Objective-C Runtime 分析

稍微了解 Mach-O 和 dyld 后，自然抑制不住重新梳理 Objective-C 底层知识的想法。OC 底层知识最核心的部分莫过于 runtime；所以本文的分析对象是 OC 的 runtime。OC runtime 源码是开放的，本文参考的版本是：objc4-750。

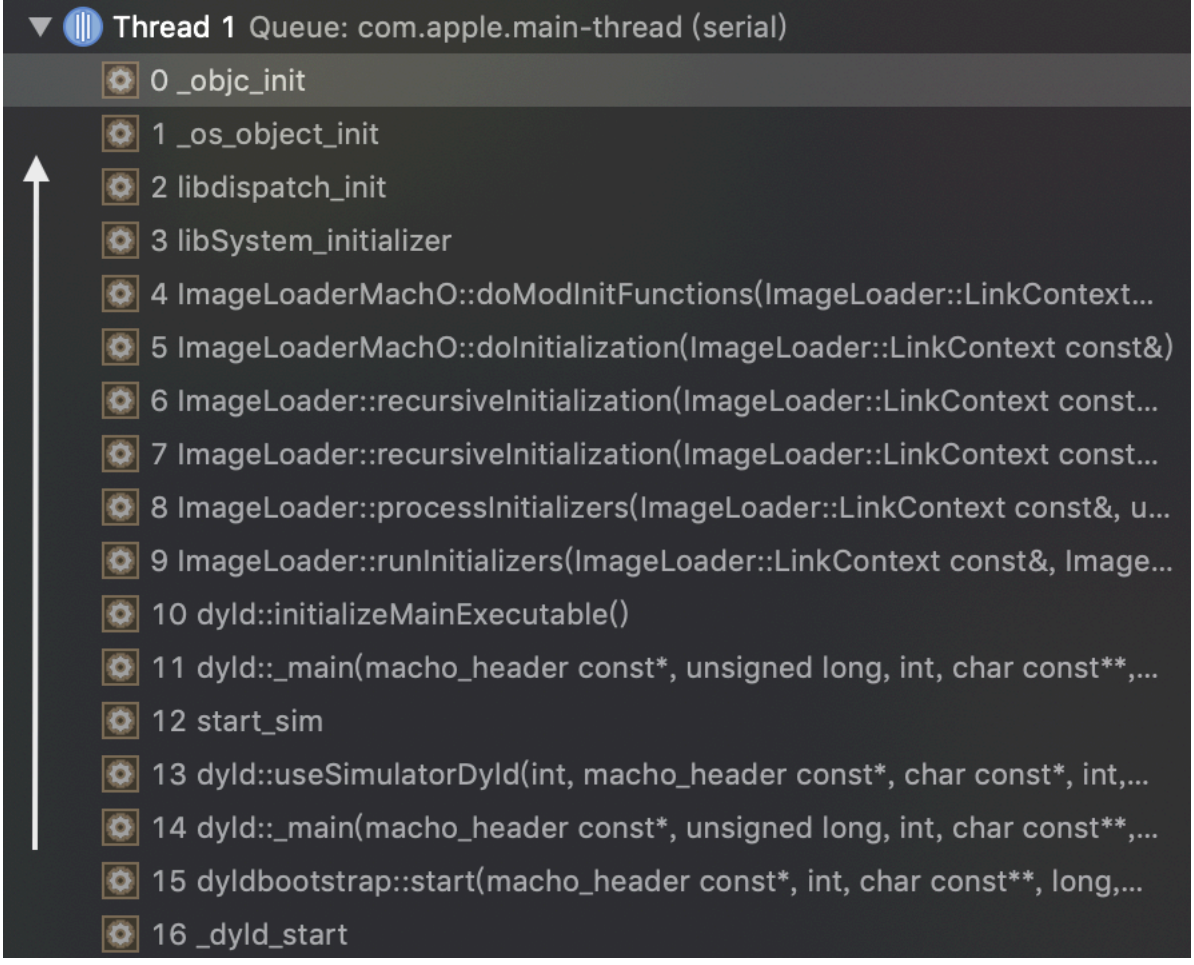
> 在分析过程中，若能将 objc-runtime 跑起来，岂不美哉？RetVal 大神制作了一份可以跑的 objc/runtime 工程：RetVal/objc-runtime。

objc/runtime 值得分析的点非常多，本文的分析主要围绕两个大问题进行：

- `_objc_init`（runtime 的入口）是如何被调用的?
- 类的加载过程是怎么样的?

从 _objc_init 说起

`_objc_init`是 libobjc runtime 的入口函数，通过给 Xcode 工程设置`_objc_init`符号断点可以证明这一点：



上图调用栈中，从 dyld 的入口函数`_dyld_start`开始，经过一系列周转，最终`_objc_init`被调用。

_objc_init 调用过程分析

在开始分析`_objc_init`之前，得搞清楚这么一个问题：`_objc_init`是如何被调用的？为啥抛出这么一个问题，上图调用链不是很明显吗？事实是，上图`libSystem_initializer`的调用逻辑稍微有些令人困惑，因为并没有在源码中找到对它的显式调用。关于这一疑点，没有在任何博客中找到解释。因此，本文略花篇幅围绕源码进行简单分析，所参考的源码包括：

- dyld-635.2
- libdispatch-1008.200.78
- libsystem-1252

- objc4-750

通过分析 libsystem、libdispatch、objc 这几个 library 的源码，很快理清了这么一条调用栈：

```
-> libSystem_initializer (定义于 libSystem 的 init.c)
  -> libdispatch_init (定义于 libdispatch 的 queue.c)
    -> _os_object_init (定义于 libdispatch 的 object.mm)
      -> _objc_init (定义于 libobjc 的 objc-os.mm)
```

而从上文截图的调用栈可以知道，`libSystem_initializer`是在`doModInitFunctions`中被调用的，后者定义于ImageLoaderMachO.cpp。

如上所述，`doModInitFunctions`中并没有显式调用`libSystem_initializer`的逻辑，如下是该函数源码的简化版：

```
void ImageLoaderMachO::doModInitFunctions(const LinkContext
    for (uint32_t i = 0; i < cmd_count; ++i) {
      if ( cmd->cmd == LC_SEGMENT_COMMAND ) {
        for (const struct macho_section* sect=sectionsStart
          const uint8_t type = sect->flags & SECTION_TYPE;
          if ( type == S_MOD_INIT_FUNC_POINTERS ) {
            Initializer* inits = (Initializer*)(sect->addr
            for (size_t j=0; j < count; ++j) {
              Initializer func = inits[j];
              if ( ! dyld::gProcessInfo->libSystemInitializ
                const char* installPath = getInstallPath();
                if ( (installPath == NULL) || (strcmp(insta
                  dyld::throwf("initializer in image (%s) t
```

```
                }
                bool haveLibSystemHelpersBefore = (dyld::gLib
                {
                  func(context.argc, context.argv, context.en
                }
                bool haveLibSystemHelpersAfter = (dyld::gLibS
                if ( !haveLibSystemHelpersBefore && haveLibSy

                  dyld::gProcessInfo->libSystemInitialized =
                }
              }
            }
          }
        }
    }
}
```

这个函数要做的事情很直接，找到镜像中`flags`字段匹配`S_MOD_INIT_FUNC_POINTERS`的 section，这种类型的 section 存储函数指针，dyld 遍历这些函数指针并调用。

不出意外，libSystem.dylib 中就有 flags 为`S_MOD_INIT_FUNC_POINTERS`的 section，可以使用 MachOView 或者 otool 工具查看确认：

```
$ otool -l /usr/lib/libSystem.dylib
...
Section
  sectname __mod_init_func
   segname __DATA
      addr 0x0000000000002218
      size 0x0000000000000008
```

```
    offset 8728
     align 2^3 (8)
    reloff 0
    nreloc 0
     flags 0x00000009
 reserved1 0
 reserved2 0
```

上述 0x09 对应的值即`S_MOD_INIT_FUNC_POINTERS`，顾名思义，它表示该字段存储的是函数指针，这些函数指针在模块（镜像）被加载时调用。

对于 libSystem.dylib 而言，该 section 名为`__mod_init_func`（一般都是这个名字），存储了一个函数指针，该函数指针恰好对应`_libSystem_initializer`符号。当`libSystem_initializer`被调用时，dyld 会对`gProcessInfo->libSystemInitialized`进行标记，表示 libSystem 已经被初始化。

补充说明

如何确保自定义函数被编译到`__mod_init_func`中呢？一种常见的做法是对函数标记`__attribute__((constructor))`。

libSystem 的初始化是一个内部行为，dyld 是如何知道它被初始化的呢？libSystem 是一个特殊的 dylib，默认情况下会被所有可执行文件所依赖，dyld 为它单独提供了一个 API：`_dyld_initializer`，当 libSystem 被初始化时，会调用该函数，进而 dyld 内部就知道了 libSystem 被初始化

了。

OK，搞清楚libSystem_initializer的调用逻辑，强行对_objc_init的调用链分析收个尾。接下来看看_objc_init做了些什么...

_objc_init 逻辑分析

如下是_objc_init的源码，非常短：

```
void _objc_init(void)
{
    static bool initialized = false;
    if (initialized) return;
    initialized = true;


    environ_init();
    tls_init();
    static_init();
    lock_init();
    exception_init();

    _dyld_objc_notify_register(&map_images, load_images, un
}
```

该函数做了两件事，首先进行一系列的初始化，本文不对这些初始化展开（其实也说不太清除）；其次调用_dyld_objc_notify_register注册，它是关键。

_dyld_objc_notify_register是 dyld 为 libobjc 提供的一

个钩子，用来注册 handlers：

```
void _dyld_objc_notify_register(_dyld_objc_notify_mapped
                                _dyld_objc_notify_init
                                _dyld_objc_notify_unmapped
```

当 dyld 完成对镜像的符号绑定时，镜像状态变为`dyld_image_state_bound`， dyld 会针对 OC 镜像，调用 libobjc 注册的 mapped handler，对镜像进行 objc 层次的处理；此后，dyld 对镜像进行初始化，触发 init handler 的调用；而 remove 镜像操作触发 unmapped handler 的调用。

> dyld 如何判断 OC 镜像呢？不同版本的 dyld 处理逻辑不同，本文所参考的 dyld，会根据镜像是否含有`__objc_image_info` section 来判断它是否是 OC 镜像。

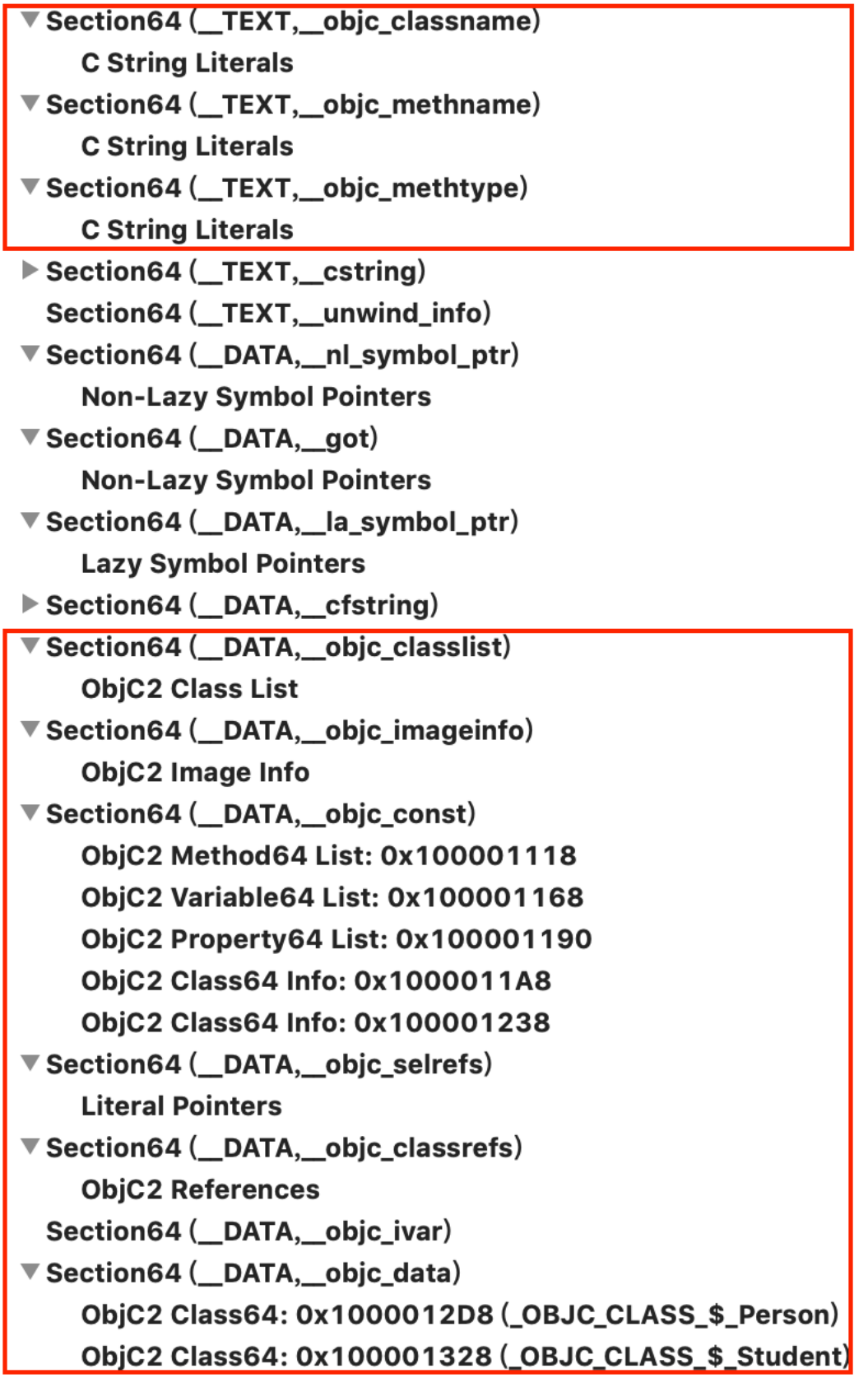
从 objc/runtime 内部来看，mapped 和 init 这俩 handler 分别对应`map_images()`和`load_images()`函数，在`map_images`中，runtime 完成了对镜像的类的构建，它

是本文分析的重点。load_images逻辑相对比较简单，所做的事情主要是调用+load方法。

从 Mach-O 视角看类结构

在分析类的构建过程之前，先站在 Mach-O 的视角窥视静态状态下的类结构。

Mach-O 是如何组织类信息的呢？源码中的类信息是内聚在一起的，类和类元素（方法、成员变量等）有着比较清晰的包含关系，但落实到二进制数据组织上，信息分散得比较厉害，下图截自某个 Mach-O 文件中描述 OC 类信息的片区：



下表对这些 section 简单描述一下：

Section Name	描述
__objc_imageinfo	记录 Objective-C 环境信息等，dyld 用它来判断镜像是否是 objc 镜像
__objc_classlist	记录镜像所定义的类，每个条目都是一个指针，指向到 __objc_data section

__objc_data	存放真正的类数据，和 __objc_classlist 条目呼应
__objc_classname	类名列表
__objc_methodname	方法名列表
__objc_methodtype	方法类型列表
__objc_selrefs	selector 列表信息，每个条目是指向到 __objc_methname 的指针，记录 selector 的名字
__objc_classrefs	类引用列表
__objc_ivar	类的成员变量列表
__objc_const	存放类的元数据，包括：method list、variable list、property list、class info

如何知道一个镜像定义了多少个类？可以从 `__objc_classlist` section 知晓，它是一个列表，每一项是一个指针，对应镜像定义的某个类，指针指向到`__objc_data` section。`__objc_data`中的条目长度是固定的：40 bytes（64 位架构）；在 runtime 阶段，libobjc 根据`__objc_classlist`对`__objc_data` section 进行解析，解析所使用的结构体是`objc_class`。

`objc_class`继承自`objc_object`，如下：

```
struct objc_object {
    isa_t isa;
};

typedef struct objc_object *id;

struct objc_class : objc_object {
```

```
    Class superclass;
    cache_t cache;
    class_data_bits_t bits;
}

typedef struct objc_class *Class;
```

> Runtime 定义了各种各样的数据结构描述类和相关元素，其中部分数据结构是用户可见的，除了objc_object、objc_class，还有：objc_selector、method_t、objc_ivar、objc_property、objc_category，另外很多数据结构于用户而言是不可见的，譬如classref、class_rw_t等。本文只在需要的时候对结构体展开分析，如无必要，一带而过。

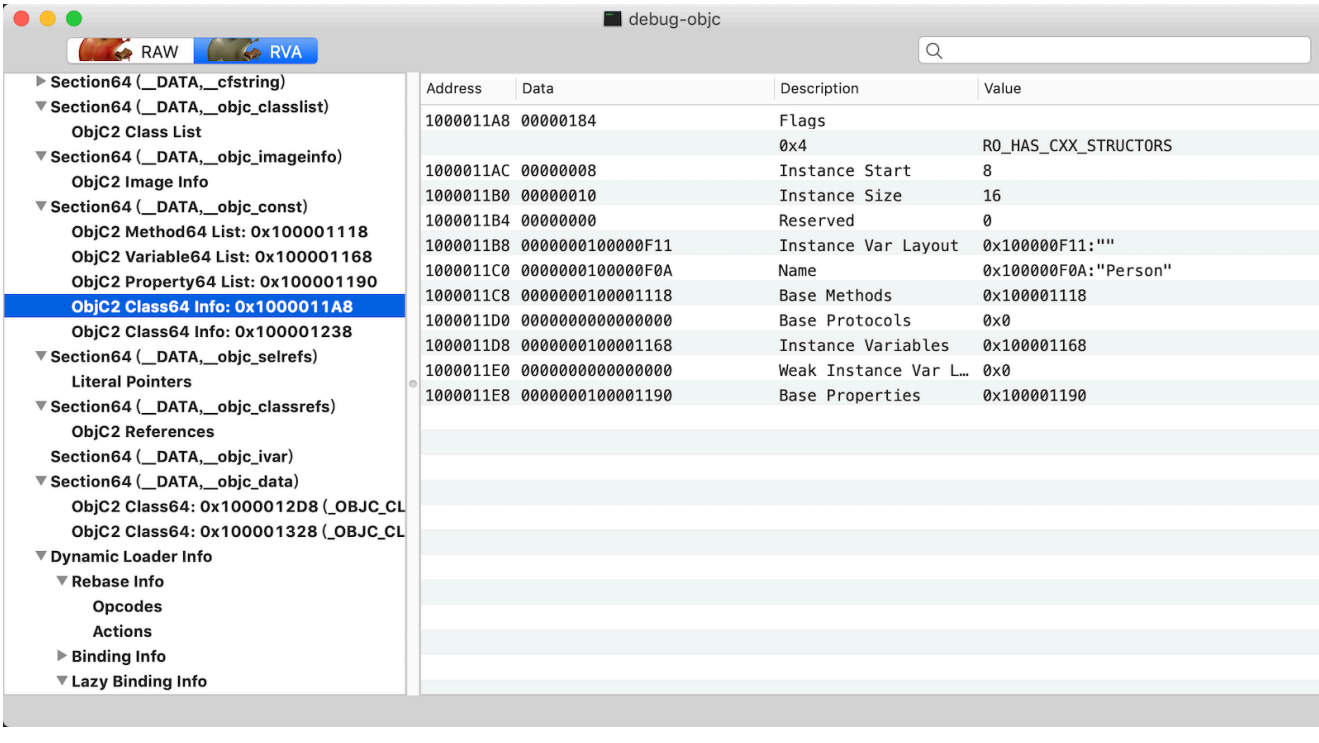
objc_object用来描述 OC 中的实例，当用口语描述实例时，总会说「XX 类的实例 x」或「x 是 XX 的实例」；objc_object的isa在程序结构上表达类似的含义，它指向了该实例所对应的类，类在 runtime 中被描述成objc_class结构。

objc_class继承自objc_object，所以它也有isa指针，指向到它的元类，对于元类而言，类本身也是一个对象；objc_class的superclass成员变量指向该类的父类；isa和superclass这两个成员变量在继承链中扮演着关键作用，满足了类的继承关系的构建，针对它们进行分析的资料太多了，本文不再赘述。cache成员变量与优化

有关，譬如缓存最近命中的方法等。对于bits字段，通过它，可以找到类的其他描述信息，包括类名、方法、成员变量等。

bits的类型class_data_bits_t是一个结构体，只有一个也叫bits的成员变量，对于 64 位机器架构，它有 64 bits，低二位用于描述 Swift 版本，bit[3]描述当前类或者父类是否有默认的retain、release、autorelease、retainCount、_tryRetain、_isDeallocating、retainWeakReference、allowsWeakReference系列方法。bits[3:47]描述的是指向到 class data 的指针，高 17 位是保留区，如下：

初始状态下，class data 的指针指向到镜像的__objc_const区域，该区域的其中一部分存储类的元数据，如下：



Runtime 对上图数据的解析使用class_ro_t结构体，该结构体于用户而言是不可见的，定义如下：

```
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;


};
```

__objc_const的 class info 条目（class_ro_t实例）的成员变量，大部分都是指针，分别描述类名、方法列表、协议列表等，从 Mach-O 视角来看，这些指针指向到__objc_classname、__objc_const method list、__objc_const variable list、__objc_const protocol list 等等，而它们又通过指针将触角伸到其他以__objc为前缀的 section。

综上，可以看到，在编译阶段，编译器就对 OC 类结构的

基本信息进行了整理，只是这些信息比较分散，零散分布到各种以__objc为前缀的 section 中。libobjc 在 runtime 阶段，特别重要的一项工作是：将 Mach-O 中以__objc为前缀的零散 section 信息提取出来进行再加工结构化。

类的加载过程

从函数map_images()开始分析类的加载过程，调用栈如下：

```
map_images
  -> map_images_nolock
    -> _read_images
```

类的主要加载过程在_read_images()完成，做了如下事情：

- discover classes. 即从镜像提取类信息，并存到名为allocatedClasses的全局 hash table 中
- remap classes. 重新调整类之间的引用
- fix up selector references. 提取方法，并注册到名为namedSelectors的全局 map table 中
- fix up objc_msgSend_fixup
- discover protocols. 提取 protocols，存储到全局 map table
- fix up @protocol references. 和类一样，protocol 也有继承关系，此过程 fixup 它们的依赖关系
- realize non-lazy classes. realize 含有+load方法或者静态实例的类

- realize future classes. realize 含有`RO_FUTURE`标识的类，这些类一般是 Core Foundation 中的类
- discover categories. 提取 categories，存储到全局 map table

上述过程最值得研究的莫过于 realize。

笔者并没有找到权威的资料来解释 realize 的含义，只能根据对源码的理解自说自话。OC 类在被使用之前（譬如调用类方法），需要进行一系列的初始化，譬如：指定 superclass、指定 isa 指针、attach categories 等等；libobjc 在 runtime 阶段就可以做这些事情，但是有些过于浪费，更好的选择是懒处理，这一举措极大优化了程序的执行速度。而 runtime 把对类的这些惰性初始化过程称为「realize」。

对于用户定义的大部分类的 realize 处理，是在类第一次被使用时进行的；但对于某些类，需要在 runtime 阶段就进行 realize 处理；其一是 non-lazy classes，包括含有`+load`方法的类，以及含有静态实例（譬如单例）的类；其二是 future classes，笔者也不太理解这个概念，但貌似是为 Core Foundation 中定义的类服务的。

接下来的重点是分析 realize 做了哪些事情，realize class 的主要逻辑发生在`realizeClass()`中，如下是其逻辑代码的简化版本：

```
static Class realizeClass(Class cls)
{
```

```
    rw = (class_rw_t *)calloc(sizeof(class_rw_t), 1);
    rw->ro = ro;
    rw->flags = RW_REALIZED|RW_REALIZING;
    cls->setData(rw);

    isMeta = ro->flags & RO_META;


    cls->chooseClassArrayIndex();


    supercls = realizeClass(remapClass(cls->superclass));
    metacls = realizeClass(remapClass(cls->ISA()));
    cls->superclass = supercls;
    cls->initClassIsa(metacls);


    if (supercls) {
        addSubclass(supercls, cls);
    } else {
        addRootClass(cls);
    }


    methodizeClass(cls);

    return cls;
}
```

关键代码是：

```
rw = (class_rw_t *)calloc(sizeof(class_rw_t), 1);

rw->ro = ro;

rw->flags = RW_REALIZED|RW_REALIZING;

cls->setData(rw);
```

前文站在 Mach-O 的视角分析了类的结构，我们得知，objc_class 的 data 指针最开始指向`class_ro_t`结构体，它包含了类名、方法列表、属性列表、协议列表等信息；但在 realize 逻辑中，libobjc 创建了一个`class_rw_t`结构体，并将 data 指针指向到该结构体。

如何看待 libobjc 的这个设计呢？可以这么理解，`class_ro_t`包含的类信息（方法、属性、协议等）都是在编译期就可以确定的，暂且称为元信息吧，在之后的逻辑中，它们显然是不希望被改变的；后续在用户层，无论是方法还是别的扩展，都是在`class_rw_t`上进行操作，这些操作都不会影响类的元信息。

`class_rw_t`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;
    Class nextSiblingClass;

    char *demangledName;
    uint32_t index;
}
```

realize 作为 objc runtime 中类的懒加载机制，在类真正要使用时再去做相应的准备工作，为确保程序的快速启动发挥了很大的作用；利用已经被 realize 的类含有`RW_REALIZED`和`RW_REALIZING`标记的特点，可以为项目找出无用类：因为没有被使用的类，一定没有被 realized。

+load

当 dyld 完成对 OC 镜像初始化时，会调用`load_images()`，它所做的工作比较简单，找出所有`+load`方法并调用：

```
load_images(const char *path __unused, const struct mach_he
{

    prepare_load_methods((const headerType *)mh);


    call_load_methods();
}
```

prepare_load_methods()遍历找出类的+load方法，并把它们添加到全局数组loadable_classes中；并找出 category 的+load方法，把它们添加到全局数组 loadable_categories中。

```
static void schedule_class_load(Class cls)
{

    schedule_class_load(cls->superclass);

    add_class_to_loadable_list(cls);
    cls->setInfo(RW_LOADED);
}

void prepare_load_methods(const headerType *mhdr)
{
    classref_t *classlist =  _getObjc2NonlazyClassList(mhdr
    for (i = 0; i < count; i++) {
        schedule_class_load(remapClass(classlist[i]));
    }


    category_t **categorylist = _getObjc2NonlazyCategoryLis
    for (i = 0; i < count; i++) {
        category_t *cat = categorylist[i];
        Class cls = remapClass(cat->cls);
        realizeClass(cls);
        add_category_to_loadable_list(cat);
    }
}

void call_load_methods(void)
```

```
{
    bool more_categories;

    do {

        while (loadable_classes_used > 0) {
            call_class_loads();
        }


        more_categories = call_category_loads();


    } while (loadable_classes_used > 0  ||  more_categories
}
```

从这段代码，可以梳理出我们所熟悉的+load调用顺序：

- 父类的+load一定比子类的+load早被调用
- 主类的+load一定比分类的+load早被调用

+load的源码相对较简单，左神的博客你真的了解 load 方法么？有更丰富的分析...

更多阅读

- 你真的了解 load 方法么？
- 深入解析 ObjC 中方法的结构
- Understanding the Objective-C Runtime