Objective-C Runtime 机制简析

Objective-C 在 C 的基础上添加了面向对象的特性，同时它是一种动态编程语言，将静态语言在编译和链接时需要做的一些事情给延后到运行时执行。例如方法的调用，只有在程序执行的时候，才能具体定位到哪个类的哪个方法。这就需要一个运行时库，就是 Runtime。

1. 类的结构和定义

在 Objective-C 中，类实际上是一个 objc_class 结构体，其定义如下：

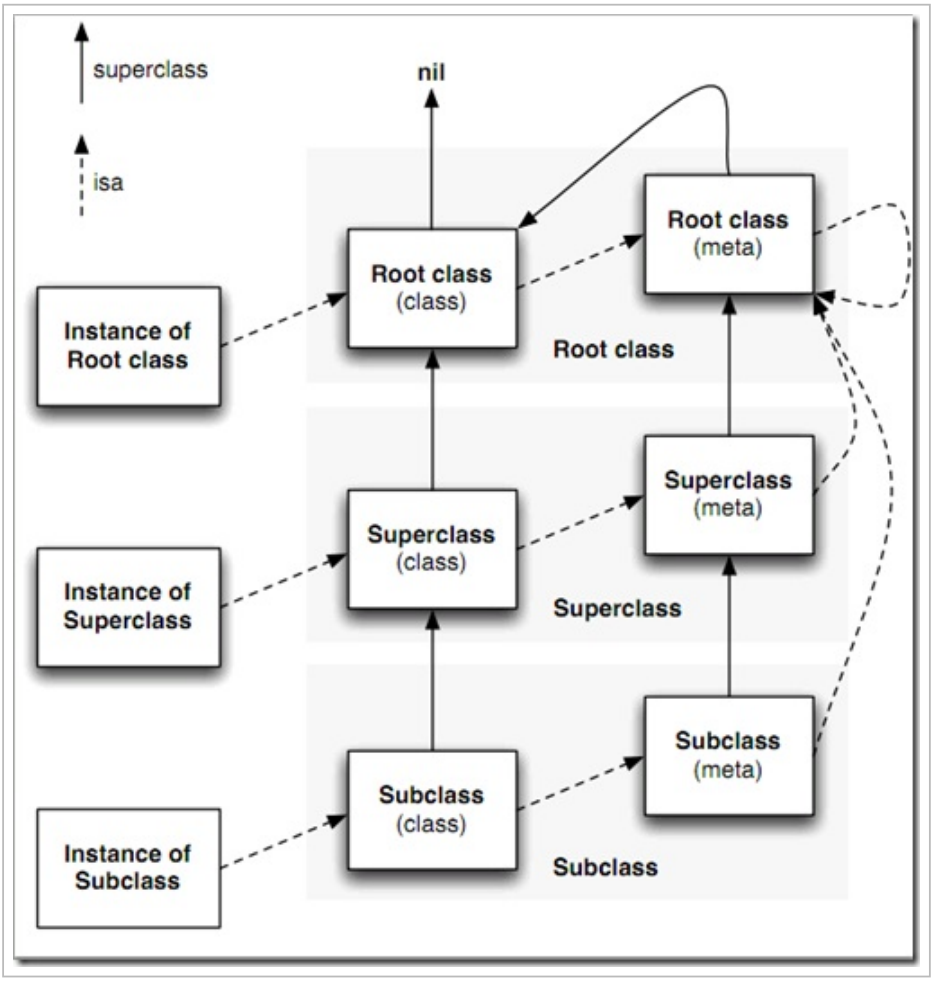
```
    typedef struct objc_class *Class;
struct objc_class {
   Class isa  OBJC_ISA_AVAILABILITY;

#if !__OBJC2__
   Class super_class                                       OBJC2_UNAVAILABLE;
   const char *name                                        OBJC2_UNAVAILABLE;
   long version                                            OBJC2_UNAVAILABLE;
   long info                                               OBJC2_UNAVAILABLE;
   long instance_size                                      OBJC2_UNAVAILABLE;
   struct objc_ivar_list *ivars                            OBJC2_UNAVAILABLE;
   struct objc_method_list **methodLists                   OBJC2_UNAVAILABLE;
   struct objc_cache *cache                                OBJC2_UNAVAILABLE;
   struct objc_protocol_list *protocols                    OBJC2_UNAVAILABLE;
#endif

} OBJC2_UNAVAILABLE;

struct objc_object {
   Class isa  OBJC_ISA_AVAILABILITY;
};
```

可以看到，在 objc2.0 中，除了 isa 指针外，objc_class 的其他成员变量皆已被弃用。其中 isa 是 objc_class 结构体的指针，它指向当前类的 meta class。

- **meta class 与 class**

  在 objc 中，class 存储类的实例方法（-），meta class 存储类的类方法（+），class 的 isa 指针指向 meta class。下文会对此详细介绍。

objc_object 结构体就是 objc 中的对象，它仅包含一个 isa 指针，指向当前对象所属的类。 我们常用的 id 实质上就是一个 objc_object 类型的指针。



如图 1.1 所示，一个对象（Instance of Subclass）的 isa 指针指向它所属的类 Subclass（class），Subclass（class）的 isa 指针指向 Subclass（meta），Subclass（meta）的 isa 指针指向 Root class（meta）。Root class（meta）的 isa 指针指向本身。

同时，Root class（meta）的父类是 Root class（class），即 NSObject，NSObject 的父类为 nil。

2. 方法的调用

在这里需要先了解几个概念

SEL

SEL 是 objc_selector 类型指针，是根据特定规则生成的方法的唯一标识。需要注意的是，只要方法名相同，生成的 SEL 就相同，与这个方法属于哪个类没有关系。

```
typedef struct objc_selector *SEL;
```

IMP

如果说，SEL 是方法名，那么 IMP 就是方法的实现。IMP 指针定义了一个方法的入口，指向了实现方法的代码块的内存地址。

```
typedef id (*IMP)(id, SEL, ...);
```

objc_method

在 objc 中，方法实质上是一个 objc_method 指针。其中，method_name 相当于 objc_method 的 hash 值，runtime 通过 method_name 找到相应的方法入口（method_imp），从而执行方法的代码块。

```
struct objc_method {
SEL method_name                                          OBJC2_UNAVAILABLE;
char *method_types                                       OBJC2_UNAVAILABLE;
IMP method_imp                                           OBJC2_UNAVAILABLE;
}                                                        OBJC2_UNAVAILABLE;
```

调用一个方法时具体做了什么?

在 Objective-C 中，方法的调用采用如下方式:

```
[object methodWithArg:arg];
```

在编译期间，以上代码会被转化为

```
objc_msgSend（object, methodWithArg, arg）
```

可以把它看作是发送消息的过，其中 object 为消息的接收体，它可能是一个对象，也可能是一个类。若为对象，则是实例方法（- 方法）；反之，则是类方法（+ 方法）。
mehodWithArg、arg 是具体的消息内容。
object 接收到消息之后，若是实例方法，则会从其所属的类 Subclass(class) 的 methodLists 去寻找 methodWithArg: 方法。若未找着，则到其父类 Superclass(class) 的 methodLists 中寻找。以此类推，直到根类 NSObject，若仍未找着，就 crash。
同理，若是类方法，则从对象所属类的 meta class 开始寻找。

3. 在 Objective-C 2.0 中的变化

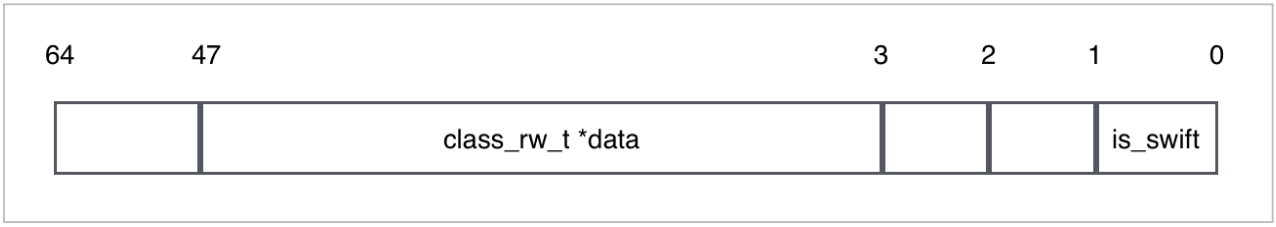
前面提到过在 objc2.0 中，objc_class 只剩下一个 isa 指针。由于 Xcode 对 API 进行了一定的封装，类的信息并未全部对开发者开放。我们不妨通过阅读 Objective-C 2.0 的源码去分析，可以通过 官网 (https://link.jianshu.com?t=https://opensource.apple.com/tarballs/objc4/) 浏览，或者从 github (https://link.jianshu.com?t=https://github.com/opensource-apple/objc4) 上下载源码。
从 objc-runtime-new.h 中可以看到 objc_class 的定义（只截取关键代码，下文同）

```
    struct objc_object {
    isa_t isa;
};
struct objc_class : objc_object {
    // Class ISA;
    Class superclass;
    cache_t cache;             // formerly cache pointer and vtable
    class_data_bits_t bits;    // class_rw_t * plus custom rr/alloc flags

    class_rw_t *data() {
        return bits.data();
    }
};
```

其中，superclass 指向父类，cache 缓存指针、方法入口等，用于提高效率。bits 用于存储类名、类版本号、方法列表、协议列表等信息，替代了 Objective-C1.0 中 methodLists、protocols 等成员变量。

class_data_bits_t 结构体

class_data_bits_t 结构体中只有一个 64 位的指针 bits，它相当于 class_rw_t 指针加上 rr/alloc 等标志位。其中 class_rw_t 指针存在于 4~47 位（从 1 开始计）。




```
    #define FAST_IS_SWIFT         (1UL<<0)
#define FAST_DATA_MASK        0x00007ffffffffff8UL
```

is_swift 标记位标示是否为 swift 的类。通过进行位运算可以得到一个 class_rw_t 类型指针。

class_rw_t 结构体的定义如下

```
    struct class_rw_t {
    uint32_t flags;
    uint32_t version;

    const class_ro_t *ro;

    method_array_t methods;
    property_array_t properties;
    protocol_array_t protocols;

    Class firstSubclass;
};
```

其中 methods 存储方法列表、properties 存储属性列表、protocols 存储协议列表。注意到这里有一个 class_ro_t 类型指针，我们会在下文详细介绍。

dyld 加载镜像

dyld 是 objc 的动态链接库，在程序运行时，会将镜像加载进内存。

- **镜像**

  工程的编译产物，包括一些动态链接库、Foundation 等等，是一些二进制文件。

  在程序初始化方法_objc_init 中注册了两个回调

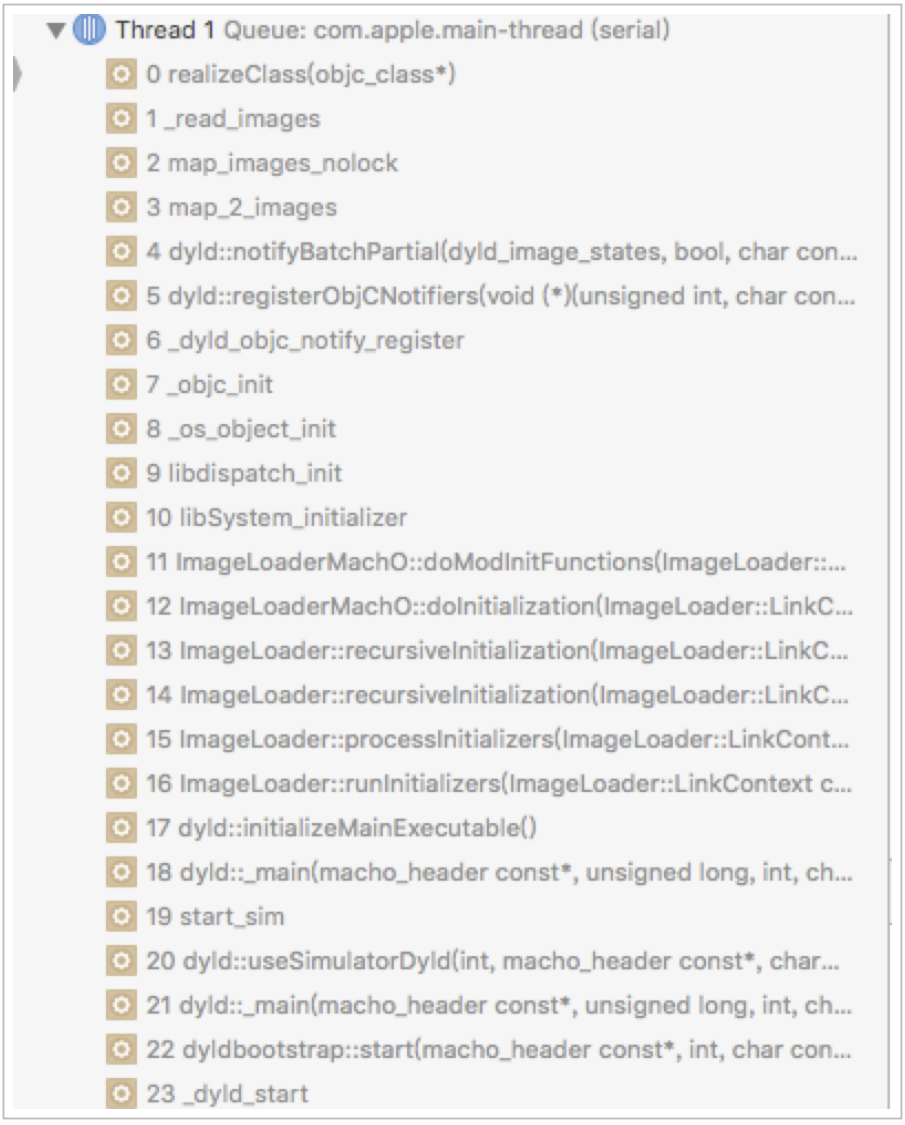
```
    dyld_register_image_state_change_handler(dyld_image_state_bound,1/*batch*/, &map_2_image
s);
 dyld_register_image_state_change_handler(dyld_image_state_dependents_initialized, 0/*not bat
ch*/, &load_images);
```

其中, map_2_images 方法的注释为：Process the given images which are being mapped in by dyld，即处理由 dyld 映射的给定镜像。它的调用如下：

map_2_images → map_images_nolock → _read_images → realizeAllClasses

realizeAllClasses 会完成对镜像中所有类的加载和预处理，它最终会调用 realizeClass 来处理每一个类，而 realizeClass 又通过调用 methodizeClass 来对类结构体的 methods 列表赋值。

可以通过添加符号断点，来直观的查看这几个方法的调用关系，如图 3.2。



+load 方法

+load 方法会在 main 方法之前被调用，所有使用到的类的 load 方法都会被调用。先调用父类的 + load 方法，再调用子类的 + load 方法；先调用主类的 + load 方法，再调用分类的 + load 方法。

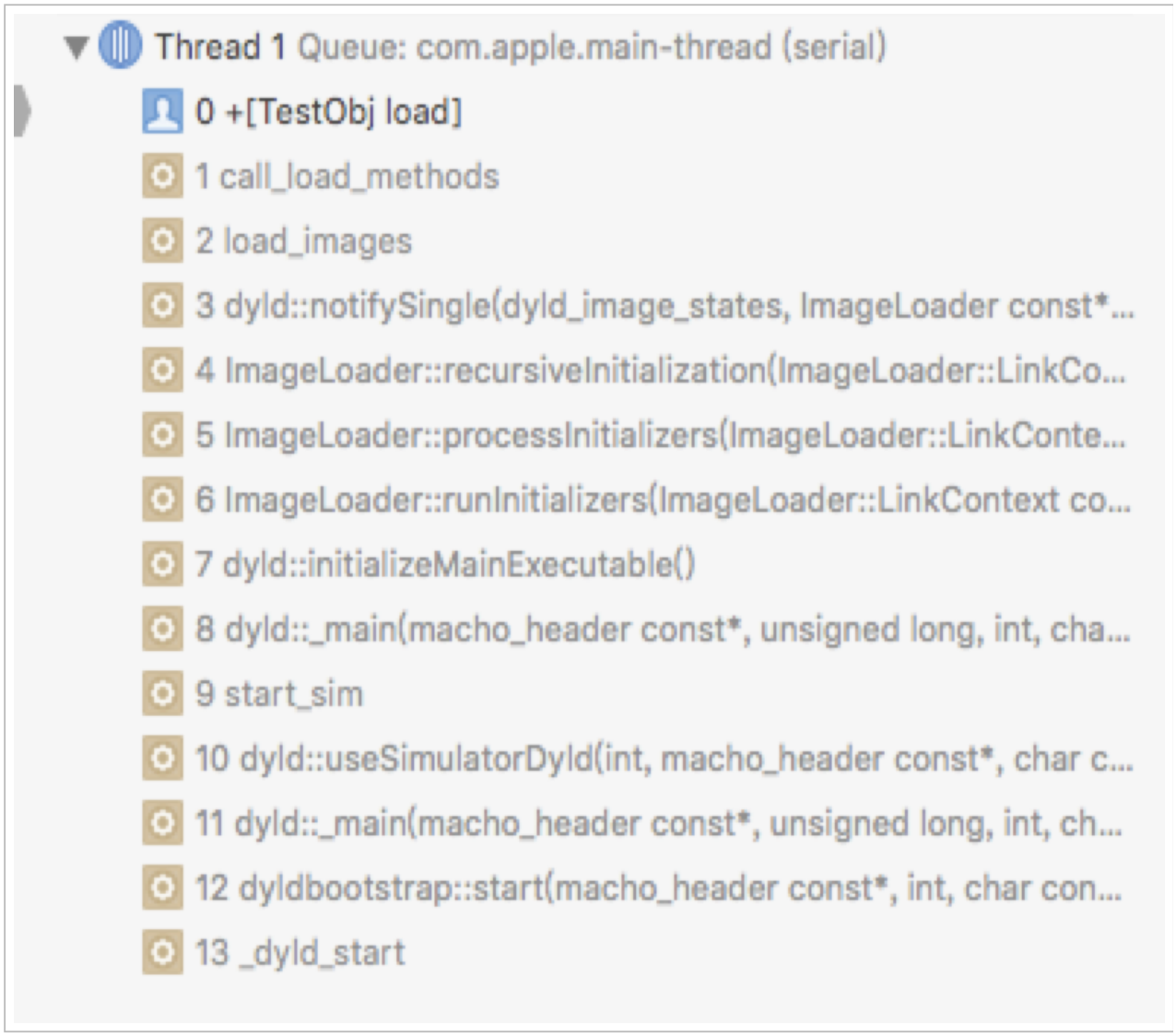


图 3.3 是 + load 方法的调用栈。load_images 方法是每个镜像加载完毕的回调。

```
    const char *
load_images(enum dyld_image_states state, uint32_t infoCount,
            const struct dyld_image_info infoList[])
{
    bool found;

    // Return without taking locks if there are no +load methods here.
    found = false;
    for (uint32_t i = 0; i < infoCount; i++) {
        if (hasLoadMethods((const headerType *)infoList[i].imageLoadAddress)) {
            found = true;
            break;
        }
    }
    if (!found) return nil;

    recursive_mutex_locker_t lock(loadMethodLock);

    // Discover load methods
    {
        rwlock_writer_t lock2(runtimeLock);
        found = load_images_nolock(state, infoCount, infoList);
    }

    // Call +load methods (without runtimeLock - re-entrant)
    if (found) {
        call_load_methods();
    }

    return nil;
}
```

load_Images 会判断镜像是否实现了 + load 方法，并且调用 load_images_nolock 方法找到所有 + load 方法，之后通过 call_load_methods 调用所有的 + load 方法。

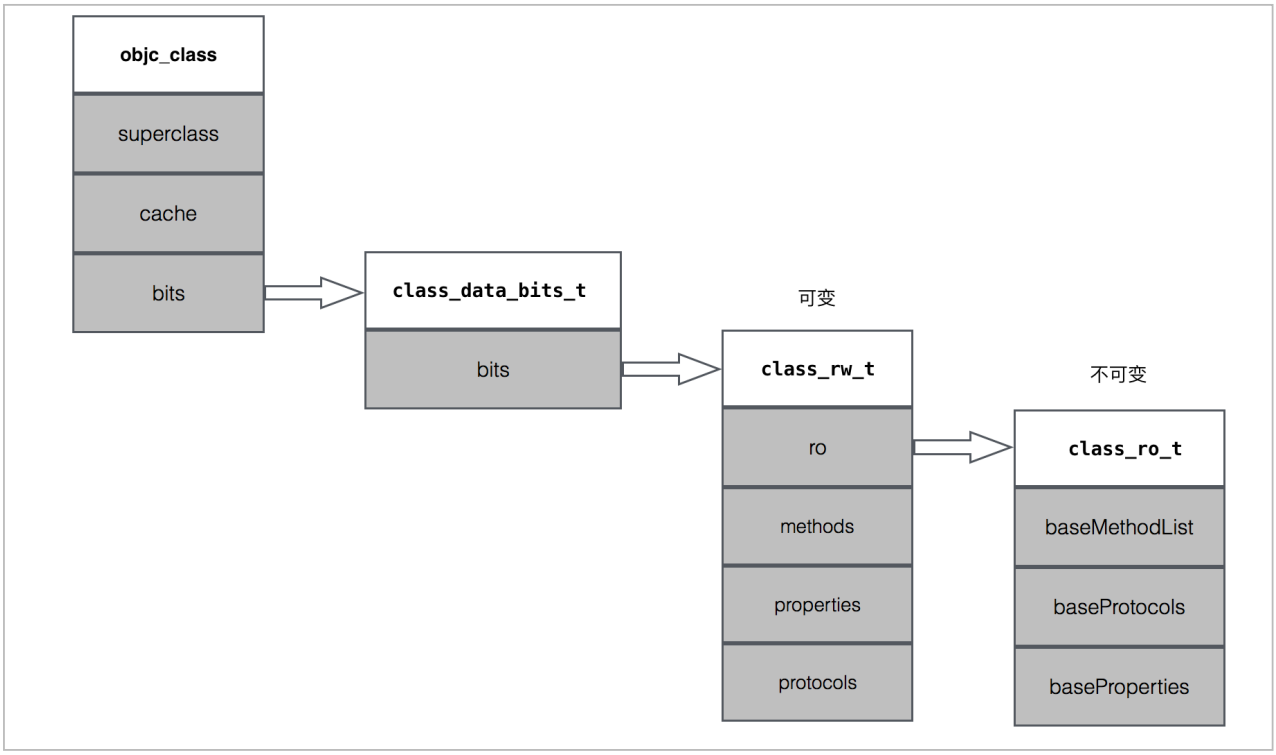
class_ro_t

class_ro_t 与 class_rw_t 的最大区别在于一个是只读的，一个是可读写的，实质上 ro 就是 readonly 的简写，rw 是 readwrite 的简写。

```
struct class_ro_t {
const char * name;
method_list_t * baseMethodList;
protocol_list_t * baseProtocols;
const ivar_list_t * ivars;
};
```

在编译之后，class_ro_t 的 baseMethodList 就已经确定。当镜像加载的时候，methodizeClass 方法会将 baseMethodList 添加到 class_rw_t 的 methods 列表中，之后会遍历 category_list，并将 category 的方法也添加到 methods 列表中。
这里的 category 指的是分类，基于此，category 能扩充一个类的方法。这是开发时经常需要使用到。
class_ro_t 在内存中是不可变的。在运行期间，动态给类添加方法，实质上是更新

class_rw_t 的 methods 列表。
baseProtocols 与 baseMethodList 类似。
objc_object、objc_class、class_rw_t、class_ro_t 的关系如图 3.4。



类的理解与方法的调用

- 对象方法：前面提过，调用对象方法，相当于给对象发送消息，例如 [obj methodWithArg: arg]。 当 obj_object 接收到消息后，通过其 isa 指针找到对应的 objc_class，objc_class 又通过其 data() 方法，查询 class_rw_t 的 methods 列表。若有，则返回；否则，到其父类寻找。以此类推，直到根类，若在根类中仍没有该方法，则 crash。
- 类方法： 在 objc 中，类本身也是一个对象。objc_class 继承自 objc_object，有一个 isa 指针，指向其所属的类，即 meta class。可以这样理解，类是 meta class 的对象。所以，当调用类方法是，例如 [classObj methodWithArg: arg]，classObj 也会通过其 isa 指针到其所属的类（meta class）中寻找。这也就是为什么说，图 1.1 里 class 存储对象方法，meta class 存储类方法。
- meta class 的 isa 指针：meta class 本身也是一个对象，它的 isa 指针指向的也是其所属的类。子 meta class 的 isa 指针指向 NSObjct 的 meta class。 NSObjct 的 meta class 的 isa 指针指向自身。当然，由于苹果进行了封装，在开发中基本不可能直接去使用 meta class。

对象的成员变量寻址

前面提过，在 objc_object 中只有一个 isa 指针。实际上当我们调用 +alloc 方法来初始化一个对象时，也仅仅在内存中生成了一个 objc_object 结构体，并根据其 instanceSize 来分配空间，将其 isa 指针指向所属的类。
类的成员变量 ivar_t 存储在 class_ro_t 中的 ivar_list_t * ivars 中，ivar_t 的定义如下：

```
    struct ivar_t {
    int32_t *offset;
    const char *name;
    const char *type;
    uint32_t size;
}
```

其中 offset 是成员变量相对于对象内存地址的偏移量，正是通过它来完成变量寻址。当我们使用对象的成员变量时，如 myObject.var ，编译器会将其转化为 object_getInstanceVariable(myObject, 'var', **value) 找到其 ivar_t 结构体 ivar，然后调用 object_getIvar(myObject, ivar) 来获取成员变量的内存地址。其计算公式如下：

```
id *location = (id *)((char *)obj + ivar_offset);
```

基于此，虽然多个对象的 isa 指针指向同一个 objc_class，但由于对象的内存地址不一样，所以它们的实例变量存储位置也不一样，从而实现对象与类之间的多对一关系。

全文完