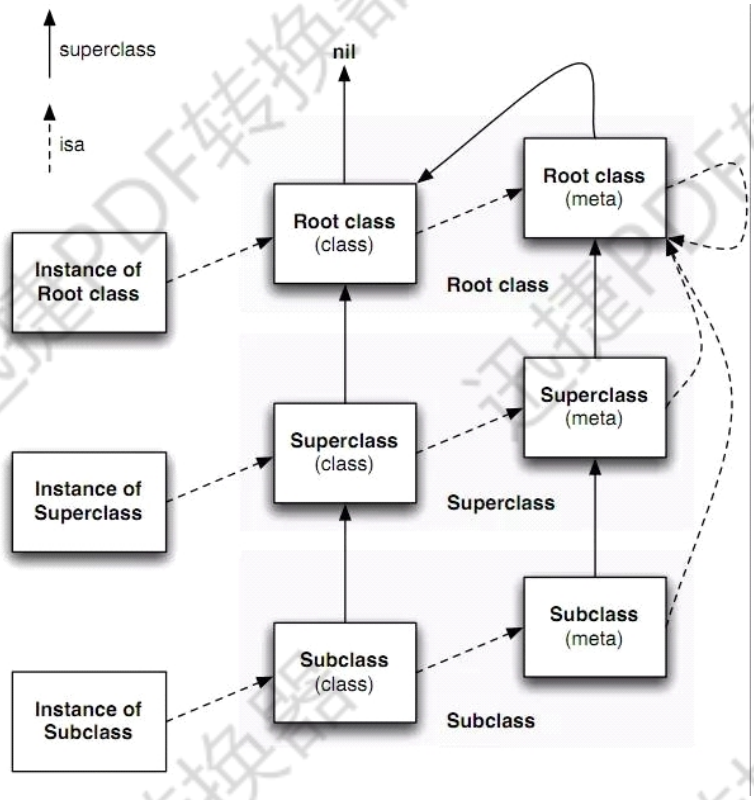
Runtime 面试题

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。

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：

```
// 可读可写
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：