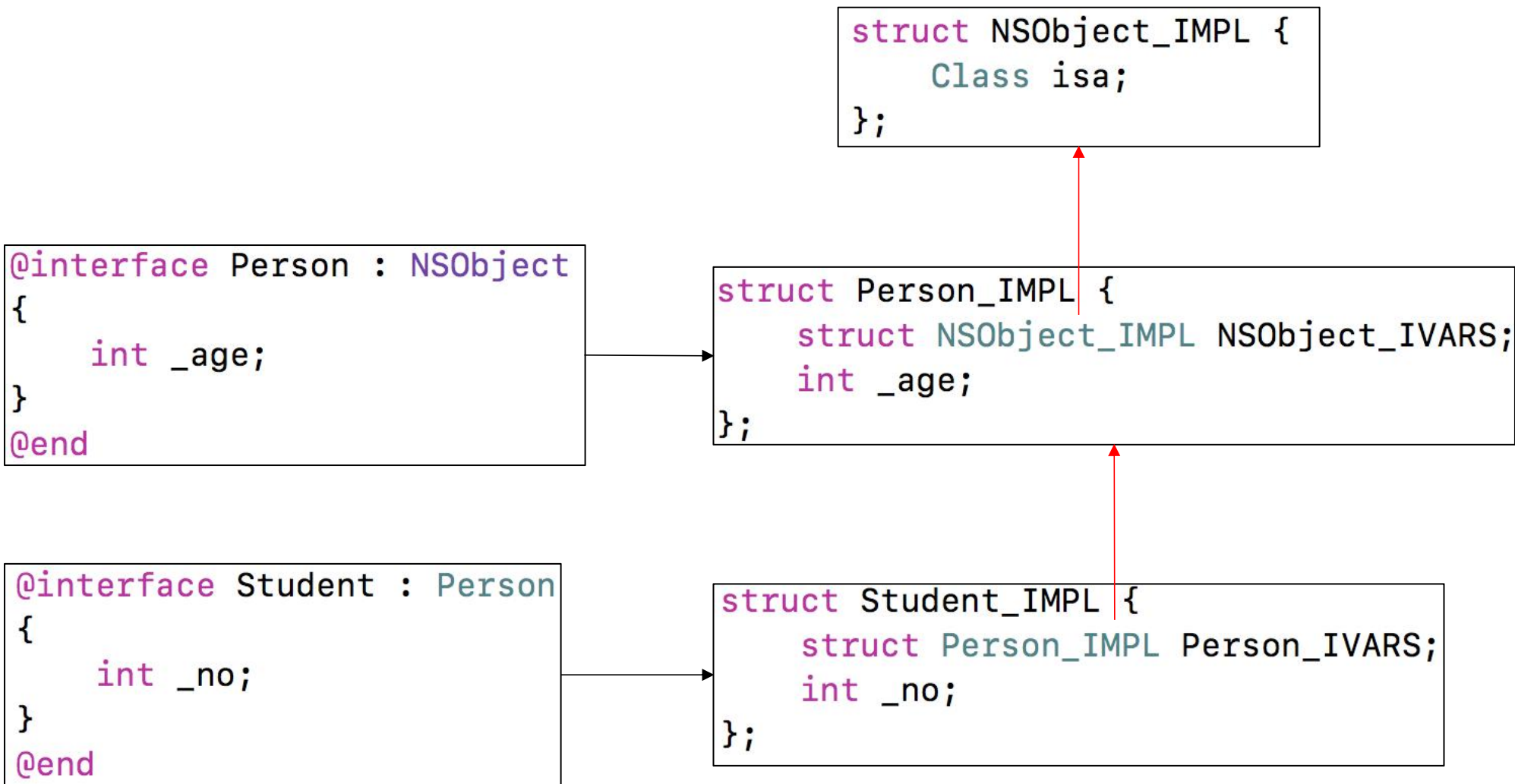
```
@interface Student : NSObject {  
    @public  
    int _no;  
    int _age;  
}  
@end
```



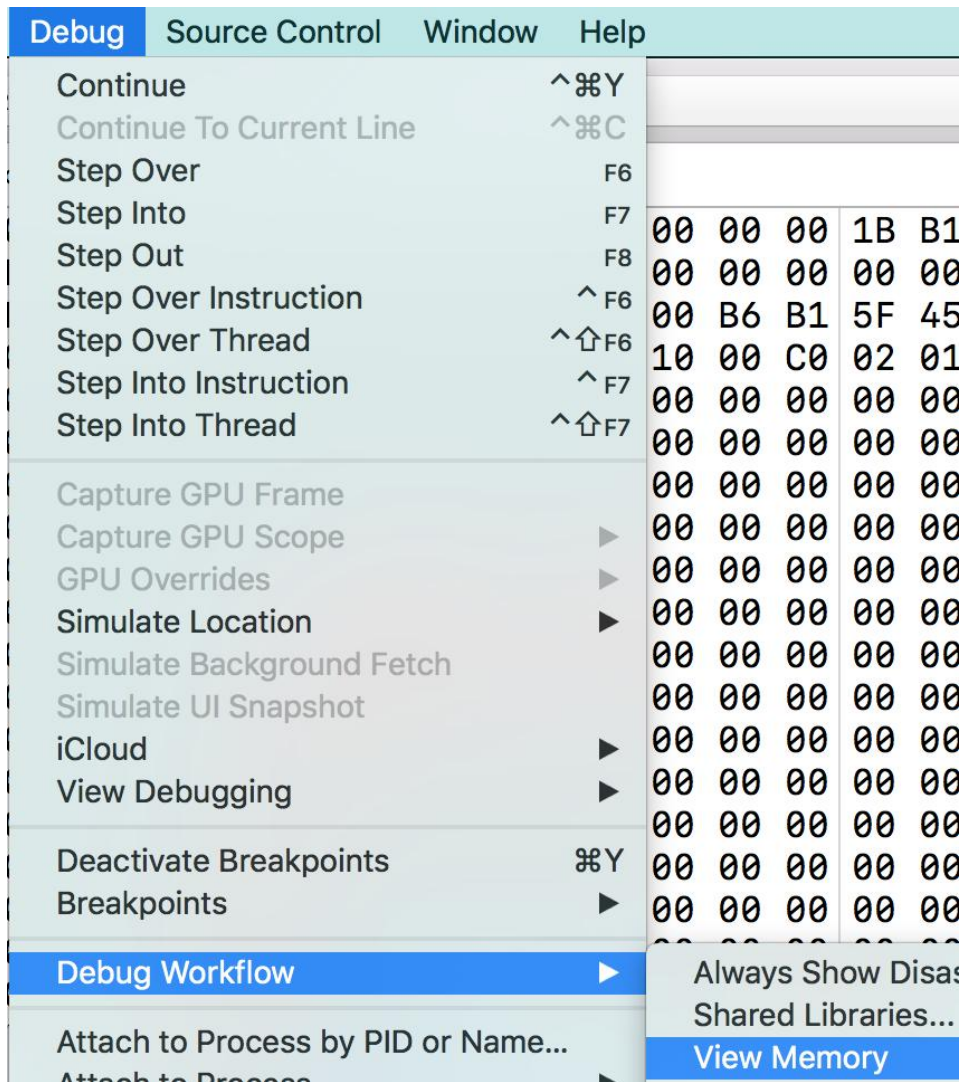
```
struct Student_IMPL {  
    struct NSObject_IMPL NSObject_IVARS;  
    int _no;  
    int _age;  
};
```

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

- 思考：一个Person对象、一个Student对象占用多少内存空间？



- Debug -> Debug Workflow -> View Memory (Shift + Command + M)



- 也可以使用LLDB指令

# 2个容易混淆的函数

- 创建一个实例对象，至少需要多少内存？

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

- `class_getInstanceSize([NSObject class]);`

- 创建一个实例对象，实际上分配了多少内存？

- `#import <malloc/malloc.h>`

- `malloc_size((__bridge const void *)obj);`

- **print**、**p**: 打印

- **po**: 打印对象

- 读取内存

- **memory read**/数量格式字节数 内存地址

- **x**/数量格式字节数 内存地址

- ✓ x/3xw 0x10010

- 修改内存中的值

- **memory write** 内存地址 数值

- ✓ memory write 0x0000010 10

- 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

- Objective-C中的对象，简称OC对象，主要可以分为3种
  - **instance**对象（**实例**对象）
  - **class**对象（**类**对象）
  - **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

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

- object1、object2是NSObject的**instance**对象（实例对象）
- 它们是不同的两个对象，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
  - isa指针
  - 其他**成员变量**

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

```
Person *p1 = [[Person alloc] init];
p1->_age = 3;

Person *p2 = [[Person alloc] init];
p2->_age = 4;
```



```
NSObject *object1 = [[NSObject alloc] init];
NSObject *object2 = [[NSObject alloc] init];
Class objectClass1 = [object1 class];
Class objectClass2 = [object2 class];
Class objectClass3 = [NSObject class];
Class objectClass4 = object_getClass(object1); // Runtime API
Class objectClass5 = object_getClass(object2); // Runtime API
```

- objectClass1 ~ objectClass5都是NSObject的class对象（类对象）
- 它们是同一个对象。每个类在内存中有且只有一个class对象
- class对象在内存中存储的信息主要包括
  - isa指针
  - superclass指针
  - 类的属性信息（@property）、类的对象方法信息（instance method）
  - 类的协议信息（protocol）、类的成员变量信息（ivar）
  - .....



```
Class objectMetaClass = object_getClass([NSObject class]); // Runtime API
```

- objectMetaClass是NSObject的meta-class对象（元类对象）
- 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
  - isa指针
  - superclass指针
  - 类的类方法信息（class method）
  - .....



- 以下代码获取的objectClass是class对象，并不是meta-class对象

```
Class objectClass = [[NSObject class] class];
```

# 查看Class是否为meta-class

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

```
BOOL result = class_isMetaClass([NSObject class]);
```

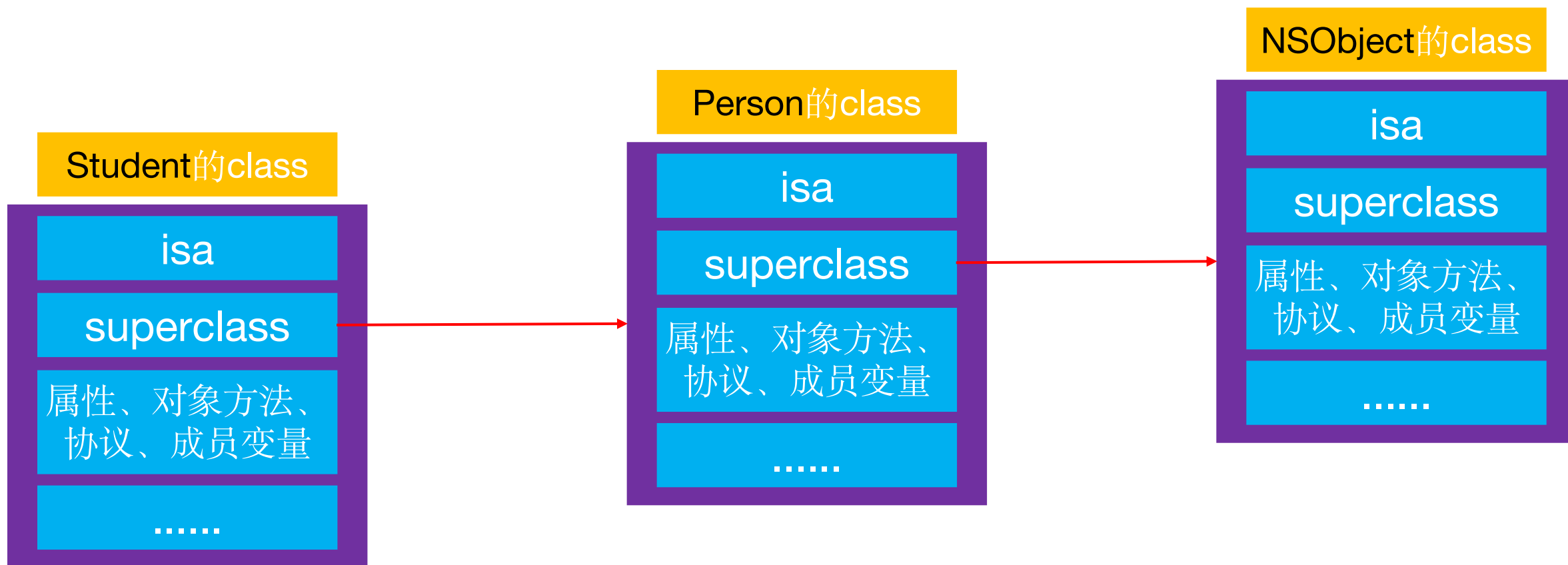


- instance的isa指向class
- 当调用对象方法时，通过instance的isa找到class，最后找到对象方法的实现进行调用
- class的isa指向meta-class
- 当调用类方法时，通过class的isa找到meta-class，最后找到类方法的实现进行调用

# class对象的superclass指针

```
@interface Student : Person
```

```
@interface Person : NSObject
```

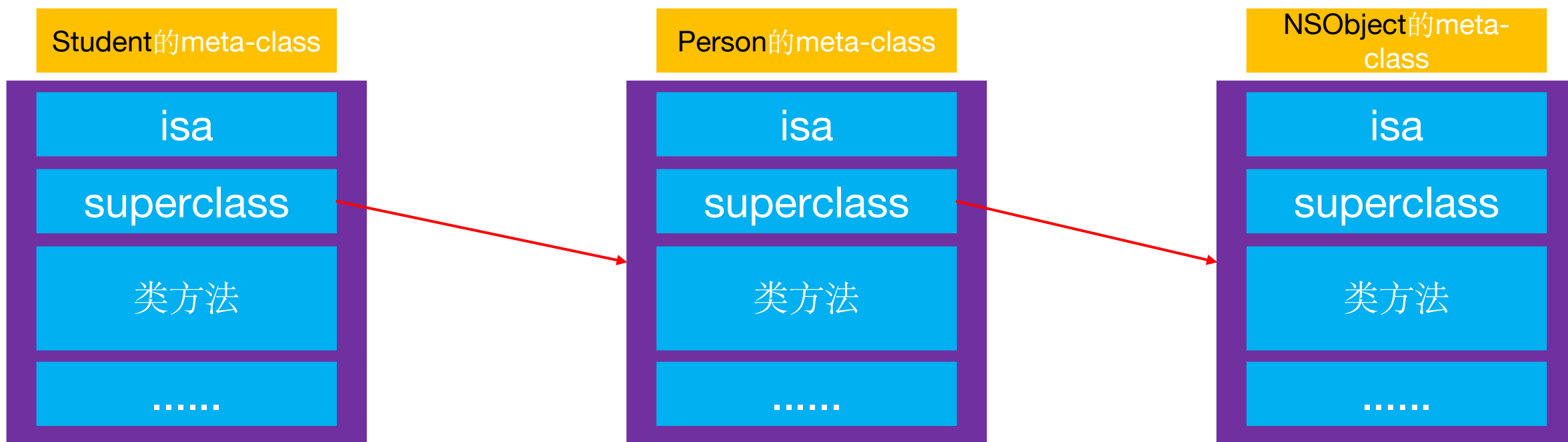


- 当Student的instance对象要调用Person的对象方法时，会先通过isa找到Student的class，然后通过superclass找到Person的class，最后找到对象方法的实现进行调用

# meta-class对象的superclass指针

```
@interface Student : Person
```

```
@interface Person : NSObject
```

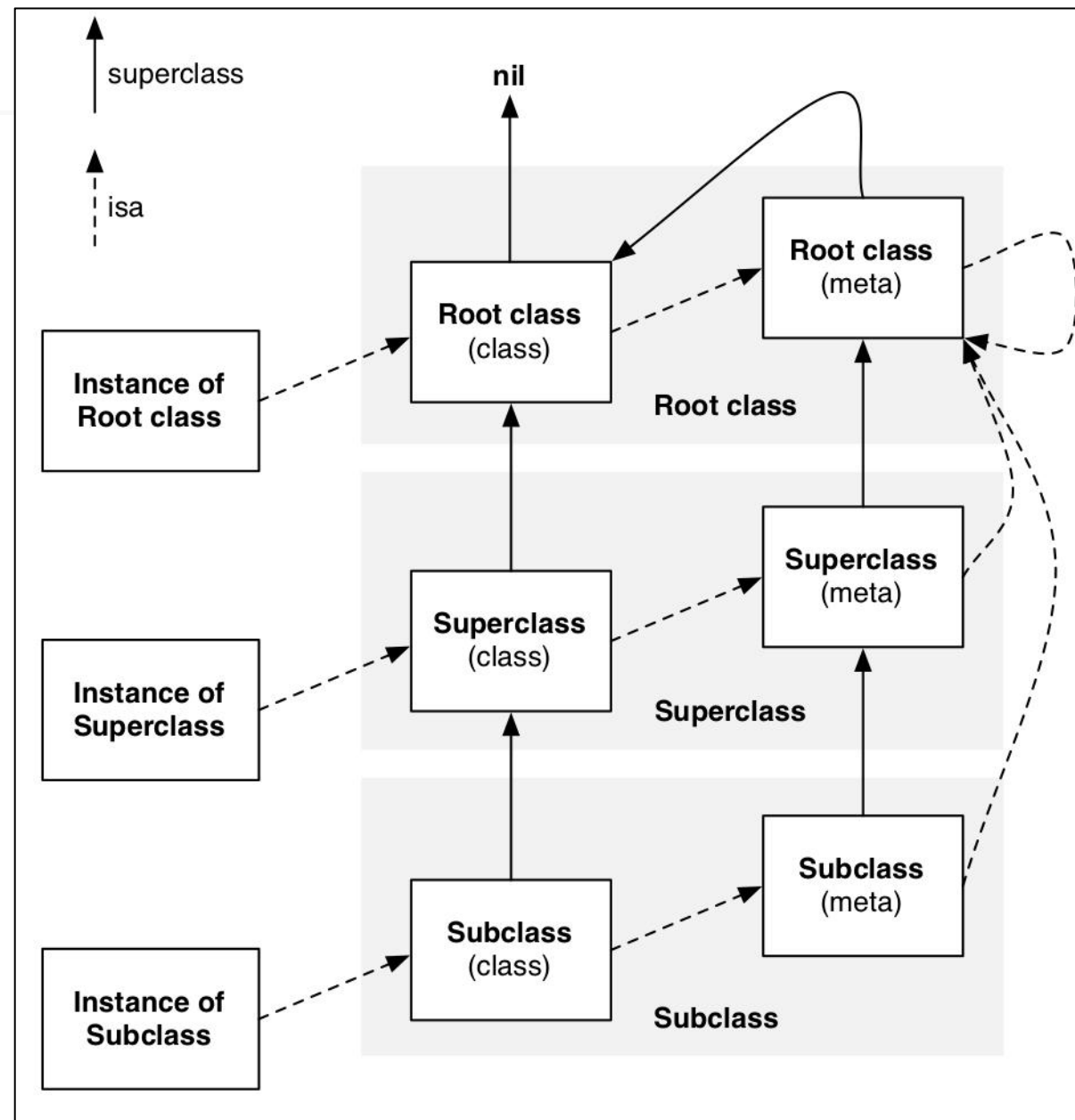


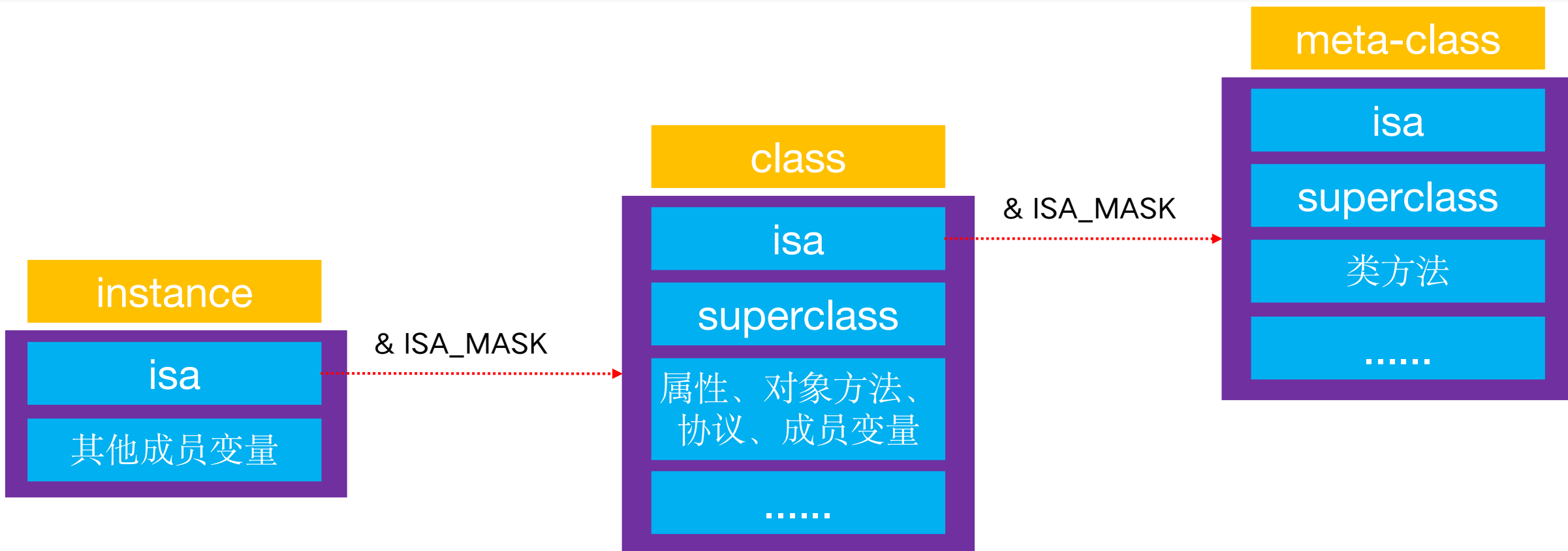
- 当Student的class要调用Person的类方法时，会先通过isa找到Student的meta-class，然后通过superclass找到Person的meta-class，最后找到类方法的实现进行调用



# isa、superclass总结

- instance的isa指向class
- class的isa指向meta-class
- meta-class的isa指向基类的meta-class
- class的superclass指向父类的class
- 如果没有父类，superclass指针为nil
- meta-class的superclass指向父类的meta-class
- 基类的meta-class的superclass指向基类的class
- 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
- 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



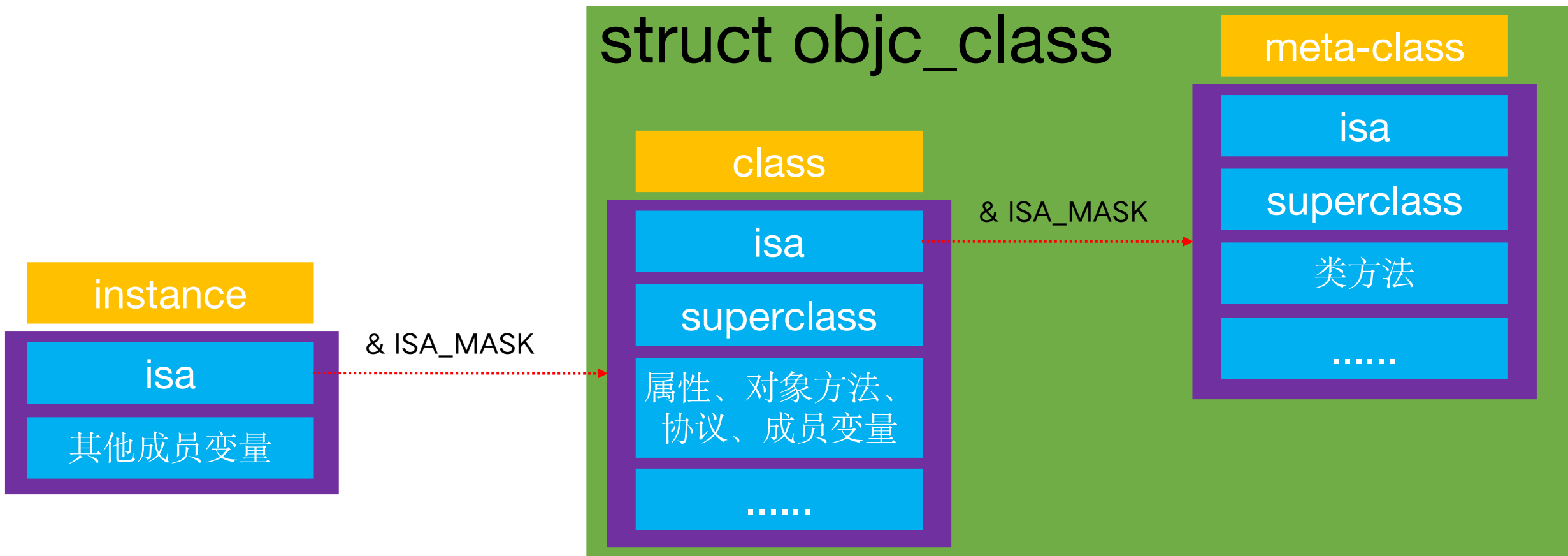


- 从64bit开始，isa需要进行一次位运算，才能计算出真实地址

```

# if __arm64__
#   define ISA_MASK      0x000000fffffffff8ULL
# elif __x86_64__
#   define ISA_MASK      0x00007fffffffffff8ULL
# endif
    
```

- <https://opensource.apple.com/tarballs/objc4/>



- class、meta-class对象的本质结构都是struct objc\_class

# 窥探struct objc\_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\_MA  
CK

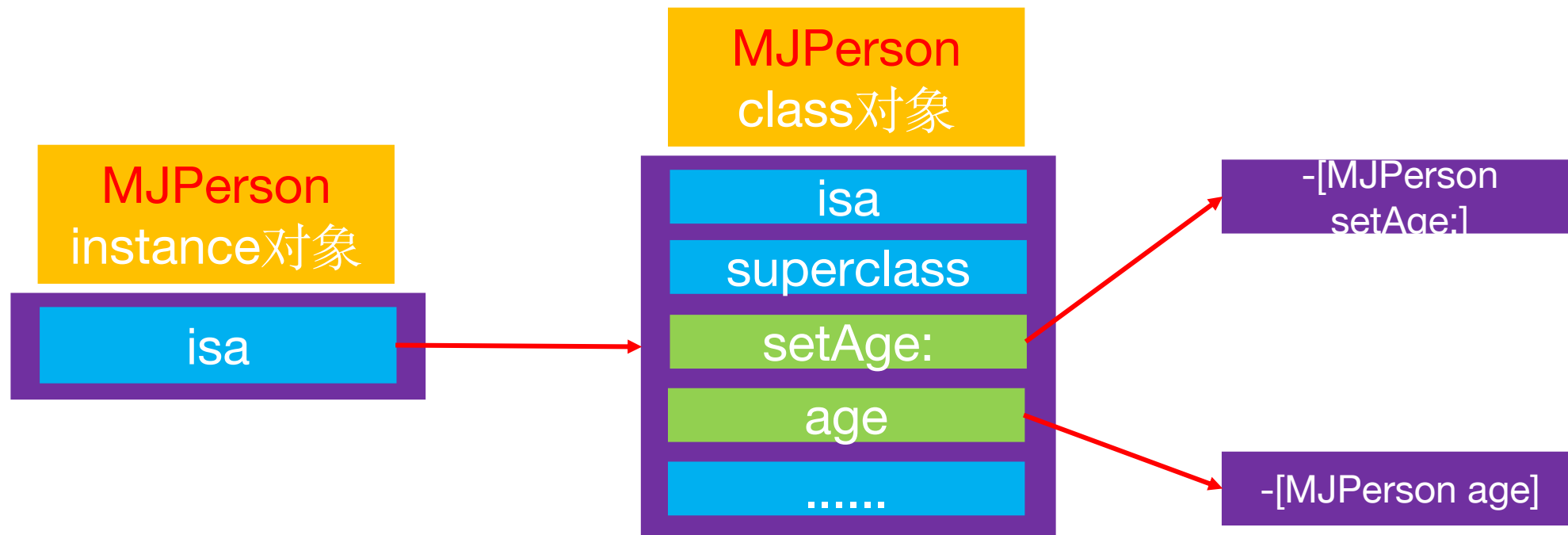
```
struct class_rw_t {  
    uint32_t flags;  
    uint32_t version;  
    const class_ro_t *ro; —————→  
    method_list_t *methods; // 方法列表  
    property_list_t *properties; // 属性列表  
    const protocol_list_t *protocols; // 协议列表  
    Class firstSubclass;  
    Class nextSiblingClass;  
    char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

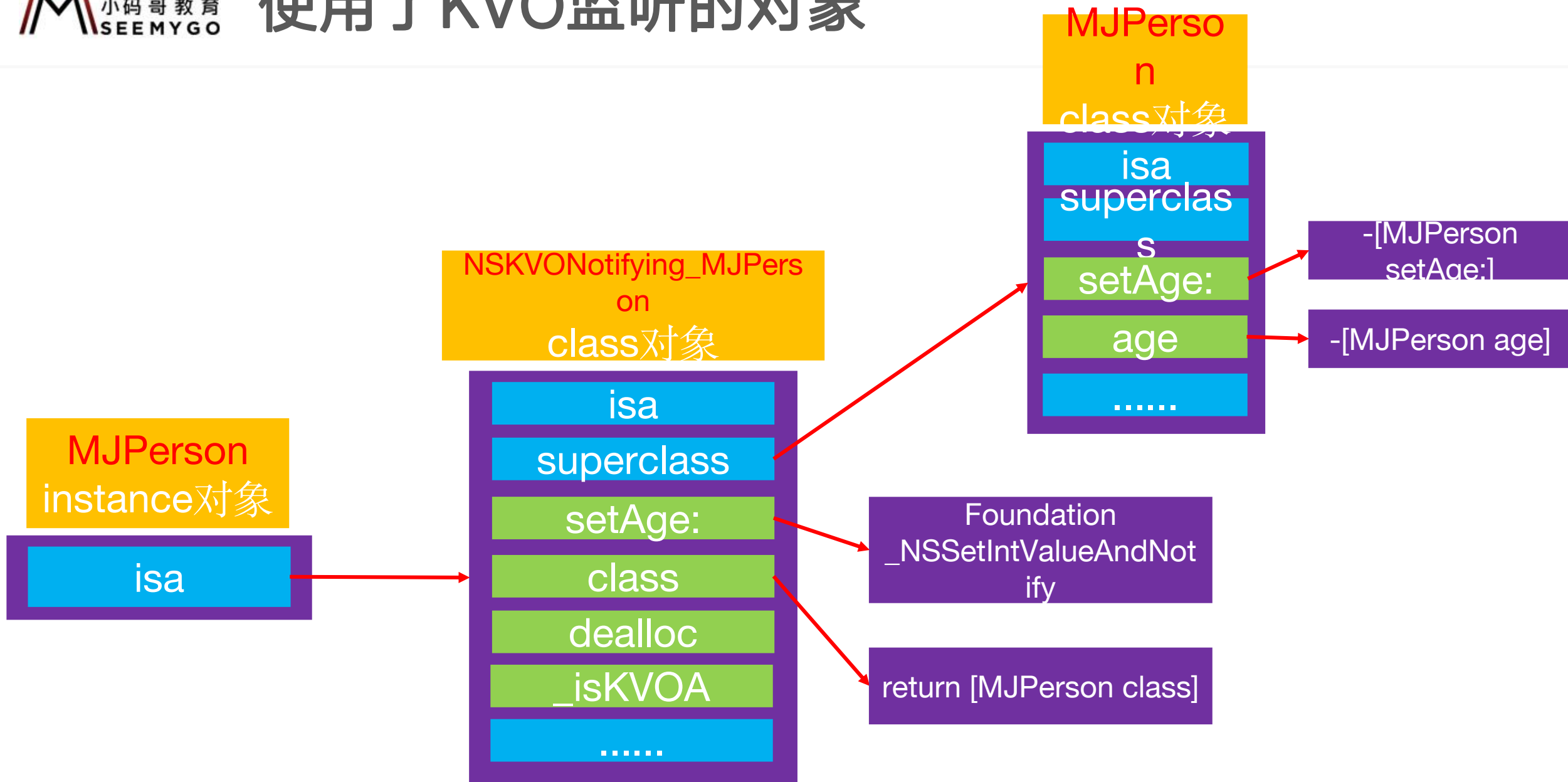
- KVO的全称是Key-Value Observing，俗称“键值监听”，可以用于监听某个对象属性值的改变



# 未使用KVO监听的对象



# 使用了KVO监听的对象





# 查看\_NSSet\*AndNotify的存在

```
~/Desktop mj$ nm Foundation | grep ValueAndNotify
000000018307f5f8 t __NSSetBoolValueAndNotify
0000000182ffb37c t __NSSetCharValueAndNotify
000000018307f868 t __NSSetDoubleValueAndNotify
0000000183039cc4 t __NSSetFloatValueAndNotify
0000000183024f30 t __NSSetIntValueAndNotify
000000018307fd50 t __NSSetLongLongValueAndNotify
000000018307fae0 t __NSSetLongValueAndNotify
0000000182ffb534 t __NSSetObjectValueAndNotify
0000000183080230 t __NSSetPointValueAndNotify
0000000183080378 t __NSSetRangeValueAndNotify
00000001830804c0 t __NSSetRectValueAndNotify
000000018307ffc0 t __NSSetShortValueAndNotify
0000000183080624 t __NSSetSizeValueAndNotify
```



# \_NSSet\*ValueAndNotify的内部实现

```
[self willChangeValueForKey:@"age"];

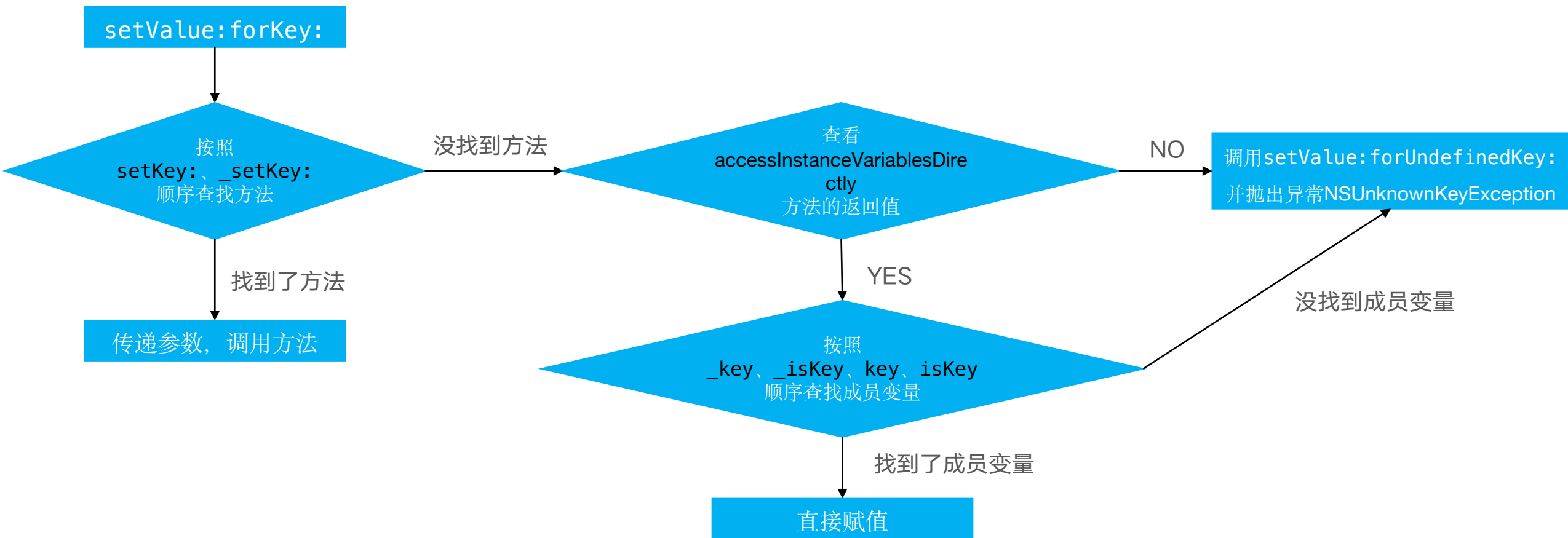
// 原来的setter实现

[self didChangeValueForKey:@"age"];
```

- 调用willChangeValueForKey:
- 调用原来的setter实现
- 调用didChangeValueForKey:
- didChangeValueForKey:内部会调用observer的observeValueForKeyPath:ofObject:change:context:方法

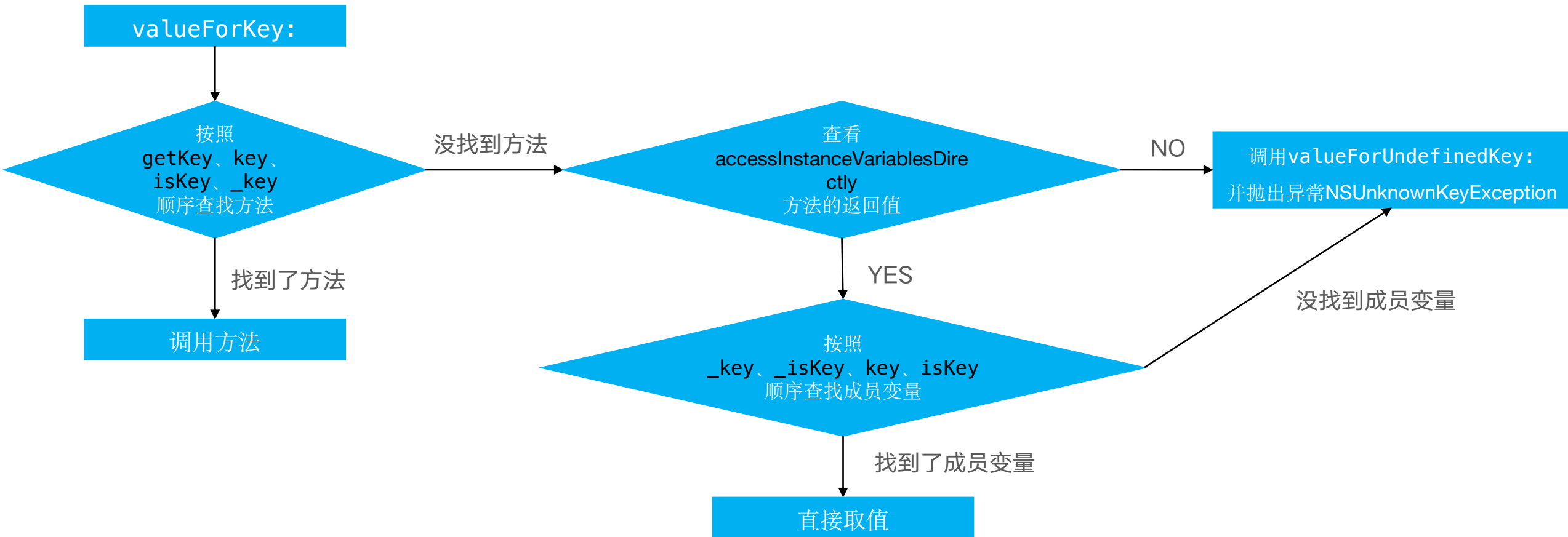
- KVC的全称是Key-Value Coding，俗称“键值编码”，可以通过一个key来访问某个属性
- 常见的API有
  - – (void)setValue:(id)value forKeyPath:(NSString \*)keyPath;
  - – (void)setValue:(id)value forKey:(NSString \*)key;
  - – (id)valueForKeyPath:(NSString \*)keyPath;
  - – (id)valueForKey:(NSString \*)key;

# setValue:forKey:的原理



- accessInstanceVariablesDirectly方法的默认返回值是YES

# valueForKey:的原理



- 定义在objc-runtime-new.h中

```
struct category_t {
    const char *name;
    classref_t cls;
    struct method_list_t *instanceMethods;
    struct method_list_t *classMethods;
    struct protocol_list_t *protocols;
    struct property_list_t *instanceProperties;
    // Fields below this point are not always present on disk.
    struct property_list_t *_classProperties;

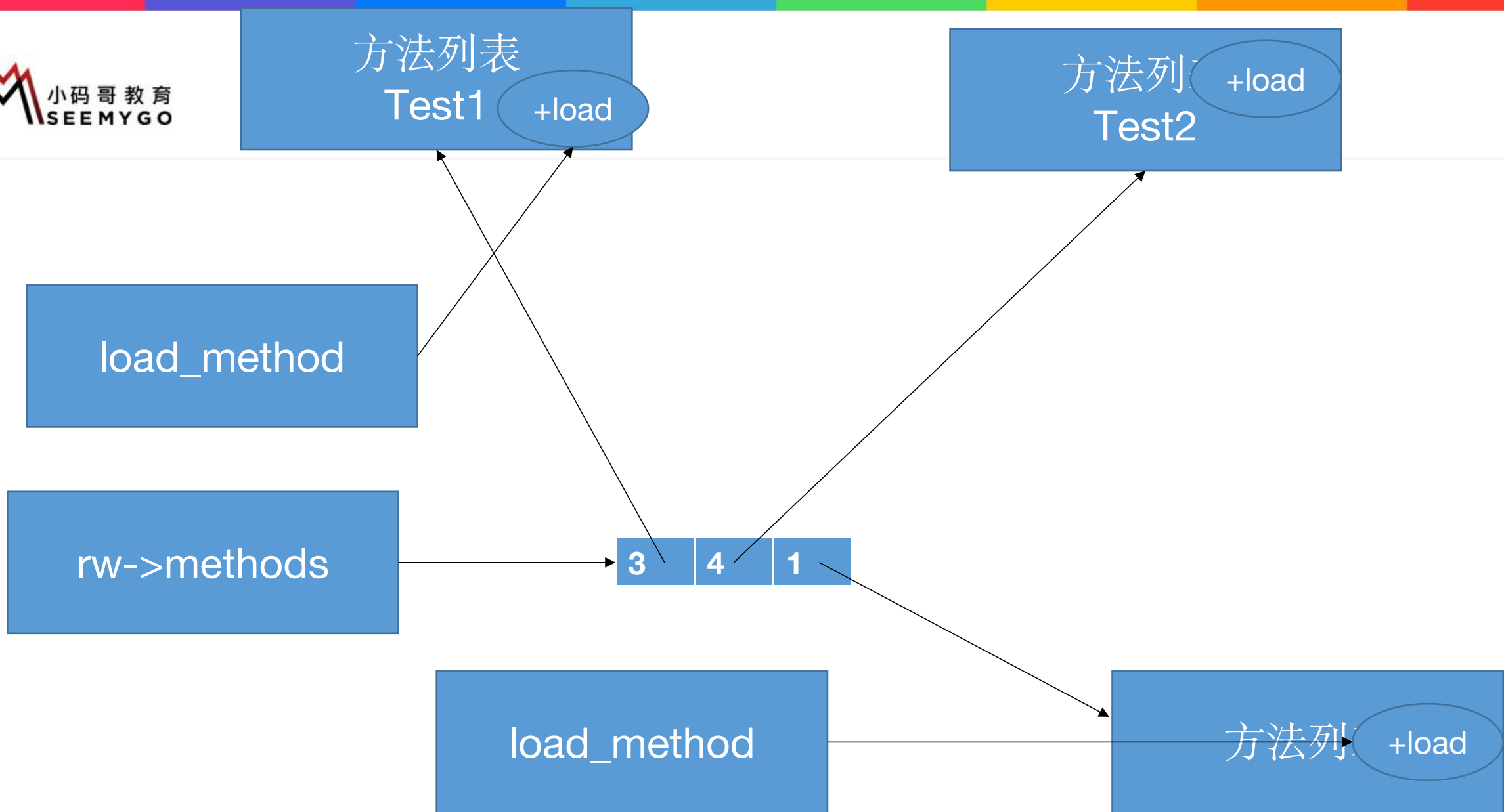
    method_list_t *methodsForMeta(bool isMeta) {
        if (isMeta) return classMethods;
        else return instanceMethods;
    }

    property_list_t *propertiesForMeta(bool isMeta, struct header_info *hi);
};
```

# Category的加载处理过程

1. 通过Runtime加载某个类的所有Category数据
2. 把所有Category的方法、属性、协议数据，合并到一个大数组中
  - ✓ 后面参与编译的Category数据，会在数组的前面
3. 将合并后的分类数据（方法、属性、协议），插入到类原来数据的前面

- 源码解读顺序
- objc-os.mm
  - ✓ \_objc\_init
  - ✓ map\_images
  - ✓ map\_images\_nolock
- objc-runtime-new.mm
  - ✓ \_read\_images
  - ✓ remethodizeClass
  - ✓ attachCategories
  - ✓ attachLists
  - ✓ realloc、memmove、memcpy



- +load方法会在runtime加载类、分类时调用
- 每个类、分类的+load，在程序运行过程中只调用一次
- 调用顺序
  1. 先调用类的+load
    - ✓ 按照编译先后顺序调用（先编译，先调用）
    - ✓ 调用子类的+load之前会先调用父类的+load
  2. 再调用分类的+load
    - ✓ 按照编译先后顺序调用（先编译，先调用）
- objc4源码解读过程：objc-os.mm
  - \_objc\_init
  - load\_images
  - prepare\_load\_methods
    - ✓ schedule\_class\_load
    - ✓ add\_class\_to\_loadable\_list
    - ✓ add\_category\_to\_loadable\_list
  - call\_load\_methods
    - ✓ call\_class\_loads
    - ✓ call\_category\_loads
    - ✓ (\*load\_method)(cls, SEL\_load)
- +load方法是根据方法地址直接调用，并不是经过objc\_msgSend函数调用



# +initialize方法

- +initialize方法会在类第一次接收到消息时调用
- 调用顺序
  - 先调用父类的+initialize，再调用子类的+initialize
  - (先初始化父类，再初始化子类，每个类只会初始化1次)
- objc4源码解读过程
  - objc-msg-arm64.s
    - ✓ objc\_msgSend
  - objc-runtime-new.mm
    - ✓ class\_getInstanceMethod
    - ✓ lookupImpOrNil
    - ✓ lookupImpOrForward
    - ✓ \_class\_initialize
    - ✓ callInitialize
    - ✓ objc\_msgSend(cls, SEL\_initialize)
- +initialize和+load的很大区别是，+initialize是通过objc\_msgSend进行调用的，所以有以下特点
  - 如果子类没有实现+initialize，会调用父类的+initialize（所以父类的+initialize可能会被调用多次）
  - 如果分类实现了+initialize，就覆盖类本身的+initialize调用

# 思考：如何实现给分类“添加成员变量”？

- 默认情况下，因为分类底层结构的限制，不能添加成员变量到分类中。但可以通过关联对象来间接实现

- 关联对象提供了以下API

- 添加关联对象

- ✓ `void objc_setAssociatedObject(id object, const void * key, id value, objc_AssociationPolicy policy)`

- 获得关联对象

- ✓ `id objc_getAssociatedObject(id object, const void * key)`

- 移除所有的关联对象

- ✓ `void objc_removeAssociatedObjects(id object)`

# key的常见用法

- `static void *MyKey = &MyKey;`
- `objc_setAssociatedObject(obj, MyKey, value, OBJC_ASSOCIATION_RETAIN_NONATOMIC)`
- `objc_getAssociatedObject(obj, MyKey)`
  
- `static char MyKey;`
- `objc_setAssociatedObject(obj, &MyKey, value, OBJC_ASSOCIATION_RETAIN_NONATOMIC)`
- `objc_getAssociatedObject(obj, &MyKey)`
  
- 使用属性名作为key
- `objc_setAssociatedObject(obj, @"property", value, OBJC_ASSOCIATION_RETAIN_NONATOMIC);`
- `objc_getAssociatedObject(obj, @"property");`
  
- 使用get方法的@selector作为key
- `objc_setAssociatedObject(obj, @selector(getter), value, OBJC_ASSOCIATION_RETAIN_NONATOMIC)`
- `objc_getAssociatedObject(obj, @selector(getter))`

# objc\_AssociationPolicy

objc_AssociationPolicy	对应的修饰符
OBJC_ASSOCIATION_ASSIGN	assign
OBJC_ASSOCIATION_RETAIN_NONATOMIC	strong, nonatomic
OBJC_ASSOCIATION_COPY_NONATOMIC	copy, nonatomic
OBJC_ASSOCIATION_RETAIN	strong, atomic
OBJC_ASSOCIATION_COPY	copy, atomic

- 实现关联对象技术的核心对象有
  - AssociationsManager
  - AssociationsHashMap
  - ObjectAssociationMap
  - ObjcAssociation
- objc4源码解读: objc-references.mm

```
class AssociationsManager {  
    static AssociationsHashMap *_map;  
};
```

```
class AssociationsHashMap : public unordered_map<disguised_ptr_t, ObjectAssociationMap>
```

```
class ObjectAssociationMap : public std::map<void *, ObjcAssociation>
```

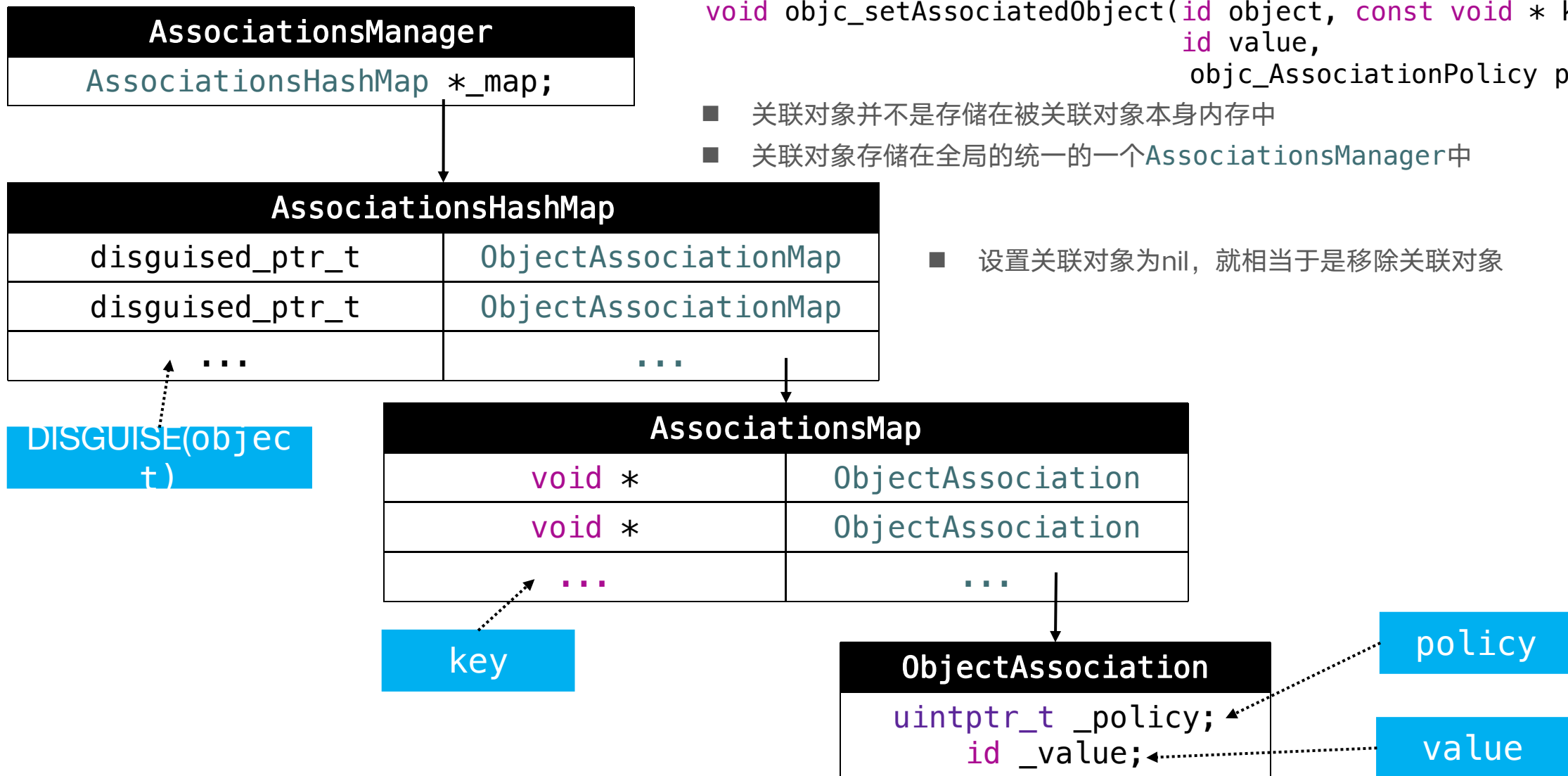
```
class ObjcAssociation {  
    uintptr_t _policy;  
    id _value;  
};
```

# 关联对象的原理

```
void objc_setAssociatedObject(id object, const void * key,
                             id value,
                             objc_AssociationPolicy policy)
```

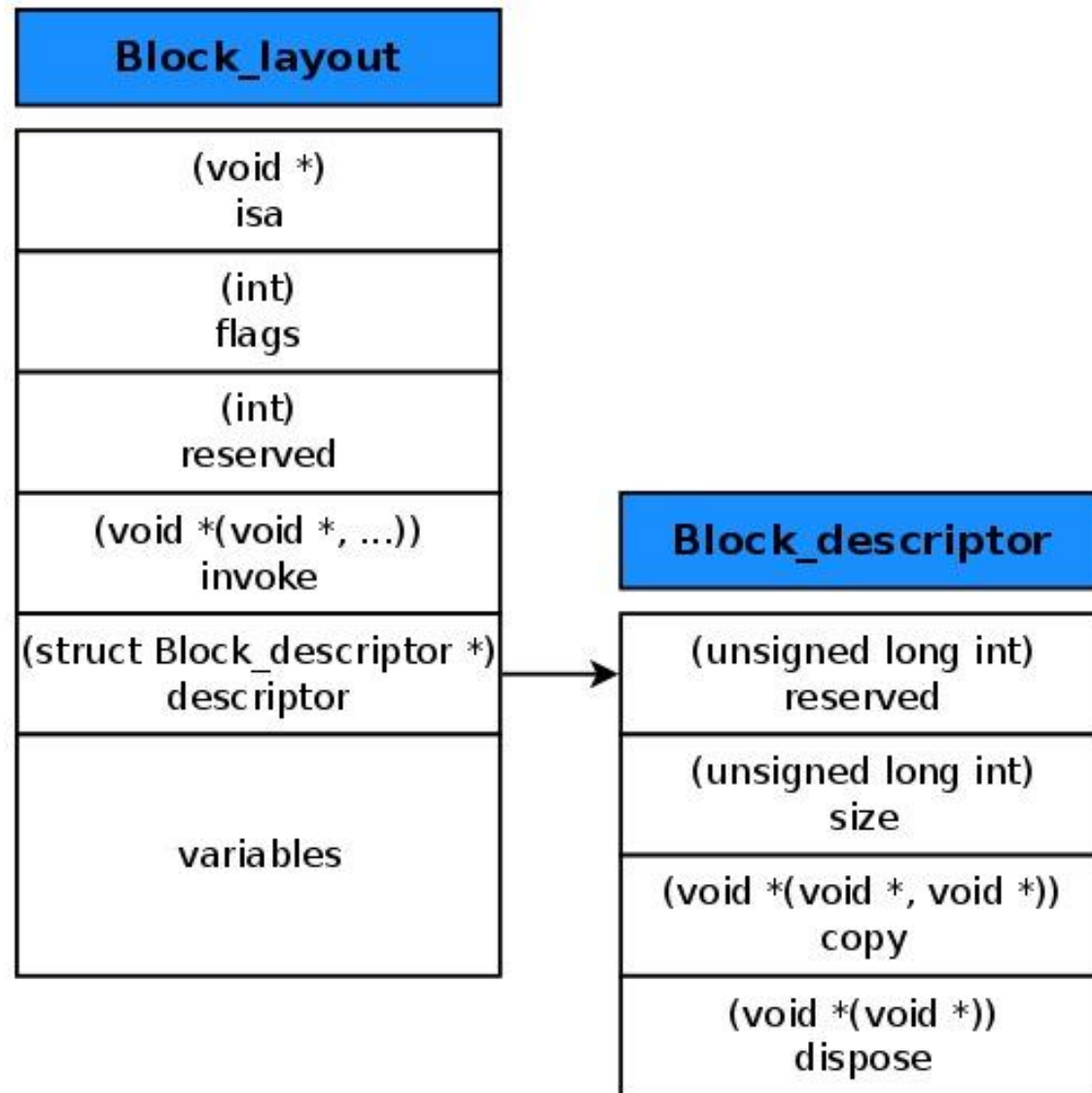
- 关联对象并不是存储在被关联对象本身内存中
- 关联对象存储在全局的统一的一个AssociationsManager中

- 设置关联对象为nil，就相当于移除关联对象



# block的本质

- block本质上也是一个OC对象，它内部也有个isa指针
- block是封装了函数调用以及函数调用环境的OC对象
- block的底层结构如右图所示



# block的变量捕获 (capture)

- 为了保证block内部能够正常访问外部的变量，block有个变量捕获机制

变量类型		捕获到block内部	访问方式
局部变量	auto	✓	值传递
	static	✓	指针传递
全局变量		✗	直接访问



# auto变量的捕获

```
int age = 20;  
void (^block)(void) = ^{  
    NSLog(@"age is %d", age);  
};
```

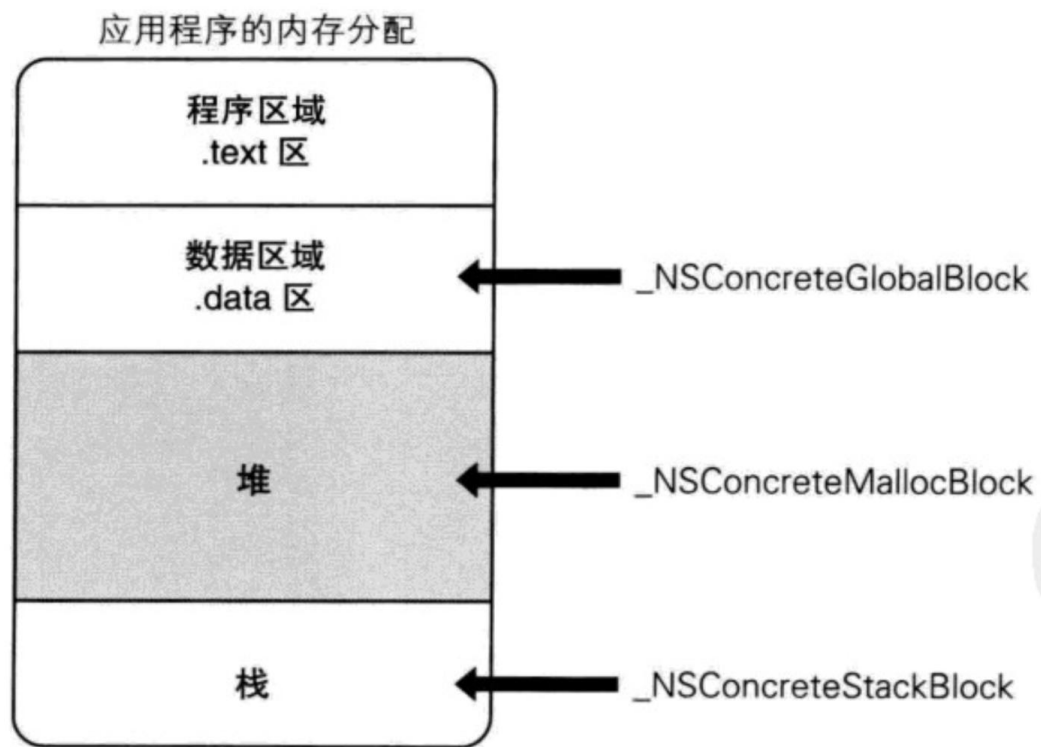
```
struct __main_block_impl_0 {  
    struct __block_impl impl;  
    struct __main_block_desc_0* Desc;  
    int age;  
};
```

```
struct __block_impl {  
    void *isa;  
    int Flags;  
    int Reserved;  
    void *FuncPtr;  
};
```

```
struct __main_block_desc_0 {  
    size_t reserved;  
    size_t Block_size;  
};
```

# block的类型

- block有3种类型，可以通过调用class方法或者isa指针查看具体类型，最终都是继承自NSBlock类型
- `__NSGlobalBlock__` ( `_NSConcreteGlobalBlock` )
- `__NSStackBlock__` ( `_NSConcreteStackBlock` )
- `__NSMallocBlock__` ( `_NSConcreteMallocBlock` )



block类型	环境
__NSGlobalBlock__	没有访问auto变量
__NSStackBlock__	访问了auto变量
__NSMallocBlock__	__NSStackBlock__调用了copy

■ 每一种类型的block调用copy后的结果如下所示

Block 的类	副本源的配置存储域	复制效果
_NSConcreteStackBlock	栈	从栈复制到堆
_NSConcreteGlobalBlock	程序的数据区域	什么也不做
_NSConcreteMallocBlock	堆	引用计数增加

- 在ARC环境下，编译器会根据情况自动将栈上的block复制到堆上，比如以下情况
  - block作为函数返回值时
  - 将block赋值给\_\_strong指针时
  - block作为Cocoa API中方法名含有usingBlock的方法参数时
  - block作为GCD API的方法参数时
- MRC下block属性的建议写法
  - @property (copy, nonatomic) void (^block)(void);
- ARC下block属性的建议写法
  - @property (strong, nonatomic) void (^block)(void);
  - @property (copy, nonatomic) void (^block)(void);

- 当block内部访问了对象类型的auto变量时
- 如果block是在栈上，将不会对auto变量产生强引用
- 如果block被拷贝到堆上
  - ✓ 会调用block内部的copy函数
  - ✓ copy函数内部会调用\_Block\_object\_assign函数
  - ✓ \_Block\_object\_assign函数会根据auto变量的修饰符（\_\_strong、\_\_weak、\_\_unsafe\_unretained）做出相应的操作，形成强引用（retain）或者弱引用
- 如果block从堆上移除
  - ✓ 会调用block内部的dispose函数
  - ✓ dispose函数内部会调用\_Block\_object\_dispose函数
  - ✓ \_Block\_object\_dispose函数会自动释放引用的auto变量（release）

函数	调用时机
copy 函数	栈上的 Block 复制到堆时
dispose 函数	堆上的 Block 被废弃时

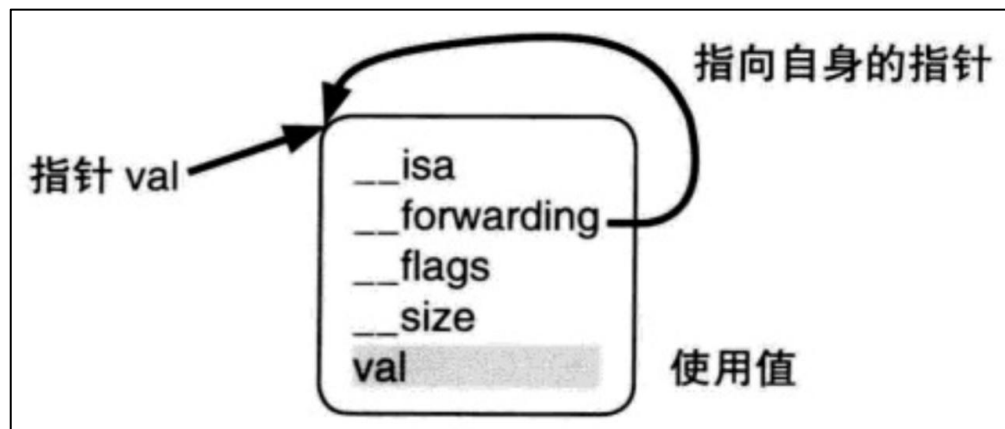
- 在使用clang转换OC为C++代码时，可能会遇到以下问题
- cannot create \_\_weak reference in file using manual reference
- 解决方案：支持ARC、指定运行时系统版本，比如
- `xcrun -sdk iphoneos clang -arch arm64 -rewrite-objc -fobjc-arc -fobjc-runtime=ios-8.0.0 main.m`

# \_\_block修饰符

- `__block`可以用于解决block内部无法修改`auto`变量值的问题
- `__block`不能修饰全局变量、静态变量（`static`）
- 编译器会将`__block`变量包装成一个对象

```
__block int age = 10;  
^{  
    NSLog(@"%d", age);  
}();
```

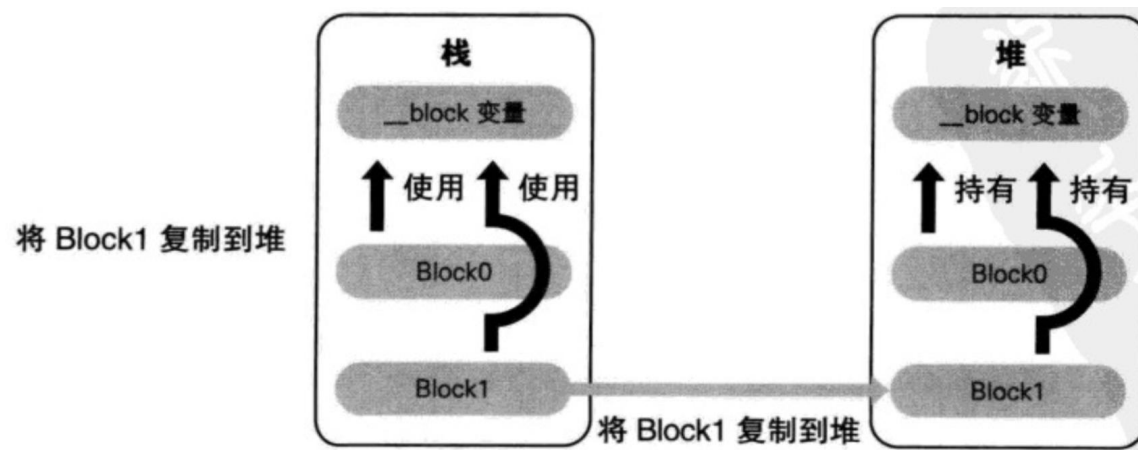
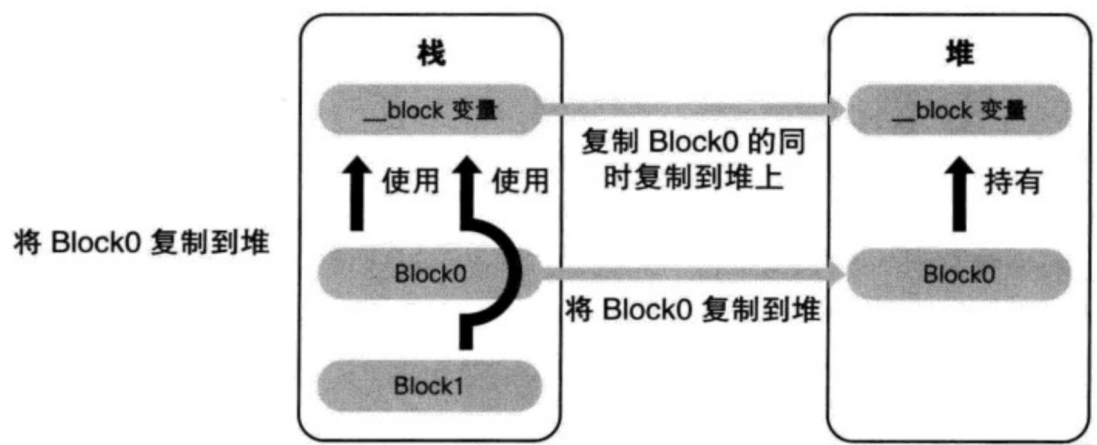
```
struct __main_block_impl_0 {  
    struct __block_impl impl;  
    struct __main_block_desc_0* Desc;  
    __Block_byref_age_0 *age; // by ref  
};
```



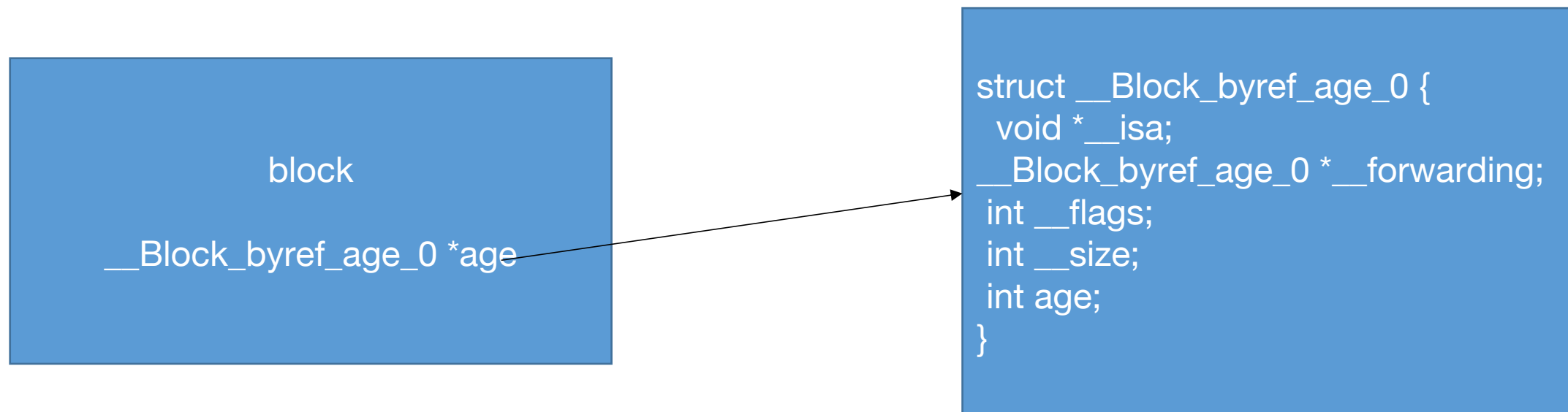
```
struct __Block_byref_age_0 {  
    void *__isa;  
    __Block_byref_age_0 *__forwarding;  
    int __flags;  
    int __size;  
    int age;  
};
```

# \_\_block的内存管理

- 当block在栈上时，并不会对\_\_block变量产生强引用
- 当block被copy到堆时
  - ✓ 会调用block内部的copy函数
  - ✓ copy函数内部会调用\_Block\_object\_assign函数
  - ✓ \_Block\_object\_assign函数会对\_\_block变量形成强引用 (retain)

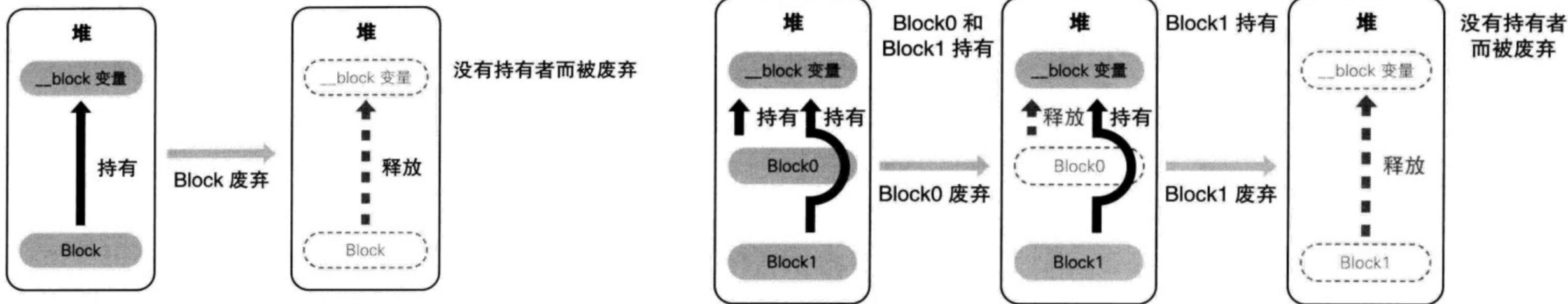




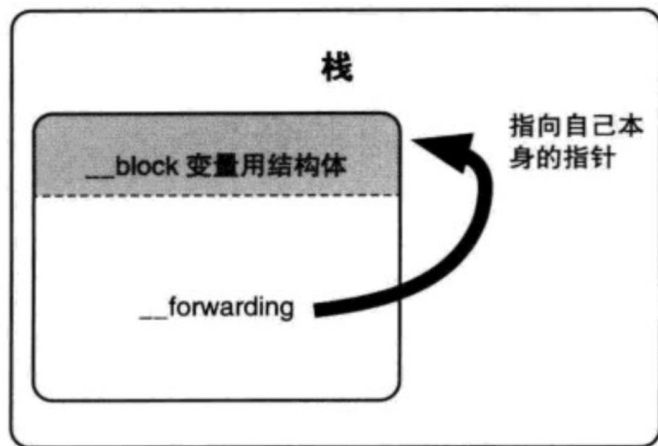


# \_\_block的内存管理

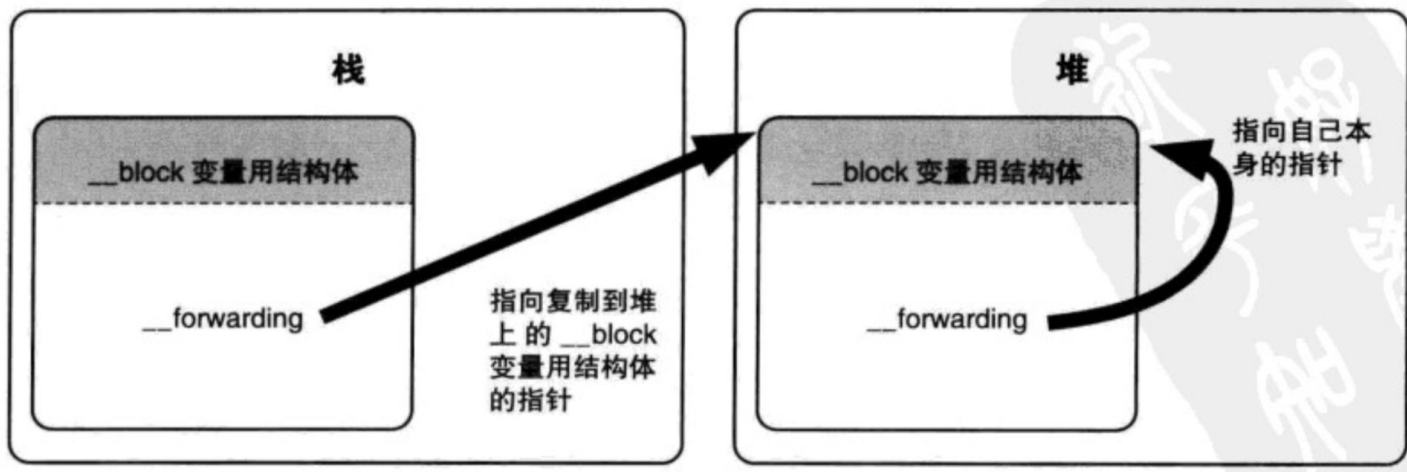
- 当block从堆中移除时
- ✓ 会调用block内部的dispose函数
- ✓ dispose函数内部会调用\_Block\_object\_dispose函数
- ✓ \_Block\_object\_dispose函数会自动释放引用的\_\_block变量 (release)



# \_\_block的\_\_forwarding指针



↓ 复制到堆之后



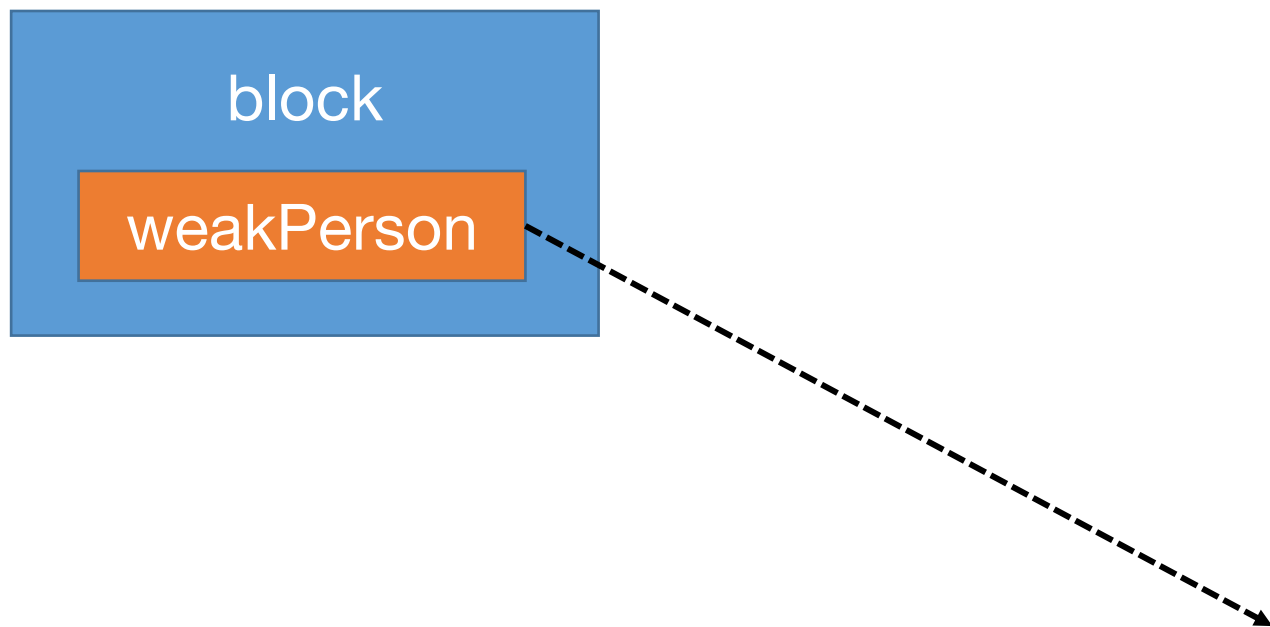
# 对象类型的auto变量、\_\_block变量

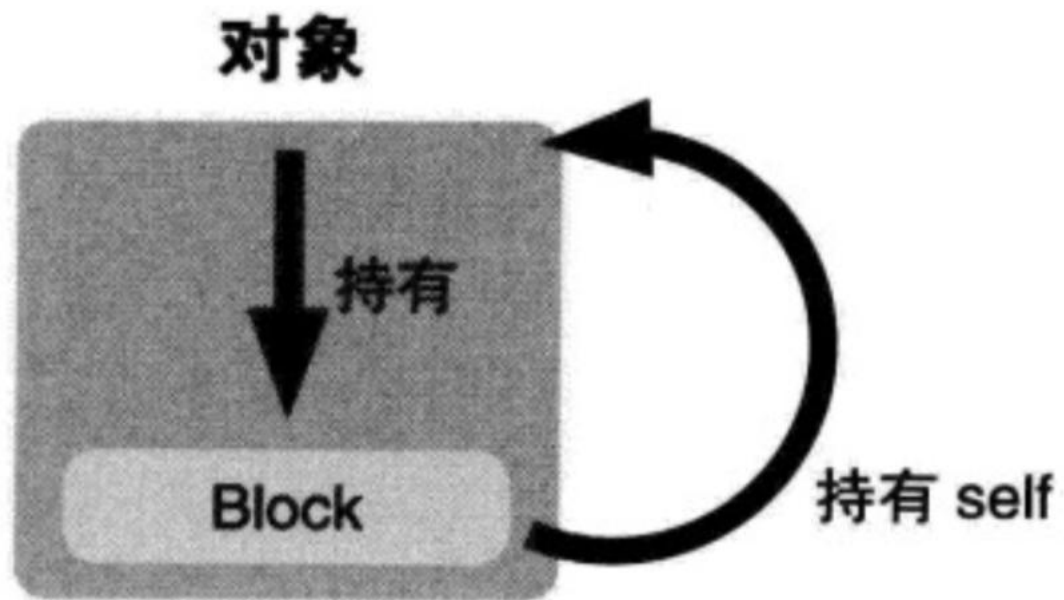
- 当block在栈上时，对它们都不会产生强引用
- 当block拷贝到堆上时，都会通过copy函数来处理它们
- \_\_block变量（假设变量名叫做a）
  - ✓ \_Block\_object\_assign((void\*)&dst->a, (void\*)src->a, 8/\*BLOCK\_FIELD\_IS\_BYREF\*/);
- 对象类型的auto变量（假设变量名叫做p）
  - ✓ \_Block\_object\_assign((void\*)&dst->p, (void\*)src->p, 3/\*BLOCK\_FIELD\_IS\_OBJECT\*/);
- 当block从堆上移除时，都会通过dispose函数来释放它们
- \_\_block变量（假设变量名叫做a）
  - ✓ \_Block\_object\_dispose((void\*)src->a, 8/\*BLOCK\_FIELD\_IS\_BYREF\*/);
- 对象类型的auto变量（假设变量名叫做p）
  - ✓ \_Block\_object\_dispose((void\*)src->p, 3/\*BLOCK\_FIELD\_IS\_OBJECT\*/);

对象	BLOCK_FIELD_IS_OBJECT
__block 变量	BLOCK_FIELD_IS_BYREF

# 被\_\_block修饰的对象类型

- 当\_\_block变量在栈上时，不会对指向的对象产生强引用
- 当\_\_block变量被copy到堆时
  - ✓ 会调用\_\_block变量内部的copy函数
  - ✓ copy函数内部会调用\_Block\_object\_assign函数
  - ✓ \_Block\_object\_assign函数会根据所指向对象的修饰符（\_\_strong、\_\_weak、\_\_unsafe\_unretained）做出相应的操作，形成强引用（retain）或者弱引用（注意：这里仅限于ARC时会retain，MRC时不会retain）
- 如果\_\_block变量从堆上移除
  - ✓ 会调用\_\_block变量内部的dispose函数
  - ✓ dispose函数内部会调用\_Block\_object\_dispose函数
  - ✓ \_Block\_object\_dispose函数会自动释放指向的对象（release）



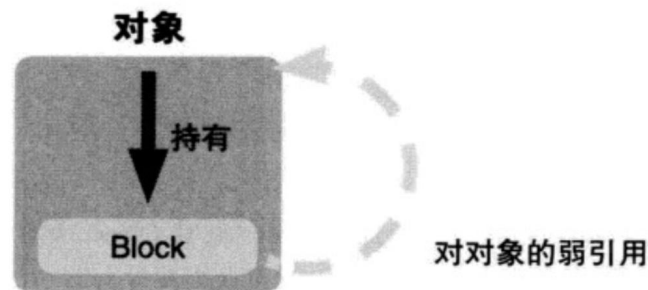


# 解决循环引用问题 - ARC

## ■ 用\_\_weak、\_\_unsafe\_unretained解决

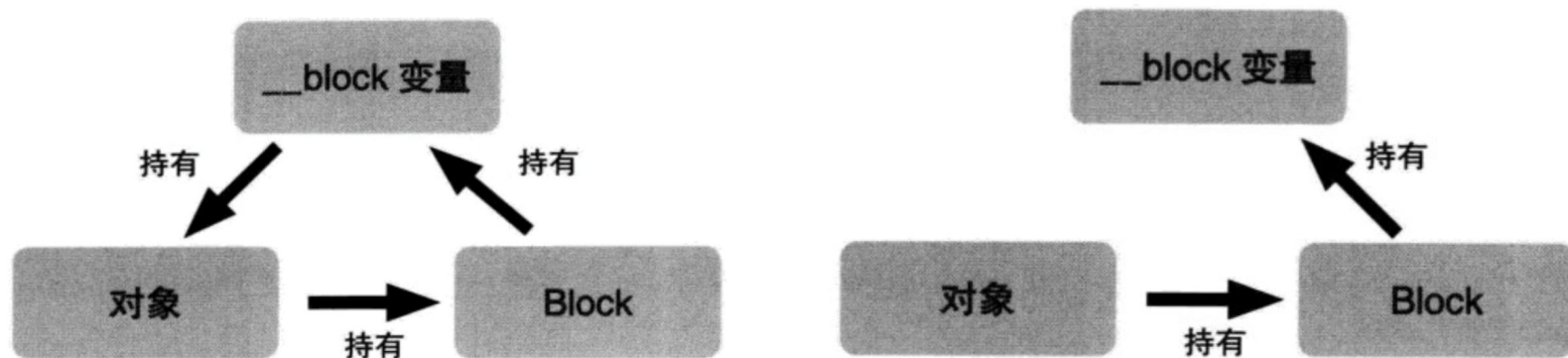
```
__weak typeof(self) weakSelf = self;
self.block = ^{
    printf("%p", weakSelf);
};
```

```
__unsafe_unretained id weakSelf = self;
self.block = ^{
    NSLog(@"%p", weakSelf);
};
```



## ■ 用\_\_block解决（必须要调用block）

```
__block id weakSelf = self;
self.block = ^{
    printf("%p", weakSelf);
    weakSelf = nil;
};
self.block();
```





## ■ 用\_\_unsafe\_unretained解决

```
__unsafe_unretained id weakSelf = self;  
self.block = ^{  
    NSLog(@"%p", weakSelf);  
};
```

## ■ 用\_\_block解决

```
__block id weakSelf = self;  
self.block = ^{  
    printf("%p", weakSelf);  
};
```