Mach-O
Mach-O文件结构
- header
- MH_PIE 随机地址空间
- dyld
- 二级命名空间
- Load commands
- Segment load command
- Section header
- Data
- __TEXT段
- __DATA段

OC之源起
Mach-O中OC相关的Segment Section
__TEXT & OC
__objc_classname
__objc_methname
__objc_methtype
__DATA & OC
- __objc_imageinfo
- __objc_classlist
- __objc_catlist
- __objc_protolist
- __objc_classrefs
- __objc_selrefs
- __objc_superrefs
- __objc_const

前言
什么是runtime?
与Runtime交互
从NSObject说起
- BUT！！！

objc_class
realizeClass
objc_object
objc_object & objc_class
id
消息相关数据结构
- SEL
- method_t
- IMP
- 消息

objc_msgSend
lookUpImpOrForward
类调用实例方法
objc_msgSendSuper
动态解析

Mach-O

Mach-O是Mach Object文件格式的缩写。它是用于可执行文件，动态库，目标代码的文件格式。作为a.out格式的替代，Mach-O格式提供了更强的扩展性，以及更快的符号表信息访问速度。

Mach-O格式为大部分基于Mach内核的操作系统所使用的，包括NeXTSTEP, Mac OS X和iOS，它们都以Mach-O格式作为其可执行文件，动态库，目标代码的文件格式。

具体到我们的iOS程序，当用XCode打包后，会生成一个.app为扩展名的文件（位于工程目录/Products文件夹下），其实.app是一个文件夹，我们用鼠标右键选择'Show Package contents'，就可以查看文件夹的内容，其中会发现有一个和我们工程同名的unix 可执行文件，这个就是iOS可执行文件，它是符合Mach-O格式的。

Mach-O文件结构

Mach-O格式如下图所示，它被分为**header**， **load commands**， **data**三大部分：

header：对Mach-O文件的一个概要说明，包括Magic Number, 支持的CPU类型等。

load commands： 当系统加载Mach-O文件时，load command会指导苹果的动态加载器(dyld)或内核，该如何加载文件的Data数据。

data： Mach-O文件的数据区，包含代码和数据。其中包含若干Segment块（注意，除了Segment块之外，还有别的内容，包括code signature，符号表之类，不要被苹果的图所误导！），每个Segment块中包含0个或多个seciton。Segment根据对应的load command被dyld加载入内存中。

我们可以使用 MachOView（一个查看MachO 格式文件信息的开源工具）工具来查看一个具体的文件的Mach-O格式。

header

我们以一个普通的iOS APP为例，看看Mach-O文件header部分的具体内容。通过MachOView打开可执行文件，可以看到header的结构：

是不是有些懵？下面我们就结合Darwin内核源码，来了解下Mach header的定义。

Mach header的定义位于Darwin源码中的 EXTERNAL_HEADERS/mach-o/loader.h 中：

32位：

```
struct mach_header {
  uint32_t  magic;    /* mach magic number identifier */
  cpu_type_t  cputype;  /* cpu specifier */
  cpu_subtype_t cpusubtype; /* machine specifier */
  uint32_t  filetype; /* type of file */
  uint32_t  ncmds;    /* number of load commands */
  uint32_t  sizeofcmds; /* the size of all the load commands */
  uint32_t  flags;    /* flags */
};
```

64位：

```
struct mach_header_64 {
  uint32_t  magic;    /* mach magic number identifier */
  cpu_type_t  cputype;  /* cpu specifier */
  cpu_subtype_t cpusubtype; /* machine specifier */
  uint32_t  filetype; /* type of file */
  uint32_t  ncmds;    /* number of load commands */
  uint32_t  sizeofcmds; /* the size of all the load commands */
  uint32_t  flags;    /* flags */
  uint32_t  reserved; /* reserved */
};
```

可以看到，32位和64位的Mach header基本相同，只不过64位header中多了一个保留参数reserved。

- magic：魔数，用来标识这是一个Mach-O文件，有32位和64位两个版本：

```
#define MH_MAGIC  0xfeedface  /* the mach magic number */
#define MH_MAGIC_64 0xfeedfacf /* the 64-bit mach magic number */
```

- cputype：支持的CUP架构类型，如arm。
- cpusubtype：在支持的CUP架构类型下，所支持的具体机器型号。在我们的例子中，APP是支持所有arm64的机型的:CUP_SUBTYPE_ARM64_ALL
- filetype: Mach-O的文件类型。包括：

```
#define MH_OBJECT 0x1   /* Target 文件：编译器对源码编译后得到的中间结果 */
#define MH_EXECUTE  0x2   /* 可执行二进制文件 */
#define MH_FVMLIB 0x3   /* VM 共享库文件（还不清楚是什么东西） */
#define MH_CORE   0x4   /* Core 文件，一般在 App Crash 产生 */
#define MH_PRELOAD  0x5   /* preloaded executable file */
#define MH_DYLIB  0x6   /* 动态库 */
#define MH_DYLINKER 0x7   /* 动态连接器 /usr/lib/dyld */
#define MH_BUNDLE 0x8   /* 非独立的二进制文件，往往通过 gcc-bundle 生成 */
#define MH_DYLIB_STUB 0x9   /* 静态链接文件（还不清楚是什么东西） */
#define MH_DSYM   0xa   /* 符号文件以及调试信息，在解析堆栈符号中常用 */
#define MH_KEXT_BUNDLE  0xb   /* x86_64 内核扩展 */
```

- ncmds：load command的数量
- sizeofcmds: 所有load command的大小
- flags： Mach-O文件的标志位。主要作用是告诉系统该如何加载这个Mach-O文件以及该文件的一些特性。有很多值，我们取常见的几种：

```
#define MH_NOUNDEFS 0x1   /* Target 文件中没有带未定义的符号，常为静态二进制文件 */
#define MH_SPLIT_SEGS 0x20  /* Target 文件中的只读 Segment 和可读写 Segment 分开
*/
#define MH_TWOLEVEL 0x80    /* 该 Image 使用二级命名空间(two name space binding)
绑定方案 */
#define MH_FORCE_FLAT 0x100 /* 使用扁平命名空间(flat name space binding)绑定（与
MH_TWOLEVEL 互斥） */
#define MH_WEAK_DEFINES 0x8000 /* 二进制文件使用了弱符号 */
#define MH_BINDS_TO_WEAK 0x10000 /* 二进制文件链接了弱符号 */
#define MH_ALLOW_STACK_EXECUTION 0x20000/* 允许 Stack 可执行 */
#define MH_PIE 0x200000  /* 加载程序在随机的地址空间，只在 MH_EXECUTE中使用 */
#define MH_NO_HEAP_EXECUTION 0x1000000 /* 将 Heap 标记为不可执行，可防止 heap
spray 攻击 */
```

MH_PIE 随机地址空间

每次系统加载进程后，都会为其随机分配一个虚拟内存空间。 在传统系统中，进程每次加载的虚拟内存是相同的。这就让黑客有可能篡改内存来破解软件。

dyld

dyld是苹果公司的动态链接库，用来把Mach-O文件加载入内存。

二级命名空间

表示其符号空间中还会包含所在库的信息。这样可以使得不同的库导出通用的符号。与其相对的是扁平命名空间。

Load commands

load commands 紧跟在header之后，用来告诉内核和dyld，如何将各个Segment加载入内存中。
load command被源码表示为struct，有若干种load command，但是共同的特点是，在其结构的开头处，必须是如下两个属性：

```
struct load_command {
  uint32_t cmd;   /* type of load command */
  uint32_t cmdsize; /* total size of command in bytes */
};
```

苹果为cmd定义了若干的宏，用来表示cmd的类型，下面列举出几种：

```
// 描述该如何将32或64位的segment 加载入内存，对应segment command类型
#define LC_SEGMENT  0x1
#define LC_SEGMENT_64 0x19
// UUID，2进制文件的唯一标识符
#define LC_UUID   0x1b
// 启动动态加载器dyld
#define LC_LOAD_DYLINKER 0xe
```

Segment load command

在这么多的load command中，需要我们重点关注的是segment load command。segment command解释了该如何将Data中的各个Segment加载入内存中，而和我们APP相关的逻辑及数据，则大部分位于各个Segment中。

而和我们的Run time相关的Segment，则位于__DATA类型Segment下。

Segment load command分为32位和64位：

```
struct segment_command { /* for 32-bit architectures */
  uint32_t  cmd;    /* LC_SEGMENT */
  uint32_t  cmdsize;  /* includes sizeof section structs */
  char    segname[16];  /* segment name */
  uint32_t  vmaddr;   /* memory address of this segment */
  uint32_t  vmsize;   /* memory size of this segment */
  uint32_t  fileoff;  /* file offset of this segment */
  uint32_t  filesize; /* amount to map from the file */
  vm_prot_t maxprot;  /* maximum VM protection */
  vm_prot_t initprot; /* initial VM protection */
  uint32_t  nsects;   /* number of sections in segment */
  uint32_t  flags;    /* flags */
};

struct segment_command_64 { /* for 64-bit architectures */
  uint32_t  cmd;    /* LC_SEGMENT_64 */
  uint32_t  cmdsize;  /* includes sizeof section_64 structs */
  char    segname[16];  /* segment name */
  uint64_t  vmaddr;   /* memory address of this segment */
```

```
  uint64_t  vmsize;   /* memory size of this segment */
  uint64_t  fileoff;  /* file offset of this segment */
  uint64_t  filesize; /* amount to map from the file */
  vm_prot_t maxprot;  /* maximum VM protection */
  vm_prot_t initprot; /* initial VM protection */
  uint32_t  nsects;   /* number of sections in segment */
  uint32_t  flags;    /* flags */
};
```

32位和64位的Segment load command基本类似，只不过在64位的结构中，把和寻址相关的数据类型，由32位的uint32_t改为了64位的uint64_t类型。

结构体的定义，看注释基本能够看懂，就是maxprot， initprot不太明白啥意思。

这里介绍一个特殊的'Segment'，叫做__PAGEZERO Segment。 这里说它特殊，是因为这个Segment其实是苹果虚拟出来的，只是一个逻辑上的段，而在Data中，根本没有对应的内容，也没有占用任何硬盘空间。

__PAGEZERO Segment在VM中被置为Read only，逻辑上占用APP最开始的4GB空间，用来处理空指针。

我们用MachOV点开**PAGEZERO Segment所对应的Segment load command，LC_SEGMENT_64(**PAGEZERO):

可以看到其vm size是4GB，但其真正的物理地址File size和offset都是0。

Section header

在Data中，程序的逻辑和数据是按照Segment（段）存储，在Segment中，又分为0或多个section，每个section中在存储实际的内容。而之所以这么做的原因在于，在section中，可以不用内存对齐达到节约内存的作用，而所有的section作为整体的Segment，又可以整体的内存对齐。

在Mach-O文件中，每一个Segment load command下面，都会包含对应Segment 下所有section的header。 section header的定义如下：

```
struct section { /* for 32-bit architectures */
  char    sectname[16]; /* name of this section */
  char    segname[16];  /* segment this section goes in */
  uint32_t  addr;   /* memory address of this section */
  uint32_t  size;   /* size in bytes of this section */
  uint32_t  offset;   /* file offset of this section */
  uint32_t  align;    /* section alignment (power of 2) */
  uint32_t  reloff;   /* file offset of relocation entries */
  uint32_t  nreloc;   /* number of relocation entries */
  uint32_t  flags;    /* flags (section type and attributes)*/
  uint32_t  reserved1;  /* reserved (for offset or index) */
  uint32_t  reserved2;  /* reserved (for count or sizeof) */
};
```

```
struct section_64 { /* for 64-bit architectures */
  char    sectname[16]; /* name of this section */
  char    segname[16];  /* segment this section goes in */
  uint64_t  addr;   /* memory address of this section */
  uint64_t  size;   /* size in bytes of this section */
  uint32_t  offset;   /* file offset of this section */
  uint32_t  align;    /* section alignment (power of 2) */
  uint32_t  reloff;   /* file offset of relocation entries */
  uint32_t  nreloc;   /* number of relocation entries */
  uint32_t  flags;    /* flags (section type and attributes)*/
  uint32_t  reserved1;  /* reserved (for offset or index) */
  uint32_t  reserved2;  /* reserved (for count or sizeof) */
  uint32_t  reserved3;  /* reserved */
};
```

这样，关于load commonds部分，其真正的结构其实和苹果提供的图片有些许的差异：

Data

Mach-O的Data部分，其实是真正存储APP 二进制数据的地方，前面的header和load command，仅是提供文件的说明以及加载信息的功能。

Data部分也被分为若干的部分，除了我们前面提到的Segment外，还包括符号表，代码签名，动态加载器信息等。

而程序的逻辑和数据，则是放在以Segment分割的Data部分中的。我们在这里，仅关心Data中的Segment的部分。

Segment根据内容的不同，分为若干类型，类型名称均是以“双下划线+大写英文”表示，有的Segment下面还会包含若干的section，section的命名是以"双下划线+小写英文"表示。

先来看Segment，Mach-O中有如下几种Segment：

```
#define SEG_PAGEZERO  "__PAGEZERO" /* 当时 MH_EXECUTE 文件时，表示空指针区域 */
#define SEG_TEXT  "__TEXT" /* 代码/只读数据段 */
#define SEG_DATA  "__DATA" /* 数据段 */
#define SEG_OBJC  "__OBJC" /* Objective-C runtime 段 */
#define SEG_LINKEDIT  "__LINKEDIT" /* 包含需要被动态链接器使用的符号和其他表，包括符号表、字符串表等 */
```

这里面注意到到SEG_OBJC，是和OC的runtime相关的。在OC 2.0中已经废弃掉**OBJC段，而是将其放入到了**DATA段中以__objc开头的section中。这些和runtime相关的sections是本文的重点，我们稍后再分析。我们先看看其他的段。

__TEXT段

TEXT是程序的只读段，用于保存我们所写的代码和字符串常量，const修饰常量等。 下面是TEXT段下常见的section：

Section	用途
TEXT.text	主程序代码
TEXT.cstring	C 语言字符串
TEXT.const	const 关键字修饰的常量
TEXT.stubs	用于 Stub 的占位代码，很多地方称之为桩代码。
TEXT.stubs_helper	当 Stub 无法找到真正的符号地址后的最终指向
TEXT.objc_methname	Objective-C 方法名称
TEXT.objc_methtype	Objective-C 方法类型
TEXT.objc_classname	Objective-C 类名称

例如，我们点击**TEXT.**objc_classname, 会看到我们程序中所使用到的类的名称：

而在**TEXT.**cstring section中，则看到我们定义的字符串常量（如@“I’m a cat!! miao miao”）：

值得注意的是，这些都是以明文形式展现的。如果我们将加密key用字符串常量或宏定义的形式存储在程序中，可以想象其安全性是得不到保障的。

__DATA段

DATA段用于存储程序中所定义的数据，可读写。DATA段下常见的sectin有：

Section	用途
DATA.data	初始化过的可变数据
DATA.la_symbol_ptr	lazy binding 的指针表，表中的指针一开始都指向 __stub_helper
__DATA.nl_symbol_ptr	非 lazy binding 的指针表，每个表项中的指针都指向一个在装载过程中，被动态链机器搜索完成的符号
DATA.const	没有初始化过的常量
DATA.cfstring	程序中使用的 Core Foundation 字符串（CFStringRefs）
DATA.bss	BSS，存放为初始化的全局变量，即常说的静态内存分配
DATA.common	没有初始化过的符号声明
DATA.objc_classlist	Objective-C 类列表
DATA.objc_protolist	Objective-C 原型
DATA.objc_imginfo	Objective-C 镜像信息
DATA.objc_selfrefs	Objective-C self 引用
DATA.objc_protorefs	Objective-C 原型引用
DATA.objc_superrefs	Objective-C 超类引用

可见，在**DATA段下，有许多以**objc开头的section，而这些section，均是和runtime的加载有关的。

我们将在后续的文章中，继续探讨这些section和runtime的关系。

总结

这次我们一起了解了XNU内核下的二进制文件格式Mach-O。它由header，load command以及data三部分组成:

我们重点应该了解的应该是data部分，因为这里存储着我们程序真正的数据和代码。 在data部分中，又区分为以Segment划分的部分以及代码签名等其他部分。 在Segment下，有区分有若干的section。常用的Segment有**PAGE_ZERO,** TEXT, **DATA(注意区分Mach-O的data和这里的**DATA段名称)。

在前传1中，我们分析了解了XNU内核所支持的二进制文件格式Mach-O。同时还留了一个小尾巴，就是Mach-O文件中和Objective-C以及runtime相关的Segment section。今天，就来了解一下它们。

OC之源起

我们知道，程序的入口点在iOS中被称之为main函数：

```
int main(int argc, char * argv[]) {
    NSString * appDelegateClassName;
    @autoreleasepool {
        // Setup code that might create autoreleased objects goes here.
        appDelegateClassName = NSStringFromClass([AppDelegate class]);
    }
    return UIApplicationMain(argc, argv, nil, appDelegateClassName);
}
```

我们所写的所有代码，它的执行的第一步，均是由main函数开始的。

但其实，在程序进入main函数之前，内核已经为我们的程序加载和运行做了许多的事情。

当我们设置符号断点`_objc_init`，可以看到如下调用堆栈信息，这些函数都是在main函数调用前，被系统调用的：

`_objc_init`是Object-C runtime的入口函数，在这里面主要功能是读取Mach-O文件OC对应的Segment seciton，并根据其中的数据代码信息，完成为OC的内存布局，以及初始化runtime相关的数据结构。

我们可以看到，`_objc_init`是被`_dyld_start`所调用起来的，`_dyld_start`是dyld的bootstrap方法，最终调用到了`_objc_init`。

dyld是苹果的动态加载器，用来加载image（注意这里image不是指图片，而是Mach -O格式的二进制文件）。

当程序启动时，系统内核首先会加载dyld， 而dyld会将我们APP所依赖的各种库加载到内存空间中，其中就包括libobjc库（OC和runtime）, 这些工作，是在APP的main函数执行前完成的。

在`_objc_init`方法中，会注册监听来自dlyd的以下事件：

```
   // Register for unmap first, in case some +load unmaps something
    _dyld_register_func_for_remove_image(&unmap_image);
    dyld_register_image_state_change_handler(dyld_image_state_bound,
                                             1/*batch*/, &map_2_images);

dyld_register_image_state_change_handler(dyld_image_state_dependents_initializ
ed, 0/*not batch*/, &load_images);
```

对应的事件说明：

- **_dyld_register_func_for_remove_image**： dyld将image移除内存
- **dyld_image_state_bound**： dyld绑定了新的image
- **dyld_image_state_dependents_initialized**： image所依赖的各种库都被初始化好了

由上面代码可以看到，当dyld绑定了新的image之后，runtime会执行`map_2_images`方法，在这个方法中，libobjc会读取当前所加载的image的Mach-O文件信息，并利用和OC相关的Segment section中的信息，对OC内存空间的各种底层数据结构进行初始化，包括class struct的填充，category绑定到对应class，填充class的method list，protocol list以及property list等。当这一切都做好后，我们就可以

在程序中尽情的调用runtime的各种黑魔法啦~ 说到底，`所谓的runtime黑魔法，只是基于OC各种底层数据结构上的应用`。

所以，要想深入的了解runtime，需要了解OC底层的各种C语言实现，这在后续的章节中会逐步介绍。而OC底层结构的初始化，则是借助于Mach-O文件格式以及dyld。

> PS：不要将OC和runtime分开理解，其实在苹果开源代码中，OC和runtime是在一个工程中objc4的。我们平常所用到的OC语言，底层90%都是由C语言加一些汇编代码实现的。比如OC中的NSObject它唯一的实例变量类型Class，对应C语言结构体objc_class。 而所谓的runtime，则是在OC这些C语言底层实现的数据结构基础上，进行的一些操作，比如交换Selector的实现，只需要交换method list的IMP。获取一个对象的说有属性名称，只需要输出对应C语言结构property list即可。

Mach-O中OC相关的Segment Section

在上一篇文章中，我们可以看到，在**TEXT段和**DATA段中，都有如下以*objc**形式命名的section，这些section，是和OC的内存布局相关的。

__TEXT & OC

__TEXT段是程序的不可变常量部分，包括了字符串常量，被const修饰的常量，同时也包括OC相关的一些seciton。

__objc_classname

这里面以字符串常量的形式，记录了我们自定义以及所引用的系统class的名称，同时也包括Category, protocol 的名称：

__objc_methname

这个seciton中记录了当前APP所调用的方法的名称：

__objc_methtype

这个seciton与__objc_methname节对应，记录了method的描述字符串：

__DATA & OC

以下内容的结论，大部分来自于`五子棋大神`的深入理解Macho文件（二）- 消失的**OBJC段与新生的DATA段**，里面涉及到许多的汇编分析，本人对汇编功力欠佳，所以仅能总结下大神的结论。

__objc_imageinfo

__objc_imageinfo section 主要用来区分OC的版本1.0 或 2.0，在OC源码中是这样定义的：

```
typedef struct {
    uint32_t version; // currently 0
    uint32_t flags;
} objc_image_info;
```

version 字段总是为0，而flags字段通过异或的方式，表面需要支持的特性。如是否需要/支持垃圾回收：

```
SupportsGC          = 1<<1,  // image supports GC
  RequiresGC          = 1<<2,  // image requires GC

if (ii.flags & (1<<1)) {
    // App wants GC.
    // Don't return yet because we need to
    // check the AppleScriptObjC exception.
    wantsGC = YES;
}
```

__objc_classlist

这个section列出了所有的class，包括meta class。

用MachOView查看该节，是这样的：

该节中存储的是一个个的指针，指针指向的地址是class结构体所在的地址。class结构体在OC中用结构体objc_class 表示。

```
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
}
```

__objc_catlist

该节中存储了OC中定义的Catgory, 存储了一个个指向__objc_category 类型的指针。

```
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;
}
```

__objc_protolist

该节中记录了所有的协议。 像上面一样，也是存储了指向protocol_t的指针。

```
struct protocol_t : objc_object {
    const char *mangledName;
    struct protocol_list_t *protocols;
    method_list_t *instanceMethods;
    method_list_t *classMethods;
    method_list_t *optionalInstanceMethods;
    method_list_t *optionalClassMethods;
    property_list_t *instanceProperties;
    uint32_t size;   // sizeof(protocol_t)
    uint32_t flags;
    // Fields below this point are not always present on disk.
    const char **extendedMethodTypes;
    const char *_demangledName;
  }
```

__objc_classrefs

该节记录了那些class被引用了。因为有些类虽然被打包入APP中，但是在程序中并没有被引用。所以，这里记录了那些类真正的被我们实例化了。（注意，这里的AppDelegate 虽然在程序中没有显示的实例化，但系统似乎将其也标识为被引用的）

__objc_selrefs

这节告诉那些SEL对应的字符串被引用了，有系统方法，也有自定义方法：

__objc_superrefs

这一个节记录了调用super message的类。比如，在son方法中，我们调用了father的方法，就会将son class记录在这里。同理，在viewDidLoad中调用了super viewDidLoad, 因此view controller class也被记录在这里。

__objc_const

这一节用来记录在OC内存初始化过程中的不可变内容。这里所谓的不可变内容并不是我们在程序中所写的const int k = 5这种常量数据（它存在__TEXT的const section中），而是在OC内存布局中不可变得部分，包括但不限于：

```
struct class_ro_t {
    uint32_t flags;
    uint32_t instanceStart;
    uint32_t instanceSize;
#ifdef __LP64__
    uint32_t reserved;
#endif

    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;

    method_list_t *baseMethods() const {
        return baseMethodList;
    }
};

// 方法列表
// Two bits of entsize are used for fixup markers.
struct method_list_t : entsize_list_tt<method_t, method_list_t, 0x3> {
    bool isFixedUp() const;
    void setFixedUp();

    uint32_t indexOfMethod(const method_t *meth) const {
        uint32_t I =
            (uint32_t)(((uintptr_t)meth - (uintptr_t)this) / entsize());
        assert(i < count);
        return I;
    }
```

```
};
// 方法实体
struct method_t {
    SEL name;
    const char *types;
    IMP imp;

    struct SortBySELAddress :
        public std::binary_function<const method_t&,
                                    const method_t&, bool>
    {
        bool operator() (const method_t& lhs,
                         const method_t& rhs)
        { return lhs.name < rhs.name; }
    };
};
```

总结

今天我们了解了Mach-O格式和OC的关系，并大致了解了各个section中的内容。

当dyld加载我们的APP的时候，会通知OC读取对应seciton的内容，进而完成OC内存数据结构的初始化工作，为之后的程序运行及runtime黑魔法做好了准备。

`注意，上面所说的工作，是在APP的main函数之前就已经结束了的`。

到此为止，关于Mach-O格式，我们已经有了基本的了解了。接下来将会进入我们的正题：理解OC内部的各种数据结构，以及它们是如何被runtime所应用的。

前言

从本篇文章开始，就进入runtime的正篇。关于runtime的源码，大家可以在github中下载：

Runtime源码

什么是runtime?

OC是一门动态语言，与C++这种静态语言不同，静态语言的各种数据结构在编译期已经决定了，不能够被修改。而动态语言却可以使我们在程序运行期，动态的修改一个类的结构，如修改方法实现，绑定实例变量等。

> OC作为动态语言，它总会想办法将静态语言在编译期决定的事情，推迟到运行期来做。所以，仅有编译器是不够的，它需要一个运行时系统(runtime system)，这也就是OC的runtime系统的意义，它是OC运行框架的基石。

与Runtime交互

我们的OC语言是离不开runtime的。我们会在三个层次上和runtime进行交互，分别是：OC源码，通过Foundation框架定义的NSObject方法，直接调用runtime提供的接口方法。

- **OC源码**：大多数情况下，我们仅使用OC语言来编写代码，如NSObject，类属性，中括号的方法调用，协议，分类等。而这一切的背后，都是由runtime来支持的。我们平常所熟知的各种类型，背后都有runtime对应的C语言结构体，及C和汇编实现。
- **NSObject**: Cocoa中大部分类均继承于NSObject，因此大多数类都继承了NSObject所提供的方法。在NSObject中，有若干方法是运行时动态决定结果的，这背后其实是runtime系统对应数据结构的支持。如`isKindOfClass`和`isMemberOfClass`检查类是否属于指定的Class的继承体系中；`responderToSelector`检查对象是否能响应指定的消息；`conformsToProtocol`检查对象是否遵循某个协议；`methodForSelector`返回指定方法实现的地址。
- **Runtime函数**：Runtime 系统是一个由一系列函数和数据结构组成，具有公共接口的动态共享库。头文件存放于`/usr/include/objc`目录下。许多函数允许你用纯C代码来重复实现 Objc 中同样的功能。虽然有一些方法构成了NSObject类的基础，但是你在写 Objc 代码时一般不会直接用到这些函数的，除非是写一些 Objc 与其他语言的桥接或是底层的debug工作。在Objective-C Runtime Reference 中有对 Runtime 函数的详细文档。 就如在我们在前传篇中提到的，所谓的runtime黑魔法，只是基于OC各种底层数据结构上的应用。

因此，要想了解runtime，就要先了解runtime中定义的各种数据结构。我们先从最基础的objc_object和objc_class开始。

从NSObject说起

我们知道，在OC中，基本上所有的类的基类，都是NSObject。因此要深入了解OC中的类的结构，就要从NSObject这个类说起。

在XCode中，我们可以通过查看定义来了解NSObject的实现：

```
@interface NSObject <NSObject> {
    Class isa  OBJC_ISA_AVAILABILITY;
}
```

NSObject仅有一个实例变量Class isa：

```
/// An opaque type that represents an Objective-C class.
typedef struct objc_class *Class;
```

Class实质上是指向objc_class的指针。而objc_class的定义又是如何呢，在XCode中，我们继续查看定义：

```
struct objc_class {
    Class _Nonnull isa  OBJC_ISA_AVAILABILITY;

#if !__OBJC2__
    Class _Nullable super_class
OBJC2_UNAVAILABLE;
```

```
    const char * _Nonnull name
OBJC2_UNAVAILABLE;
    long version
OBJC2_UNAVAILABLE;
    long info
OBJC2_UNAVAILABLE;
    long instance_size
OBJC2_UNAVAILABLE;
    struct objc_ivar_list * _Nullable ivars
OBJC2_UNAVAILABLE;
    struct objc_method_list * _Nullable * _Nullable methodLists
     OBJC2_UNAVAILABLE;
    struct objc_cache * _Nonnull cache
OBJC2_UNAVAILABLE;
    struct objc_protocol_list * _Nullable protocols
OBJC2_UNAVAILABLE;
#endif

} OBJC2_UNAVAILABLE;
```

OK，到这里，对OC中类的结构的探索，我们先暂停一下。网上可以搜到很多博客，都是按照上面`objc_class`的定义来继续讲解的。

BUT！！！

难道都没有看到`OBJC2_UNAVAILABLE`这个提示吗？？？在OC 2.0中，这种关于`objc_class`的定义已经废弃掉了啊！许多博客都是在根本没有深入了解的情况下，就开始人云亦云，其实自己未必知道自己在说什么。这样做是很不负责任的，结果往往是误人子弟，自己不明白，还把别人带到了坑里。

吐槽完毕，我们继续来探索OC类的定义。前面说到，关于`objc_class`的定义，我们在XCode里面是看不到其真实的定义了，那么到哪里继续深入呢？看runtime源码。

在runtime源码的objc-runtime-new.h中，可以看到objc_class在OC 2.0中的定义。

objc_class

```
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();
    }
    void setData(class_rw_t *newData) {
        bits.setData(newData);
    }
  // 省略其他方法
```

```
  。。。
}
```

可以看到，objc_class继承自objc_object , 即在runtime中，class也被看做一种对象。 在objc_class中，有三个数据成员：

- **Class superclass**： 同样是Class类型，表明当前类的父类。
- **cache_t cache**： cache用于优化方法调用，其对应的数据结构如是：

```
struct cache_t {
    struct bucket_t *_buckets;
    mask_t _mask;
    mask_t _occupied;

  // 省略其余方法
  。。。
}

typedef uintptr_t cache_key_t;

struct bucket_t {
private:
    cache_key_t _key;
    IMP _imp;

public:
    inline cache_key_t key() const { return _key; }
    inline IMP imp() const { return (IMP)_imp; }
    inline void setKey(cache_key_t newKey) { _key = newKey; }
    inline void setImp(IMP newImp) { _imp = newImp; }

    void set(cache_key_t newKey, IMP newImp);
};
```

cache的核心是有一个类型为`bucket_t`的指针，它指向了一个以`_key`和`IMP`对应的缓存节点。

这里我们第一次遇到`uintptr_t`类型（_key） 。在runtime中，`uintptr_t`定义为

```
#ifndef _UINTPTR_T
#define _UINTPTR_T
typedef unsigned long   uintptr_t;
#endif /* _UINTPTR_T */
```

可以理解为`void *`。

runtime方法调用的流程是，当要调用一个方法时，先不去Class的方法列表中查找，而是先去找`cache_t cache` 。当系统调用过一个方法后，会将其实现`IMP`和`key`存放到cache中，因为理论上一个方法调用过后，被再次调用的概率很大。关于方法调用，我们将会在别的章节描述。

- **class_data_bits_t bits**：这是Class的核心，其本质是一个可以被Mask的指针类型。根据不同的Mask，可以取出不同的值。

```
struct class_data_bits_t {

    // Values are the FAST_ flags above.
    uintptr_t bits;

  public:
    class_rw_t* data() {
        return (class_rw_t *)(bits & FAST_DATA_MASK);
    }
    void setData(class_rw_t *newData)
    {
        assert(!data()  ||  (newData->flags & (RW_REALIZING | RW_FUTURE)));
        // Set during realization or construction only. No locking needed.
        // Use a store-release fence because there may be concurrent
        // readers of data and data's contents.
        uintptr_t newBits = (bits & ~FAST_DATA_MASK) | (uintptr_t)newData;
        atomic_thread_fence(memory_order_release);
        bits = newBits;
    }
    。。。
```

`class_data_bits_t bits` 仅含有一个成员`uintptr_t bits`， 可以理解为一个**复合指针**。什么意思呢，就是`bits`不仅包含了指针，同时包含了`Class`的各种异或flag，来说明`Class`的属性。把这些信息复合在一起，仅用一个uint指针`bits`来表示。当需要取出这些信息时，需要用对应的以FAST_ 前缀开头的flag掩码对`bits`做按位与操作。

例如，我们需要取出Classs的核心信息class_rw_t, 则需要调用方法：

```
class_rw_t* data() {
        return (class_rw_t *)(bits & FAST_DATA_MASK);
    }
```

该方法返回一个`class_rw_t *`，需要对`bits`进行`FAST_DATA_MASK` 的与操作。

bits在内存中有三种位排列方式：

32位

0	1	2-31
FAST_IS_SWIFT	FAST_HAS_DEFAULT_RR	FAST_DATA_MAS

64位兼容版

0	1	2	3-46	47-63
FAST_IS_SWIFT	FAST_HAS_DEFAULT_RR	FAST_REQUIRES_RAW_ISA	FAST_DATA_MASK	空闲

64位不兼容版

0	1	2	3-46	47	48	49
FAST_IS_SWIFT	FAST_REQUIRES_RAW_ISA	FAST_HAS_CXX_DTOR	FAST_DATA_MASK	FAST_HAS_CXX_CTOR	FAST_HAS_DEFAULT_AWZ	FA

不兼容版本的宏定义如下：

```
// class is a Swift class
#define FAST_IS_SWIFT           (1UL<<0)
// class's instances requires raw isa
#define FAST_REQUIRES_RAW_ISA   (1UL<<1)
// class or superclass has .cxx_destruct implementation
//   This bit is aligned with isa_t->hasCxxDtor to save an instruction.
#define FAST_HAS_CXX_DTOR       (1UL<<2)
// data pointer
#define FAST_DATA_MASK          0x00007ffffffffff8UL
// class or superclass has .cxx_construct implementation
#define FAST_HAS_CXX_CTOR       (1UL<<47)
// class or superclass has default alloc/allocWithZone: implementation
// Note this is is stored in the metaclass.
#define FAST_HAS_DEFAULT_AWZ    (1UL<<48)
// class or superclass has default retain/release/autorelease/retainCount/
//   _tryRetain/_isDeallocating/retainWeakReference/allowsWeakReference
#define FAST_HAS_DEFAULT_RR     (1UL<<49)
// summary bit for fast alloc path: !hasCxxCtor and
//   !instancesRequireRawIsa and instanceSize fits into shiftedSize
#define FAST_ALLOC              (1UL<<50)
// instance size in units of 16 bytes
//   or 0 if the instance size is too big in this field
//   This field must be LAST
#define FAST_SHIFTED_SIZE_SHIFT 51
```

让我们再看一下Class的核心结构class_rw_t：

```
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;         // 类不可修改的原始核心

    // 下面三个array，method,property，protocol，可以被runtime 扩展，如Category
    method_array_t methods;
    property_array_t properties;
    protocol_array_t protocols;

    // 和继承相关的东西
    Class firstSubclass;
```

```
    Class nextSiblingClass;

    // Class对应的 符号名称
    char *demangledName;

  // 以下方法省略
  ...
}

struct class_ro_t {
    uint32_t flags;
    uint32_t instanceStart;
    uint32_t instanceSize;
#ifdef __LP64__
    uint32_t reserved;
#endif

    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;

    method_list_t *baseMethods() const {
        return baseMethodList;
    }
};
```

可以看到，在`class_ro_t`中包含了类的名称，以及`method_list_t`， `protocol_list_t`，`ivar_list_t`， `property_list_t`这些类的基本信息。 在`class_ro_t`的信息是不可修改和扩展的。

而在更外一层`class_rw_t`中，有三个数组`method_array_t`，`property_array_t`，`protocol_array_t`：

```
struct class_rw_t {

  ...
    const class_ro_t *ro;          // 类不可修改的原始核心

    // 下面三个array，method,property，protocol，可以被runtime 扩展，如Category
    method_array_t methods;
    property_array_t properties;
    protocol_array_t protocols;
  ...
}
```

这三个数组是可以被runtime动态扩展的。

`objc_class` 中包含`class_data_bits_t`，`class_data_bits_t` 中通过`FAST_DATA_MASK`获取指向`class_rw_t`类型的指针，而在`class_rw_t`中包含`class_ro_t`，类的核心const信息。

realizeClass

在`objc_class`的`data()`方法最初返回的是`const class_ro_t *`类型，也就是类的基本信息。因为在调用`realizeClass`方法前，Category定义的各种方法，属性还没有附加到class上，因此只能够返回类的基本信息。

而当我们调用`realizeClass`时，会在函数内部将Category中定义的各种扩展附加到class上，同时改写data()的返回值为`class_rw_t *`类型，核心代码如下：

```
  const class_ro_t *ro;
    class_rw_t *rw;
  ro = (const class_ro_t *)cls->data();
    if (ro->flags & RO_FUTURE) {
        // This was a future class. rw data is already allocated.
        rw = cls->data();
        ro = cls->data()->ro;
        cls->changeInfo(RW_REALIZED|RW_REALIZING, RW_FUTURE);
    } else {
        // Normal class. Allocate writeable class data.
        rw = (class_rw_t *)calloc(sizeof(class_rw_t), 1);
        rw->ro = ro;
        rw->flags = RW_REALIZED|RW_REALIZING;
        cls->setData(rw);
    }
```

得出结论，在`class`没有调用`realizeClass`之前，不是真正完整的类。

objc_object

OC的底层实现是runtime，在runtime这一层，对象被定义为`objc_object`结构体，类被定义为了`objc_class`结构体。而`objc_class`继承于`objc_object`， 因此，类可以看做是一类**特殊的对象**。

现在就来看`objc_object`是如何定义的：

```
struct objc_object {
private:
    isa_t isa;

public:

    // ISA() assumes this is NOT a tagged pointer object
    Class ISA();

    // getIsa() allows this to be a tagged pointer object
    Class getIsa();

  // 省略其余方法
  ...
}
```

可以看到， `objc_object`的定义很简单，仅包含一个`isa_t`类型。

```
union isa_t
{
    isa_t() { }
    isa_t(uintptr_t value) : bits(value) { }

    Class cls;
    uintptr_t bits;

  // 省略其余
  。。。
}
```

`isa_t`是一个联合，可以表示`Class cls`或`uintptr_t bits`类型。实际上在OC 2.0里面，多数时间用的是`uintptr_t bits`。`bits`是一个64位的数据，每一位或几位都表示了关于当前对象的信息。具体的细节，我们将会在后续章节中作出详细描述。在这里我们先了解`isa_t`是一个联合类型就好。

objc_object & objc_class

如果我们再回头看一下`objc_object`和`objc_class`的定义，可以发现`object`和`class`是你中有我，我中有你的：

```
struct objc_object {
private:
    isa_t isa; // unit联合，可以表示Class类型，表明Object所属的类
  。。。
}

struct objc_class : objc_object { // objc_class继承自objc_object，表明objc_class
也是一个objc_object
   Class superclass; // super class 是一个objc_class * 指针
   。。。
}
```

如果用UML图表示的话：

可以看到，`objc_class`也是一个`objc_object`类型，这意味着，`objc_class`中也有一个属性`isa`，而这个`isa`，可以表示当前类属于（注意不是继承）哪个类。而这种说明类是属于哪个类的类，我们称之为**元类**（meta-class）。

这里再重申一遍，元类不是类的父类。至于元类的用途，我们将会在OC的消息转发中详细讲解。现在只需要知道，每一个类都有一个与其对应的元类。

id

我们可以用id表示任意类型的类实例变量。在runtime中，id是这样定义的：

```
typedef struct objc_object *id;
```

其实是一个`objc_object *`，因为`objc_object`的`isa`存在，所有runtime是可以知道id类型对应的真正类型的。这个和C里面的`void *`还是有区别的。

关于`NSObject`，`objc_class`和`objc_object`三者之间的关系，我们可以用下面的图来更清晰的了解：

当我们用中括号`[]`调用OC函数的时候，实际上会进入消息发送和消息转发流程：

> 消息发送（Messaging），runtime系统会根据SEL查找对用的IMP，查找到，则调用函数指针进行方法调用；若查找不到，则进入动态消息解析和转发流程，如果动态解析和消息转发失败，则程序crash并记录日志。

在进入正题之前，我们先思考两个问题：

1. 类实例可以调用类方法吗？ 类可以调用实例方法吗？ 为什么？
2. 下面代码输出什么？

```
@interface Father : NSObject
@end
```

```
@implementation Father
@end

@interface Son : Father
- (void)showClass;
@end

@implementation Son
- (void)showClass {
    NSLog(@"self class = %@, super class = %@", [self class], [super class]);
}

...
Son *son = [Son new];
[son showClass];    // 这里输出什么?
...
```

消息相关数据结构

SEL

`SEL`被称之为消息选择器，它相当于一个key，在类的消息列表中，可以根据这个key，来查找到对应的消息实现。

在runtime中，SEL的定义是这样的：

```
/// An opaque type that represents a method selector.
typedef struct objc_selector *SEL;
```

它是一个不透明的定义，似乎苹果故意隐藏了它的实现。目前SEL仅是一个字符串。

`这里要注意，即使消息的参数类型不同(注意，不是指参数数量不同)或方法所属的类也不同，但只要方法名相同，SEL也是一样的。所以，SEL单独并不能作为唯一的Key，必须结合消息发送的目标Class，才能找到最终的IMP。`

我们可以通过OC编译器命令`@selector()`或runtime函数`sel_registerName`，来获取一个SEL类型的方法选择器。

method_t

当需要发送消息的时候，runtime会在Class的方法列表中寻找方法的实现。在方法列表中方法是以结构体`method_t`存储的。

```
struct method_t {
    SEL name;
```

```
    const char *types;
    IMP imp;

    struct SortBySELAddress :
        public std::binary_function<const method_t&,
                                    const method_t&, bool>
    {
        bool operator() (const method_t& lhs,
                         const method_t& rhs)
        { return lhs.name < rhs.name; }
    };
};
```

可以看到`method_t`包含一个`SEL`作为key，同时有一个指向函数实现的指针`IMP`。`method_t`还包含一个属性`const char *types`;

`types`是一个C字符串，用于表明方法的返回值和参数类型。一般是这种格式的：

```
v24@0:8@16
```

关于SEL type，可以参考Type Encodings

IMP

`IMP`实际是一个函数指针，用于实际的方法调用。在runtime中定义是这样的：

```
/// A pointer to the function of a method implementation.
#if !OBJC_OLD_DISPATCH_PROTOTYPES
typedef void (*IMP)(void /* id, SEL, ... */ );
#else
typedef id _Nullable (*IMP)(id _Nonnull, SEL _Nonnull, ...);
#endif
```

`IMP`是由编译器生成的，如果我们知道了`IMP`的地址，则可以绕过runtime消息发送的过程，直接调用函数实现。关于这一点，我们稍后会谈到。

在消息发送的过程中，runtime就是根据`id`和`SEL`来唯一确定`IMP`并调用之的。

消息

当我们用`[]`向OC对象发送消息时，编译器会对应的代码修改为`objc_msgSend`, 其定义如下：

```
OBJC_EXPORT id _Nullable
objc_msgSend(id _Nullable self, SEL _Nonnull op, ...)
    OBJC_AVAILABLE(10.0, 2.0, 9.0, 1.0, 2.0);
```

其实，除了`objc_msgSend`，编译器还会根据实际情况，将消息发送改写为下面四个msgSend之一：

```
objc_msgSend
objc_msgSend_stret

objc_msgSendSuper
objc_msgSendSuper_stret
```

当我们将消息发送给`super class`的时候，编译器会将消息发送改写为`**SendSuper`的格式，如调用`[super viewDidLoad]`，会被编译器改写为`objc_msgSendSuper`的形式。

`objc_msgSendSuper`的定义如下：

```
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);
```

可以看到，调用`super`方法时，`msgSendSuper`的第一个参数不是`id self`，而是一个`objc_super *`。`objc_super`定义如下：

```
struct objc_super {
    /// Specifies an instance of a class.
    __unsafe_unretained _Nonnull id receiver;
    /// Specifies the particular superclass of the instance to message.
    __unsafe_unretained _Nonnull Class super_class;

};
```

`objc_super`包含两个数据，`receiver`指调用`super`方法的对象，即子类对象，而`super_class`表示子类的`Super Class`。

这就说明了在消息过程中调用了`super`方法和没有调用`super`方法，还是略有差异的。我们将会在下面讲解。

至于`**msgSend`中以`_stret`结尾的，表明方法返回值是一个结构体类型。

在`objc_msgSend`的内部，会依次执行：

1. 检测selector是否是应该忽略的，比如在Mac OS X开发中，有了垃圾回收机制，就不会响应`retain`，`release`这些函数。
2. 判断当前`receiver`是否为`nil`，若为`nil`，则不做任何响应，即向`nil`发送消息，系统不会`crash`。
3. 检查`Class`的`method cache`，若`cache`未命中，则进而查找`Class`的`method list`。
4. 若在`Class`的`method list`中未找到对应的`IMP`，则进行动态方法解析
5. 若无动态方法添加，则进入消息转发机制
6. 若消息转发失败，程序crash

objc_msgSend

objc_msgSend 的伪代码实现如下：

```
id objc_msgSend(id self, SEL cmd, ...) {
  if(self == nil)
    return 0;
  Class cls = objc_getClass(self);
  IMP imp = class_getMethodImplementation(cls, cmd);
  return imp?imp(self, cmd, ...):0;
}
```

而在runtime源码中，`objc_msgSend`方法其实是用汇编写的。为什么用汇编？ **一是因为`objc_msgSend`的返回值类型是可变的，需要用到汇编的特性； 二是因为汇编可以提高代码的效率。**

对应`arm64`，其汇编源码是这样的（有所删减）：

```
  ENTRY _objc_msgSend
  UNWIND _objc_msgSend, NoFrame
  MESSENGER_START

  cmp x0, #0      // nil check and tagged pointer check
  b.le  LNilOrTagged    //  (MSB tagged pointer looks negative)
  ldr x13, [x0]   // x13 = isa
  and x16, x13, #ISA_MASK // x16 = class
LGetIsaDone:
  CacheLookup NORMAL    // calls imp or objc_msgSend_uncached

LNilOrTagged:
  b.eq  LReturnZero   // nil check
END_ENTRY _objc_msgSend
```

虽然不懂汇编，但是结合注释，还是能够猜大体意思的。

首先，系统通过`cmp x0, #0`检测`receiver`是否为`nil`。如果为`nil`，则进入`LNilOrTagged`，返回`0`；

如果不为`nil`，则现将`receiver`的`isa`存入`x13`寄存器；

在`x13`寄存器中，取出`isa`中的`class`，放到`x16`寄存器中；

调用`CacheLookup NORMAL`，在这个函数中，首先查找`class`的`cache`，如果未命中，则进入`objc_msgSend_uncached`。

`objc_msgSend_uncached` 也是汇编，实现如下：

```
STATIC_ENTRY __objc_msgSend_uncached
  UNWIND __objc_msgSend_uncached, FrameWithNoSaves

  // THIS IS NOT A CALLABLE C FUNCTION
  // Out-of-band x16 is the class to search

  MethodTableLookup
  br  x17

  END_ENTRY __objc_msgSend_uncached
```

其内部调用了`MethodTableLookup`， `MethodTableLookup`是一个汇编的宏定义，其内部会调用C语言函数`_class_lookupMethodAndLoadCache3`：

```
IMP _class_lookupMethodAndLoadCache3(id obj, SEL sel, Class cls)
{
    return lookUpImpOrForward(cls, sel, obj,
                              YES/*initialize*/, NO/*cache*/,
YES/*resolver*/);
}
```

最终，会调用到`lookUpImpOrForward`来寻找`class`的`IMP`实现或进行消息转发。

lookUpImpOrForward

`lookUpImpOrForward`方法的目的在于根据`class`和`SEL`，在`class`或其`super class`中找到并返回对应的实现`IMP`，同时，`cache`所找到的`IMP`到当前`class`中。如果没有找到对应`IMP`， `lookUpImpOrForward`会进入消息转发流程。

`lookUpImpOrForward` 的简化版实现如下：

```
IMP lookUpImpOrForward(Class cls, SEL sel, id inst,
                       bool initialize, bool cache, bool resolver)
{
    IMP imp = nil;
    bool triedResolver = NO;

    runtimeLock.assertUnlocked();

    // 先在class的cache中查找imp
    if (cache) {
        imp = cache_getImp(cls, sel);
        if (imp) return imp;
    }

   runtimeLock.read();
```

```
    if (!cls->isRealized()) {
        runtimeLock.unlockRead();
        runtimeLock.write();
    // 如果class没有被relize，先relize
        realizeClass(cls);

        runtimeLock.unlockWrite();
        runtimeLock.read();
    }

    if (initialize  &&  !cls->isInitialized()) {
        runtimeLock.unlockRead();
        // 如果class没有init，则先init
        _class_initialize (_class_getNonMetaClass(cls, inst));
        runtimeLock.read();
    }


retry:
    runtimeLock.assertReading();

    // relaized并init了class，再试一把cache中是否有imp
    imp = cache_getImp(cls, sel);
    if (imp) goto done;

    // 先在当前class的method list中查找有无imp
    {
        Method meth = getMethodNoSuper_nolock(cls, sel);
        if (meth) {
            log_and_fill_cache(cls, meth->imp, sel, inst, cls);
            imp = meth->imp;
            goto done;
        }
    }

    // 在当前class中没有找到imp，则依次向上查找super class的方法列表
    {
        unsigned attempts = unreasonableClassCount();
        // 进入for循环，沿着继承链，依次向上查找super class的方法列表
        for (Class curClass = cls->superclass;
             curClass != nil;
             curClass = curClass->superclass)
        {
            // Halt if there is a cycle in the superclass chain.
            if (--attempts == 0) {
                _objc_fatal("Memory corruption in class list.");
            }

            // 先找super class的cache
```

```
            imp = cache_getImp(curClass, sel);
            if (imp) {
                if (imp != (IMP)_objc_msgForward_impcache) {
                    // 在super class 的cache中找到imp，将imp存储到当前class（注意，
不是super  class）的cache中
                    log_and_fill_cache(cls, imp, sel, inst, curClass);
                    goto done;
                }
                else {
                    // Found a forward:: entry in a superclass.
                    // Stop searching, but don't cache yet; call method
                    // resolver for this class first.
                    break;
                }
            }

            // 在Super class的cache中没有找到，调用getMethodNoSuper_nolock在super
class的方法列表中查找对应的实现
            Method meth = getMethodNoSuper_nolock(curClass, sel);
            if (meth) {
                log_and_fill_cache(cls, meth->imp, sel, inst, curClass);
                imp = meth->imp;
                goto done;
            }
        }
    }

    // 在class和其所有的super class中均未找到imp，进入动态方法解析流程resolveMethod
    if (resolver  &&  !triedResolver) {
        runtimeLock.unlockRead();
        _class_resolveMethod(cls, sel, inst);
        runtimeLock.read();
        // Don't cache the result; we don't hold the lock so it may have
        // changed already. Re-do the search from scratch instead.
        triedResolver = YES;
        goto retry;
    }

    // 如果在class，super classes和动态方法解析 都不能找到这个imp，则进入消息转发流程，
尝试让别的class来响应这个SEL

  // 消息转发结束，cache结果到当前class
    imp = (IMP)_objc_msgForward_impcache;
    cache_fill(cls, sel, imp, inst);

 done:
    runtimeLock.unlockRead();

    return imp;
```

```
}
```

通过上的源码，我们可以很清晰的知晓runtime的消息处理流程：

> 1. 尝试在当前`receiver`对应的`class`的`cache`中查找`imp`
> 2. 尝试在`class`的方法列表中查找`imp`
> 3. 尝试在`class`的所有`super classes`中查找`imp`（先看`Super class`的`cache`，再看`super class`的方法列表）
> 4. 上面3步都没有找到对应的`imp`，则尝试动态解析这个`SEL`
> 5. 动态解析失败，尝试进行消息转发，让别的`class`处理这个`SEL` 在查找`class`的方法列表中是否有`SEL`的对应实现时，是调用函数`getMethodNoSuper_nolock`：

```
static method_t *
getMethodNoSuper_nolock(Class cls, SEL sel)
{
    runtimeLock.assertLocked();

    assert(cls->isRealized());
    // fixme nil cls?
    // fixme nil sel?

    for (auto mlists = cls->data()->methods.beginLists(),
              end = cls->data()->methods.endLists();
         mlists != end;
         ++mlists)
    {
        method_t *m = search_method_list(*mlists, sel);
        if (m) return m;
    }

    return nil;
}
```

方法实现很简单，就是在`class`的方法列表`methods`中，根据`SEL`查找对应的`imp`。

> PS：这里顺便说一下`Category`覆盖类原始方法的问题，由于在`methods`中是线性查找的，会返回第一个和`SEL`匹配的`imp`。而在`class`的`realizeClass`方法中，会调用`methodizeClass`来初始化`class`的方法列表。在`methodizeClass`方法中，会将`Category`方法和`class`方法合并到一个列表，同时，会确保`Category`方法位于`class`方法前面，这样，在runtime寻找`SEL`的对应实现时，会先找到`Category`中定义的`imp`返回，从而实现了原始方法覆盖的效果。 关于`Category`的底层实现，我们会在后续章节中详细讲解。

关于消息的查找，可以用下图更清晰的解释：

runtime用`isa`找到`receiver`对应的`class`，用`superClass`找到`class`的父类。

这里用蓝色的表示实例方法的消息查找流程：通过类对象实例的`isa`查找到对象的`class`，进行查找。

用紫色表示类方法的消息查找流程: 通过类的`isa`找到类对应的元类, 沿着元类的`super class`链一路查找

关于*元类*，我们在上一章中已经提及，元类是**“类的类”**。因为在runtime中，类也被看做是一种对象，而对象就一定有其所属的类，因此，类所属的类，被称为类的***元类(meta class)***。

我们所定义的类方法，其实是存储在元类的方法列表中的。

关于元类的更多描述，可以查看这里。

类调用实例方法

这里有一个很好玩的地方，注意到在`SEL`查找链的最上方：`Root Class`和`Root meta Class`。

我们上面说到，对于类方法，是沿着紫色的路线依次查找`Super`类方法列表。一路上各个节点，都是元类(meta class)。而注意到，`Root meta Class`的`super class`竟然是`Root class`! 也就是说，当在`Root meta Class`中找不到类方法时，会转而到`Root class`中查找类方法。而在`Root class`中存储的，其实都是实例方法。

> 换句话说，我们在通过类方法的形式调用Root class中的实例方法，在OC中， 也是可以被解析的!

比如，在NSObject中，有一个实例方法：

```
@interface NSObject <NSObject> {
...
- (IMP)methodForSelector:(SEL)aSelector;
...
}
```

然后，我们自定义类：

```
@interface MyObj : NSObject
- (void)showY;
@end

@implementation MyObj
- (void)showY {
    NSLog(@"AB");
}
@end
```

我们分别通过类方法的形式调用`methodForSelector`和`showY`：

```
 [MyObj methodForSelector:@selector(test)];
 [MyObj showY];
```

会发现，编译器允许`methodForSelector`的调用，并能够正常运行。 而对于`showY`，则会编译错误。

至于原因，就是因为对于`Root meta class`，其实会到`Root class`中寻找对应的SEL实现。

类似的，还有一个好玩的例子，在子类中，我们重写NSObject的respondsToSelector方法，然后通过类方法和实例方法两种形式来调用，看看分别会发生什么情况：

```
@interface NSObject <NSObject> {
...
- (BOOL)respondsToSelector:(SEL)aSelector;
...
}

@interface MyObj : NSObject
@end

@implementation MyObj
- (BOOL)respondsToSelector:(SEL)aSelector {
    NSLog(@"It is my overwrite");
    return YES;
}
@end
```

然后通过类方法和实例方法分别调用

```
 [MyObj respondsToSelector:@selector(test)];

 MyObj *obj = [[MyObj alloc] init];
 [obj respondsToSelector:@selector(test)];
```

这两种调用方式会有什么不同？ 这当做一个思考题，如果大家理解了上面IMP的查找流程，那么应该能够知道答案。

objc_msgSendSuper

看完了`objc_msgSend`方法的调用流程，我们再来看一下`objc_msgSendSuper`是如何调用的。 当我们在代码里面显示的调用`super`方法时，runtime就会调用`objc_msgSendSuper`来完成消息发送。

`objc_msgSendSuper`的定义如下：

```
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);
```

当我使用`super`关键字时，在这里，`super`并不代表某个确定的对象，而是编译器的一个符号，编译器会将`super`替换为`objc_super *`类型来传入`objc_msgSendSuper`方法中。

而objc_super结构体定义如下：

```
struct objc_super {
    /// Specifies an instance of a class.
    __unsafe_unretained _Nonnull id receiver;
    /// Specifies the particular superclass of the instance to message.
    __unsafe_unretained _Nonnull Class super_class;

};
```

第一个成员`id receiver`，表明要将消息发送给谁。它应该是我们的类实例（注意，是当前类实例，而不是super） 第二个成员`Class super_class`，表明要到哪里去寻找`SEL`所对应的`IMP`。它应该是我们类实例所对应类的`super class`。（即要直接到`super class`中寻找`IMP`，而略过当前`class`的`method list`）

简单来说，当调用`super method`时，runtime会到`super class`中找到`IMP`，然后发送到当前`class`的实例上。因此，虽然`IMP`的实现是用的`super class`，但是，最终作用对象，仍然是当前`class`的实例。这也就是为什么

```
NSLog(@"%@ %@",[self class],  [super class]);
```

会输出同样的内容，即`[self class]`的内容。

我们来看一下objc_msgSendSuper的汇编实现：

```
  ENTRY _objc_msgSendSuper
  MESSENGER_START

  ldr r9, [r0, #CLASS]  // r9 = struct super->class
  CacheLookup NORMAL
  // cache hit, IMP in r12, eq already set for nonstret forwarding
  ldr r0, [r0, #RECEIVER] // load real receiver
  MESSENGER_END_FAST
  bx  r12     // call imp

  CacheLookup2 NORMAL
  // cache miss
  ldr r9, [r0, #CLASS]  // r9 = struct super->class
  ldr r0, [r0, #RECEIVER] // load real receiver
  MESSENGER_END_SLOW
  b __objc_msgSend_uncached

  END_ENTRY _objc_msgSendSuper
```

可以看到，它就是在`struct super->class`的`method list`中寻找对应的`IMP`，而`real receiver`则是`super->receiver`，即当前类实例。

如果在`super class`的`cache`中没有找到`IMP`的话，则同样会调用`__objc_msgSend_uncached`，这和`objc_msgSend`是一样的，最终都会调用到

```
IMP _class_lookupMethodAndLoadCache3(id obj, SEL sel, Class cls)
{
    return lookUpImpOrForward(cls, sel, obj,
                              YES/*initialize*/, NO/*cache*/,
YES/*resolver*/);
}
```

只不过，这里传入`lookUpImpOrForward`里面的`cls`，使用了`super class`而已。

动态解析

如果在类的继承体系中，没有找到相应的`IMP`，runtime首先会进行消息的动态解析。所谓动态解析，就是给我们一个机会，将方法实现在运行时动态的添加到当前的类中。然后，runtime会重新尝试走一遍消息查找的过程：

```
// 在class和其所有的super class中均未找到imp，进入动态方法解析流程resolveMethod
    if (resolver  &&  !triedResolver) {
        runtimeLock.unlockRead();
        _class_resolveMethod(cls, sel, inst);
        runtimeLock.read();
        // Don't cache the result; we don't hold the lock so it may have
        // changed already. Re-do the search from scratch instead.
        triedResolver = YES;
        goto retry;
    }
```

在源码中，可以看到，runtime会调用`_class_resolveMethod`，让用户进行动态方法解析，而且设置标记`triedResolver = YES`，仅执行一次。当动态解析完毕，不管用户是否作出了相应处理，runtime都会`goto retry`， 重新尝试查找一遍类的消息列表。

根据是调用的实例方法或类方法，runtime会在对应的类中调用如下方法：

```
+ (BOOL)resolveInstanceMethod:(SEL)sel  // 动态解析实例方法
+ (BOOL)resolveClassMethod:(SEL)sel     // 动态解析类方法
```

resolveInstanceMethod

`+ (BOOL)resolveInstanceMethod:(SEL)sel`用来动态解析实例方法，我们需要在运行时动态的将对应的方法实现添加到类实例所对应的类的消息列表中：

```
+ (BOOL)resolveInstanceMethod:(SEL)sel {
    if (sel == @selector(singSong)) {
        class_addMethod([self class], sel, class_getMethodImplementation([self
class], @selector(unrecoginzedInstanceSelector)), "v@:");
        return YES;
    }
    return [super resolveInstanceMethod:sel];
}

- (void)unrecoginzedInstanceSelector {
    NSLog(@"It is a unrecoginzed instance selector");
}
```

resolveClassMethod

`+ (BOOL)resolveClassMethod:(SEL)sel`用于动态解析类方法。 我们同样需要将类的实现动态的添加到相应类的消息列表中。

但这里需要注意，调用类方法的'对象'实际也是一个类，而类所对应的类应该是元类。要添加类方法，我们必须把方法的实现添加到元类的方法列表中。

在这里，我们就不能够使用`[self class]`了，它仅能够返回当前的类。而是需要使用`object_getClass(self)`，它其实会返回`isa`所指向的类，即类所对应的元类（注意，因为现在是在类方法里面，`self`所指的是`Class`，而通过`object_getClass(self)`获取`self`的类，自然是元类）。

```
+ (BOOL)resolveClassMethod:(SEL)sel {
    if (sel == @selector(payMoney)) {
        class_addMethod(object_getClass(self), sel,
class_getMethodImplementation(object_getClass(self),
@selector(unrecognizedClassSelector)), "v@:");
        return YES;
    }
    return [class_getSuperclass(self) resolveClassMethod:sel];
}

+ (void)unrecognizedClassSelector {
    NSLog(@"It is a unrecoginzed class selector");
}
```

这里主要弄清楚，类，元类，实例方法和类方法在不同地方存储，就清楚了。

关于`class`方法和`object_getClass`方法的区别：

当`self`是实例对象时，`[self class]`与`object_getClass(self)`等价，因为前者会调用后者，都会返回对象实例所对应的类。

当`self`是类对象时，`[self class]`返回类对象自身，而`object_getClass(self)`返回类所对应的元类。

消息转发

当动态解析失败，则进入消息转发流程。所谓消息转发，是将当前消息转发到其它对象进行处理。

```
- (id)forwardingTargetForSelector:(SEL)aSelector  // 转发实例方法
+ (id)forwardingTargetForSelector:(SEL)aSelector  // 转发类方法，id需要返回类对象
```

```
- (id)forwardingTargetForSelector:(SEL)aSelector
{
    if(aSelector == @selector(mysteriousMethod:)){
        return alternateObject;
    }
    return [super forwardingTargetForSelector:aSelector];
}
```

```
+ (id)forwardingTargetForSelector:(SEL)aSelector {
  if(aSelector == @selector(xxx)) {
    return NSClassFromString(@"Class name");
  }
  return [super forwardingTargetForSelector:aSelector];
}
```

如果`forwardingTargetForSelector`没有实现，或返回了`nil`或`self`，则会进入另一个转发流程。它会依次调用`- (NSMethodSignature *)methodSignatureForSelector:(SEL)aSelector`，然后runtime会根据该方法返回的值，组成一个`NSInvocation`对象，并调用`-(void)forwardInvocation:(NSInvocation *)anInvocation`。注意，当调用到`forwardInvocation`时，无论我们是否实现了该方法，系统都默认消息已经得到解析，不会引起crash。如果方法签名未返回，runtime则会调用`- (void)doesNotRecognizeSelector:(SEL)aSelector`并crash。

整个消息转发流程可以用下图表示：

`注意，和动态解析不同，由于消息转发实际上是将消息转发给另一种对象处理。而动态解析仍是尝试在当前类范围内进行处理。`

消息转发 & 多继承

通过消息转发流程，我们可以模拟实现OC的多继承机制。详情可以参考官方文档。

直接调用IMP

runtime的消息解析，究其根本，实际上就是根据SEL查找到对应的IMP，并调用之。如果我们可以直接知道IMP的所在，就不用再走消息机制这一层了。似乎不走消息机制会提高一些方法调用的速度，但现实是这样的吗?

我们比较一下:

```
CGFloat BNRTimeBlock (void (^block)(void)) {
    mach_timebase_info_data_t info;
    if (mach_timebase_info(&info) != KERN_SUCCESS) return -1.0;

    uint64_t start = mach_absolute_time ();
    block ();
    uint64_t end = mach_absolute_time ();
    uint64_t elapsed = end - start;

    uint64_t nanos = elapsed * info.numer / info.denom;
    return (CGFloat)nanos / NSEC_PER_SEC;

} // BNRTimeBlock

 Son *mySon1 = [Son new];
 setter ss = (void (*)(id, SEL, BOOL))[mySon1
methodForSelector:@selector(setFilled:)];
    CGFloat timeCost1 = BNRTimeBlock(^{
        for (int i = 0; i < 1000; ++i) {
            ss(mySon1, @selector(setFilled:), YES);
        }
    });

CGFloat timeCost2 = BNRTimeBlock(^{
        for (int i = 0; i < 1000; ++i) {
            [mySon1 setFilled:YES];
        }
    });
```

将timeCost1和timeCost2打印出来，你会发现，仅仅相差0.000001秒，几乎可以忽略不计。这样是因为在消息机制中，有缓存的存在。

总结

在本文中，我们了解了OC语言中方法调用实现的底层机制——消息机制。并了解了`self method`和`super method`的异同。 最后，让我们回答文章开头的两个问题:

1. 类实例可以调用类方法吗? 类可以调用实例方法吗? 为什么? 类实例不可用调用类方法，因为类实例查找消息`IMP`的流程仅会沿着继承链查找`class`的`method list`，而对于类方法来说，是

存于`meta class`的`method list`的，因此类实例通过`objc_msgSend`方法是找不到对应的实现的。

类大多数情况下是不能够调用实例方法的，除非实例方法定义在`root class`——`NSObject`中。因为，当调用类方法时，会在`meta class`的继承链的`method list`查找对应的`IMP`，而`root meta class`对应的`super class`是`NSObject`，因此在`NSObject`中定义的实例方法，其实是可以通过类方法形式来调用的。

2. 下面代码输出什么?

```
@interface Father : NSObject
@end

@implementation Father
@end

@interface Son : Father
- (void)showClass;
@end

@implementation Son
- (void)showClass {
    NSLog(@"self class = %@, super class = %@", [self class], [super class]);
}

...
Son *son = [Son new];
[son showClass];    // 这里输出什么?
...
```

会输出

```
self class = Son, super class = Son
```

至于原因，可以到本文关于`objc_msgSendSuper`相关讲解中查看。

方法替换，又称为`method swizzling`，是一个比较著名的runtime黑魔法。网上有很多的实现，我们这里直接讲最正规的实现方式以及其背后的原理。

Method Swizzling

在进行方法替换前，我们要考虑两种情况:

1. 要替换的方法在`target class`中有实现
2. 要替换的方法在`target class`中没有实现，而是在其父类中实现 对于第一种情况，很简单，我们直接调用`method_exchangeImplementations`即可达成方法。

而对于第二种情况，我们要仔细想想了。 因为在`target class`中没有对应的方法实现，方法实际上是在`target class`的父类中实现的，因此当我们要交换方法实现时，其实是交换了`target class`父类的实现。这样当其他地方调用这个父类的方法时，也会调用我们所替换的方法，这显然使我们不想要的。

比如，我想替换UIViewController类中的`methodForSelector:`方法，其实该方法是在其父类`NSObject`类中实现的。如果我们直接调用`method_exchangeImplementations`，则会替换掉`NSObject`的方法。这样当我们在别的地方，比如UITableView中再调用`methodForSelector:`方法时，其实会调用到父类NSObject，而NSObject的实现，已经被我们替换了。

为了避免这种情况，我们在进行方法替换前，需要检查`target class`是否有对应方法的实现，如果没有，则要讲方法动态的添加到`class`的`method list`中。

```
+(void)load{
    static dispatch_once_t onceToken;
    dispatch_once(&onceToken, ^{
        //要特别注意你替换的方法到底是哪个性质的方法
        // When swizzling a Instance method, use the following:
                Class class = [self class];

        // When swizzling a class method, use the following:
       // Class class = object_getClass((id)self);

        SEL originalSelector = @selector(systemMethod_PrintLog);
        SEL swizzledSelector = @selector(ll_imageName);

        Method originalMethod = class_getInstanceMethod(class,
originalSelector);
        Method swizzledMethod = class_getInstanceMethod(class,
swizzledSelector);

        BOOL didAddMethod =
        class_addMethod(class,
                        originalSelector,
                        method_getImplementation(swizzledMethod),
                        method_getTypeEncoding(swizzledMethod));

        if (didAddMethod) {
            class_replaceMethod(class,
                                swizzledSelector,
                                method_getImplementation(originalMethod),
                                method_getTypeEncoding(originalMethod));
        } else {
            method_exchangeImplementations(originalMethod, swizzledMethod);
        }
    });
}
```

来解释一下： 这里我们用`class_addMethod`方法来检查`target class`是否有方法实现。如果`target class`没有实现对应方法的话，则`class_addMethod`会返回`true`，同时，会将方法添加到`target class`中。如果`target class`已经有对应的方法实现的话，则`class_addMethod`调用失败，返回`false`，这时，我们直接调用`method_exchangeImplementations`方法来对调`originalMethod`和`swizzledMethod`即可。

这里有两个细节，一个是在`class_addMethod`方法中，我们传入的`SEL`是`originalSelector`，而实现是`swizzledMethod`的`IMP`，这样就等同于调换了方法。当`add method`成功后，我们又调用

```
if (didAddMethod) {
    class_replaceMethod(class, swizzledSelector,
method_getImplementation(originalMethod),
method_getTypeEncoding(originalMethod));
}
```

`class_replaceMethod`方法其实在内部会首先尝试调用`class_addMethod`，将方法添加到`class`中，如果添加失败，则说明`class`已经存在该方法，这时，会调用`method_setImplementation`来设置方法的`IMP`。

在`if (didAddMethod)`中，我们将`swizzledMethod`的`IMP`设置为了`originalMethod`的`IMP`，完成了方法交换。

第二个细节是这段注释：

```
+(void)load {
    //要特别注意你替换的方法到底是哪个性质的方法
    // When swizzling a Instance method, use the following:
    Class class = [self class];

    // When swizzling a class method, use the following:
    // Class class = object_getClass((id)self);
...
}
```

结合`+(void)load`方法的调用时机，它是由runtime在将`class`加载入内存中所调用的类方法。因此，我们一般会在这里面进行方法交换，因为时机是很靠前的。

这里要注意，在类方法中，`self`是一个`类对象`而不是实例对象。

当我们要替换类方法时，其实是要替换类对象所对应`元类`中的方法，要获取类对象的`元类`，需要调用`object_getClass`方法，它会返回`ISA()`，而类对象的`ISA()`,恰好是`元类`。

当我们要替换实例方法时，需要找到实例所对应的类，这时，就需要调用`[self class]`，虽然`self`是类对象，但是`+ class`会返回类对象自身，也就是实例对象所对应的类。

这段话说的比较绕，如果模糊的同学可以结合上一章最后类和元类的关系进行理解。

附带`class`方法的实现源码：

```
//NSObject.mm

+ (Class)class {
    return self;
}

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

Method swizzling原理

就如之前所说，runtime中所谓的黑魔法，只不过是基于runtime底层数据结构的应用而已。

现在，我们就一次剖析在method swizzling中所用到的runtime函数以及其背后实现和所依赖的数据结构。

class & object_getClass

要进行方法替换，首先要清楚我们要替换哪个类中的方法，即`target class`：

```
// When swizzling a Instance method, use the following:
        Class class = [self class];

// When swizzling a class method, use the following:
        Class class = object_getClass((id)self);
```

我们有两种方式获取`Class`对象，NSObject的`class`方法以及runtime函数`object_getClass`。这两种方法的具体实现，还是有差别的。

class

先看NSObject的方法`class`，其实有两个版本，一个是实例方法，一个是类方法，其源码如下：

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

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

当调用者是类对象时，会调用类方法版本，返回类对象自身。而调用者是实例对象时，会调用实例方法版本，在该版本中，又会调用runtime方法`object_getClass`。

那么在`object_getClass`中，又做了什么呢?

object_getClass

```
Class object_getClass(id obj)
{
    if (obj) return obj->getIsa();
    else return Nil;
}
```

实现很简单，就是调用了对象的`getIsa()`方法。这里我们可以简单的理解为就是返回了对象的`isa`指针。

如果对象是实例对象，`isa`返回实例对象所对应的类对象。 如果对象是类对象，`isa`返回类对象所对应的元类对象。

我们在回过头来看这段注释（注意这里的前提是在`+load()`方法中，`self`是类对象）：

```
// When swizzling a Instance method, use the following:
Class class = [self class];

// When swizzling a class method, use the following:
Class class = object_getClass((id)self);
```

当我们要调换实例方法，则需要修改实例对象所对应的类对象的方法列表，因为这里的self已经是一个类对象，所有调用class方法其实会返回其自身，即实例对象对应的类对象：

```
// When swizzling a Instance method, use the following:
Class class = [self class];
```

当我们要调换类方法，则需要修改类对象所对应的元类对象的方法列表，因此要调用`object_class`方法，它会返回对象的`isa`，而类对象的`isa`，则恰是类对象对应的`元类`对象:

```
// When swizzling a class method, use the following:
Class class = object_getClass((id)self);
```

class_getInstanceMethod

确认了`class`后，我们就需要准备方法调用的原材料：`originalMethod method`和`swizzled method`。Method数据类型在runtime中的定义为：

```
typedef struct method_t *Method;

struct method_t {
    SEL name;
    const char *types;
    IMP imp;
```

```
    struct SortBySELAddress :
        public std::binary_function<const method_t&,
                                    const method_t&, bool>
    {
        bool operator() (const method_t& lhs,
                         const method_t& rhs)
        { return lhs.name < rhs.name; }
    };
};
```

我们所说的类的方法列表中，就是存储的`method_t`类型。

Method数据类型的实例，如果自己创建的话，会比较麻烦，尤其是如何填充`IMP`，但我们可以从现有的`class`方法列表中取出一个`method`来。很简单，只需要调用`class_getInstanceMethod`方法。

`class_getInstanceMethod`方法究竟做了什么呢？就像我们刚才说的一样，它就是在指定的类对象中的方法列表中去取SEL所对应的Method。

```
/***********************************************************************
* class_getInstanceMethod.  Return the instance method for the
* specified class and selector.
**********************************************************************/
Method class_getInstanceMethod(Class cls, SEL sel)
{
    if (!cls  ||  !sel) return nil;
    lookUpImpOrNil(cls, sel, nil,
                   NO/*initialize*/, NO/*cache*/, YES/*resolver*/);
    return _class_getMethod(cls, sel);
}
```

`class_getInstanceMethod`首先调用了`lookUpImpOrNil`，其实它的内部实现和普通的消息流程是一样的（内部会调用上一章中说所的消息查找函数`lookUpImpOrForward`），只不过对于消息转发得到的`IMP`，会替换为`nil`。

在进行了一波消息流程之后，调用`_class_getMethod`方法

```
static Method _class_getMethod(Class cls, SEL sel)
{
    rwlock_reader_t lock(runtimeLock);
    return getMethod_nolock(cls, sel);
}

static method_t *
getMethod_nolock(Class cls, SEL sel)
{
    method_t *m = nil;
    runtimeLock.assertLocked();
    assert(cls->isRealized());
```

```
    // 核心：沿着继承链，向上查找第一个SEL所对应的method
    while (cls  &&  ((m = getMethodNoSuper_nolock(cls, sel))) == nil) {
        cls = cls->superclass;
    }

    return m;
}

// getMethodNoSuper_nolock 方法实质就是在查找class的消息列表
static method_t *
getMethodNoSuper_nolock(Class cls, SEL sel)
{
    runtimeLock.assertLocked();

    assert(cls->isRealized());
    // fixme nil cls?
    // fixme nil sel?

    for (auto mlists = cls->data()->methods.beginLists(),
              end = cls->data()->methods.endLists();
         mlists != end;
         ++mlists)
    {
        method_t *m = search_method_list(*mlists, sel);
        if (m) return m;
    }

    return nil;
}
```

class_addMethod

当我们获取到`target class`和`swizzled method`后，首先尝试调用`class_addMethod`方法将`swizzled method`添加到`target class`中。

这样做的目的在于：如果`target class`中没有要替换的`original method`，则会直接将`swizzled method`作为`original method`的实现添加到`target class`中。如果`target class`中确实存在`original method`，则`class_addMethod`会失败并返回`false`，我们就可以直接调用`method_exchangeImplementations`方法来实现方法替换。这就是下面一段逻辑代码的意义：

```
BOOL didAddMethod = class_addMethod(class, originalSelector,
method_getImplementation(swizzledMethod),
method_getTypeEncoding(swizzledMethod));
if (didAddMethod) {
    class_replaceMethod(class, swizzledSelector,
method_getImplementation(originalMethod),
method_getTypeEncoding(originalMethod));
} else {
    method_exchangeImplementations(originalMethod, swizzledMethod);
}
```

我们先来看`class_addMethod`是怎么实现的。其实到了这里，相信大家不用看代码也能猜的出来，`class_addMethod`其实就是将我们提供的`method`，插入到`target class`的方法列表中。事实是这样的吗，看源码：

```
BOOL class_addMethod(Class cls, SEL name, IMP imp, const char *types)
{
    if (!cls) return NO;

    rwlock_writer_t lock(runtimeLock);
    return ! addMethod(cls, name, imp, types ?: "", NO);
}

static IMP addMethod(Class cls, SEL name, IMP imp, const char *types, bool
replace)
{
    IMP result = nil;

    runtimeLock.assertWriting();

    assert(types);
    assert(cls->isRealized());

    method_t *m;
    if ((m = getMethodNoSuper_nolock(cls, name))) {
        // 方法已经存在
        if (!replace) { // 如果选择不替换，则返回原始的方法，添加方法失败
            result = m->imp;
        } else {  // 如果选择替换，则返回原始方法，同时，替换为新的方法
            result = _method_setImplementation(cls, m, imp);
        }
    } else {
        // 方法不存在，则在class的方法列表中添加方法，并返回nil
        method_list_t *newlist;
        newlist = (method_list_t *)calloc(sizeof(*newlist), 1);
        newlist->entsizeAndFlags =
            (uint32_t)sizeof(method_t) | fixed_up_method_list;
        newlist->count = 1;
```

```
        newlist->first.name = name;
        newlist->first.types = strdupIfMutable(types);
        newlist->first.imp = imp;

        prepareMethodLists(cls, &newlist, 1, NO, NO);
        cls->data()->methods.attachLists(&newlist, 1);
        flushCaches(cls);

        result = nil;
    }

    return result;
}
```

源码证明，我们的猜想是正确的!

class_replaceMethod

如果`class_addMethod`返回成功，则说明我们已经为`target class`添加上了`SEL`为`original SEL`，并且其实现是`swizzled method`。至此，我们方法交换完成了一半，现在我们将`swizzled method`替换为`original method`。

```
 if (didAddMethod) {
            class_replaceMethod(class,
                                swizzledSelector,
                                method_getImplementation(originalMethod),
                                method_getTypeEncoding(originalMethod));
 }
```

这里，我们调用了`class_replaceMethod`方法。它的内部逻辑是这样的：

1. 如果`target class`中没有`SEL`的对应实现，则会为`target class`添加上对应实现。
2. 如果`target class`中已经有了`SEL`对应的方法，则会将`SEL`对应的原始`IMP`，替换为新的`IMP`。

```
IMP class_replaceMethod(Class cls, SEL name, IMP imp, const char *types)
{
    if (!cls) return nil;

    rwlock_writer_t lock(runtimeLock);
    return addMethod(cls, name, imp, types ?: "", YES);
}

static IMP
addMethod(Class cls, SEL name, IMP imp, const char *types, bool replace)
{
    IMP result = nil;
```

```
    runtimeLock.assertWriting();

    assert(types);
    assert(cls->isRealized());

    method_t *m;
    if ((m = getMethodNoSuper_nolock(cls, name))) {
        // 方法已经存在
        if (!replace) { // 如果选择不替换，则返回原始的方法，添加方法失败
            result = m->imp;
        } else {  // 如果选择替换，则返回原始方法，同时，替换为新的方法
            result = _method_setImplementation(cls, m, imp);
        }
    } else {
        // 方法不存在，则在class的方法列表中添加方法，并返回nil
        method_list_t *newlist;
        newlist = (method_list_t *)calloc(sizeof(*newlist), 1);
        newlist->entsizeAndFlags =
            (uint32_t)sizeof(method_t) | fixed_up_method_list;
        newlist->count = 1;
        newlist->first.name = name;
        newlist->first.types = strdupIfMutable(types);
        newlist->first.imp = imp;

        prepareMethodLists(cls, &newlist, 1, NO, NO);
        cls->data()->methods.attachLists(&newlist, 1);
        flushCaches(cls);

        result = nil;
    }

    return result;
}
```

通过源码对比可以发现，`class_addMethod`和`class_replaceMethod`其实都是调用的`addMethod`方法，区别只是`bool replace`参数，一个是`NO`，不会替换原始实现，另一个是`YES`，会替换原始实现。

method_exchangeImplementations

如果`class_addMethod`失败，则说明`target class`中的`original method`是在`target class`中有定义的，这时候，我们直接调用`method_exchangeImplementations`交换实现即可。`method_exchangeImplementations`实现很简单，就是交换两个`Method`的`IMP`：

```
void method_exchangeImplementations(Method m1, Method m2)
{
    if (!m1  ||  !m2) return;
```

```
    rwlock_writer_t lock(runtimeLock);

    IMP m1_imp = m1->imp;
    m1->imp = m2->imp;
    m2->imp = m1_imp;


    // RR/AWZ updates are slow because class is unknown
    // Cache updates are slow because class is unknown
    // fixme build list of classes whose Methods are known externally?

    flushCaches(nil);

    updateCustomRR_AWZ(nil, m1);
    updateCustomRR_AWZ(nil, m2);
}
```

值得注意的地方

在写这篇博文的时候，笔者曾做过这个实验，在UIViewController的Category中，测试

```
- (void)exchangeImp {
  Class aClass = object_getClass(self);
    SEL originalSelector = @selector(viewWillAppear:);
    SEL swizzledSelector = @selector(sw_viewWillAppearXXX:);

    Method originalMethod = class_getInstanceMethod(aClass, originalSelector);
    Method swizzledMethod = class_getInstanceMethod(aClass, swizzledSelector);
  IMP result = class_replaceMethod(aClass,
originalSelector,method_getImplementation(swizzledMethod),
method_getTypeEncoding(swizzledMethod));
  NSLog(@"result is %p", result);
}
```

因为在`class_replaceMethod`方法中，如果`target class`已经存在`SEL`对应的方法实现，则会返回其`old IMP`，并替换为`new IMP`。本来以为`result`会返回`viewWillAppear:`的实现，但结果却是返回了`nil`。这是怎么回事呢?

究其根本，原来是因为我是在UIViewController的子类ViewController中调用的`exchangeImp`方法，那么`object_getClass(self)`，其实会返回子类ViewController而不是UIViewController。

在`class_replaceMethod`中，runtime仅会查找当前类`aClass`，即ViewController的方法列表，而不会向上查询其父类UIViewController的方法列表。这样自然就找不到`viewWillAppear:`的实现啦。

而对于`class_getInstanceMethod`，runtime除了查找当前类，还会沿着继承链向上查找对应的`Method`。

所以，这里就造成了，`class_getInstanceMethod`可以得到`viewWillAppear:`对应的`Method`，而在`class_replaceMethod`中，却找不到`viewWillAppear:`对应的`IMP`。

如果不了解背后的实现，确实很难理解这种看似矛盾的结果。

在平日编程中或阅读第三方代码时，`category`可以说是无处不在。`category`也可以说是OC作为一门动态语言的一大特色。`category`为我们动态扩展类的功能提供了可能，或者我们也可以把一个庞大的类进行功能分解，按照`category`进行组织。

关于`category`的使用无需多言，今天我们来深入了解一下，`category`是如何在runtime中实现的。

category的数据结构

`category`对应到runtime中的结构体是`struct category_t`(位于objc-runtime-new.h)：

```
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_t`的定义很简单。从定义中看出，`category` 中的 **可为**：添加实例方法(`instanceMethods`)，类方法(`classMethods`)，协议(`protocols`)和实例属性(`instanceProperties`) **不可为**：不能够添加实例变量（关于实例属性和实例变量的区别，我们将会在别的章节中探讨）。

category的加载

知道了`category`的数据结构，我们来深入探究一下`category`是如何在runtime中实现的。

原理很简单：runtime会分别将`category` 结构体中的`instanceMethods`, `protocols`, `instanceProperties`添加到`target class`的实例方法列表，协议列表，属性列表中，会将`category`结构体中的`classMethods`添加到`target class`所对应的元类的实例方法列表中。其本质就相当于runtime在运行时期，修改了`target class`的结构。

经过这一番修改，`category`中的方法，就变成了`target class`方法列表中的一部分，其调用方式也就一模一样啦~

现在，就来看一下具体是怎么实现的。

首先，我们在`Mach-O`格式和runtime 介绍过在`Mach-O`文件中，`category`数据会被存放在`__DATA`段下的`__objc_catlist section`中。

当OC被`dyld`加载起来时，OC进入其入口点函数`_objc_init`：

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

    // fixme defer initialization until an objc-using image is found?
    environ_init();
    tls_init();
    static_init();
    lock_init();
    exception_init();

    _dyld_objc_notify_register(&map_images, load_images, unmap_image);
}
```

我们忽略一堆`init`方法，重点来看`_dyld_objc_notify_register`方法。该方法会向`dyld`注册监听`Mach-O`中OC相关`section`被加载入\载出内存的事件。

具体有三个事件：

- **`_dyld_objc_notify_mapped`**(对应`&map_images`回调)：当dyld已将images加载入内存时。
- **`_dyld_objc_notify_init`**(对应`load_images`回调)：当dyld初始化image后。OC调用类的`+load`方法，就是在这时进行的。
- **`_dyld_objc_notify_unmapped`**(对应`unmap_image`回调)：当dyld将images移除内存时。

而`category`写入`target class`的方法列表，则是在`_dyld_objc_notify_mapped`，即将`Mach-O`相关`sections`都加载到内存之后所发生的。

我们可以看到其对应回调为`map_images`方法。

在`map_images`最终会调用`_read_images`方法来读取OC相关`sections`，并以此来初始化OC内存环境。`_read_images`的极简实现版如下，可以看到，rumtime是如何根据`Mach-O`各个`section`的信息来初始化其自身的：

```
void _read_images(header_info **hList, uint32_t hCount, int totalClasses, int
unoptimizedTotalClasses)
{
```

```
    static bool doneOnce;
    TimeLogger ts(PrintImageTimes);

    runtimeLock.assertWriting();

    if (!doneOnce) {
        doneOnce = YES;

        ts.log("IMAGE TIMES: first time tasks");
    }


    // Discover classes. Fix up unresolved future classes. Mark bundle
classes.

    for (EACH_HEADER) {

        classref_t *classlist = _getObjc2ClassList(hi, &count);
        for (i = 0; i < count; i++) {
            Class cls = (Class)classlist[I];
            Class newCls = readClass(cls, headerIsBundle,
headerIsPreoptimized);
        }
    }

    ts.log("IMAGE TIMES: discover classes");

    // Fix up remapped classes
    // Class list and nonlazy class list remain unremapped.
    // Class refs and super refs are remapped for message dispatching.

    for (EACH_HEADER) {
        Class *classrefs = _getObjc2ClassRefs(hi, &count);
        for (i = 0; i < count; i++) {
            remapClassRef(&classrefs[I]);
        }
        // fixme why doesn't test future1 catch the absence of this?
        classrefs = _getObjc2SuperRefs(hi, &count);
        for (i = 0; i < count; i++) {
            remapClassRef(&classrefs[I]);
        }
    }

    ts.log("IMAGE TIMES: remap classes");


    for (EACH_HEADER) {
        if (hi->isPreoptimized()) continue;
```

```
        bool isBundle = hi->isBundle();
        SEL *sels = _getObjc2SelectorRefs(hi, &count);
        UnfixedSelectors += count;
        for (i = 0; i < count; i++) {
            const char *name = sel_cname(sels[i]);
            sels[i] = sel_registerNameNoLock(name, isBundle);
        }
    }

    ts.log("IMAGE TIMES: fix up selector references");


    // Discover protocols. Fix up protocol refs.
    for (EACH_HEADER) {
        extern objc_class OBJC_CLASS_$_Protocol;
        Class cls = (Class)&OBJC_CLASS_$_Protocol;
        assert(cls);
        NXMapTable *protocol_map = protocols();
        bool isPreoptimized = hi->isPreoptimized();
        bool isBundle = hi->isBundle();

        protocol_t **protolist = _getObjc2ProtocolList(hi, &count);
        for (i = 0; i < count; i++) {
            readProtocol(protolist[i], cls, protocol_map,
                         isPreoptimized, isBundle);
        }
    }

    ts.log("IMAGE TIMES: discover protocols");

    // Fix up @protocol references
    // Preoptimized images may have the right
    // answer already but we don't know for sure.
    for (EACH_HEADER) {
        protocol_t **protolist = _getObjc2ProtocolRefs(hi, &count);
        for (i = 0; i < count; i++) {
            remapProtocolRef(&protolist[I]);
        }
    }

    ts.log("IMAGE TIMES: fix up @protocol references");

    // Realize non-lazy classes (for +load methods and static instances)
    for (EACH_HEADER) {
        classref_t *classlist =
        _getObjc2NonlazyClassList(hi, &count);
        for (i = 0; i < count; i++) {
            Class cls = remapClass(classlist[i]);
            if (!cls) continue;
```

```
            realizeClass(cls);
        }
    }

    ts.log("IMAGE TIMES: realize non-lazy classes");

    // Realize newly-resolved future classes, in case CF manipulates them
    if (resolvedFutureClasses) {
        for (i = 0; i < resolvedFutureClassCount; i++) {
            realizeClass(resolvedFutureClasses[I]);
            resolvedFutureClasses[i]-
>setInstancesRequireRawIsa(false/*inherited*/);
        }
        free(resolvedFutureClasses);
    }
    ts.log("IMAGE TIMES: realize future classes");

    // Discover categories.
    for (EACH_HEADER) {
        category_t **catlist =
        _getObjc2CategoryList(hi, &count);
        bool hasClassProperties = hi->info()->hasCategoryClassProperties();

        for (i = 0; i < count; i++) {
            category_t *cat = catlist[I];
            Class cls = remapClass(cat->cls);

            bool classExists = NO;
            if (cat->instanceMethods ||  cat->protocols
                ||  cat->instanceProperties)
            {
                addUnattachedCategoryForClass(cat, cls, hi);
            }

            if (cat->classMethods  ||  cat->protocols
                ||  (hasClassProperties && cat->_classProperties))
            {
                addUnattachedCategoryForClass(cat, cls->ISA(), hi);
            }
        }
    }

    ts.log("IMAGE TIMES: discover categories");
}
```

大致的逻辑是，runtime调用`_getObjc2XXX`格式的方法，依次来读取对应的`section`内容，并根据其结果初始化其自身结构。

`_getObjc2XXX`方法有如下几种，可以看到他们都一一对应了`Mach-O`中相关的OC `seciton`。

```
//      function name                  content type     section name
GETSECT(_getObjc2SelectorRefs,        SEL,             "__objc_selrefs");
GETSECT(_getObjc2MessageRefs,         message_ref_t,   "__objc_msgrefs");
GETSECT(_getObjc2ClassRefs,           Class,           "__objc_classrefs");
GETSECT(_getObjc2SuperRefs,           Class,           "__objc_superrefs");
GETSECT(_getObjc2ClassList,           classref_t,      "__objc_classlist");
GETSECT(_getObjc2NonlazyClassList,    classref_t,      "__objc_nlclslist");
GETSECT(_getObjc2CategoryList,        category_t *,    "__objc_catlist");
GETSECT(_getObjc2NonlazyCategoryList, category_t *,    "__objc_nlcatlist");
GETSECT(_getObjc2ProtocolList,        protocol_t *,    "__objc_protolist");
GETSECT(_getObjc2ProtocolRefs,        protocol_t *,    "__objc_protorefs");
GETSECT(getLibobjcInitializers,       Initializer,     "__objc_init_func");
```

可以看到，我们使用的类，协议和`category`，都是在`_read_images`方法中读取出来的。 在读取`cateogry`的方法`_getObjc2CategoryList(hi, &count)`中，读取的是`Mach-O`文件的`__objc_catlist`段。

我们重点关注和category相关的代码：

```
  // Discover categories.
    for (EACH_HEADER) {
        category_t **catlist =
        _getObjc2CategoryList(hi, &count);
        bool hasClassProperties = hi->info()->hasCategoryClassProperties();

        for (i = 0; i < count; i++) {
            category_t *cat = catlist[I];
            Class cls = remapClass(cat->cls);

            bool classExists = NO;
            // 如果Category中有实例方法，协议，实例属性，会改写target class的结构
            if (cat->instanceMethods ||  cat->protocols
                ||  cat->instanceProperties)
            {
                addUnattachedCategoryForClass(cat, cls, hi);
                if (cls->isRealized()) {
                    remethodizeClass(cls);
                    classExists = YES;
                }
                if (PrintConnecting) {
                    _objc_inform("CLASS: found category -%s(%s) %s",
                                 cls->nameForLogging(), cat->name,
                                 classExists ? "on existing class" : "");
                }
            }
            // 如果category中有类方法，协议，或类属性(目前OC版本不支持类属性)，会改写
target class的元类结构
            if (cat->classMethods  ||  cat->protocols
```

```
                ||  (hasClassProperties && cat->_classProperties))
            {
                addUnattachedCategoryForClass(cat, cls->ISA(), hi);
                if (cls->ISA()->isRealized()) {
                    remethodizeClass(cls->ISA());
                }
                if (PrintConnecting) {
                    _objc_inform("CLASS: found category +%s(%s)",
                                 cls->nameForLogging(), cat->name);
                }
            }
        }
    }

    ts.log("IMAGE TIMES: discover categories");
}
```

`discover categories`的逻辑如下：

1. 先调用`_getObjc2CategoryList`读取`__objc_catlist seciton`下所记录的所有`category`。并存放到`category_t *`数组中。
2. 依次读取数组中的`category_t * cat`
3. 对每一个`cat`，先调用`remapClass(cat->cls)`，并返回一个`objc_class *`对象`cls`。这一步的目的在于找到到`category`对应的类对象`cls`。
4. 找到`category`对应的类对象`cls`后，就开始进行对`cls`的修改操作了。首先，如果`category`中有实例方法，协议，和实例属性之一的话，则直接对`cls`进行操作。如果`category`中包含了类方法，协议，类属性（不支持）之一的话，还要对`cls`所对应的`元类(cls->ISA())`进行操作。
5. 不管是对`cls`还是`cls的元类`进行操作，都是调用的方法`addUnattachedCategoryForClass`。但这个方法并不是`category`实现的关键，其内部逻辑只是将`class`和其对应的`category`做了一个映射。这样，以`class`为`key`，就可以取到所其对应的所有的`category`。
6. 做好`class`和`category`的映射后，会调用`remethodizeClass`方法来修改`class`的`method list`结构，这才是runtime实现`category`的关键所在。

remethodizeClass

既然`remethodizeClass`是`category`的实现核心，那么我们就单独一节，细看一下该方法的实现：

```
/***********************************************************************
* remethodizeClass
* Attach outstanding categories to an existing class.
* Fixes up cls's method list, protocol list, and property list.
* Updates method caches for cls and its subclasses.
* Locking: runtimeLock must be held by the caller
**********************************************************************/
static void remethodizeClass(Class cls)
{
```

```
    category_list *cats;
    bool isMeta;

    runtimeLock.assertWriting();

    isMeta = cls->isMetaClass();

    // Re-methodizing: check for more categories
    if ((cats = unattachedCategoriesForClass(cls, false/*not realizing*/))) {
        if (PrintConnecting) {
            _objc_inform("CLASS: attaching categories to class '%s' %s",
                         cls->nameForLogging(), isMeta ? "(meta)" : "");
        }

        attachCategories(cls, cats, true /*flush caches*/);
        free(cats);
    }
}
```

该段代码首先通过`unattachedCategoriesForClass`取出还未被附加到`class`上的`category list`，然后调用`attachCategories`将这些`category`附加到`class`上。

`attachCategories`的实现如下：

```
// Attach method lists and properties and protocols from categories to a
class.
// Assumes the categories in cats are all loaded and sorted by load order,
// oldest categories first.
static void
attachCategories(Class cls, category_list *cats, bool flush_caches)
{
    if (!cats) return;
    if (PrintReplacedMethods) printReplacements(cls, cats);

    bool isMeta = cls->isMetaClass();

    // 首先分配method_list_t *， property_list_t *， protocol_list_t *的数组空间,
数组大小等于category的个数
    method_list_t **mlists = (method_list_t **)
        malloc(cats->count * sizeof(*mlists));
    property_list_t **proplists = (property_list_t **)
        malloc(cats->count * sizeof(*proplists));
    protocol_list_t **protolists = (protocol_list_t **)
        malloc(cats->count * sizeof(*protolists));

    // Count backwards through cats to get newest categories first
    int mcount = 0;
    int propcount = 0;
    int protocount = 0;
```

```
    int i = cats->count;
    bool fromBundle = NO;
    while (i--) {  // 依次读取每一个category，将其methods，property，protocol添加到
mlists，proplist，protolist中存储
        auto& entry = cats->list[I];

        method_list_t *mlist = entry.cat->methodsForMeta(isMeta);
        if (mlist) {
            mlists[mcount++] = mlist;
            fromBundle |= entry.hi->isBundle();
        }

        property_list_t *proplist =
            entry.cat->propertiesForMeta(isMeta, entry.hi);
        if (proplist) {
            proplists[propcount++] = proplist;
        }

        protocol_list_t *protolist = entry.cat->protocols;
        if (protolist) {
            protolists[protocount++] = protolist;
        }
    }

  // 取出class的data()数据，其实是class_rw_t * 指针，其对应结构体实例存储了class的基本
信息
    auto rw = cls->data();

    prepareMethodLists(cls, mlists, mcount, NO, fromBundle);
    rw->methods.attachLists(mlists, mcount);  // 将category中的method 添加到
class中
    free(mlists);
    if (flush_caches  &&  mcount > 0) flushCaches(cls); // 如果需要，同时刷新
class的method list cache


    rw->properties.attachLists(proplists, propcount); // 将category的property添
加到class中
    free(proplists);

    rw->protocols.attachLists(protolists, protocount); // 将category的protocol
添加到class中
    free(protolists);
}
```

到此为止，我们就完成了`category`的加载工作。可以看到，最终，`cateogry`被加入到了对应`class`的方法，协议以及属性列表中。

最后我们再看一下`attachLists`方法是如何将两个list合二为一的：

```
    void attachLists(List* const * addedLists, uint32_t addedCount) {
        if (addedCount == 0) return;

        if (hasArray()) {
            // many lists -> many lists
            uint32_t oldCount = array()->count;
            uint32_t newCount = oldCount + addedCount;
            setArray((array_t *)realloc(array(),
array_t::byteSize(newCount)));
            array()->count = newCount;
            memmove(array()->lists + addedCount, array()->lists,
                    oldCount * sizeof(array()->lists[0]));
            memcpy(array()->lists, addedLists,
                   addedCount * sizeof(array()->lists[0]));
        }
        else if (!list  &&  addedCount == 1) {
            // 0 lists -> 1 list
            list = addedLists[0];
        }
        else {
            // 1 list -> many lists
            List* oldList = list;
            uint32_t oldCount = oldList ? 1 : 0;
            uint32_t newCount = oldCount + addedCount;
            setArray((array_t *)malloc(array_t::byteSize(newCount)));
            array()->count = newCount;
            if (oldList) array()->lists[addedCount] = oldList;
            memcpy(array()->lists, addedLists,
                   addedCount * sizeof(array()->lists[0]));
        }
    }
```

仔细看会发现，`attachLists`方法其实是使用的‘头插’的方式将新的list插入原有list中的。即，新的list会插入到原始list的头部。

这也就说明了，为什么`category`中的方法，会‘覆盖’`class`的原始方法。其实并没有真正的‘覆盖’，而是由于cateogry中的方法被排到了原始方法的前面，那么在消息查找流程中，会返回首先被查找到的cateogry方法的实现。

category和+load方法

在面试时，可能被问到这样的问题：

在类的`+load`方法中，可以调用分类方法吗？ 要回答这个问题，其实要搞清load方法的调用时机和`category`附加到`class`上的先后顺序。

如果在`load`方法被调用前，`category`已经完成了附加到`class`上的流程，则对于上面的问题，答案是肯定的。

我们回到runtime的入口函数来看一下，

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

    // fixme defer initialization until an objc-using image is found?
    environ_init();
    tls_init();
    static_init();
    lock_init();
    exception_init();

    _dyld_objc_notify_register(&map_images, load_images, unmap_image);
}
```

runtime在入口点分别向`dyld`注册了三个事件监听：`mapped oc sections`， `init oc section`以及`unmapped oc sections`。

而这三个事件的顺序是: `mapped oc sections` -> `init oc section` -> `unmapped oc sections`

在`mapped oc sections`事件中，我们已经看过其源码，runtime会依次读取Mach-O文件中的`oc sections`，并根据这些信息来初始化runtime环境。这其中就包括`cateogry`的加载。

之后，当runtime环境都初始化完毕，在`dyld`的`init oc section`事件中，runtime会调用每一个加载到内存中的类的`+load`方法。

这里我们注意到，`+load`方法的调用是在`cateogry`加载之后的。因此，在`+load`方法中，是可以调用`category`方法的。

调用已被`category`‘覆盖’的方法

前面我们已经知道，类中的方法并不是真正的被`category`‘覆盖’，而是被放到了类方法列表的后面，消息查找时找不到而已。我们当然也可以手动来找到并调用它，代码如下：

```
@interface Son : NSObject
- (void)sayHi;
@end

@implementation Son
- (void)sayHi {
    NSLog(@"Son say hi!");
}
@end

// son 的分类，覆写了sayHi方法
@interface Son (Good)
```

```
- (void)sayHi;
- (void)saySonHi;
@end

- (void)sayHi {
    NSLog(@"Son's category good say hi");
}

- (void)saySonHi {
    unsigned int methodCount = 0;
    Method *methodList = class_copyMethodList([self class], &methodCount);

    SEL sel = @selector(sayHi);
    NSString *originalSelName = NSStringFromSelector(sel);
    IMP lastIMP = nil;
    for (NSInteger i = 0; i < methodCount; ++i) {
        Method method = methodList[I];
        NSString *selName = NSStringFromSelector(method_getName(method));
        if ([originalSelName isEqualToString:selName]) {
            lastIMP = method_getImplementation(method);
        }
    }

    if (lastIMP != nil) {
        typedef void(*fn)(id, SEL);
        fn f = (fn)lastIMP;
        f(self, sel);
    }
    free(methodList);

}

// 分别调用sayHi 和 saySonHi
Son *mySon1 = [Son new];
[mySon1 sayHi];
[mySon1 saySonHi];
```

输出为：

果然，我们调用到了原始的`sayHi`方法。

category和关联对象

众所周知，`category`是不支持向类添加实例变量的。这在源码中也可以看出，`cateogry`仅支持实例方法、类方法、协议、和实例属性（注意，实例属性并不等于实例变量）。

但是，runtime也给我提供了一个折中的方式，虽然不能够向类添加实例变量，但是runtime为我们提供了方法，可以向类的实例对象添加关联对象。

所谓关联对象，就是为目标对象添加一个关联的对象，并能够通过`key`来查找到这个关联对象。说的形象一点，就像我们去跳舞，runtime可以给我们分配一个舞伴一样。

这种关联是对象和对象级别的，而不是类层次上的。当你为一个类实例添加一个关联对象后，如果你再创建另一个类实例，这个新建的实例是没有关联对象的。

我们可以通过重写`set/get`方法的形式，来自动为我们的实例添加关联对象。

```
MyClass+Category1.h:

#import "MyClass.h"

@interface MyClass (Category1)

@property(nonatomic,copy) NSString *name;

@end
```

```
MyClass+Category1.m:

#import "MyClass+Category1.h"
#import <objc/runtime.h>

@implementation MyClass (Category1)

+ (void)load
{
    NSLog(@"%@",@"load in Category1");
}

- (void)setName:(NSString *)name
{
    objc_setAssociatedObject(self,
                             "name",
                             name,
                             OBJC_ASSOCIATION_COPY);
}

- (NSString*)name
{
    NSString *nameObject = objc_getAssociatedObject(self, "name");
    return nameObject;
}

@end
```

代码很简单，我们重点关注一下其背后的实现。

objc_setAssociatedObject

我们要设置关联对象，需要调用`objc_setAssociatedObject`方法将对象关联到目标对象上。我们需要传入4个参数：`target object`，`associated key`，`associated value`，`objc_AssociationPolicy`。

`objc_AssociationPolicy`是一个枚举，可以取值为：

```
typedef OBJC_ENUM(uintptr_t, objc_AssociationPolicy) {
    OBJC_ASSOCIATION_ASSIGN = 0,           /**< Specifies a weak reference to
the associated object. */
    OBJC_ASSOCIATION_RETAIN_NONATOMIC = 1, /**< Specifies a strong reference
to the associated object.
                                            *   The association is not made
atomically. */
    OBJC_ASSOCIATION_COPY_NONATOMIC = 3,   /**< Specifies that the associated
object is copied.
                                            *   The association is not made
atomically. */
    OBJC_ASSOCIATION_RETAIN = 01401,       /**< Specifies a strong reference
to the associated object.
                                            *   The association is made
atomically. */
    OBJC_ASSOCIATION_COPY = 01403          /**< Specifies that the associated
object is copied.
                                            *   The association is made
atomically. */
};
```

分别和property的属性定义一一匹配。

当我们为对象设置关联对象的时候，所关联的对象到底存在了那里呢？我们看源码：

```
void objc_setAssociatedObject(id object, const void *key, id value,
objc_AssociationPolicy policy) {
    _object_set_associative_reference(object, (void *)key, value, policy);
}
```

```
void _object_set_associative_reference(id object, void *key, id value,
uintptr_t policy) {
    // retain the new value (if any) outside the lock.
    ObjcAssociation old_association(0, nil);
    id new_value = value ? acquireValue(value, policy) : nil;
    {
```

```
        AssociationsManager manager; // 这是一个单例，内部保存一个全局的static
AssociationsHashMap *_map; 用于保存所有的关联对象。
        AssociationsHashMap &associations(manager.associations());
        disguised_ptr_t disguised_object = DISGUISE(object); // 取反object 地址
作为accociative key
        if (new_value) {
            // break any existing association.
            AssociationsHashMap::iterator i =
associations.find(disguised_object);
            if (i != associations.end()) {
                // secondary table exists
                ObjectAssociationMap *refs = i->second;
                ObjectAssociationMap::iterator j = refs->find(key);
                if (j != refs->end()) {
                    old_association = j->second;
                    j->second = ObjcAssociation(policy, new_value);
                } else {
                    (*refs)[key] = ObjcAssociation(policy, new_value);
                }
            } else {
                // create the new association (first time).
                ObjectAssociationMap *refs = new ObjectAssociationMap;
                associations[disguised_object] = refs;
                (*refs)[key] = ObjcAssociation(policy, new_value);
                object->setHasAssociatedObjects(); // 将object标记为 has
AssociatedObjects
            }
        } else { // 如果传入的关联对象值为nil，则断开关联
            // setting the association to nil breaks the association.
            AssociationsHashMap::iterator i =
associations.find(disguised_object);
            if (i !=  associations.end()) {
                ObjectAssociationMap *refs = i->second;
                ObjectAssociationMap::iterator j = refs->find(key);
                if (j != refs->end()) {
                    old_association = j->second;
                    refs->erase(j);
                }
            }
        }
    }
    // release the old value (outside of the lock).
    if (old_association.hasValue()) ReleaseValue()(old_association); // 释放掉
old关联对象。(如果多次设置同一个key的value,这里会释放之前的value)
}
```

大体流程为：

1. 根据关联的policy，调用`id new_value = value ? acquireValue(value, policy) :`

`nil;`，`acquireValue`方法会根据`poilcy`是`retain`或`copy`，对`value`做引用`+1`操作或`copy`操作，并返回对应的`new_value`。（如果传入的`value`为`nil`，则返回`nil`，不做任何操作）`acquireValue`实现代码是：

```
static id acquireValue(id value, uintptr_t policy) {
    switch (policy & 0xFF) {
    case OBJC_ASSOCIATION_SETTER_RETAIN:
        return objc_retain(value);
    case OBJC_ASSOCIATION_SETTER_COPY:
        return ((id(*)(id, SEL))objc_msgSend)(value, SEL_copy);
    }
    return value;
}
```

2. 获取到`new_value`后，根据是否有`new_value`的值，进入不同流程。如果`new_value`存在，则对象与目标对象关联。实质是存入到全局单例`AssociationsManager manager`的对象关联表中。如果`new_value`不存在，则释放掉之前目标对象及关联`key`所存储的关联对象。实质是在`AssociationsManager`中删除掉关联对象。
3. 最后，释放掉之前以同样`key`存储的关联对象。

其中，起到关键作用的在于`AssociationsManager manager`，它是一个全局单例，其成员变量为`static AssociationsHashMap *_map`，用于存储目标对象及其关联的对象。`_map`中的数据存储结构如下图所示：

仔细看这一段代码，会发现有个问题：当我们第一次为目标对象创建关联对象时，会在`AssociationsManager manager`的`ObjectAssociationMap`中插入一个以`disguised_object`为`key`的节点，用于存储该目标对象所关联的对象。

但是，上面代码中，仅有释放`old_association`关联对象的代码，而没有释放保存在`AssociationsManager manager`中节点的代码。那么，`AssociationsManager manager`中的节点是什么时候被释放的呢?

在对象的销毁逻辑里，会调用`objc_destructInstance`，实现如下：

```
void *objc_destructInstance(id obj)
{
    if (obj) {
        // Read all of the flags at once for performance.
        bool cxx = obj->hasCxxDtor();
        bool assoc = obj->hasAssociatedObjects();

        // This order is important.
        if (cxx) object_cxxDestruct(obj); // 调用C++析构函数
        if (assoc) _object_remove_assocations(obj); // 移除所有的关联对象，并将其自身从AssociationsManager的map中移除
        obj->clearDeallocating(); // 清理ARC ivar
```

```
    }

    return obj;
}
```

`obj`的关联对象会在`_object_remove_assocations`方法中全部移除，同时，会将`obj`自身从`AssociationsManager`的`map`中移除：

```
void _object_remove_assocations(id object) {
    vector< ObjcAssociation,ObjcAllocator<ObjcAssociation> > elements;
    {
        AssociationsManager manager;
        AssociationsHashMap &associations(manager.associations());
        if (associations.size() == 0) return;
        disguised_ptr_t disguised_object = DISGUISE(object);
        AssociationsHashMap::iterator i = associations.find(disguised_object);
        if (i != associations.end()) {
            // copy all of the associations that need to be removed.
            ObjectAssociationMap *refs = i->second;
            for (ObjectAssociationMap::iterator j = refs->begin(), end = refs-
>end(); j != end; ++j) {
                elements.push_back(j->second);
            }
            // remove the secondary table.
            delete refs;
            associations.erase(i);
        }
    }
    // the calls to releaseValue() happen outside of the lock.
    for_each(elements.begin(), elements.end(), ReleaseValue());
}
```

概述

当我们创建一个对象时：

```
SWHunter *hunter = [[SWHunter alloc] init];
```

上面这行代码在`栈`上创建了`hunter`指针，并在`堆`上创建了一个`SWHunter`对象。目前，iOS并不支持在栈上创建对象。

iOS 内存分区

iOS的内存管理是基于虚拟内存的。虚拟内存能够让每一个进程都能够在逻辑上“独占”整个设备的内存。

iOS又将虚拟内存按照地址由低到高划分为如下五个区：

- 代码区： 存放APP二进制代码
- 常量区：存放程序中定义的各种常量， 包括字符串常量，各种被`const`修饰的常量
- 全局/静态区: 全局变量，静态变量就放在这里
- 堆区：在程序运行时调用`alloc`， `copy`， `mutablecopy`， `new`会在堆上分配内存。堆内存需要程序员手动释放，这在ARC中是通过引用计数的形式表现的。堆分配地址不连续，但整体是地址从低到高地址分配
- 栈区：存放局部变量，当变量超出作用域时，内存会被系统自动释放。栈上的地址连续分配，在内存地址由高向低增长

> 在程序运行时，代码区，常量区以及全局静态区的大小是固定的，会变化的只有栈和堆的大小。而栈的内存是有操作系统自动释放的，我们平常说所的iOS内存引用计数，其实是针对堆上的对象来说的。

下面，我们就来看一下，在runtime中，是如何通过引用计数来管理内存的。

tagged pointer

首先，来想这么一个问题，在平常的编程中，我们使用的NSNumber对象来表示数字，最大会有多大？几万？几千万？甚至上亿？

我相信，对于绝大多数程序来说，用不到上亿的数字。同样，对于字符串类型，绝大多数时间，字符个数也在8个以内。

再想另一个方面，自2013年苹果推出iphone5s之后，iOS的寻址空间扩大到了64位。我们可以用63位来表示一个数字（一位做符号位），这是一个什么样的概念？231=2147483648，也达到了20多亿，而263这个数字，用到的概率基本为零。比如NSNumber *num=@10000的话，在内存中则会留下很多无用的空位。这显然浪费了内存空间。

苹果当然也发现了这个问题，于是就引入了`tagged pointer`。`tagged pointer`是一种特殊的“指针”，其特殊在于，其实它存储的并不是地址，而是真实的数据和一些附加的信息。

在引入`tagged pointer` 之前，iOS对象的内存结构如下所示:

显然，本来4字节就可以表示的数值，现在却用了8字节，明显的内存浪费。而引入了`tagged pointer` 后, 其内存布局如下:

可以看到，利用`tagged pointer`后，“指针”又存储了其本身，也存储了和对象相关的标记。这时的`tagged pointer`里面存储的不是地址，而是一个数据集合。同时，其占用的内存空间也由16字节缩减为8字节。

我们可以在WWDC2013的《Session 404 Advanced in Objective-C》视频中，看到苹果对于Tagged Pointer特点的介绍：

1. Tagged Pointer专门用来存储小的对象，例如NSNumber, NSDate, NSString。
2. Tagged Pointer指针的值不再是地址了，而是真正的值。所以，实际上它不再是一个对象了，它只是一个披着对象皮的普通变量而已。所以，它的内存并不存储在堆中，也不需要malloc和free。
3. 在内存读取上有着3倍的效率，创建时比以前快106倍。 运行如下代码：

```
    NSMutableString *mutableStr = [NSMutableString string];
    NSString *immutable = nil;
  #define _OBJC_TAG_MASK (1UL<<63)
    char c = 'a';
    do {
        [mutableStr appendFormat:@"%c", c++];
        immutable = [mutableStr copy];
        NSLog(@"%p %@ %@", immutable, immutable, immutable.class);
    }while(((uintptr_t)immutable & _OBJC_TAG_MASK) == _OBJC_TAG_MASK);
```

输出为：

我们看到，字符串由‘a’增长到‘abcdefghi’的过程中，其地址开头都是0xa 而结尾也很有规律，是1到9递增，正好对应着我们的字符串长度，同时，其输出的class类型为`NSTaggedPointerString`。在字符串长度在`9`个以内时，iOS其实使用了`tagged pointer`做了优化的。

直到字符串长度大于`9`，字符串才真正成为了`__NSCFString`类型。

我们回头分析一下上面的代码。 首先，iOS需要一个标志位来判断当前指针是真正的指针还是`tagged pointer`。这里有一个宏定义`_OBJC_TAG_MASK (1UL<<63)`，它表明如果64位数据中，最高位是`1`的话，则表明当前是一个`tagged pointer`类型。

然后，既然使用了`tagged pointer`，那么就失去了iOS对象的数据结构，但是，系统还是需要有个标志位表明当前的`tagged pointer`表示的是什么类型的对象。这个标志位，也是在最高4位来表示的。我们将`0xa`转换为二进制，得到`1010`，其中最高位`1xxx`表明这是一个`tagged pointer`，而剩下的3位`010`，表示了这是一个`NSString`类型。`010`转换为十进制即为`2`。也就是说，标志位是`2`的`tagger pointer`表示这是一个`NSString`对象。

在runtime源码的objc-internal.h中，有关于标志位的定义如下：

```
{
    OBJC_TAG_NSAtom            = 0,
    OBJC_TAG_1                 = 1,
    OBJC_TAG_NSString          = 2,
    OBJC_TAG_NSNumber          = 3,
    OBJC_TAG_NSIndexPath       = 4,
    OBJC_TAG_NSManagedObjectID = 5,
    OBJC_TAG_NSDate            = 6,
    OBJC_TAG_RESERVED_7        = 7,

    OBJC_TAG_First60BitPayload = 0,
    OBJC_TAG_Last60BitPayload  = 6,
```

```
    OBJC_TAG_First52BitPayload = 8,
    OBJC_TAG_Last52BitPayload  = 263,

    OBJC_TAG_RESERVED_264      = 264
};
```

最后，让我们再尝试分析一下`NSString`类型的`tagged pointer`是如何实现的。

我们前面已经知道，在总共64位数据中，高4位被用于标志`tagged pointer`以及对象类型标识。低`1`位用于记录字符串字符个数，那么还剩下`59`位可以让我们表示数据内容。

对于字符串格式，怎么来表示内容呢？自然的，我们想到了`ASCII码`。对应`ASCII码`，`a`用16进制ASCII码表示为`0x61`，`b`为`0x62`，依次类推。在字符串长度增加到8个之前，`tagged pointer`的内容如下。可以看到，从最低2位开始，分别为61，62，63... 这正对应了字符串中字符的ASCII码。

直到字符串增加到`7`个之上，我们仍然可以分辨出`tagged pointer`中的标志位以及字符串长度，但是中间的内容部分，却不符合ASCII的编码规范了。

这是因为，iOS对字符串使用了压缩算法，使得`tagged pointer`表示的字符串长度最大能够达到`9`个。关于具体的压缩算法，我们就不再讨论了。由于苹果内部会对实现逻辑作出修改，因此我们只要知道有`tagged pointer`的概念就好了。有兴趣的同学可以看采用`Tagged Pointer`的字符串，但其内容也有些过时了，和我们的实验结果并不一致。

我们顺便看一下NSNumber的tagged pointer实现：

```
    NSNumber *number1 = @(0x1);
    NSNumber *number2 = @(0x20);
    NSNumber *number3 = @(0x3F);
    NSNumber *numberFFFF = @(0xFFFFFFFFFFFEFE);
    NSNumber *maxNum = @(MAXFLOAT);
    NSLog(@"number1 pointer is %p class is %@", number1, number1.class);
    NSLog(@"number2 pointer is %p class is %@", number2, number2.class);
    NSLog(@"number3 pointer is %p class is %@", number3, number3.class);
    NSLog(@"numberffff pointer is %p class is %@", numberFFFF,
numberFFFF.class);
    NSLog(@"maxNum pointer is %p class is %@", maxNum, maxNum.class);
```

可以看到，对于`MAXFLOAT`，系统无法进行优化，输出的是一个正常的`NSNumber`对象地址。而对于其他的`number`值，系统采用了`tagged pointer`，其'地址'都是以`0xb`开头，转换为二进制就是`1011`，首位`1`表示这是一个`tagged pointer`，而`011`转换为十进制是`3`，参考前面`tagged pointer`的类型枚举，这是一个`NSNumber`类型。接下来几位，就是以16进制表示的`NSNumber`的值，而对于最后一位，应该是一个标志位，具体作用，笔者也不是很清楚。

isa

由于一个`tagged pointer`所指向的并不是一个真正的OC对象，它其实是没有`isa`属性的。

在runtime中，可以这样获取isa的内容：

```
#define _OBJC_TAG_SLOT_SHIFT 60
#define _OBJC_TAG_EXT_SLOT_MASK 0xff

inline Class
objc_object::getIsa()
{
  // 如果不是tagged pointer，则返回ISA()
    if (!isTaggedPointer()) return ISA();

  // 如果是tagged pointer，取出高4位的内容，查找对应的class
    uintptr_t ptr = (uintptr_t)this;

    uintptr_t slot = (ptr >> _OBJC_TAG_SLOT_SHIFT) & _OBJC_TAG_SLOT_MASK;
    return objc_tag_classes[slot];

}
```

在runtime中，还有专用的方法用于判断指针是`tagged pointer`还是普通指针：

```
#   define _OBJC_TAG_MASK (1UL<<63)
static inline bool
_objc_isTaggedPointer(const void * _Nullable ptr)
{
    return ((uintptr_t)ptr & _OBJC_TAG_MASK) == _OBJC_TAG_MASK;
}
```

isa 指针（NONPOINTER_ISA）

对象的`isa`指针，用来表明对象所属的类类型。 但是如果`isa`指针仅表示类型的话，对内存显然也是一个极大的浪费。于是，就像`tagged pointer`一样，对于`isa`指针，苹果同样进行了优化。`isa`指针表示的内容变得更为丰富，除了表明对象属于哪个类之外，还附加了引用计数`extra_rc`，是否有被`weak`引用标志位`weakly_referenced`，是否有附加对象标志位`has_assoc`等信息。

这里，我们仅关注`isa`中和内存引用计数有关的`extra_rc` 以及相关内容。

首先，我们回顾一下`isa`指针是怎么在一个对象中存储的。下面是runtime相关的源码：

```
@interface NSObject <NSObject> {
    Class isa  OBJC_ISA_AVAILABILITY;
}

typedef struct objc_class *Class;
```

```
// ============ 注意！从这一行开始，其定义就和在XCode中objc.h看到的定义不一致，我们需要阅读runtime的源码，才能看到其真实的定义！下面是简化版的定义：============
struct objc_class : objc_object {
    Class superclass;
    cache_t cache;             // formerly cache pointer and vtable
    class_data_bits_t bits;    // class_rw_t * plus custom rr/alloc flags
}


struct objc_object {
private:
    isa_t isa;
}

union isa_t
{
    isa_t() { }
    isa_t(uintptr_t value) : bits(value) { }

    Class cls;
    uintptr_t bits;

# if __arm64__
#   define ISA_MASK        0x0000000ffffffff8ULL
#   define ISA_MAGIC_MASK  0x000003f000000001ULL
#   define ISA_MAGIC_VALUE 0x000001a000000001ULL
    struct {
        uintptr_t nonpointer        : 1;
        uintptr_t has_assoc         : 1;
        uintptr_t has_cxx_dtor      : 1;
        uintptr_t shiftcls          : 33; // MACH_VM_MAX_ADDRESS 0x1000000000
        uintptr_t magic             : 6;
        uintptr_t weakly_referenced : 1;
        uintptr_t deallocating      : 1;
        uintptr_t has_sidetable_rc  : 1;
        uintptr_t extra_rc          : 19;
#       define RC_ONE   (1ULL<<45)
#       define RC_HALF  (1ULL<<18)
    };
}
```

结合下面的图，我们可以更清楚的了解runtime中对象和类的结构定义，显然，类也是一种对象，这就是类对象的含义。

从图中可以看出，我们所谓的`isa`指针，最后实际上落脚于`isa_t`的`联合类型`。`联合类型`是C语言中的一种类型，简单来说，就是一种`n选1`的关系。比如`isa_t`中包含有`cls`，`bits`，`struct`三个变量，它们的内存空间是重叠的。在实际使用时，仅能够使用它们中的一种，你把它当做`cls`，就不能当`bits`访问，你把它当`bits`，就不能用`cls`来访问。

联合的作用在于，用更少的空间，表示了更多的可能的类型，虽然这些类型是不能够共存的。

将注意力集中在`isa_t`联合上，我们该怎样理解它呢?

首先它有两个构造函数`isa_t()`,`isa_t(uintptr_value)`,这两个定义很清晰，无需多言。

然后它有三个数据成员`Class cls`,`uintptr_t bits`,`struct`。其中`uintptr_t`被定义为`typedef unsigned long uintptr_t`，占据64位内存。

关于上面三个成员，`uintptr_t bits`和`struct`其实是一个成员，它们都占据64位内存空间，之前已经说过，联合类型的成员内存空间是重叠的。在这里，由于`uintptr_t bits`和`struct`都是占据64位内存，因此它们的内存空间是完全重叠的。而你将这块64位内存当做是`uintptr_t bits`还是`struct`，则完全是逻辑上的区分，在内存空间上，其实是一个东西。即`uintptr_t bits`和`struct`是一个东西的两种表现形式。

实际上在runtime中，任何对`struct`的操作和获取某些值，如`extra_rc`，实际上都是通过对`uintptr_t bits`做位操作实现的。`uintptr_t bits`和`struct`的关系可以看做，`uintptr_t bits`向外提供了操作`struct`的接口，而`struct`本身则说明了`uintptr_t bits`中各个二进制位的定义。

理解了`uintptr_t bits`和`struct`关系后，则`isa_t`其实可以看做有两个可能的取值，`Class cls`或`struct`。如下图所示：

当`isa_t`作为`Class cls`使用时，这符合了我们之前一贯的认知：`isa`是一个指向对象所属`Class`类型的指针。然而，仅让一个64位的指针表示一个类型，显然不划算。

因此，绝大多数情况下，苹果采用了优化的`isa`策略，即，`isa_t`类型并不等同而`Class cls`,而是`struct`。这种情况对于我们自己创建的类对象以及系统对象都是如此，稍后我们会对这一结论进行验证。

先让我们集中精力来看一下struct的结构：

```
# if __arm64__
#   define ISA_MASK        0x0000000ffffffff8ULL
#   define ISA_MAGIC_MASK  0x000003f000000001ULL
#   define ISA_MAGIC_VALUE 0x000001a000000001ULL
    struct {
        uintptr_t nonpointer        : 1;
        uintptr_t has_assoc         : 1;
        uintptr_t has_cxx_dtor      : 1;
        uintptr_t shiftcls          : 33; // MACH_VM_MAX_ADDRESS 0x1000000000
        uintptr_t magic             : 6;
        uintptr_t weakly_referenced : 1;
```

```
        uintptr_t deallocating      : 1;
        uintptr_t has_sidetable_rc  : 1;
        uintptr_t extra_rc          : 19;
#       define RC_ONE   (1ULL<<45)
#       define RC_HALF  (1ULL<<18)
    };
```

`struct`共占用64位，从低位到高位依次是`nonpointer`到`extra_rc`。成员后面的`：`表明了该成员占用几个`bit`。成员的含义如下：

成员	位	含义
nonpointer	1bit	标志位。1(奇数)表示开启了isa优化，0(偶数)表示没有启用isa优化。所以，我们可以通过判断isa是否为奇数来判断对象是否启用了isa优化。
has_assoc	1bit	标志位。表明对象是否有关联对象。没有关联对象的对象释放的更快。

|has_cxx_dtor |1bit |标志位。表明对象是否有C++或ARC析构函数。没有析构函数的对象释放的更快。 |shiftcls |33bit |类指针的非零位。| |magic |6bit |固定为0x1a，用于在调试时区分对象是否已经初始化。| |weakly_referenced |1bit |标志位。用于表示该对象是否被别的对象弱引用。没有被弱引用的对象释放的更快。| |deallocating |1bit |标志位。用于表示该对象是否正在被释放。| |has_sidetable_rc |1bit |标志位。用于标识是否当前的引用计数过大，无法在isa中存储，而需要借用sidetable来存储。（这种情况大多不会发生）| |extra_rc |19bit |对象的引用计数减1。比如，一个object对象的引用计数为7，则此时extra_rc的值为6。|

由上表可以看出，和对象引用计数相关的有两个成员：`extra_rc`和`has_sidetable_rc`。iOS用19位的`extra_rc`来记录对象的引用次数，当`extra_rc`不够用时，还会借助`sidetable`来存储计数值，这时，`has_sidetable_rc`会被标志为`1`。

我们可以算一下，对于19位的`extra_rc`，其数值可以表示`2^19 - 1 = 524287`。52万多，相信绝大多数情况下，都够用了。

现在，我们来真正的验证一下，我们上述的结论。注意，做验证试验时，必须要使用真机，因为模拟器默认是不开启`isa`优化的。

要做验证试验，我们必须要得到`isa_t`的值。在苹果提供的公共接口中，是无法获取到它的。不过，通过对象指针，我们确实是可以获取到`isa_t`的值。

让我们看一下当我们创建一个对象时，实际上是获得到了什么。

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

我们得到了`obj`这个对象，实质上`obj`是一个指向对象的指针，即`obj == NSObject *`。

而在`NSObject`中，又有唯一的成员`Class isa`，而`Class`实质上是`objc_class *`。这样，我们可以用`objc_class *`替换掉`NSObject`，得到`obj == objc_class **`

再看objc_class的定义：

```
struct objc_class : objc_object {
  。。。
}
```

`objc_class`继承自`objc_object`， 因此，在`objc_class` 内存布局的首地址肯定存放的是继承自`objc_object`的内容。从内存布局的角度，我们可以将`objc_class`替换为 `objc_object` 。得到：`obj == objc_object **`

而`objc_object`的定义如下，仅含有一个成员`isa_t`:

```
struct objc_object {
private:
    isa_t isa;
}
```

因此，我们又可以将`objc_object` 替换为`isa_t`。得到：`obj == isa_t **`

好了，这里到了关键的地方，从现在看，我们得到的`obj`应该是一个指向 `isa_t *` 的指针，即 `obj` 是一个指针的指针， `obj`指向一个指针。 但是， `obj`真的是指向了一个指针吗?

我们再来看一下`isa_t`的定义，我们看标志为***注意！！！*** 的地方:

```
# if __arm64__
#   define ISA_MASK        0x0000000ffffffff8ULL
#   define ISA_MAGIC_MASK  0x000003f000000001ULL
#   define ISA_MAGIC_VALUE 0x000001a000000001ULL
    struct {
        uintptr_t nonpointer        : 1;   // 注意！！！ 标志位，表明isa_t *是否是一个真正的指针！！！
        uintptr_t has_assoc         : 1;
        uintptr_t has_cxx_dtor      : 1;
        uintptr_t shiftcls          : 33; // MACH_VM_MAX_ADDRESS 0x1000000000
        uintptr_t magic             : 6;
        uintptr_t weakly_referenced : 1;
        uintptr_t deallocating      : 1;
        uintptr_t has_sidetable_rc  : 1;
        uintptr_t extra_rc          : 19;
#       define RC_ONE   (1ULL<<45)
#       define RC_HALF  (1ULL<<18)
    };
```

也就是说，当开启了`isa_t`优化， `nonpointer`置位为1， 这时， `isa_t *`其实不是一个地址，而是一个实实在在有意义的值，也就是说，苹果用`isa_t *` 所占用的64位空间，表示了一个有意义的值，而这64位值的定义，就符合我们上面`struct`的定义。

这时，我们可以将`isa_t *`改写为`isa_t`，这是因为`isa_t *`的64位并没有指向任何地址，而是实际表示了`isa_t`的内容。

继续上面的公式推导，得到结论:`obj == *isa_t`

哈哈，有意思吗？`obj`实际上是指向`isa_t`的指针。绕了这里大一圈，结论竟如此直白。

如果我们想得到`isa_t`的值，只需要做`*obj`操作即可，即

```
NSLog(@"isa_t = %p", *obj);
```

之所以用%p输出，是因为我们要`isa_t*`本身的值，而不是要取它指向的值。

得出了这个结论，我们就可以通过`obj`打印出`isa_t`中存储的内容了（中间需要做几次类型转换，但是实质和上面是一样的）：

```
NSLog(@"isa_t = %p", *(void **)(__bridge void*)obj);
```

我们的实验代码如下：

```
@interface MyObj : NSObject
@end
@implementation MyObj
@end

@interface ViewController ()
@property(nonatomic, strong) MyObj *obj1;
@property(nonatomic, strong) MyObj *obj2;
@property(nonatomic, weak) MyObj *weakRefObj;
@end

@implementation ViewController
- (void)viewDidLoad {
    [super viewDidLoad];
    MyObj *obj = [[MyObj alloc] init];
    NSLog(@"1. obj isa_t = %p", *(void **)(__bridge void*)obj);
    _obj1 = obj;
    MyObj *tmpObj = obj;
    NSLog(@"2. obj isa_t = %p", *(void **)(__bridge void*)obj);
}

- (void)viewDidAppear:(BOOL)animated {
    [super viewDidAppear:animated];
    NSLog(@"3. obj isa_t = %p", *(void **)(__bridge void*)_obj1);
    _obj2 = _obj1;
    NSLog(@"4. obj isa_t = %p", *(void **)(__bridge void*)_obj1);
    _weakRefObj = _obj1;
    NSLog(@"5. obj isa_t = %p", *(void **)(__bridge void*)_obj1);
    NSObject *attachObj = [[NSObject alloc] init];
    objc_setAssociatedObject(_obj1, "attachKey", attachObj,
OBJC_ASSOCIATION_RETAIN_NONATOMIC);
    NSLog(@"6. obj isa_t = %p", *(void **)(__bridge void*)_obj1);
```

```
}
@end
```

其输出为：

直观的可以看到isa_t的内容都是奇数，说明开启了`isa`优化。（`nonpointer == 1`）

接下来我们一行行的分析代码以及相应的`isa_t`内容变化：

首先在`viewDidLoad`方法中，我们创建了一个`MyObj`实例，并接着打印出`isa_t`的内容，这时候，`MyObj`的引用计数应该是`1`：

```
- (void)viewDidLoad {
    ...
    MyObj *obj = [[MyObj alloc] init];
    NSLog(@"1. obj isa_t = %p", *(void **)(__bridge void*)obj);
    ...
}
```

对应的输出内容为`0x1a1000a0ff9`：

大家可以在图中直观的看到`isa_t`此时各位的内容，注意到`extra_rc`此时为`0`，因为引用计数等于`extra_rc + 1`，因此，`MyObj`对象的引用计数为`1`，和我们的预期一致。

接下来执行

```
  _obj1 = obj;
    MyObj *tmpObj = obj;
    NSLog(@"2. obj isa_t = %p", *(void **)(__bridge void*)obj);
```

由于`_obj1`对`MyObj`对象是强引用，同时，`tmpObj`的赋值也默认是强引用，`obj`的引用计数加`2`，应该等于`3`。

输出为`0x41a1000a0ff9`：

引用计数等于`extra_rc + 1 = 2 + 1 = 3`，符合预期。

然后，程序执行到了`viewDidAppear`方法，并立刻输出`MyObj`对象的引用计数。因为此时栈上变量`obj`，`tmpObj`已经释放，因此引用计数应该`减2`，等于`1`。

```
- (void)viewDidAppear:(BOOL)animated {
    [super viewDidAppear:animated];
    NSLog(@"3. obj isa_t = %p", *(void **)(__bridge void*)_obj1);
  ...
}
```

输出为 `0x1a1000a0ff9`:

引用计数等于`extra_rc + 1 = 0 + 1 = 1`,符合预期。

接下来我们又赋值了一个强引用`_obj2`,引用计数`加1`，等于`2`。

```
  ...
   _obj2 = _obj1;
    NSLog(@"4. obj isa_t = %p", *(void **)(__bridge void*)_obj1);
  ...
```

输出为`0x21a1000a0ff9`:

引用计数等于`extra_rc + 1 = 1 + 1 = 2`,符合预期。

接下来，我们又将`MyObj`对象赋值给一个`weak`引用，此时，引用计数应该保持不变，但是`weakly_referenced`位应该置`1`。

```
  ...
  _weakRefObj = _obj1;
    NSLog(@"5. obj isa_t = %p", *(void **)(__bridge void*)_obj1);
    ...
```

输出`0x25a1000a0ff9`:

可以看到引用计数仍是`2`，但是`weakly_referenced`位已经置位`1`，符合预期。

最后，我们向`MyObj对象`添加了一个关联对象，此时，`isa_t`的其他位应该保持不变，只有`has_assoc`标志位应该置位`1`。

```
  ...
  NSObject *attachObj = [[NSObject alloc] init];
    objc_setAssociatedObject(_obj1, "attachKey", attachObj,
OBJC_ASSOCIATION_RETAIN_NONATOMIC);
    NSLog(@"6. obj isa_t = %p", *(void **)(__bridge void*)_obj1);
    ...
```

输出`0x25a1000a0ffb`：

可以看到，其他位保持不变，只有`has_assoc`被设置为`1`，符合预期。

OK，通过上面的分析，你现在应该很清楚rumtime里面`isa`究竟是怎么回事了吧?

> PS: 笔者所实验的环境为iPhone5s + iOS 10。

SideTable

其实在绝大多数情况下，仅用优化的`isa_t`来记录对象的引用计数就足够了。只有在`19位`的`extra_rc`盛放不了那么大的引用计数时，才会借助`SideTable`出马。

`SideTable`是一个全局的引用计数表，它记录了所有对象的引用计数。

为了弄清`extra_rc`和`sidetable`的关系，我们首先看runtime添加对象引用计数时的简化代码。不过在看代码之前，我们需要弄清楚slowpath和fastpath是干啥的。

我们在runtime源码中有时候，有时在if语句中会看到类似下面这些代码：

```
if (fastpath(cls->canAllocFast())){
...
}

if (slowpath(!newisa.nonpointer)) {
...
}
```

其实将`fastpath`和`slowpath`去掉是完全不影响任何功能的。之所以将`fastpath`和`slowpath`放到`if`语句中，是为了告诉编译器，`if`中的条件是大概率(`fastpath`)还是小概率(`slowpath`)事件，从而让编译器对代码进行优化。知道了这些，我们就可以来继续看源码了：

```
#       define RC_HALF  (1ULL<<18)
ALWAYS_INLINE id
objc_object::rootRetain(bool tryRetain, bool handleOverflow)
{
  // 如果是tagged pointer，直接返回this，因为tagged pointer不用记录引用次数
    if (isTaggedPointer()) return (id)this;
    // transcribeToSideTable用于表示extra_rc是否溢出，默认为false
    bool transcribeToSideTable = false;

    isa_t oldisa;
    isa_t newisa;

    do {
        transcribeToSideTable = false;
        oldisa = LoadExclusive(&isa.bits); // 将isa_t提取出来
```

```
        newisa = oldisa;
        if (slowpath(!newisa.nonpointer)) {  // 如果没有采用isa优化，则返回
sidetable记录的内容，此处slowpath表明这不是一个大概率事件
            return sidetable_retain();
        }
        // 如果对象正在析构，则直接返回nil
        if (slowpath(tryRetain && newisa.deallocating)) {
            return nil;
        }
        // 采用了isa优化，做extra_rc++，同时检查是否extra_rc溢出，若溢出，则extra_rc减
半，并将另一半转存至sidetable
        uintptr_t carry;
        newisa.bits = addc(newisa.bits, RC_ONE, 0, &carry);  // extra_rc++

        if (slowpath(carry)) { // 有carry值，表示extra_rc 溢出
            // newisa.extra_rc++ overflowed
            if (!handleOverflow) {  // 如果不处理溢出情况，则在这里会递归调用一次，再
进来的时候，handleOverflow会被rootRetain_overflow设置为true，从而进入到下面的溢出处理流
程
                return rootRetain_overflow(tryRetain);
            }

            // 进行溢出处理：逻辑很简单，先在extra_rc中引用计数减半，同时把
has_sidetable_rc设置为true，表明借用了sidetable。然后把另一半放到sidetable中
            sideTableLocked = true;
            transcribeToSideTable = true;
            newisa.extra_rc = RC_HALF;
            newisa.has_sidetable_rc = true;
        }
    } while (slowpath(!StoreExclusive(&isa.bits, oldisa.bits, newisa.bits)));
// 将oldisa 替换为 newisa，并赋值给isa.bits(更新isa_t)，如果不成功，do while再试一遍

    //isa的extra_rc溢出，将一半的refer count值放到sidetable中
    if (slowpath(transcribeToSideTable)) {
        // Copy the other half of the retain counts to the side table.
        sidetable_addExtraRC_nolock(RC_HALF);
    }

    return (id)this;
}
```

添加对象引用计数的源码逻辑还算清晰，重点看当`extra_rc`溢出后，runtime是怎么处理的。

在iOS中，`extra_rc`占有19位，也就是最大能够表示`2^19-1`, 用二进制表示就是`19个1`。
当`extra_rc`等于`2^19`时，溢出，此时的二进制位是一个`1`后面跟`19个0`， 即`10000...00`。将会溢出的值`2^19除以2`，相当于将`10000...00`向右移动一位。也就等于`RC_HALF(1ULL<<18)`，即一个`1`后面跟`18个0`。

然后，调用

```
sidetable_addExtraRC_nolock(RC_HALF);
```

将另一半的引用计数`RC_HALF`放到`sidetable`中。

SideTable数据结构

在runtime中，通过`SideTable`来管理对象的引用计数以及`weak`引用。这里要注意，一张`SideTable`会管理多个对象，而并非一个。 而这一个个的`SideTable`又构成了一个集合，叫`SideTables`。`SideTables`在系统中是全局唯一的。

`SideTable`， `SideTables`的关系如下图所示（这张图会随着分析的深入逐渐扩充）：

`SideTables`的类型是是`template<typename T> class StripedMap, StripedMap<SideTable>`。我们可以简单的理解为一个`64 * sizeof(SideTable)`的哈希线性数组。

```
#if TARGET_OS_IPHONE && !TARGET_OS_SIMULATOR
    enum { StripeCount = 8 };
#else
    enum { StripeCount = 64 };
#endif
```

每个对象可以通过`StripedMap`所对应的哈希算法，找到其对应的`SideTable`。`StripedMap`的哈希算法如下，其参数是对象的地址。

```
static unsigned int indexForPointer(const void *p) {
        uintptr_t addr = reinterpret_cast<uintptr_t>(p);
        return ((addr >> 4) ^ (addr >> 9)) % StripeCount; // 这里 %StripeCount
保证了所有的对象对应的SideTable均在这个64长度数组中。
    }
```

注意到这个`SideTables`哈希数组是全局的，因此，对于我们APP中所有的对象的引用计数，也就都存在于这些`SideTable`中。

具体到每个`SideTable`， 其中有存储了若干对象的引用计数。`SideTable`的定义如下：

```
struct SideTable {
    spinlock_t slock;
    RefcountMap refcnts;
    weak_table_t weak_table;

    SideTable() {
        memset(&weak_table, 0, sizeof(weak_table));
    }

    ~SideTable() {
        _objc_fatal("Do not delete SideTable.");
    }
};
```

`SideTable`包含三个成员：

- **`spinlock_t slock`**：自旋锁。防止多线程访问`SideTable`冲突
- **`RefcountMap refcnts`**：用于存储对象引用计数的`map`
- **`weak_table_t weak_table`**： 用于存储对象弱引用的`map`

这里我们暂且不去管`weak_table`， 先看存储对象引用计数的成员`RefcountMap refcnts`。

`RefcountMap`类型实际是`DenseMap`，这是一个模板类。

```
typedef objc::DenseMap<DisguisedPtr<objc_object>,size_t,true> RefcountMap;
```

关于`DenseMap`的实际定义，有点复杂，暂时不想看:(

这里只需要将`RefcountMap`简单的的理解为是一个`map`，`key`是`DisguisedPtr<objc_object>`，`value`是对象的引用计数。同时，这个`map`还有个加强版功能，当引用计数为`0`时，会自动将对象数据清除。

这也是`objc::DenseMap<DisguisedPtr<objc_object>,size_t,true> RefcountMap`的含义，即模板类型分别对应：`key,DisguisedPtr<objc_object>`类型。`value，size_t`类型。是否清除为`vlaue==0`的数据，`true`。

`DisguisedPtr<objc_object>`中的采样方法是：

```
    static uintptr_t disguise(T* ptr) {
        return -(uintptr_t)ptr;
    }
    // 将T按照模板替换为objc_object，即是：
    static uintptr_t disguise(objc_object* ptr) {
        return -(uintptr_t)ptr;
    }
```

所以，对象引用计数`map RefcountMap`的`key`是：`-(object *)`，就是对象地址取负。`value`就是该对象的引用计数。

我们来看一下OC是如何获取对象引用计数的：

```
inline uintptr_t
objc_object::rootRetainCount()
{
    //case 1： 如果是tagged pointer，则直接返回this，因为tagged pointer是不需要引用计数的
    if (isTaggedPointer()) return (uintptr_t)this;

    // 将objcet对应的sidetable上锁
    sidetable_lock();
    isa_t bits = LoadExclusive(&isa.bits);
    ClearExclusive(&isa.bits);
    // case 2： 如果采用了优化的isa指针
    if (bits.nonpointer) {
        uintptr_t rc = 1 + bits.extra_rc; // 先读取isa.extra_rc
        if (bits.has_sidetable_rc) { // 如果extra_rc不够大， 还需要读取sidetable中的数据
            rc += sidetable_getExtraRC_nolock(); // 总引用计数= rc + sidetable count
        }
        sidetable_unlock();
        return rc;
    }
    // case 3：如果没采用优化的isa指针，则直接返回sidetable中的值
    sidetable_unlock(); // 将objcet对应的sidetable解锁，因为sidetable_retainCount()中会上锁
    return sidetable_retainCount();
}
```

可以看到，runtime在获取对象引用计数的时候，是考虑了三种情况:

- (1) tagged pointer;
- (2) 优化的isa;
- (3) 未优化的isa。

我们来看一下`(2)优化的isa`的情况下： 首先，会读取`extra_rc中`的数据，因为`extra_rc`中存储的是引用计数减一，所以这里要加回去。 如果`extra_rc`不够大的话，还需要读取`sidetable`，调用`sidetable_getExtraRC_nolock`：

```
#define SIDE_TABLE_RC_SHIFT 2

size_t
objc_object::sidetable_getExtraRC_nolock()
{
    assert(isa.nonpointer);
    SideTable& table = SideTables()[this];
    RefcountMap::iterator it = table.refcnts.find(this);
    if (it == table.refcnts.end()) return 0;
    else return it->second >> SIDE_TABLE_RC_SHIFT;
}
```

注意，这里在返回引用计数前，还做了个`右移2位`的位操作`it->second >> SIDE_TABLE_RC_SHIFT`。这是因为在`sidetable`中，引用计数的低2位不是用来记录引用次数的，而是分别表示对象是否有弱引用计数，以及是否在`deallocing`，这估计是为了兼容未优化的`isa`而设计的：

```
#define SIDE_TABLE_WEAKLY_REFERENCED (1UL<<0)
#define SIDE_TABLE_DEALLOCATING      (1UL<<1)  // MSB-ward of weak bit
```

所以，在`sidetable`中做加引用加一操作时，需要在第3位上+1：

```
#define SIDE_TABLE_RC_ONE            (1UL<<2)  // MSB-ward of deallocating bit
refcntStorage += SIDE_TABLE_RC_ONE;
```

这里`sidetable`的引用计数值还有一个`SIDE_TABLE_RC_PINNED`状态，表明这个引用计数太大了，连`sidetable`都表示不出来：

```
#define SIDE_TABLE_RC_PINNED         (1UL<<(WORD_BITS-1))
```

OK，到此为止，我们就学习完了runtime中所有的引用计数实现方式。接下来我们还会继续看和引用计数相关的两个概念：弱引用和autorelease。

Weekly reference

再来回看一下`sidetable`的定义如下：

```
struct SideTable {
    spinlock_t slock;           // 自旋锁，防止多线程访问冲突
    RefcountMap refcnts;        // 对象引用计数map
    weak_table_t weak_table;    // 对象弱引用map
}
```

`spinlock_t slock`、`RefcountMap refcnts`的定义我们已经清楚，下面就来看一下`weak_table_t weak_table`，它记录了所有弱引用对象的集合。

`weak_table_t`定义如下：

```
/**
 * The global weak references table. Stores object ids as keys,
 * and weak_entry_t structs as their values.
 */
struct weak_table_t {
    weak_entry_t *weak_entries;        // hash数组，用来存储弱引用对象的相关信息weak_entry_t
    size_t    num_entries;             // hash数组中的元素个数
    uintptr_t mask;                    // hash数组长度-1，会参与hash计算。（注意，这里是hash数组的长度，而不是元素个数。比如，数组长度可能是64，而元素个数仅存了2个）
    uintptr_t max_hash_displacement;   // 可能会发生的hash冲突的最大次数
};
```

`weak_table_t`包含一个`weak_entry_t`类型的数组，可以通过`hash算法`找到对应`object`在数组中的`index`。这种结构，和`sidetables`类似，不同的是，`weak_table_t`是可以动态扩展的，而不是写死的64个。

`weak_entries`实质上是一个hash数组，数组中存储`weak_entry_t`类型的元素。`weak_entry_t`的定义如下：

```
typedef DisguisedPtr<objc_object *> weak_referrer_t;

#define PTR_MINUS_2 62
/**
 * The internal structure stored in the weak references table.
 * It maintains and stores
 * a hash set of weak references pointing to an object.
 * If out_of_line_ness != REFERRERS_OUT_OF_LINE then the set
 * is instead a small inline array.
 */
#define WEAK_INLINE_COUNT 4

// out_of_line_ness field overlaps with the low two bits of inline_referrers[1].
// inline_referrers[1] is a DisguisedPtr of a pointer-aligned address.
// The low two bits of a pointer-aligned DisguisedPtr will always be 0b00
// (disguised nil or 0x80..00) or 0b11 (any other address).
// Therefore out_of_line_ness == 0b10 is used to mark the out-of-line state.
#define REFERRERS_OUT_OF_LINE 2

struct weak_entry_t {
    DisguisedPtr<objc_object> referent; // 被弱引用的对象

    // 引用该对象的对象列表，联合。 引用个数小于4，用inline_referrers数组。 用个数大于4，用动态数组weak_referrer_t *referrers
```

```
    union {
        struct {
            weak_referrer_t *referrers;                       // 弱引用该对象的对象列表的动态数组
            uintptr_t        out_of_line_ness : 2;            // 是否使用动态数组标记位
            uintptr_t        num_refs : PTR_MINUS_2;          // 动态数组中元素的个数
            uintptr_t        mask;                            // 用于hash确定动态数组index，值实际上是动态数组空间长度-1（它和num_refs不一样，这里是记录的是数组中位置的个数，而不是数组中实际存储的元素个数）。
            uintptr_t        max_hash_displacement;           // 最大的hash冲突次数（说明了最多做max_hash_displacement次hash冲突，肯定会找到对应的数据）
        };
        struct {
            // out_of_line_ness field is low bits of inline_referrers[1]
            weak_referrer_t  inline_referrers[WEAK_INLINE_COUNT];
        };
    };

    bool out_of_line() {
        return (out_of_line_ness == REFERRERS_OUT_OF_LINE);
    }

    weak_entry_t& operator=(const weak_entry_t& other) {
        memcpy(this, &other, sizeof(other));
        return *this;
    }

    weak_entry_t(objc_object *newReferent, objc_object **newReferrer)
        : referent(newReferent)
    {
        inline_referrers[0] = newReferrer;
        for (int i = 1; i < WEAK_INLINE_COUNT; i++) {
            inline_referrers[i] = nil;
        }
    }
};
```

根据注释，`DisguisedPtr`方法返回的`hash值`得最低2个字节应该是`0b00`或`0b11`，因此可以用`out_of_line_ness == 0b10`来表明当前是否在使用定长`数组或动态数组来保存引用该对象的列表。

这样`sidetable`中的`weak_table_t weak_table`成员的结构如下所示：

回头再来看一下，会发现在`weak talbe`中存在`两个hash 表`。

一个是`weak_table_t`自身。它可以通过对象地址做hash（`hash_pointer(objc_object *) & weak_table->mask`），直接找到`weak_entries`中该对象对应的`weak_entry_t`。

另一个是`weak_entry_t`中的`weak_referrer_t *referrers`。它可以通过弱引用该对象的对象指针的指针做hash（`w_hash_pointer(objc_object **) & (entry->mask)`），直接找到对象指针的指针在`referrers`中对应的`weak_referrer_t *`。

虽然`weak_table_t`和`referrers`是表示意义不同的hash表，但他们的实现是一样的，可以看做是同一种hash表。而且还设计的很有技巧。下面，我们就来详细学习一下hash 表示怎么实现的。

weak table的实现细节

由于`weak_entries`和`referrers`中的实现类似，这里我们就以`weak_table_t`为例，来分析hash表的实现。

`weak_table_t`定义如下：

```
/**
 * The global weak references table. Stores object ids as keys,
 * and weak_entry_t structs as their values.
 */
struct weak_table_t {
    weak_entry_t *weak_entries;         // hash数组，用来存储弱引用对象的相关信息 weak_entry_t
    size_t    num_entries;              // hash数组中的元素个数
    uintptr_t mask;                     // hash数组长度-1，用于和hash值做位与计算，来确定数组下标。（注意，这里是hash数组的长度，而不是元素个数。比如，数组长度可能是64，而元素个数仅存了2个）
    uintptr_t max_hash_displacement;    // 可能会发生的hash冲突的最大次数
};
```

hash定位

当向`weak_table_t`中插入或查找某个元素时，是通过如下hash算法的(以查找为例)：

```
static weak_entry_t *
weak_entry_for_referent(weak_table_t *weak_table, objc_object *referent)
{
    assert(referent);

    weak_entry_t *weak_entries = weak_table->weak_entries;

    if (!weak_entries) return nil;

    size_t begin = hash_pointer(referent) & weak_table->mask;
    size_t index = begin;
    size_t hash_displacement = 0;
```

```
    while (weak_table->weak_entries[index].referent != referent) {
        index = (index+1) & weak_table->mask;
        if (index == begin) bad_weak_table(weak_table->weak_entries); // 触发
bad weak table crash
        hash_displacement++;
        if (hash_displacement > weak_table->max_hash_displacement) {
            return nil;
        }
    }

    return &weak_table->weak_entries[index];
}
```

首先，确定hash值可能对应的数组下标`begin`：

```
size_t begin = hash_pointer(referent) & weak_table->mask;
```

`hash_pointer(referent)`将会对`referent`进行`hash操作`：

```
static inline uint32_t ptr_hash(uint64_t key)
{
    key ^= key >> 4;
    key *= 0x8a970be7488fda55;
    key ^= __builtin_bswap64(key);
    return (uint32_t)key;
}
```

这个算法不用深究，知道就是一个hash操作就好了。

有技巧的是后半部分`& weak_table->mask`，将hash值和mask做位与运算。 之前说过，`mask`的值等于`数组长度-1`。而在下面的小节你会了解到，`hash`数组的长度会以`64`，`128`，`256`规律递增。总之，数组长度表现为二进制会是1000...0这种形式，即首位1，后面跟n个0。而这个值减1的话，则会变为011...1这种形式，即首位0，后面跟n个1，这即mask的二进制形式。那么用`mask & hash_pointer(referent)`时，就会保留`hash_pointer(referent)`的后n位的值，而首位被位与操作置为了0。那么这个值肯定是小于首位是1的数值的，也就是肯定会小于数组的长度。

> 因此，`begin`是一个小于数组长度的一个数组下标，且这个下标对应着目标元素的hash值。

确定了初始的数组下标后，就开始尝试确定元素的真正位置：

```
 while (weak_table->weak_entries[index].referent != referent) {
        index = (index+1) & weak_table->mask; // hash冲突，做index+1，尝试下一个相邻位置，& weak_table->mask 确保了index不会越界，而且会使index自动find数组一圈
        if (index == begin) bad_weak_table(weak_table->weak_entries); // 在数组中转了一圈还没找到目标元素，触发bad weak table crash
        hash_displacement++;
        if (hash_displacement > weak_table->max_hash_displacement) { // 如果hash冲突大于了最大可能的冲突次数，则说明目标对象不存在于数组中，返回nil
            return nil;
        }
  }
```

这里，产生了hash冲突后，系统会依次线性循环寻找目标对象的位置。直到找了一圈又回到了起点或大于了可能的hash冲突值。这个`max_hash_displacement`值是在每个元素插入的时候更新的，它总是记录在插入时，所发生的hash冲突的最大值。因此在查找时，hash冲突的次数肯定不会大于这个值。

这里最巧妙的是这条语句：

```
index = (index+1) & weak_table->mask
```

它即会让你向下一个相邻位置寻找，同时当寻找到最后一个位置时，它又会自动让你从数组的第一个位置开始寻找。这一切，都归功于二进制运算的巧妙运用。

hash表自动扩容

这里的`weak table`的大小是不固定的。当插入新元素时，会调用`weak_grow_maybe`方法，来判断是否要做hash表的扩容。该方法实现如下：

```
#define TABLE_SIZE(entry) (entry->mask ? entry->mask + 1 : 0)

// Grow the given zone's table of weak references if it is full.
static void weak_grow_maybe(weak_table_t *weak_table)
{
    size_t old_size = TABLE_SIZE(weak_table);

    // Grow if at least 3/4 full.
    if (weak_table->num_entries >= old_size * 3 / 4) { // 当大于现有长度的3/4时，会做数组扩容操作。
        weak_resize(weak_table, old_size ? old_size*2 : 64);  // 初次会分配64个位置，之后在原有基础上*2
    }
}
```

这里的扩容会调用`weak_resize`方法。每次扩容都会是原有长度的一倍。这样，每次扩容的新增空间都会比上一次要大一倍，而不是固定的扩容n个空间。这么做的目的在于，系统认为，当你有扩容需求时，之后又扩容需求的概率就会变大，为了防止频繁的申请内存，所以，每次扩容强度都会比上一次要大。

hash表自动收缩

当从`weak table`中删除元素时，系统会调用`weak_compact_maybe`判断是否需要收缩hash数组的空间：

```
// Shrink the table if it is mostly empty.
static void weak_compact_maybe(weak_table_t *weak_table)
{
    size_t old_size = TABLE_SIZE(weak_table);

    // Shrink if larger than 1024 buckets and at most 1/16 full.
    if (old_size >= 1024  && old_size / 16 >= weak_table->num_entries) { // 当前数组长度大于1024，且实际使用空间最多只有1/16时，需要做收缩操作
        weak_resize(weak_table, old_size / 8); // 缩小8倍
        // leaves new table no more than 1/2 full
    }
}
```

hash表resize

无论是扩容还是收缩，最终都会调用到`weak_resize`方法：

```
static void weak_resize(weak_table_t *weak_table, size_t new_size)
{
    size_t old_size = TABLE_SIZE(weak_table);

    weak_entry_t *old_entries = weak_table->weak_entries;  // 先把老数据取出来
    weak_entry_t *new_entries = (weak_entry_t *)   // 在为新的size申请内存
        calloc(new_size, sizeof(weak_entry_t));

    // 重置weak_table的各成员
    weak_table->mask = new_size - 1;
    weak_table->weak_entries = new_entries;
    weak_table->max_hash_displacement = 0;
    weak_table->num_entries = 0;  // restored by weak_entry_insert below

    if (old_entries) {
        weak_entry_t *entry;
        weak_entry_t *end = old_entries + old_size;
        for (entry = old_entries; entry < end; entry++) {
            if (entry->referent) { // 依次将老的数据插入到新的内存空间
                weak_entry_insert(weak_table, entry);
            }
        }
        free(old_entries); // 释放老的内存空间
    }
```

```
}

/**
 * Add new_entry to the object's table of weak references.
 * Does not check whether the referent is already in the table.
 */
static void weak_entry_insert(weak_table_t *weak_table, weak_entry_t
*new_entry)
{
    weak_entry_t *weak_entries = weak_table->weak_entries;
    assert(weak_entries != nil);

    size_t begin = hash_pointer(new_entry->referent) & (weak_table->mask);
    size_t index = begin;
    size_t hash_displacement = 0;
    while (weak_entries[index].referent != nil) {
        index = (index+1) & weak_table->mask;
        if (index == begin) bad_weak_table(weak_entries);
        hash_displacement++;
    }

    weak_entries[index] = *new_entry;
    weak_table->num_entries++;

    if (hash_displacement > weak_table->max_hash_displacement) { // 这里记录最大
的hash冲突次数，当查找元素时，hash冲突肯定不会大于这个值
        weak_table->max_hash_displacement = hash_displacement;
    }
}
```

OK, 上面就是对runtime中weak引用的相关数据结构的分析。关于weak引用数据，是存在于hash表中的。 这关于hash算法映射到数组下标，以及hash表动态的扩容/收缩，还是很有意思的。

autoreleasepool

在iOS中，除了需要手动retain，release（现在已经交给了ARC自动生成）外，我们还可以将对象扔到自动释放池中，由自动释放池来自动管理这些对象。我们可以这样使用autoreleasepool：

```
int main(int argc, char * argv[]) {
    @autoreleasepool {
        NSString *a =  [NSString stringWithFormat:@"%d", 1];
    }
}
```

用`clang -rewrite-objc`重写后，得到：

```
int main(int argc, char * argv[]) {
    /* @autoreleasepool */ { __AtAutoreleasePool __autoreleasepool;
        NSString *a = ((NSString * _Nonnull (*)(id, SEL, NSString * _Nonnull,
...))(void *)objc_msgSend)((id)objc_getClass("NSString"),
sel_registerName("stringWithFormat:"), (NSString
*)&__NSConstantStringImpl__var_folders_8k_3pbszhls2czcmz0w335cvc0w0000gn_T_mai
n_1a8fc0_mi_1, 1);

    }
}
```

这时会发现，`@autoreleasepool`被改写为了`__AtAutoreleasePool __autoreleasepool`这样一个对象。`__AtAutoreleasePool`的定义为：

```
struct __AtAutoreleasePool {
  __AtAutoreleasePool() {atautoreleasepoolobj = objc_autoreleasePoolPush();}
  ~__AtAutoreleasePool() {objc_autoreleasePoolPop(atautoreleasepoolobj);}
  void * atautoreleasepoolobj;
};
```

于是，关于`@autoreleasepool`的代码可以被改写为：

```
objc_autoreleasePoolPush();
// Do your code
objc_autoreleasePoolPop(atautoreleasepoolobj);
```

置于`@autoreleasepool`的`{}`中的代码实际上是被一个`push`和`pop`操作所包裹。当`push`时，会压栈一个`autoreleasepage`，在`{}`中的所有的`autorelease`对象都会放到这个`page`中。当`pop`时，会出栈一个`autoreleasepage`，同时，所有存储于这个`page`的对象都会做`release`操作。这就是`autoreleasepool`的实现原理。

`objc_autoreleasePoolPush()`和`objc_autoreleasePoolPop(atautoreleasepoolobj)`的实现如下：

```
void *
objc_autoreleasePoolPush(void)
{
    return AutoreleasePoolPage::push();
}

void
objc_autoreleasePoolPop(void *ctxt)
{
    AutoreleasePoolPage::pop(ctxt);
}
```

它们都分别调用了`AutoreleasePoolPage`类的静态方法`push`和`pop`。`AutoreleasePoolPage`是`runtime`中`autoreleasepool`的核心实现，下面，我们就来了解一下它。

AutoreleasePoolPage

`AutoreleasePoolPage`在runtime中的定义如下：

```
class AutoreleasePoolPage
{
  magic_t const magic;                 // 魔数，用于自身的完整性校验
                                          16字节
    id *next;                            // 指向autorelePool page中的下一个可用位置
                                          8字节
    pthread_t const thread;              // 和autorelePool page中相关的线程
                                           8字节
    AutoreleasePoolPage * const parent; // autoreleasPool page双向链表的前向指针
                                           8字节
    AutoreleasePoolPage *child;          // autoreleasPool page双向链表的后向指针
                                           8字节
    uint32_t const depth;                // 当前autoreleasPool page在双向链表中的
位置（深度）                                 4字节
    uint32_t hiwat;                      // high water mark. 最高水位，可用近似理解
为autoreleasPool page双向链表中的元素个数       4字节

    // SIZE-sizeof(*this) bytes of contents follow
}
```

每个`AutoreleasePoolPage`的大小为一个`SIZE`，即内存管理中一个页的大小。这在`Mac`中是`4KB`，而在iOS中，这里没有相关代码，估计差不多。

对象指针栈

由源码可用看出，在`AutoreleasePoolPage`类中共有7个成员属性，大小为56Bytes，按照一个Page是4KB计算，显然还有4040Bytes没有用。而这4040Bytes空间，就用来存储`AutoreleasePoolPage`所管理的对象指针。因此，一个`AutoreleasePoolPage`的内存布局如下图：

在autoreleasepool中的对象指针是按照栈的形式存储的，栈低是一个`POOL_BOUNDARY`哨兵，之后对象指针依次入栈存储。

POOL_BOUNDARY

在图中可用看到，除了`AutoreleasePoolPage`类中的7个成员之外，还有一个叫`POOL_BOUNDARY`，其实这是一个`nil`指针，`AutoreleasePoolPage`中的`next`指针用来指向栈中下一个入栈位置。

```
#   define POOL_BOUNDARY nil
```

它作为一个哨兵，当需要将`AutoreleasePoolPage`中存储的对象指针依次出栈时，会执行到`POOL_BOUNDARY`为止。

双向链表

在图中也可以看出，单个的`AutoreleasePoolPage`是以栈的形式存储的。 当加入到`autoreleasepool`中的元素太多时，一个`AutoreleasePoolPage`就不够用的了。这时候我们需要新创建一个`AutoreleasePoolPage`，多个`AutoreleasePoolPage`之间通过`双向链表`的形式串起来。

成员`parent`和`child`就是用来链接双向链表上下的指针。

下面我们就结合具体的代码，来看一下`AutoreleasePoolPage`是如何在系统中发挥作用的。

Push

当用户调用`@autoreleasepool{}`的时候，系统首先会调用`AutoreleasePoolPage`的`push()`方法，来创建或获取当前的`hotPage`，并向对象栈中插入一个`POOL_BOUNDARY`。

```
    static inline void *push()
     {
         id *dest;
         dest = autoreleaseFast(POOL_BOUNDARY);
         assert(dest == EMPTY_POOL_PLACEHOLDER || *dest == POOL_BOUNDARY);
         return dest;
     }
```

我们也可以调用`autorelease(id obj)`方法将某个特定的对象指针插入到`AutoreleasePoolPage`中：

```
static inline id autorelease(id obj)
    {
        assert(obj);
        assert(!obj->isTaggedPointer());  // 注意这个assert，由于tagged pointer不遵循引用计数规则，所以也不会有autorelease操作。
        id *dest __unused = autoreleaseFast(obj);
        assert(!dest  ||  dest == EMPTY_POOL_PLACEHOLDER  ||  *dest == obj);
        return obj;
    }
```

可以看到，无论是`push`还是`autorelease`方法，最后都是调用了`autoreleaseFast(obj)`，该方法会将一个`id`放入到`autoreleasePage`中。：

```
    static inline id *autoreleaseFast(id obj)
    {
        AutoreleasePoolPage *page = hotPage();
        if (page && !page->full()) {
            return page->add(obj);
        } else if (page) {
            return autoreleaseFullPage(obj, page);
        } else {
            return autoreleaseNoPage(obj);
        }
    }
```

可以看到方法实现逻辑也很简单：

1. 首先取出当前的`hotPage`，所谓`hotPage`，就是在`autoreleasePage`链表中正在使用的`autoreleasePage`节点。 如果有`hotPage`，且`hotPage`还没满，这将`obj`加入到`page`中。
2. 如果有`hotPage`，但是已经满了，则进入`page full`逻辑（`autoreleaseFullPage`）。
3. 如果没有`hotPage`，进入`no page`逻辑(`autoreleaseNoPage`)。

hotPage

`hotPage`是`autoreleasePage`链表中正在使用的`autoreleasePage`节点。实质上是指向`autoreleasepage`的指针，并存储于线程的TSD(线程私有数据：`Thread-specific Data`)中：

```
    static inline AutoreleasePoolPage *hotPage()
    {
        AutoreleasePoolPage *result = (AutoreleasePoolPage *)
            tls_get_direct(key);
        if ((id *)result == EMPTY_POOL_PLACEHOLDER) return nil;
        if (result) result->fastcheck();
        return result;
    }
```

从这段代码可以看出，**`autoreleasepool`是和线程绑定的，一个线程对应一个`autoreleasepool`。而`autoreleasepool`虽然叫做自动释放池，其实质上是一个双向链表。**

在介绍`runloop`的时候，我们也曾提到过，`runloop`和`线程`也是`一一对应`的，并且在当前线程的`runloop`指针，也会存储到线程的`TSD`中。这是runtime对于TSD的一个应用。

add object

如果有`hot page`，先判断`page`是否已经`full`了，判断逻辑是`next*`是否等于`end()`：

```
bool full() {
        return next == end();
    }
```

关于`begin()`和`end()`，定义如下，结合`page`的图示，应该比较容易理解：

```
   id * begin() {
        return (id *) ((uint8_t *)this+sizeof(*this));
    }

    id * end() {
        return (id *) ((uint8_t *)this+SIZE);
    }
```

如果`page`没有满，这调用`page`的`add`方法：

```
id *add(id obj)
    {
        assert(!full());

        id *ret = next;  // faster than `return next-1` because of aliasing
        *next = obj;
        next++;
        return ret;
    }
```

逻辑比较简单，就是将`obj`置于`next`的位置，`next++`，然后返回`obj`的位置。

autoreleaseFullPage

如果`hot page`满了，就需要在链表中'加页'，同时将新页置为hot page：

```
  static __attribute__((noinline))
    id *autoreleaseFullPage(id obj, AutoreleasePoolPage *page)
    {
        // The hot page is full.
        // Step to the next non-full page, adding a new page if necessary.
        // Then add the object to that page.
        assert(page == hotPage());
        assert(page->full()  ||  DebugPoolAllocation);

        do {
            if (page->child) page = page->child;
            else page = new AutoreleasePoolPage(page);
        } while (page->full());
```

```
        setHotPage(page);
        return page->add(obj);
    }
```

这一段代码重点需要关注的是寻找可用`page`的`do while`逻辑。 其实注释中已经写得很清楚，系统会首先尝试在`hot page`的`child pages`中挑出第一个没有满的`page`，如果没有符合要求的`child page`，则只能创建一个新的`new AutoreleasePoolPage(page)`。

最后，将挑选出的`page`作为当前线程的`hot page`（实际上存储到了TSD中），并将`obj`存到新的`hot page`中。

autoreleaseNoPage

若当前线程没有`hot Page`，则说明当前的线程还未建立起`autorelease pool`。这时，就会调用`autoreleaseNoPage`：

```
  static __attribute__((noinline))
    id *autoreleaseNoPage(id obj)
    {
        // "No page" could mean no pool has been pushed
        // or an empty placeholder pool has been pushed and has no contents
yet
        assert(!hotPage());

        bool pushExtraBoundary = false;
        if (haveEmptyPoolPlaceholder()) { // 如果当前线程只有一个虚拟的空池，则这次需要真正创建一个page
            // We are pushing a second pool over the empty placeholder pool
            // or pushing the first object into the empty placeholder pool.
            // Before doing that, push a pool boundary on behalf of the pool
            // that is currently represented by the empty placeholder.
            pushExtraBoundary = true;
        }else if (obj == POOL_BOUNDARY  &&  !DebugPoolAllocation) { // 如果obj == POOL_BOUNDARY，这里苹果有个小心机，它不会真正创建page，而是在线程的TSD中做了一个空池的标志
            // We are pushing a pool with no pool in place,
            // and alloc-per-pool debugging was not requested.
            // Install and return the empty pool placeholder.
            return setEmptyPoolPlaceholder();
        }

        // We are pushing an object or a non-placeholder'd pool.

        // 创建线程的第一个page，并置为hot page。
        AutoreleasePoolPage *page = new AutoreleasePoolPage(nil);
        setHotPage(page);
```

```
        // 如果之前只是做了空池标记，这里还需要在栈中补上POOL_BOUNDARY，作为栈底哨兵
        if (pushExtraBoundary) {
            page->add(POOL_BOUNDARY);
        }

        // Push the requested object or pool.  注意，这里的注释，进入page的不光可以有object，还可以是pool。
        return page->add(obj);
    }
```

当系统发现当前线程没有对应的`autoreleasepool`时，我们自然的想到需要为线程创建一个`page`。但是苹果其实在这里是耍了一个小心机的，当在创建第一个`page`时，苹果并不会真正创建一个`page`，因为它害怕创建了`page`后，并没有真正的`object`需要插入`page`，这样就造成了无谓的内存浪费。

在没有第一个真正的`object`入栈之前，苹果是这样做的：仅仅在线程的`TSD`中做了一个`EMPTY_POOL_PLACEHOLDER`标记，并返回它。这里没有真正的`new`一个`AutoreleasePoolPage`。

Pop

当`autoreleasepool`需要被释放时，会调用`Pop`方法。而`Pop`方法需要接受一个`void *token`参数，来告诉池子，需要一直释放到`token`对应的那个`page`：

```
static inline void pop(void *token)
    {
        AutoreleasePoolPage *page;
        id *stop;

        if (token == (void*)EMPTY_POOL_PLACEHOLDER) {
            // Popping the top-level placeholder pool.
            if (hotPage()) {
                // Pool was used. Pop its contents normally.
                // Pool pages remain allocated for re-use as usual.
                pop(coldPage()->begin());
            } else {
                // Pool was never used. Clear the placeholder.
                setHotPage(nil);
            }
            return;
        }

        page = pageForPointer(token);
        stop = (id *)token;
        if (*stop != POOL_BOUNDARY) {
            if (stop == page->begin()  &&  !page->parent) {
                // Start of coldest page may correctly not be POOL_BOUNDARY:
                // 1. top-level pool is popped, leaving the cold page in place
                // 2. an object is autoreleased with no pool
```

```
            } else {
                // 这是为了兼容旧的SDK，看来在新的SDK里面，token 可能的取值只有两
个:POOL_BOUNDARY，page->begin() && !page->parent
                // Error. For bincompat purposes this is not
                // fatal in executables built with old SDKs.
                return badPop(token);
            }
        }
    // 对page中的object做objc_release操作，一直到stop
        page->releaseUntil(stop);

        // memory: delete empty children 删除多余的child，节约内存
         if (page->child) {
            // hysteresis: keep one empty child if page is more than half full
            if (page->lessThanHalfFull()) {
                page->child->kill();
            }
            else if (page->child->child) {
                page->child->child->kill();
            }
        }
    }
```

何时需要`autoreleasePool`

OK，以上就是autoreleasepool的内容。那么在ARC的环境下，我们何时需要用@autoreleasepool呢?

一般的，有下面两种情况:

创建子线程。当我们创建子线程的时候，需要将子线程的`runloop`用`@autoreleasepool`包裹起来，进而达到自动释放内存的效果。因为系统并不会为子线程自动包裹一个`@autoreleasepool`，这样加入到`autoreleasepage`中的元素就得不到释放。 在大循环中创建`autorelease`对象。当我们在一个循环中创建`autorelease`对象(不是用alloc创建的对象)，该对象会加入到`autoreleasepage`中，而这个page中的对象，会等到外部池子结束才会释放。在主线程的`runloop`中，会将所有的对象的释放权都交给了`RunLoop`的释放池，而`RunLoop`的释放池会等待这个事件处理之后才会释放，因此就会使对象无法及时释放，堆积在内存造成内存泄露。

前言

提起弱引用，大家都知道它的作用:

- 1. 不会添加引用计数
- 2. 当所引用的对象释放后，引用者的指针自动置为nil

那么，围绕它背后的实现，是怎么样的呢？在许多公司面试时，都会问到这个问题。那么，今天就带大家一起分析一下weak引用是怎么实现的，希望能够搞清楚每一个细节。

Store as weak

当我们要weak引用一个对象，我们可以这么做：

```
int main(int argc, char * argv[]) {
    @autoreleasepool {
        NSObject *obj = [[NSObject alloc] init];
        __weak NSObject *weakObj = obj;
    }
}
```

创建了一个`NSObject`对象`obj`，然后用`weakObj`对`obj`做弱引用。 当我们对一个对象做`weak`引用的时候，其背后是通过`runtime`来支持的。当把一个对象做`weak`引用时，会调用`runtime`方法`objc_initWeak`：

objc_initWeak

```
id objc_initWeak(id *location, id newObj)
{
    if (!newObj) {
        *location = nil;
        return nil;
    }

    return storeWeak<DontHaveOld, DoHaveNew, DoCrashIfDeallocating>
        (location, (objc_object*)newObj);
}
```

该方法接受两个参数：

- `id *location`： `__weak`指针的地址，即例子中的weak指针取地址： `&weakObj`。它是一个指针的地址。之所以要存储指针的地址，是因为最后我们要讲`__weak`指针指向的内容置为`nil`，如果仅存储指针的话，是不能够完成这个功能的。
- `id newObj`：所引用的对象。即例子中的`obj`。

有一个返回值`id`： 会返回`obj`自身，但其值已经做了更改（isa_t中的weak_ref位置1）

`objc_initWeak`实质是调用了`storeWeak`方法。看这个方法的名字，就可以猜到是将`weak`引用存到某个地方，没错，实际上苹果就是这么做的。

storeWeak

`storeWeak`方法有点长，这也是`weak`引用的核心实现部分。其实核心也就实现了两个功能：

- 将`weak`指针的地址`location`存入到`obj`对应的`weak_entry_t`的哈希数组中，用于在`obj`析构

时，通过该数组找到所有其`weak`指针引用，并将指针指向的地址（`location`）置为nil。关于`weak_entry_t`，在上一篇中已有介绍。

- 如果启用了`isa`优化，则将`obj`的`isa_t`的`weakly_referenced`位置为1。置为1的作用主要是为了标记`obj`被`weak`引用了，当`dealloc`时，runtime会根据`weakly_referenced`标志位来判断是否需要查找`obj`对应的`weak_entry_t`，并将引用置为`nil`。

```
// Template parameters.
enum HaveOld { DontHaveOld = false, DoHaveOld = true };
enum HaveNew { DontHaveNew = false, DoHaveNew = true };
enum CrashIfDeallocating {
    DontCrashIfDeallocating = false, DoCrashIfDeallocating = true
};

template <HaveOld haveOld, HaveNew haveNew,
          CrashIfDeallocating crashIfDeallocating>
static id storeWeak(id *location, objc_object *newObj)
{
    assert(haveOld  ||  haveNew);
    if (!haveNew) assert(newObj == nil);

    Class previouslyInitializedClass = nil;
    id oldObj;
    SideTable *oldTable;
    SideTable *newTable;

    // Acquire locks for old and new values.
    // Order by lock address to prevent lock ordering problems.
    // Retry if the old value changes underneath us.
 retry:
    if (haveOld) { // 如果weak ptr之前弱引用过一个obj，则将这个obj所对应的SideTable取出，赋值给oldTable
        oldObj = *location;
        oldTable = &SideTables()[oldObj];
    } else {
        oldTable = nil; // 如果weak ptr之前没有弱引用过一个obj，则oldTable = nil
    }
    if (haveNew) { // 如果weak ptr要weak引用一个新的obj，则将该obj对应的SideTable取出，赋值给newTable
        newTable = &SideTables()[newObj];
    } else {
        newTable = nil; // 如果weak ptr不需要引用一个新obj，则newTable = nil
    }

    // 加锁操作，防止多线程中竞争冲突
    SideTable::lockTwo<haveOld, haveNew>(oldTable, newTable);

    // location 应该与 oldObj 保持一致，如果不同，说明当前的 location 已经处理过 oldObj 可是又被其他线程所修改
```

```
    if (haveOld  &&  *location != oldObj) {
        SideTable::unlockTwo<haveOld, haveNew>(oldTable, newTable);
        goto retry;
    }

    // Prevent a deadlock between the weak reference machinery
    // and the +initialize machinery by ensuring that no
    // weakly-referenced object has an un-+initialized isa.
    if (haveNew  &&  newObj) {
        Class cls = newObj->getIsa();
        if (cls != previouslyInitializedClass  &&
            !((objc_class *)cls)->isInitialized())  // 如果cls还没有初始化，先初始化，再尝试设置weak
        {
            SideTable::unlockTwo<haveOld, haveNew>(oldTable, newTable);
            _class_initialize(_class_getNonMetaClass(cls, (id)newObj));

            // If this class is finished with +initialize then we're good.
            // If this class is still running +initialize on this thread
            // (i.e. +initialize called storeWeak on an instance of itself)
            // then we may proceed but it will appear initializing and
            // not yet initialized to the check above.
            // Instead set previouslyInitializedClass to recognize it on retry.
            previouslyInitializedClass = cls; // 这里记录一下previouslyInitializedClass， 防止改if分支再次进入

            goto retry; // 重新获取一遍newObj，这时的newObj应该已经初始化过了
        }
    }

    // Clean up old value, if any.
    if (haveOld) {
        weak_unregister_no_lock(&oldTable->weak_table, oldObj, location); // 如果weak_ptr之前弱引用过别的对象oldObj，则调用weak_unregister_no_lock，在oldObj的weak_entry_t中移除该weak_ptr地址
    }

    // Assign new value, if any.
    if (haveNew) { // 如果weak_ptr需要弱引用新的对象newObj
        // (1) 调用weak_register_no_lock方法，将weak ptr的地址记录到newObj对应的weak_entry_t中
        newObj = (objc_object *)
            weak_register_no_lock(&newTable->weak_table, (id)newObj, location,
                                  crashIfDeallocating);
        // weak_register_no_lock returns nil if weak store should be rejected

        // (2) 更新newObj的isa的weakly_referenced bit标志位
        // Set is-weakly-referenced bit in refcount table.
```

```
        if (newObj  &&  !newObj->isTaggedPointer()) {
            newObj->setWeaklyReferenced_nolock();
        }

        // Do not set *location anywhere else. That would introduce a race.
        //  (3) *location 赋值，也就是将weak ptr直接指向了newObj。可以看到，这里并没有
将newObj的引用计数+1
        *location = (id)newObj; // 将weak ptr指向object
    }
    else {
        // No new value. The storage is not changed.
    }

    // 解锁，其他线程可以访问oldTable, newTable了
    SideTable::unlockTwo<haveOld, haveNew>(oldTable, newTable);

    return (id)newObj; // 返回newObj，此时的newObj与刚传入时相比，weakly-referenced
bit位置1
}
```

下面我们就一起来分析下`storeWeak`方法。

`storeWeak`方法实质上接受5个参数，其中`HaveOld haveOld`，`HaveNew haveNew`，`CrashIfDeallocating crashIfDeallocating`这三个参数是以`模板枚举`的方式传入的，其实这是三个`bool`参数，分别表示：`weak ptr`之前是否已经指向了一个弱引用，`weak ptr`是否需要指向一个新引用， 如果被弱引用的对象正在析构，此时再弱引用该对象，是否应该`crash`。

具体到`objc_initWeak`，这三个参数的值分别为`false`，`true`，`true`。

这时，传入storeWeak的参数中，haveOld被设置为false，表明weakObj之前并没有weak指向其他的对象。

那么，什么时候storeWeak的参数haveOld被设置为true呢？当我们的weakObj已经指向一个weak对象，又要指向新的weak对象时，storeWeak的haveOld参数会被置为true：

```
int main(int argc, char * argv[]) {
    @autoreleasepool {
        NSObject *obj = [[NSObject alloc] init];
        __weak NSObject *weakObj = obj; // 这里会调用objc_initWeak方法，storeWeak
的haveOld == false
        NSObject *obj2 = [[NSObject alloc] init];
        weakObj = obj2;  // 这里会调用objc_storeWeak方法，storeWeak的haveOld ==
true，会将之前的引用先移除
    }
}
```

objc_storeWeak方法的实现如下：

```
/**
```

```
 * This function stores a new value into a __weak variable. It would
 * be used anywhere a __weak variable is the target of an assignment.
 *
 * @param location The address of the weak pointer itself
 * @param newObj The new object this weak ptr should now point to
 *
 * @return \e newObj
 */
id
objc_storeWeak(id *location, id newObj)
{
    return storeWeak<DoHaveOld, DoHaveNew, DoCrashIfDeallocating>
        (location, (objc_object *)newObj);
}
```

`storeWeak` 另外两个参数是`id *location`, `objc_object *newObj`，这两个参数和objc_initWeak是一样的，分别代表weak 指针的地址，以及被weak引用的对象。

接下来函数体里的内容，大家可以结合注释，应该能够看个明白。

这里涉及到两个关键的函数：

```
weak_unregister_no_lock  // 将 weak ptr地址 从obj的weak_entry_t中移除
weak_register_no_lock   // 将 weak ptr地址 注册到obj对应的weak_entry_t中
```

这里我们先看注册函数：

weak_register_no_lock

```
id weak_register_no_lock(weak_table_t *weak_table, id referent_id,
                         id *referrer_id, bool crashIfDeallocating)
{
    objc_object *referent = (objc_object *)referent_id;
    objc_object **referrer = (objc_object **)referrer_id;

    // 如果referent为nil 或 referent 采用了TaggedPointer计数方式，直接返回，不做任何操
作
    if (!referent  ||  referent->isTaggedPointer()) return referent_id;

    // 确保被引用的对象可用（没有在析构，同时应该支持weak引用）
    bool deallocating;
    if (!referent->ISA()->hasCustomRR()) {
        deallocating = referent->rootIsDeallocating();
    }
    else {
        BOOL (*allowsWeakReference)(objc_object *, SEL) =
            (BOOL(*)(objc_object *, SEL))
            object_getMethodImplementation((id)referent,
```

```
                                            SEL_allowsWeakReference);
        if ((IMP)allowsWeakReference == _objc_msgForward) {
            return nil;
        }
        deallocating =
            ! (*allowsWeakReference)(referent, SEL_allowsWeakReference);
    }
    // 正在析构的对象，不能够被弱引用
    if (deallocating) {
        if (crashIfDeallocating) {
            _objc_fatal("Cannot form weak reference to instance (%p) of "
                        "class %s. It is possible that this object was "
                        "over-released, or is in the process of
deallocation.",
                        (void*)referent, object_getClassName((id)referent));
        } else {
            return nil;
        }
    }

    // now remember it and where it is being stored
    // 在 weak_table中找到referent对应的weak_entry,并将referrer加入到weak_entry中
    weak_entry_t *entry;
    if ((entry = weak_entry_for_referent(weak_table, referent))) { // 如果能找到
weak_entry,则讲referrer插入到weak_entry中
        append_referrer(entry, referrer);   // 将referrer插入到weak_entry_t的引
用数组中
    }
    else { // 如果找不到，就新建一个
        weak_entry_t new_entry(referent, referrer);
        weak_grow_maybe(weak_table);
        weak_entry_insert(weak_table, &new_entry);
    }

    // Do not set *referrer. objc_storeWeak() requires that the
    // value not change.

    return referent_id;
}
```

注意看开头的地方：

```
    // 如果referent为nil 或 referent 采用了TaggedPointer计数方式，直接返回，不做任何操
作
    if (!referent  ||  referent->isTaggedPointer()) return referent_id;
```

这里再次出现了`taggedPointer`的身影，若引用计数使用了`taggedPointer`，则不会做任何引用计数。

接着，会判断`referent_id`是否能够被weak 引用。这里主要判断`referent_id`是否正在被析构以及`referent_id`是否支持weak引用。如果`referent_id`不能够被weak引用，则直接返回`nil`。

接下来，如果`referent_id`能够被weak引用，则将`referent_id`对应的`weak_entry_t`从`weak_table`的`weak_entry_t`哈希数组中找出来

```
if ((entry = weak_entry_for_referent(weak_table, referent))) { // 如果能找到
weak_entry,则讲referrer插入到weak_entry中
    append_referrer(entry, referrer);     // 将referrer插入到weak_entry_t的引用数
组中
}
```

如果`entry`不存在，则会新建一个`referent_id`所对应的`weak_entry_t`：

```
else { // 如果找不到，就新建一个
    weak_entry_t new_entry(referent, referrer);  // 创建一个新的weak_entry_t ，并
将referrer插入到weak_entry_t的引用数组中
    weak_grow_maybe(weak_table);  // weak_table的weak_entry_t 数组是否需要动态增
长，若需要，则会扩容一倍
    weak_entry_insert(weak_table, &new_entry); // 将weak_entry_t插入到
weak_table中
}
```

将`referrer`插入到对应的`weak_entry_t`的引用数组后，我们的weak工作基本也就结束了。 最后，只需要返回被引用的对象即可：

```
return referent_id;
```

关于`referrer`是如何插入到`weak_entry_t`中的，其hash算法是怎么样的，则是利用函数`append_referrer`：

```
static void append_referrer(weak_entry_t *entry, objc_object **new_referrer)
{
    if (! entry->out_of_line()) { // 如果weak_entry 尚未使用动态数组，走这里
        // Try to insert inline.
        for (size_t i = 0; i < WEAK_INLINE_COUNT; i++) {
            if (entry->inline_referrers[i] == nil) {
                entry->inline_referrers[i] = new_referrer;
                return;
            }
        }

        // 如果inline_referrers的位置已经存满了，则要转型为referrers，做动态数组。
        // Couldn't insert inline. Allocate out of line.
        weak_referrer_t *new_referrers = (weak_referrer_t *)
            calloc(WEAK_INLINE_COUNT, sizeof(weak_referrer_t));
        // This constructed table is invalid, but grow_refs_and_insert
```

```
        // will fix it and rehash it.
        for (size_t i = 0; i < WEAK_INLINE_COUNT; i++) {
            new_referrers[i] = entry->inline_referrers[i];
        }
        entry->referrers = new_referrers;
        entry->num_refs = WEAK_INLINE_COUNT;
        entry->out_of_line_ness = REFERRERS_OUT_OF_LINE;
        entry->mask = WEAK_INLINE_COUNT-1;
        entry->max_hash_displacement = 0;
    }

    // 对于动态数组的附加处理:
    assert(entry->out_of_line()); // 断言:  此时一定使用的动态数组

    if (entry->num_refs >= TABLE_SIZE(entry) * 3/4) { // 如果动态数组中元素个数大于或等于数组位置总空间的3/4，则扩展数组空间为当前长度的一倍
        return grow_refs_and_insert(entry, new_referrer); // 扩容，并插入
    }

    // 如果不需要扩容，直接插入到weak_entry中
    // 注意，weak_entry是一个哈希表，key: w_hash_pointer(new_referrer) value: new_referrer

    // 细心的人可能注意到了，这里weak_entry_t 的hash算法和 weak_table_t的hash算法是一样的，同时扩容/减容的算法也是一样的
    size_t begin = w_hash_pointer(new_referrer) & (entry->mask); // '& (entry->mask)' 确保了 begin的位置只能大于或等于 数组的长度
    size_t index = begin;  // 初始的hash index
    size_t hash_displacement = 0;  // 用于记录hash冲突的次数，也就是hash再位移的次数
    while (entry->referrers[index] != nil) {
        hash_displacement++;
        index = (index+1) & entry->mask;  // index + 1，移到下一个位置，再试一次能否插入。(这里要考虑到entry->mask取值，一定是: 0x111, 0x1111, 0x11111, ... ，因为数组每次都是*2增长，即8， 16， 32，对应动态数组空间长度-1的mask，也就是前面的取值。)
        if (index == begin) bad_weak_table(entry); // index == begin 意味着数组绕了一圈都没有找到合适位置，这时候一定是出了什么问题。
    }
    if (hash_displacement > entry->max_hash_displacement) { // 记录最大的hash冲突次数，max_hash_displacement意味着: 我们尝试至多max_hash_displacement次，肯定能够找到object对应的hash位置
        entry->max_hash_displacement = hash_displacement;
    }
    // 将ref存入hash数组，同时，更新元素个数num_refs
    weak_referrer_t &ref = entry->referrers[index];
    ref = new_referrer;
    entry->num_refs++;
}
```

weak_unregister_no_lock

如果`weak ptr`在指向`obj`之前，已经weak引用了其他的对象，则需要先将`weak ptr`从其他对象的`weak_entry_t`的hash数组中移除。在`storeWeak`方法中，会调用`weak_unregister_no_lock`来做移除操作：

```
    if (haveOld) {
        weak_unregister_no_lock(&oldTable->weak_table, oldObj, location); // 如果weak_ptr之前弱引用过别的对象oldObj，则调用weak_unregister_no_lock，在oldObj的weak_entry_t中移除该weak_ptr地址
    }
```

`weak_unregister_no_lock`的实现如下：

```
void weak_unregister_no_lock(weak_table_t *weak_table, id referent_id,
                             id *referrer_id)
{
    objc_object *referent = (objc_object *)referent_id;
    objc_object **referrer = (objc_object **)referrer_id;

    weak_entry_t *entry;

    if (!referent) return;

    if ((entry = weak_entry_for_referent(weak_table, referent))) { // 查找到referent所对应的weak_entry_t
        remove_referrer(entry, referrer);  // 在referent所对应的weak_entry_t的hash数组中，移除referrer

        // 移除元素之后， 要检查一下weak_entry_t的hash数组是否已经空了
        bool empty = true;
        if (entry->out_of_line()  &&  entry->num_refs != 0) {
            empty = false;
        }
        else {
            for (size_t i = 0; i < WEAK_INLINE_COUNT; i++) {
                if (entry->inline_referrers[i]) {
                    empty = false;
                    break;
                }
            }
        }

        if (empty) { // 如果weak_entry_t的hash数组已经空了，则需要将weak_entry_t从weak_table中移除
            weak_entry_remove(weak_table, entry);
        }
```

```
    }
```

`weak_unregister_no_lock`的实现逻辑比较简单。

首先，它会在`weak_table`中找出`referent`对应的`weak_entry_t` 在`weak_entry_t`中移除`referrer` 移除元素后，判断此时`weak_entry_t`中是否还有元素 （`empty==true?` ） 如果此时`weak_entry_t`已经没有元素了，则需要将`weak_entry_t`从`weak_table`中移除

OK，上面的所有就是当我们将一个`obj`作`weak`引用时，所发生的事情。那么，当`obj`释放时，所有weak引用它的指针又是如何自动设置为nil的呢？接下来我们来看一下obj释放时，所发生的事情。

Dealloc

当对象引用计数为0时，runtime会调用`_objc_rootDealloc`方法来析构对象，实现如下：

```
void _objc_rootDealloc(id obj)
{
    assert(obj);

    obj->rootDealloc();
}
```

它会调用`objc_object`的`rootDealloc`方法：

```
inline void objc_object::rootDealloc()
{
    if (isTaggedPointer()) return;  // fixme necessary?

    if (fastpath(isa.nonpointer  &&
                 !isa.weakly_referenced  &&
                 !isa.has_assoc  &&
                 !isa.has_cxx_dtor  &&
                 !isa.has_sidetable_rc))
    {
        assert(!sidetable_present());
        free(this);
    }
    else {
        object_dispose((id)this);
    }
}
```

`rootDealloc`的实现逻辑如下：

- 1. 判断`object`是否采用了`Tagged Pointer`计数，如果是，则不进行任何析构操作。关于这一点，我们可以看出，用`Tagged Pointer`计数的对象，是不会析构的。`Tagged Pointer`计数的对象在内存中应该是类似于字符串常量的存在，多个对象指针其实会指向同一块内存

地址。

- 2. 接下来判断对象是否采用了优化的isa计数方式（`isa.nonpointer`）。如果是，则判断是否能够进行快速释放（`free(this)` 用C函数释放内存）。可以进行快速释放的前提是：对象没有被weak引用`!isa.weakly_referenced`，没有关联对象`!isa.has_assoc`，没有自定义的C++析构方法`!isa.has_cxx_dtor`，没有用到sideTable来做引用计数`!isa.has_sidetable_rc`。
- 3. 其余的，则进入`object_dispose((id)this)`慢释放分支。

如果`obj`被weak引用了，应该进入`object_dispose((id)this)`分支：

```
id object_dispose(id obj)
{
    if (!obj) return nil;

    objc_destructInstance(obj);
    free(obj);

    return nil;
}
```

`object_dispose`方法中，会先调用`objc_destructInstance(obj)`来析构`obj`，再用`free(obj)`来释放内存。

`objc_destructInstance`的实现如下：

```
void *objc_destructInstance(id obj)
{
    if (obj) {
        // Read all of the flags at once for performance.
        bool cxx = obj->hasCxxDtor();
        bool assoc = obj->hasAssociatedObjects();

        // This order is important.
        if (cxx) object_cxxDestruct(obj); // 调用C++析构函数
        if (assoc) _object_remove_assocations(obj); // 移除所有的关联对象，并将其自身从Association Manager的map中移除
        obj->clearDeallocating(); // 清理相关的引用
    }
    return obj;
}
```

在`objc_destructInstance`中，会清理相关的引用：`obj->clearDeallocating()`：

```
inline void objc_object::clearDeallocating()
{
    if (slowpath(!isa.nonpointer)) {
        // Slow path for raw pointer isa.
        sidetable_clearDeallocating();
    }
    else if (slowpath(isa.weakly_referenced  ||  isa.has_sidetable_rc)) {
        // Slow path for non-pointer isa with weak refs and/or side table
data.
        clearDeallocating_slow();
    }

    assert(!sidetable_present());
}
```

`clearDeallocating`中有两个分支，先判断`obj`是否采用了优化`isa`引用计数。没有，则要清理`obj`存储在`sideTable`中的引用计数等信息，这个分支在当前64位设备中应该不会进入，不必关心。 如果启用了`isa`优化，则判断是否使用了`sideTable`，使用的原因是因为做了weak引用（`isa.weakly_referenced`） 或 使用了`sideTable`的辅助引用计数（`isa.has_sidetable_rc`）。符合这两种情况之一，则进入慢析构路径：

```
   // Slow path for non-pointer isa with weak refs and/or side table data.
   clearDeallocating_slow();
```

```
NEVER_INLINE void objc_object::clearDeallocating_slow()
{
    assert(isa.nonpointer  &&  (isa.weakly_referenced ||
isa.has_sidetable_rc));

    SideTable& table = SideTables()[this]; // 在全局的SideTables中，以this指针为
key，找到对应的SideTable
    table.lock();
    if (isa.weakly_referenced) { // 如果obj被弱引用
        weak_clear_no_lock(&table.weak_table, (id)this); // 在SideTable的
weak_table中对this进行清理工作
    }
    if (isa.has_sidetable_rc) { // 如果采用了SideTable做引用计数
        table.refcnts.erase(this); // 在SideTable的引用计数中移除this
    }
    table.unlock();
}
```

这里调用了`weak_clear_no_lock`来做`weak_table`的清理工作，同时将所有weak引用该对象的`ptr`置为`nil`：

```
void weak_clear_no_lock(weak_table_t *weak_table, id referent_id)
```

```
{
    objc_object *referent = (objc_object *)referent_id;

    weak_entry_t *entry = weak_entry_for_referent(weak_table, referent); // 找到referent在weak_table中对应的weak_entry_t
    if (entry == nil) {
        /// XXX shouldn't happen, but does with mismatched CF/objc
        //printf("XXX no entry for clear deallocating %p\n", referent);
        return;
    }

    // zero out references
    weak_referrer_t *referrers;
    size_t count;

    // 找出weak引用referent的weak 指针地址数组以及数组长度
    if (entry->out_of_line()) {
        referrers = entry->referrers;
        count = TABLE_SIZE(entry);
    }
    else {
        referrers = entry->inline_referrers;
        count = WEAK_INLINE_COUNT;
    }

    for (size_t i = 0; i < count; ++i) {
        objc_object **referrer = referrers[i]; // 取出每个weak ptr的地址
        if (referrer) {
            if (*referrer == referent) { // 如果weak ptr确实weak引用了referent，则将weak ptr设置为nil，这也就是为什么weak 指针会自动设置为nil的原因
                *referrer = nil;
            }
            else if (*referrer) { // 如果所存储的weak ptr没有weak 引用referent，这可能是由于runtime代码的逻辑错误引起的，报错
                _objc_inform("__weak variable at %p holds %p instead of %p. "
                             "This is probably incorrect use of "
                             "objc_storeWeak() and objc_loadWeak(). "
                             "Break on objc_weak_error to debug.\n",
                             referrer, (void*)*referrer, (void*)referent);
                objc_weak_error();
            }
        }
    }

    weak_entry_remove(weak_table, entry); // 由于referent要被释放了，因此referent的weak_entry_t也要移除出weak_table
}
```

OK，上面就是为什么当对象析构时，所有弱引用该对象的指针都会被设置为nil的原因。

总结

纵观weak引用的底层实现，其实原理很简单。就是将所有弱引用`obj`的指针地址都保存在`obj`对应的`weak_entry_t`中。当`obj`要析构时，会遍历`weak_entry_t`中保存的弱引用指针地址，并将弱引用指针指向`nil`，同时，将`weak_entry_t`移除出`weak_table`。

这里涉及到runtime 四个重要的数据结构：
`SideTables`，`SideTable`，`weak_table`，`weak_entry_t`.

关于它们，我们在Objective-C runtime机制(5)——iOS 内存管理中已有涉及。

为了加深对runtime的理解，在接下来的一章中，我们会依次分析这四个数据结构。

在runtime中，有四个数据结构非常重要，分别是`SideTables`，`SideTable`，`weak_table_t`和`weak_entry_t`。它们和对象的引用计数，以及weak引用相关。

关系

先说一下这四个数据结构的关系。 在runtime内存空间中，`SideTables`是一个8个元素长度(在MacOS为64个)的hash数组，里面存储了`SideTable`。`SideTables`的hash键值就是一个对象`obj`的`address`。

因此可以说，一个`obj`，对应了一个`SideTable`。但是一个`SideTable`，会对应多个`obj`。因为`SideTable`的数量只有8个，所以会有很多`obj`共用同一个`SideTable`。

而在一个`SideTable`中，又有两个成员，分别是

```
RefcountMap refcnts;          // 对象引用计数相关 map
weak_table_t weak_table;      // 对象弱引用相关 table
```

其中，`refcents`是一个`hash map`，其`key`是`obj`的地址，而`value`，则是`obj`对象的`引用计数`。

而`weak_table`则存储了弱引用`obj`的指针的地址，其本质是一个以`obj地址`为`key`，弱引用`obj的指针的地址`作为`value`的`hash表`。`hash表`的节点类型是`weak_entry_t`。

这四个数据结构的关系如下图：

SideTables

先来说一下最外层的`SideTables`。`SideTables`可以理解为一个全局的`hash数组`，里面存储了`SideTable`类型的数据，其长度为8(Mac OS 为 64)。

`SideTabls`可以通过全局的静态函数获取：

```
static StripedMap<SideTable>& SideTables() {
    return *reinterpret_cast<StripedMap<SideTable>*>(SideTableBuf);
}
```

可以看到，`SideTabls` 实质类型为模板类型`StripedMap` 。

StripedMap

我们继续来看`StripedMap`模板的定义：

```
enum { CacheLineSize = 64 };

// StripedMap<T> is a map of void* -> T, sized appropriately
// for cache-friendly lock striping.
// For example, this may be used as StripedMap<spinlock_t>
// or as StripedMap<SomeStruct> where SomeStruct stores a spin lock.
template<typename T>
class StripedMap {
#if TARGET_OS_IPHONE && !TARGET_OS_SIMULATOR
    enum { StripeCount = 8 };
#else
    enum { StripeCount = 64 };
#endif

    struct PaddedT {
        T value alignas(CacheLineSize); // T value 64字节对齐
    };

    PaddedT array[StripeCount]; // 所有PaddedT struct 类型数据被存储在array数组中。
iOS 设备 StripeCount == 64

    static unsigned int indexForPointer(const void *p) { // 该方法以void *作为
key 来获取void *对应在StripedMap 中的位置
        uintptr_t addr = reinterpret_cast<uintptr_t>(p);
        return ((addr >> 4) ^ (addr >> 9)) % StripeCount; // % StripeCount 防止
index越界
    }

 public:
    // 取值方法 [p],
    T& operator[] (const void *p) {
        return array[indexForPointer(p)].value;
    }
    const T& operator[] (const void *p) const {
        return const_cast<StripedMap<T>>(this)[p];
    }
```

```
    // Shortcuts for StripedMaps of locks.
    void lockAll() {
        for (unsigned int i = 0; i < StripeCount; i++) {
            array[i].value.lock();
        }
    }

    void unlockAll() {
        for (unsigned int i = 0; i < StripeCount; i++) {
            array[i].value.unlock();
        }
    }

    void forceResetAll() {
        for (unsigned int i = 0; i < StripeCount; i++) {
            array[i].value.forceReset();
        }
    }

    void defineLockOrder() {
        for (unsigned int i = 1; i < StripeCount; i++) {
            lockdebug_lock_precedes_lock(&array[i-1].value, &array[i].value);
        }
    }

    void precedeLock(const void *newlock) {
        // assumes defineLockOrder is also called
        lockdebug_lock_precedes_lock(&array[StripeCount-1].value, newlock);
    }

    void succeedLock(const void *oldlock) {
        // assumes defineLockOrder is also called
        lockdebug_lock_precedes_lock(oldlock, &array[0].value);
    }

    const void *getLock(int i) {
        if (i < StripeCount) return &array[i].value;
        else return nil;
    }

#if DEBUG
    StripedMap() {
        // Verify alignment expectations.
        uintptr_t base = (uintptr_t)&array[0].value;
        uintptr_t delta = (uintptr_t)&array[1].value - base;
        ASSERT(delta % CacheLineSize == 0);
        ASSERT(base % CacheLineSize == 0);
    }
#else
```

```
    constexpr StripedMap() {}
#endif
};
```

通过开头的英文注释， `// StripedMap is a map of void* -> T, sized appropriately` 可以知道， StripedMap 是一个以void *为hash key， T为vaule的hash 表。 hash定位的算法如下：

```
    static unsigned int indexForPointer(const void *p) { // 该方法以void *作为key 来获取void *对应在StripedMap 中的位置
        uintptr_t addr = reinterpret_cast<uintptr_t>(p);
        return ((addr >> 4) ^ (addr >> 9)) % StripeCount; // % StripeCount 防止index越界
    }
```

把地址指针右移4位异或地址指针右移9位，为什么这么做，也不用关心。我们只要关心重点是最后的值要取余StripeCount，来防止index越界就好。

StripedMap的所有T类型数据都被封装到PaddedT中：

```
    struct PaddedT {
        T value alignas(CacheLineSize); // T value 64字节对齐
    };
```

之所以再次封装到PaddedT （有填充的T）中，是为了字节对齐，估计是存取hash值时的效率考虑。

接下来，这些PaddedT被放到数组array中：

```
 PaddedT array[StripeCount]; // 所有PaddedT struct 类型数据被存储在array数组中。iOS设备 StripeCount == 64
```

然后，苹果为array数组写了一些公共的存取数据的方法，主要是调用indexForPointer方法，使得外部传入的对象地址指针直接hash到对应的array节点：

```
 // 取值方法 [p],
    T& operator[] (const void *p) {
        return array[indexForPointer(p)].value;
    }
    const T& operator[] (const void *p) const {
        return const_cast<StripedMap<T>>(this)[p];
    }
```

接下来是一堆锁的操作，由于SideTabls是一个全局的hash表，因此当然必须要带锁访问。StripedMap提供了一些便捷的锁操作方法：

```
// Shortcuts for StripedMaps of locks.
    void lockAll() {
        for (unsigned int i = 0; i < StripeCount; i++) {
```

```
            array[i].value.lock();
        }
    }

    void unlockAll() {
        for (unsigned int i = 0; i < StripeCount; i++) {
            array[i].value.unlock();
        }
    }

    void forceResetAll() {
        for (unsigned int i = 0; i < StripeCount; i++) {
            array[i].value.forceReset();
        }
    }

    void defineLockOrder() {
        for (unsigned int i = 1; i < StripeCount; i++) {
            lockdebug_lock_precedes_lock(&array[i-1].value, &array[i].value);
        }
    }

    void precedeLock(const void *newlock) {
        // assumes defineLockOrder is also called
        lockdebug_lock_precedes_lock(&array[StripeCount-1].value, newlock);
    }

    void succeedLock(const void *oldlock) {
        // assumes defineLockOrder is also called
        lockdebug_lock_precedes_lock(oldlock, &array[0].value);
    }

    const void *getLock(int i) {
        if (i < StripeCount) return &array[i].value;
        else return nil;
    }
```

可以看到，所有的`StripedMap`锁操作，最终是调用的`array[i].value`的相关操作。因此，对于模板的抽象数据`T类型`，必须具备相关的`lock`操作接口。

因此，要用`StripedMap`作为模板`hash表`，对于`T类型`还是有所要求的。而在`SideTables`中，`T`即为`SideTable`类型，我们稍后会看到`SideTable`是如何符合`StripedMap`的数据类型要求的。

分析完了`StripedMap`，也就分析完了`SideTables`这个全局的`大hash表`。现在就来继续分析`SideTables`中存储的数据`SideTable`吧。

SideTable

`SideTable` 翻译过来的意思是“边桌”，可以放一下小东西。这里，主要存放了OC对象的引用计数和弱引用相关信息。定义如下:

```
struct SideTable {
    spinlock_t slock;            // 自旋锁，防止多线程访问冲突
    RefcountMap refcnts;         // 对象引用计数map
    weak_table_t weak_table;     // 对象弱引用map

    SideTable() {
        memset(&weak_table, 0, sizeof(weak_table));
    }

    ~SideTable() {
        _objc_fatal("Do not delete SideTable.");
    }

    // 锁操作 符合StripedMap对T的定义
    void lock() { slock.lock(); }
    void unlock() { slock.unlock(); }
    void forceReset() { slock.forceReset(); }

    // Address-ordered lock discipline for a pair of side tables.

    template<HaveOld, HaveNew>
    static void lockTwo(SideTable *lock1, SideTable *lock2);
    template<HaveOld, HaveNew>
    static void unlockTwo(SideTable *lock1, SideTable *lock2);
};
```

`SideTable`的定义很清晰，有三个成员:

- **`spinlock_t slock`**: 自旋锁，用于上锁/解锁 `SideTable`。
- **`RefcountMap refcnts`** ： 以`DisguisedPtr<objc_object>`为key的hash表，用来存储OC对象的引用计数(仅在未开启isa优化 或 在isa优化情况下isa_t的引用计数溢出时才会用到)。
- **`weak_table_t weak_table`** : 存储对象弱引用指针的hash表。是OC weak功能实现的核心数据结构。

除了三个成员外，苹果为`SideTable`还写了构造和析构函数:

```
// 构造函数
    SideTable() {
        memset(&weak_table, 0, sizeof(weak_table));
    }

    //析构函数(看看函数体，苹果设计的SideTable其实不希望被析构，不然会引起fatal 错误)
    ~SideTable() {
        _objc_fatal("Do not delete SideTable.");
    }
```

通过析构函数可以知道，`SideTable`是不能被析构的。

最后是一堆锁的操作，用于多线程访问`SideTable`， 同时，也符合我们上面提到的`StripedMap`中关于`value`的`lock`接口定义：

```
// 锁操作 符合StripedMap对T的定义
    void lock() { slock.lock(); }
    void unlock() { slock.unlock(); }
    void forceReset() { slock.forceReset(); }

    // Address-ordered lock discipline for a pair of side tables.

    template<HaveOld, HaveNew>
    static void lockTwo(SideTable *lock1, SideTable *lock2);
    template<HaveOld, HaveNew>
    static void unlockTwo(SideTable *lock1, SideTable *lock2);
```

spinlock_t slock

`spinlock_t`的最终定义实际上是一个`uint32_t`类型的非公平的自旋锁。所谓非公平，就是说获得锁的顺序和申请锁的顺序无关，也就是说，第一个申请锁的线程有可能会是最后一个获得到该锁，或者是刚获得锁的线程会再次立刻获得到该锁，造成`饥饿等待`。 同时，在OC中， `_os_unfair_lock_opaque`也记录了获取它的线程信息，只有获得该锁的线程才能够解开这把锁。

```
typedef struct os_unfair_lock_s {
  uint32_t _os_unfair_lock_opaque;
} os_unfair_lock, *os_unfair_lock_t;
```

关于自旋锁的实现，苹果并未公布，但是大体上应该是通过操作`_os_unfair_lock_opaque`这个`uint32_t`的值，当大于0时，锁可用，当等于或小于0时，需要锁等待。

RefcountMap refcnts

`RefcountMap refcnts`用来存储OC对象的引用计数。它实质上是一个以`objc_object`为`key`的hash表，其`vaule`就是OC对象的引用计数。同时，当OC对象的引用计数变为0时，会自动将相关的信息从hash表中剔除。RefcountMap的定义如下：

```
// RefcountMap disguises its pointers because we
// don't want the table to act as a root for `leaks`.
typedef objc::DenseMap<DisguisedPtr<objc_object>,size_t,true> RefcountMap;
```

实质上是模板类型`objc::DenseMap`。模板的三个类型参数`DisguisedPtr<objc_object>`， `size_t`, `true`分别表示`DenseMap`的`hash key类型`， `value类型`，是否需要在`value==0`的时候自动释放掉响应的`hash节点`，这里是`true`。

而DenseMap这个模板类型又继承与另一个Base 模板类型DenseMapBase：

```
template<typename KeyT, typename ValueT,
         bool ZeroValuesArePurgeable = false,
         typename KeyInfoT = DenseMapInfo<KeyT> >
class DenseMap
    : public DenseMapBase<DenseMap<KeyT, ValueT, ZeroValuesArePurgeable,
KeyInfoT>,
                          KeyT, ValueT, KeyInfoT, ZeroValuesArePurgeable>
```

关于`DenseMap`的定义，苹果写了一大坨，有些复杂，这里就不去深究了，有兴趣的同学可以自己去看下相关的源码部分。

weak_table_t weak_table

重点来了，`weak_table_t weak_table`用来存储OC对象弱引用的相关信息。我们知道，`SideTables`一共只有8个节点，而在我们的APP中，一般都会不只有8个对象，因此，多个对象一定会重用同一个`SideTable`节点，也就是说，一个`weak_table`会存储多个对象的弱引用信息。因此在一个`SideTable`中，又会通过`weak_table`作为`hash表`再次分散存储每一个对象的弱引用信息。

`weak_table_t`的定义如下：

```
/**
 * The global weak references table. Stores object ids as keys,
 * and weak_entry_t structs as their values.
 */
struct weak_table_t {
    weak_entry_t *weak_entries;         // hash数组，用来存储弱引用对象的相关信息
weak_entry_t
    size_t    num_entries;              // hash数组中的元素个数
    uintptr_t mask;                     // hash数组长度-1，会参与hash计算。（注意，这
里是hash数组的长度，而不是元素个数。比如，数组长度可能是64，而元素个数仅存了2个）
    uintptr_t max_hash_displacement;    // 可能会发生的hash冲突的最大次数，用于判断是
否出现了逻辑错误（hash表中的冲突次数绝不会超过改值）
};
```

`weak_table_t`是一个典型的hash结构。其中`weak_entry_t *weak_entries`是一个动态数组，用来存储`weak_table_t`的数据元素`weak_entry_t`。 剩下的三个元素将会用于hash表的相关操作。`weak_table`的hash定位操作如下所示：

```
static weak_entry_t *
weak_entry_for_referent(weak_table_t *weak_table, objc_object *referent)
{
    assert(referent);

    weak_entry_t *weak_entries = weak_table->weak_entries;
```

```
    if (!weak_entries) return nil;

    size_t begin = hash_pointer(referent) & weak_table->mask;  // 这里通过 &
weak_table->mask的位操作，来确保index不会越界
    size_t index = begin;
    size_t hash_displacement = 0;
    while (weak_table->weak_entries[index].referent != referent) {
        index = (index+1) & weak_table->mask;
        if (index == begin) bad_weak_table(weak_table->weak_entries); // 触发
bad weak table crash
        hash_displacement++;
        if (hash_displacement > weak_table->max_hash_displacement) { // 当hash
冲突超过了可能的max hash 冲突时，说明元素没有在hash表中，返回nil
            return nil;
        }
    }

    return &weak_table->weak_entries[index];
}
```

上面的定位操作还是比较清晰的，首先通过

```
 size_t begin = hash_pointer(referent) & weak_table->mask;
```

来尝试确定hash的初始位置。注意，这里做了`& weak_table->mask`位操作来确保index不会越界，这同我们平时用到的取余%操作是一样的功能。只不过这里改用了位操作，提升了效率。

然后，就开始对比hash表中的数据是否与目标数据相等`while (weak_table->weak_entries[index].referent != referent)`，如果不相等，则`index +1`， 直到`index == begin`（绕了一圈）或超过了可能的`hash冲突最大值`。

这是`weak_table_t`如何进行hash定位的相关操作。

关于`weak_table_t`中如何添加/删除元素，我们在上一章Objective-C runtime机制(6)——weak引用的底层实现原理中已有分析，在这里我们不再展开。

weak_entry_t

`weak_table_t`中存储的元素是`weak_entry_t`类型，每个`weak_entry_t`类型对应了一个OC对象的弱引用信息。

`weak_entry_t`的结构和`weak_table_t`很像，同样也是一个hash表，其存储的元素是`weak_referrer_t`，实质上是弱引用该对象的指针的指针，即 `objc_object **new_referrer`，通过操作指针的指针，就可以使得weak 引用的指针在对象析构后，指向`nil`。

```
// The address of a __weak variable.
// These pointers are stored disguised so memory analysis tools
// don't see lots of interior pointers from the weak table into objects.

typedef DisguisedPtr<objc_object *> weak_referrer_t;
```

`weak_entry_t`的定义如下：

```
/**
 * The internal structure stored in the weak references table.
 * It maintains and stores
 * a hash set of weak references pointing to an object.
 * If out_of_line_ness != REFERRERS_OUT_OF_LINE then the set
 * is instead a small inline array.
 */
#define WEAK_INLINE_COUNT 4

// out_of_line_ness field overlaps with the low two bits of
inline_referrers[1].
// inline_referrers[1] is a DisguisedPtr of a pointer-aligned address.
// The low two bits of a pointer-aligned DisguisedPtr will always be 0b00
// (disguised nil or 0x80..00) or 0b11 (any other address).
// Therefore out_of_line_ness == 0b10 is used to mark the out-of-line state.
#define REFERRERS_OUT_OF_LINE 2

struct weak_entry_t {
    DisguisedPtr<objc_object> referent; // 被弱引用的对象

    // 引用该对象的对象列表，联合。 引用个数小于4，用inline_referrers数组。 用个数大于4，
用动态数组weak_referrer_t *referrers
    union {
        struct {
            weak_referrer_t *referrers;                       // 弱引用该对象的对
象指针地址的hash数组
            uintptr_t        out_of_line_ness : 2;            // 是否使用动态hash
数组标记位
            uintptr_t        num_refs : PTR_MINUS_2;          // hash数组中的元素
个数
            uintptr_t        mask;                            // hash数组长度-1，
会参与hash计算。（注意，这里是hash数组的长度，而不是元素个数。比如，数组长度可能是64，而元素
个数仅存了2个）素个数）。
            uintptr_t        max_hash_displacement;           // 可能会发生的hash
冲突的最大次数，用于判断是否出现了逻辑错误（hash表中的冲突次数绝不会超过改值）
        };
        struct {
            // out_of_line_ness field is low bits of inline_referrers[1]
            weak_referrer_t  inline_referrers[WEAK_INLINE_COUNT];
        };
```

```
    };

    bool out_of_line() {
        return (out_of_line_ness == REFERRERS_OUT_OF_LINE);
    }

    weak_entry_t& operator=(const weak_entry_t& other) {
        memcpy(this, &other, sizeof(other));
        return *this;
    }

    weak_entry_t(objc_object *newReferent, objc_object **newReferrer)
        : referent(newReferent) // 构造方法，里面初始化了静态数组
    {
        inline_referrers[0] = newReferrer;
        for (int i = 1; i < WEAK_INLINE_COUNT; i++) {
            inline_referrers[i] = nil;
        }
    }
};
```

`weak_entry_t`的结构也比较清晰：

- `DisguisedPtr<objc_object> referent`： 弱引用对象指针摘要。其实可以理解为弱引用对象的指针，只不过这里使用了摘要的形式存储。（所谓摘要，其实是把地址取负）。
- `union`： 接下来是一个联合，union有两种形式：定长数组`weak_referrer_t inline_referrers[WEAK_INLINE_COUNT]` 和 动态数组 `weak_referrer_t *referrers`。这两个数组是用来存储弱引用该对象的指针的指针的，同样也使用了指针摘要的形式存储。当弱引用该对象的指针数目小于等于`WEAK_INLINE_COUNT`时，使用定长数组。当超过`WEAK_INLINE_COUNT`时，会将定长数组中的元素转移到动态数组中，并之后都是用动态数组存储。关于定长数组/动态数组 切换这部分，我们在稍后详细分析。
- `bool out_of_line()`： 该方法用来判断当前的`weak_entry_t`是使用的定长数组还是动态数组。当返回`true`，此时使用的动态数组，当返回`false`，使用静态数组。
- `weak_entry_t& operator=(const weak_entry_t& other)`： 赋值方法
- `weak_entry_t(objc_object *newReferent, objc_object **newReferrer)`： 构造方法。

定长数组 / 动态数组

`weak_entry_t`会存储所有弱引用该对象的指针的指针。存储类型为`weak_referrer_t`，其实就是弱引用指针的指针。但是是以指针摘要的形式存储的：

```
typedef DisguisedPtr<objc_object *> weak_referrer_t;
```

`weak_entry_t`会将`weak_referrer_t`存储到hash数组中，而这个hash数组会有两种形态：定长数组/动态数组：

```
    union {
        // 动态数组模式
        struct {
            weak_referrer_t *referrers;                         // 弱引用该对象的对象指针地址的hash数组
            uintptr_t        out_of_line_ness : 2;              // 是否使用动态hash数组标记位
            uintptr_t        num_refs : PTR_MINUS_2;            // hash数组中的元素个数
            uintptr_t        mask;                              // hash数组长度-1，会参与hash计算。（注意，这里是hash数组的长度，而不是元素个数。比如，数组长度可能是64，而元素个数仅存了2个）素个数）。
            uintptr_t        max_hash_displacement;             // 可能会发生的hash冲突的最大次数，用于判断是否出现了逻辑错误（hash表中的冲突次数绝不会超过改值）
        };
    // 定长数组模式
        struct {
            // out_of_line_ness field is low bits of inline_referrers[1]
            weak_referrer_t  inline_referrers[WEAK_INLINE_COUNT];
        };
    };

    bool out_of_line() {
        return (out_of_line_ness == REFERRERS_OUT_OF_LINE);
    }
```

当弱引用指针个数少于等于`WEAK_INLINE_COUNT`时，会使用定长数组`inline_referrers`。而当大于`WEAK_INLINE_COUNT`时，则会转换到动态数组模式`weak_referrer_t *referrers`。

之所以做定长/动态数组的切换，应该是苹果考虑到弱引用的指针个数一般不会超过`WEAK_INLINE_COUNT`个。这时候使用定长数组，不需要动态的申请内存空间，而是一次分配一块连续的内存空间。这会得到运行效率上的提升。

至于`weak_entry_t`是使用的定长/动态数组，苹果提供了方法：

```
#define REFERRERS_OUT_OF_LINE 2

    bool out_of_line() {
        return (out_of_line_ness == REFERRERS_OUT_OF_LINE);
    }
```

该方法的实质是测试定长数组第二个元素值的2进制位第2位是否等于2。因为根据苹果的注释，`inline_referrers[1]`中存储的是`pointer-aligned DisguisedPtr`，即指针对齐的指针摘要，其最低位一定是`0b00`或`0b11`，因此可以用`0b10`表示使用了动态数组。

下面我就来看一下weak_entry_t 中是如何插入元素的：

```
/**
```

```
 * Add the given referrer to set of weak pointers in this entry.
 * Does not perform duplicate checking (b/c weak pointers are never
 * added to a set twice).
 *
 * @param entry The entry holding the set of weak pointers.
 * @param new_referrer The new weak pointer to be added.
 */
static void append_referrer(weak_entry_t *entry, objc_object **new_referrer)
{
    if (! entry->out_of_line()) { // 如果weak_entry 尚未使用动态数组，走这里
        // Try to insert inline.
        for (size_t i = 0; i < WEAK_INLINE_COUNT; i++) {
            if (entry->inline_referrers[i] == nil) {
                entry->inline_referrers[i] = new_referrer;
                return;
            }
        }

        // 如果inline_referrers的位置已经存满了，则要转型为referrers，做动态数组。
        // Couldn't insert inline. Allocate out of line.
        weak_referrer_t *new_referrers = (weak_referrer_t *)
            calloc(WEAK_INLINE_COUNT, sizeof(weak_referrer_t));
        // This constructed table is invalid, but grow_refs_and_insert
        // will fix it and rehash it.
        for (size_t i = 0; i < WEAK_INLINE_COUNT; i++) {
            new_referrers[i] = entry->inline_referrers[I];
        }
        entry->referrers = new_referrers;
        entry->num_refs = WEAK_INLINE_COUNT;
        entry->out_of_line_ness = REFERRERS_OUT_OF_LINE;
        entry->mask = WEAK_INLINE_COUNT-1;
        entry->max_hash_displacement = 0;
    }

    // 对于动态数组的附加处理：
    assert(entry->out_of_line()); // 断言： 此时一定使用的动态数组

    if (entry->num_refs >= TABLE_SIZE(entry) * 3/4) { // 如果动态数组中元素个数大于或等于数组位置总空间的3/4，则扩展数组空间为当前长度的一倍
        return grow_refs_and_insert(entry, new_referrer); // 扩容，并插入
    }

    // 如果不需要扩容，直接插入到weak_entry中
    // 注意，weak_entry是一个哈希表，key：w_hash_pointer(new_referrer) value: new_referrer

    // 细心的人可能注意到了，这里weak_entry_t 的hash算法和 weak_table_t的hash算法是一样的，同时扩容/减容的算法也是一样的
```

```
    size_t begin = w_hash_pointer(new_referrer) & (entry->mask); // '& (entry->mask)' 确保了 begin的位置只能大于或等于 数组的长度
    size_t index = begin;  // 初始的hash index
    size_t hash_displacement = 0;  // 用于记录hash冲突的次数，也就是hash再位移的次数
    while (entry->referrers[index] != nil) {
        hash_displacement++;
        index = (index+1) & entry->mask;  // index + 1，移到下一个位置，再试一次能否插入。（这里要考虑到entry->mask取值，一定是：0x111，0x1111，0x11111，... ，因为数组每次都是*2增长，即8， 16， 32，对应动态数组空间长度-1的mask，也就是前面的取值。）
        if (index == begin) bad_weak_table(entry); // index == begin 意味着数组绕了一圈都没有找到合适位置，这时候一定是出了什么问题。
    }
    if (hash_displacement > entry->max_hash_displacement) { // 记录最大的hash冲突次数，max_hash_displacement意味着：我们尝试至多max_hash_displacement次，肯定能够找到object对应的hash位置
        entry->max_hash_displacement = hash_displacement;
    }
    // 将ref存入hash数组，同时，更新元素个数num_refs
    weak_referrer_t &ref = entry->referrers[index];
    ref = new_referrer;
    entry->num_refs++;
}
```

代码可以分成两部分理解，一部分是使用定长数组的情况：

```
  if (! entry->out_of_line()) { // 如果weak_entry 尚未使用动态数组，走这里
        // Try to insert inline.
        for (size_t i = 0; i < WEAK_INLINE_COUNT; i++) {
            if (entry->inline_referrers[i] == nil) {
                entry->inline_referrers[i] = new_referrer;
                return;
            }
        }

        // 如果inline_referrers的位置已经存满了，则要转型为referrers，做动态数组。
        // Couldn't insert inline. Allocate out of line.
        weak_referrer_t *new_referrers = (weak_referrer_t *)
            calloc(WEAK_INLINE_COUNT, sizeof(weak_referrer_t));
        // This constructed table is invalid, but grow_refs_and_insert
        // will fix it and rehash it.
        for (size_t i = 0; i < WEAK_INLINE_COUNT; i++) {
            new_referrers[i] = entry->inline_referrers[I];
        }
        entry->referrers = new_referrers;
        entry->num_refs = WEAK_INLINE_COUNT;
        entry->out_of_line_ness = REFERRERS_OUT_OF_LINE;
        entry->mask = WEAK_INLINE_COUNT-1;
        entry->max_hash_displacement = 0;
    }
```

定长数组的逻辑很简单，直接安装数组顺序，将new_referrer插入即可。如果定长数组已经用尽，则将定长数组转型为动态数组：

```
weak_referrer_t *new_referrers = (weak_referrer_t *)
            calloc(WEAK_INLINE_COUNT, sizeof(weak_referrer_t));
...

entry->referrers = new_referrers; // hash数组由 entry->inline_referrers转换为
entry->referrers
```

要注意，定长数组转换为动态数组后，新的元素并没有插入到数组中，而仅是将原来定长数组中的内容转移到了动态数组中。新元素的插入逻辑，在下面动态数组部分：

```
    ...
    // 对于动态数组的附加处理：
    assert(entry->out_of_line()); // 断言：  此时一定使用的动态数组

    if (entry->num_refs >= TABLE_SIZE(entry) * 3/4) { // 如果动态数组中元素个数大
于或等于数组位置总空间的3/4，则扩展数组空间为当前长度的一倍
        return grow_refs_and_insert(entry, new_referrer); // 扩容，并插入
    }

    // 如果不需要扩容，直接插入到weak_entry中
    // 注意，weak_entry是一个哈希表，key：w_hash_pointer(new_referrer) value:
new_referrer

    // 细心的人可能注意到了，这里weak_entry_t 的hash算法和 weak_table_t的hash算法是一
样的，同时扩容/减容的算法也是一样的
    size_t begin = w_hash_pointer(new_referrer) & (entry->mask); // '& (entry-
>mask)' 确保了 begin的位置只能大于或等于 数组的长度
    size_t index = begin;  // 初始的hash index
    size_t hash_displacement = 0;  // 用于记录hash冲突的次数，也就是hash再位移的次数
    while (entry->referrers[index] != nil) {
        hash_displacement++;
        index = (index+1) & entry->mask;  // index + 1，移到下一个位置，再试一次能
否插入。（这里要考虑到entry->mask取值，一定是：0x111，0x1111，0x11111，... ，因为数组
每次都是*2增长，即8， 16， 32，对应动态数组空间长度-1的mask，也就是前面的取值。）
        if (index == begin) bad_weak_table(entry); // index == begin 意味着数组
绕了一圈都没有找到合适位置，这时候一定是出了什么问题。
    }
    if (hash_displacement > entry->max_hash_displacement) { // 记录最大的hash冲
突次数，max_hash_displacement意味着：我们尝试至多max_hash_displacement次，肯定能够找
到object对应的hash位置
        entry->max_hash_displacement = hash_displacement;
    }
    // 将ref存入hash数组，同时，更新元素个数num_refs
    weak_referrer_t &ref = entry->referrers[index];
```

```
    ref = new_referrer;
    entry->num_refs++;
}
```

其实这部分的逻辑和`weak_table_t`中插入`weak_entry_t`是非常类似的。都使用了`mask`取余来解决hash冲突。

我们可以再细看一下动态数组是如何动态扩容的：

```
   if (entry->num_refs >= TABLE_SIZE(entry) * 3/4) { // 如果动态数组中元素个数大于或等于数组位置总空间的3/4，则扩展数组空间为当前长度的一倍
        return grow_refs_and_insert(entry, new_referrer); // 扩容，并插入
    }
```

```
/**
 * Grow the entry's hash table of referrers. Rehashes each
 * of the referrers.
 *
 * @param entry Weak pointer hash set for a particular object.
 */
__attribute__((noinline, used))
static void grow_refs_and_insert(weak_entry_t *entry,
                                 objc_object **new_referrer)
{
    assert(entry->out_of_line());

    size_t old_size = TABLE_SIZE(entry);
    size_t new_size = old_size ? old_size * 2 : 8; // 每次扩容为上一次容量的2倍

    size_t num_refs = entry->num_refs;
    weak_referrer_t *old_refs = entry->referrers;
    entry->mask = new_size - 1;

    entry->referrers = (weak_referrer_t *)
        calloc(TABLE_SIZE(entry), sizeof(weak_referrer_t));
    entry->num_refs = 0;
    entry->max_hash_displacement = 0;

    // 这里可以看到，旧的数据需要依次转移到新的内存中
    for (size_t i = 0; i < old_size && num_refs > 0; i++) {
        if (old_refs[i] != nil) {
            append_referrer(entry, old_refs[i]); // 将旧的数据转移到新的动态数组中
            num_refs--;
        }
    }
    // Insert
    append_referrer(entry, new_referrer);
    if (old_refs) free(old_refs); // 释放旧的内存
```

```
}
```

通过代码可以看出，每一次动态数组的扩容，都需要将旧的数据重新插入到新的数组中。

总结

OK，上面就是在runtime中，关于对象引用计数和weak引用相关的数据结构。搞清楚了它们之间的关系以及各自的实现细节，相信大家会对runtime有更深的理解。

在我们前面的几章中，分析了OC的runtime一些底层的数据结构以及实现机制。今天，我们就从一个OC对象的生命周期的角度，来解析在runtime底层是如何实现的。

我们创建一个对象（或对象引用）有几种方式?

```
Student *student = [[Student alloc] init];
Student *student2 = [Student new];

__weak Student *weakStudent = [Student new];

NSDictionary *dict = [[NSDictionary alloc] init];
NSDictionary *autoreleaseDict = [NSDictionary dictionary];
```

有很多种方式，我们就来依次看一下这些方式的背后实现。

alloc

要创建一个对象，第一步就是需要为对象分配内存。在创建内存时，我们会调用`alloc`方法。查看`runtime`的`NSObject +alloc`方法实现：

```
+ (id)alloc {
    return _objc_rootAlloc(self);
}
```

```
// Base class implementation of +alloc. cls is not nil.
// Calls [cls allocWithZone:nil].
id _objc_rootAlloc(Class cls)
{
    return callAlloc(cls, false/*checkNil*/, true/*allocWithZone*/);
}
```

alloc方法会将self作为参数传入`_objc_rootAlloc(Class cls)`方法中，**注意，因为`alloc`是一个类方法，因此此时的self是一个Class类型**。

最终该方法会落脚到`callAlloc`方法。

```
static ALWAYS_INLINE id callAlloc(Class cls, bool checkNil, bool
allocWithZone=false)
```

```
static ALWAYS_INLINE id
callAlloc(Class cls, bool checkNil, bool allocWithZone=false)
{
    if (slowpath(checkNil && !cls)) return nil;

#if __OBJC2__
    if (fastpath(!cls->ISA()->hasCustomAWZ())) {
        // No alloc/allocWithZone implementation. Go straight to the
allocator.
        // fixme store hasCustomAWZ in the non-meta class and
        // add it to canAllocFast's summary
        if (fastpath(cls->canAllocFast())) { // 如果可以fast alloc，走这里
            // No ctors, raw isa, etc. Go straight to the metal.
            bool dtor = cls->hasCxxDtor();
            id obj = (id)calloc(1, cls->bits.fastInstanceSize()); // 直接调用
calloc方法，申请1块大小为bits.fastInstanceSize()的内存
            if (slowpath(!obj)) return callBadAllocHandler(cls);
            obj->initInstanceIsa(cls, dtor);
            return obj;
        }
        else { // 如果不可以fast alloc，走这里，
            // Has ctor or raw isa or something. Use the slower path.
            id obj = class_createInstance(cls, 0); // （1）需要读取cls 的
class_ro_t 中的instanceSize，并使之大于16 byte，Because : CF requires all objects
be at least 16 bytes. （2）initInstanceIsa
            if (slowpath(!obj)) return callBadAllocHandler(cls);
            return obj;
        }
    }
#endif

    // No shortcuts available.
    if (allocWithZone) return [cls allocWithZone:nil];
    return [cls alloc];
}
```

在callAlloc方法里面，做了三件事：

1. 调用`calloc`方法，为类实例分配内存
2. 调用`obj->initInstanceIsa(cls, dtor)`方法，初始化`obj`的`isa`
3. 返回`obj`

在第一件事中，调用`calloc`方法，你需要提供需要申请内存的大小。在OC中有两条分支： （1）can alloc fast （2）can't alloc fast

对于可以`alloc fast`的类，应该是经过编译器优化的类。这种类的实例大小直接被放到了bits中

```
struct class_data_bits_t {

    // Values are the FAST_ flags above.
    uintptr_t bits;
  ...
}
```

而不需要通过`bits`找到`class_rw_t->class_ro_t->instanceSize`。省略了这一条查找路径，而是直接读取位值，其创建实例的速度自然比不能`alloc fast`的类要快。

而对于不能`alloc fast`的类，则会进入第二条路径，代码会通过上面所说的通过`bits`找到`class_rw_t->class_ro_t->instanceSize`来确定需要申请内存的大小。

当申请了对象的内存后，还需要初始化类实例对象的isa成员变量：

```
 obj->initInstanceIsa(cls, hasCxxDtor);
```

```
inline void
objc_object::initInstanceIsa(Class cls, bool hasCxxDtor)
{
    assert(!cls->instancesRequireRawIsa());
    assert(hasCxxDtor == cls->hasCxxDtor());

    initIsa(cls, true, hasCxxDtor);
}
```

```
inline void
objc_object::initIsa(Class cls, bool nonpointer, bool hasCxxDtor)
{
    assert(!isTaggedPointer());

    if (!nonpointer) { // 如果没有启用isa 优化，则直接将cls赋值给isa.cls，来表明当前
object 是哪个类的实例
        isa.cls = cls;
    } else { // 如果启用了isa 优化，则初始化isa的三个内容(1) isa基本的内容，包括
nonpointer置1以及设置OC magic vaule (2)置位has_cxx_dtor (3) 记录对象所属类的信息。 通
过 newisa.shiftcls = (uintptr_t)cls >> 3;
        assert(!DisableNonpointerIsa);
        assert(!cls->instancesRequireRawIsa());

        isa_t newisa(0);

#if SUPPORT_INDEXED_ISA
        assert(cls->classArrayIndex() > 0);
        newisa.bits = ISA_INDEX_MAGIC_VALUE;
```

```
        // isa.magic is part of ISA_MAGIC_VALUE
        // isa.nonpointer is part of ISA_MAGIC_VALUE
        newisa.has_cxx_dtor = hasCxxDtor;
        newisa.indexcls = (uintptr_t)cls->classArrayIndex();
#else
        newisa.bits = ISA_MAGIC_VALUE;
        // isa.magic is part of ISA_MAGIC_VALUE
        // isa.nonpointer is part of ISA_MAGIC_VALUE
        newisa.has_cxx_dtor = hasCxxDtor;
        newisa.shiftcls = (uintptr_t)cls >> 3;
#endif

        // This write must be performed in a single store in some cases
        // (for example when realizing a class because other threads
        // may simultaneously try to use the class).
        // fixme use atomics here to guarantee single-store and to
        // guarantee memory order w.r.t. the class index table
        // ...but not too atomic because we don't want to hurt instantiation
        isa = newisa;
    }
}
```

结合代码注释，以及我们在iOS 内存管理中提到的关于`isa`的描述，应该可以理解`isa`初始化的逻辑。

init

我们再来看一下init方法：

```
- (id)init {
    return _objc_rootInit(self);
}
```

```
id _objc_rootInit(id obj)
{
    // In practice, it will be hard to rely on this function.
    // Many classes do not properly chain -init calls.
    return obj;
}
```

实现很简单，就是将自身返回，没有做任何其他操作。

__strong

```
Student *student = [[Student alloc] init];
Student *student2 = [Student new];
```

在等号的右边，我们通过alloc和new的方式创建了两个OC对象。而在左面，我们通过Student *的方式来引用这些对象。

在OC中，对对象所有的引用都是有所有权修饰符的，所有权修饰符会告诉编译器，该如何处理对象的引用关系。如果代码中没有显示指明所有权修饰符，则默认为`__strong`所有权。

因此上面代码实际是：

```
__strong Student *student = [[Student alloc] init];
__strong Student *student2 = [Student new];
```

对于new方法，苹果的文档解释为：

> Allocates a new instance of the receiving class, sends it an initmessage, and returns the initialized object.

其实就是`alloc + init`方法的简写。因此，这里的两种创建实例对象的方式可以理解是一个。

那么，当所有权修饰符是`__strong`时，runtime是如何管理对象引用的呢?

runtime会通过`void objc_storeStrong(id *location, id obj)`方法来处理`__strong`引用。这里的`location`就是引用指针，即`Student *student`，而`obj`就是被引用的对象，即`Student`实例。

```
void objc_storeStrong(id *location, id obj)
{
    id prev = *location;
    if (obj == prev) {
        return;
    }
    objc_retain(obj);       //1.  retain obj
    *location = obj;        //2.  将location 指向 obj
    objc_release(prev);   //3. release location之前指向的obj
}
```

代码逻辑很简单，主要是调用了`objc_retain`和`objc_release`两个方法。 我们分别来看一下它们的实现。

objc_retain

```
id objc_retain(id obj)
{
    if (!obj) return obj;
    if (obj->isTaggedPointer()) return obj;
    return obj->retain();
}

inline id objc_object::retain()
{
```

```
    assert(!isTaggedPointer());

    if (fastpath(!ISA()->hasCustomRR())) {
        return rootRetain();
    }

    return ((id(*)(objc_object *, SEL))objc_msgSend)(this, SEL_retain);
}
```

可以看到，`objc_retain`方法最终会调到`objc_object`类的`rootRetain`方法：

```
ALWAYS_INLINE id
objc_object::rootRetain()
{
    return rootRetain(false, false);
}
```

```
ALWAYS_INLINE id
objc_object::rootRetain(bool tryRetain, bool handleOverflow)
{
    if (isTaggedPointer()) return (id)this;

    bool sideTableLocked = false;
    bool transcribeToSideTable = false;

    isa_t oldisa;
    isa_t newisa;

    do {
        transcribeToSideTable = false;
        oldisa = LoadExclusive(&isa.bits);
        newisa = oldisa;
        if (slowpath(!newisa.nonpointer)) {  // 如果没有采用isa优化，则返回
sidetable记录的内容，用slowpath表明这不是一个大概率事件
            ClearExclusive(&isa.bits);
            if (!tryRetain && sideTableLocked) sidetable_unlock();
            if (tryRetain) return sidetable_tryRetain() ? (id)this : nil;
            else return sidetable_retain();
        }
        // don't check newisa.fast_rr; we already called any RR overrides
        if (slowpath(tryRetain && newisa.deallocating)) {
            ClearExclusive(&isa.bits);
            if (!tryRetain && sideTableLocked) sidetable_unlock();
            return nil;
        }
        // 采用了isa优化，做extra_rc++，同时检查是否extra_rc溢出，若溢出，则extra_rc减
半，并将另一半转存至sidetable
        uintptr_t carry;
```

```
        newisa.bits = addc(newisa.bits, RC_ONE, 0, &carry);  // extra_rc++

        if (slowpath(carry)) { // 有carry值，表示extra_rc 溢出
            // newisa.extra_rc++ overflowed
            if (!handleOverflow) {  // 如果不处理溢出情况，则在这里会递归调用一次，再进来的时候，handleOverflow会被rootRetain_overflow设置为true，从而进入到下面的溢出处理流程
                ClearExclusive(&isa.bits);
                return rootRetain_overflow(tryRetain);
            }
            // Leave half of the retain counts inline and
            // prepare to copy the other half to the side table.
            if (!tryRetain && !sideTableLocked) sidetable_lock();

            // 进行溢出处理：逻辑很简单，先在extra_rc中留一半计数，同时把has_sidetable_rc设置为true，表明借用了sidetable，然后把另一半放到sidetable中
            sideTableLocked = true;
            transcribeToSideTable = true;
            newisa.extra_rc = RC_HALF;
            newisa.has_sidetable_rc = true;
        }
    } while (slowpath(!StoreExclusive(&isa.bits, oldisa.bits, newisa.bits)));
// 将oldisa 替换为 newisa，并赋值给isa.bits(更新isa_t)，如果不成功，do while再试一遍

    if (slowpath(transcribeToSideTable)) { //isa的extra_rc溢出，将一半的refer count值放到sidetable中
        // Copy the other half of the retain counts to the side table.
        sidetable_addExtraRC_nolock(RC_HALF);
    }

    if (slowpath(!tryRetain && sideTableLocked)) sidetable_unlock();
    return (id)this;
}
```

这一段`rootRetain`方法，在我们之前的文章iOS 内存管理已经做过分析。

我们就在总结一下rootRetain方法的流程：

- 1. 取出当前对象的`isa.bits`值
- 2. `isa.bits`分别赋值给`oldisa`和`newisa`
- 3. 根据`isa_t`的标志位`newisa.nonpointer`，来判断`runtime`是否只开启了`isa`优化。
- 4. 如果`newisa.nonpointer`为`0`，则走老的流程，调用`sidetable_retain`方法，在`SideTable`中找到`this`对应的节点，`side table refcntStorage + 1`
- 5. 如果`newisa.nonpointer`为1，则在`newisa.extra_rc`上做引用计数+1操作。同时，需要判断是否计数溢出。
- 6. 如果`newisa.extra_rc`溢出，则进行溢出处理：`newisa.extra_rc`计数减半，将计数的另一半放到`SideTable`中。并设置`newisa.has_sidetable_rc = true`，表明引用计数借用了`SideTable`

- 7. 最后，调用`StoreExclusive`，更新对象的`isa.bits`。

总结：

`__strong`引用会使得被引用对象计数`+1`，同时，会使得之前的饮用对象计数`-1`。

__weak

```
__weak Student *weakStudent = [Student new];
```

当使用`__weak`所有权修饰符来引用对象时？会发生什么呢？

当`weakStudent`弱引用`Student`对象时，会调用`objc_initWeak`方法。当`weakStudent`超出其作用域要销毁时，会调用`objc_destoryWeak`方法。 我们分别看一下它们的实现：

```
/**
 * Initialize a fresh weak pointer to some object location.
 * It would be used for code like:
 *
 * (The nil case)
 * __weak id weakPtr;
 * (The non-nil case)
 * NSObject *o = ...;
 * __weak id weakPtr = o;
 *
 * This function IS NOT thread-safe with respect to concurrent
 * modifications to the weak variable. (Concurrent weak clear is safe.)
 *
 * @param location Address of __weak ptr.
 * @param newObj Object ptr.
 */

// @param location __weak 指针的地址
// @param newObj 被弱引用的对象指针
// @return __weak 指针

id
objc_initWeak(id *location, id newObj)
{
    if (!newObj) {
        *location = nil;
        return nil;
    }

    return storeWeak<DontHaveOld, DoHaveNew, DoCrashIfDeallocating>
        (location, (objc_object*)newObj);
}
```

```
template <HaveOld haveOld, HaveNew haveNew,
          CrashIfDeallocating crashIfDeallocating>
static id
storeWeak(id *location, objc_object *newObj)
{
    assert(haveOld  ||  haveNew);
    if (!haveNew) assert(newObj == nil);

    Class previouslyInitializedClass = nil;
    id oldObj;
    SideTable *oldTable;
    SideTable *newTable;

    // Acquire locks for old and new values.
    // Order by lock address to prevent lock ordering problems.
    // Retry if the old value changes underneath us.
 retry:
    if (haveOld) { // 如果weak ptr之前弱引用过一个obj，则将这个obj所对应的SideTable取出，赋值给oldTable
        oldObj = *location;
        oldTable = &SideTables()[oldObj];
    } else {
        oldTable = nil; // 如果weak ptr之前没有弱引用过一个obj，则oldTable = nil
    }
    if (haveNew) { // 如果weak ptr要weak引用一个新的obj，则将该obj对应的SideTable取出，赋值给newTable
        newTable = &SideTables()[newObj];
    } else {
        newTable = nil; // 如果weak ptr不需要引用一个新obj，则newTable = nil
    }

    // 加锁操作，防止多线程中竞争冲突
    SideTable::lockTwo<haveOld, haveNew>(oldTable, newTable);

    // location 应该与 oldObj 保持一致，如果不同，说明当前的 location 已经处理过 oldObj 可是又被其他线程所修改
    if (haveOld  &&  *location != oldObj) {
        SideTable::unlockTwo<haveOld, haveNew>(oldTable, newTable);
        goto retry;
    }

    // Prevent a deadlock between the weak reference machinery
    // and the +initialize machinery by ensuring that no
    // weakly-referenced object has an un-+initialized isa.
    if (haveNew  &&  newObj) {
        Class cls = newObj->getIsa();
        if (cls != previouslyInitializedClass  &&
            !((objc_class *)cls)->isInitialized())  // 如果cls还没有初始化，先初始化，再尝试设置weak
```

```
        {
            SideTable::unlockTwo<haveOld, haveNew>(oldTable, newTable);
            _class_initialize(_class_getNonMetaClass(cls, (id)newObj));

            // If this class is finished with +initialize then we're good.
            // If this class is still running +initialize on this thread
            // (i.e. +initialize called storeWeak on an instance of itself)
            // then we may proceed but it will appear initializing and
            // not yet initialized to the check above.
            // Instead set previouslyInitializedClass to recognize it on
retry.
            previouslyInitializedClass = cls; // 这里记录一下
previouslyInitializedClass， 防止改if分支再次进入

            goto retry; // 重新获取一遍newObj，这时的newObj应该已经初始化过了
        }
    }

    // Clean up old value, if any.
    if (haveOld) {
        weak_unregister_no_lock(&oldTable->weak_table, oldObj, location); //
如果weak_ptr之前弱引用过别的对象oldObj，则调用weak_unregister_no_lock，在oldObj的
weak_entry_t中移除该weak_ptr地址
    }

    // Assign new value, if any.
    if (haveNew) { // 如果weak_ptr需要弱引用新的对象newObj
        // (1) 调用weak_register_no_lock方法，将weak ptr的地址记录到newObj对应的
weak_entry_t中
        newObj = (objc_object *)
            weak_register_no_lock(&newTable->weak_table, (id)newObj, location,
                                  crashIfDeallocating);
        // weak_register_no_lock returns nil if weak store should be rejected

        // (2) 更新newObj的isa的weakly_referenced bit标志位
        // Set is-weakly-referenced bit in refcount table.
        if (newObj  &&  !newObj->isTaggedPointer()) {
            newObj->setWeaklyReferenced_nolock();
        }

        // Do not set *location anywhere else. That would introduce a race.
        //  (3) *location 赋值，也就是将weak ptr直接指向了newObj。可以看到，这里并没有
将newObj的引用计数+1
        *location = (id)newObj; // 将weak ptr指向object
    }
    else {
        // No new value. The storage is not changed.
    }
```

```
    // 解锁，其他线程可以访问oldTable，newTable了
    SideTable::unlockTwo<haveOld, haveNew>(oldTable, newTable);

    return (id)newObj; // 返回newObj，此时的newObj与刚传入时相比，weakly-referenced
bit位置1
}
```

可以看到，`storeWeak`函数会根据`haveOld`参数来决定是否需要处理`weak`指针之前弱引用的对象。我们这里的`weakStudent`是第一次弱引用对象（`a fresh weak pointer`），因此，`haveOld = false`。当`haveOld = false`时，storeWeak函数做的事情如下：

- 1. 取出引用对象对应的SideTable节点SideTable *newTable;
- 2. 调用weak_register_no_lock方法，将weak pointer的地址记录到对象对应的weak_entry_t中。
- 3. 更新对象isa的weakly_referenced bit标志位，表明该对象被弱引用了。
- 4. 将weak pointer指向对象
- 5. 返回对象

关于`weak_register_no_lock`以及`weak`相关的数据结构，我们在weak引用的底层实现原理有相关探讨，就不再复述。

下面看另一种情况：

```
 __weak Son *son = [Son new];
 son = [Son new];
1
2
当weakStudent再次指向另一个对象时，则不会调用objc_initWeak方法，而是会调用
objc_storeWeak方法：

/**
 * This function stores a new value into a __weak variable. It would
 * be used anywhere a __weak variable is the target of an assignment.
 *
 * @param location The address of the weak pointer itself
 * @param newObj The new object this weak ptr should now point to
 *
 * @return \e newObj
 */
id
objc_storeWeak(id *location, id newObj)
{
    return storeWeak<DoHaveOld, DoHaveNew, DoCrashIfDeallocating>
        (location, (objc_object *)newObj);
}
```

其实还是调用了`storeWeak`方法，只不过`DontHaveOld`参数换成了`DoHaveOld`。

当传入`DoHaveOld`时，storeWeak会进入分支：

```
  // Clean up old value, if any.
    if (haveOld) {
        weak_unregister_no_lock(&oldTable->weak_table, oldObj, location); //
如果weak_ptr之前弱引用过别的对象oldObj，则调用weak_unregister_no_lock，在oldObj的
weak_entry_t中移除该weak_ptr地址
    }
```

```
void weak_unregister_no_lock(weak_table_t *weak_table, id referent_id,
                         id *referrer_id)
{
    objc_object *referent = (objc_object *)referent_id;
    objc_object **referrer = (objc_object **)referrer_id;

    weak_entry_t *entry;

    if (!referent) return;

    if ((entry = weak_entry_for_referent(weak_table, referent))) { // 查找到
referent所对应的weak_entry_t
        remove_referrer(entry, referrer);  // 在referent所对应的weak_entry_t的
hash数组中，移除referrer

        // 移除元素之后， 要检查一下weak_entry_t的hash数组是否已经空了
        bool empty = true;
        if (entry->out_of_line()  &&  entry->num_refs != 0) {
            empty = false;
        }
        else {
            for (size_t i = 0; i < WEAK_INLINE_COUNT; i++) {
                if (entry->inline_referrers[i]) {
                    empty = false;
                    break;
                }
            }
        }

        if (empty) { // 如果weak_entry_t的hash数组已经空了，则需要将weak_entry_t从
weak_table中移除
            weak_entry_remove(weak_table, entry);
        }
    }

    // Do not set *referrer = nil. objc_storeWeak() requires that the
    // value not change.
}
```

在`weak_unregister_no_lock`方法中，将`weak pointer`的地址从对象的`weak_entry_t`中移除，同时会判断`weak_entry_t`是否已经空了，如果空了，则需要把`weak_entry_t`从`weak_table`中移除。

总结：`__weak`引用对象时，会在对象的`weak_entry_t`中登记该`weak pointer`的地址（这也就是为什么当对象释放时，`weak pointe`r会被置为`nil`）。如果`weak pointer`之前已经弱引用过其他对象，则要先将`weak pointer`地址从其他对象的`weak_entry_t`中移除，同时，需要对`weak_entry_t`进行判空逻辑。

autorelease

```
NSDictionary *dict = [[NSDictionary alloc] init];
NSDictionary *autoreleaseDict = [NSDictionary dictionary];
```

当我们创建NSDictionary对象时，有这么两种方式。那么，这两种方式有什么区别呢?

在ARC时代，若方法名以下列词语开头，则其返回对象归调用者所有（意为需调用者负责释放内存，但对ARC来说，其实并没有手动release的必要）

- alloc
- new
- copy
- mutableCopy

而不使用这些词语开头的方法，如`[NSDictionary dictionary]`， 根据苹果官方文档，当调用`[NSDictionary dictionary]`时：

> This method is declared primarily for use with mutable subclasses of NSDictionary. If you don't want a temporary object, you can also create an empty dictionary using alloc and init.

似乎是说，当调用`[NSDictionary dictionary]`的形式时，会产生一个临时的对象。类似的，还有`[NSArray array]`,`[NSData data]`。

关于这种形式生成的变量，则表示"方法所返回的对象并不归调用者所有"。在这种情况下，返回的对象会自动释放。

其实我们可以理解为：当调用`dictionary`形式生成对象时，NSDictionary对象的引用计数管理，就不需要用户参与了（这在MRC时代有很大的区别，但是对于ARC来说，其实和alloc形式没有太大的区别了）。用`[NSDictionary dictionary]`其实相当于代码`[[NSDictionary alloc] init] autorelease];`,这里会将`NSDictionary`对象交给了`autorelease pool`来管理。

> 事实是这样的吗？我们查看`[NSDictionary dictionary]`的汇编代码(`Product->Perform Action->Assemble`)，可以看到，编译器会调用`objc_retainAutoreleasedReturnValue`方法。而`objc_retainAutoreleasedReturnValue`又是什么鬼？这其实是编译器的一个优化，前面我们说`[NSDictionary dictionary]`会在方法内部为`NSDictionary`实例调用`autorelease`，而如果这时候在外面用一个强引用来引用这个`NSDictionary`对象的话，还是需要调用一个`retain`，而此时的`autorelease`和`retain`其实是可以相互抵消的。于是，编译器就给了一个优化，不是直接调用`autorelease`方法，而是调

> 用`objc_retainAutoreleasedReturnValue`来做这样的判断，如果`autorelease`后面紧跟了`retain`，则将`autorelease`和`retain`都抵消掉，不再代码里面出现。（详见《Effective Objective-C 2.0》 P126）。

OK，上面是一些题外话，我们回到`autorelease`的主题上来。在ARC时代，我们通过如下形式使用autorelease：

```
@autorelease {
    // do your code
}
```

实质上，编译器会将如上形式的代码转换为：

```
objc_autoreleasePoolPush();
// do your code
objc_autoreleasePoolPop();
```

查看它们在runtime中的定义：

```
void *objc_autoreleasePoolPush(void)
{
    return AutoreleasePoolPage::push();
}

 static inline void *push()
    {
        id *dest;
        if (DebugPoolAllocation) {
            // Each autorelease pool starts on a new pool page.
            dest = autoreleaseNewPage(POOL_BOUNDARY);
        } else {
            dest = autoreleaseFast(POOL_BOUNDARY);
        }
        assert(dest == EMPTY_POOL_PLACEHOLDER || *dest == POOL_BOUNDARY);
        return dest;
    }
```

```
 static inline id *autoreleaseFast(id obj)
    {
        AutoreleasePoolPage *page = hotPage();
        if (page && !page->full()) {
            return page->add(obj);
        } else if (page) {
            return autoreleaseFullPage(obj, page);
        } else {
            return autoreleaseNoPage(obj);
        }
    }
```

可以看到，当`push`到`autorelease`时，最终会调用到`autoreleaseFast`， 在`autoreleaseFast`中，会首先取出当前线程的`hotPage`，根据当前`hotPage`的三种状态：

1. `hot page`存在且未满，调用`page->add(obj)`
2. `hot page`存在但已满， 调用`autoreleaseFullPage(obj, page)`
3. `hot page`不存在，调用 `autoreleaseNoPage(obj)`

关于这三个方法的实现细节，我们在iOS 内存管理有详细的分析。

当需要`pop autorelease pool`时，则会调用`objc_autoreleasePoolPop()`：

```
void objc_autoreleasePoolPop(void *ctxt)
{
    AutoreleasePoolPage::pop(ctxt);
}

static inline void pop(void *token)
    {
        AutoreleasePoolPage *page;
        id *stop;

        if (token == (void*)EMPTY_POOL_PLACEHOLDER) {
            // Popping the top-level placeholder pool.
            if (hotPage()) {
                // Pool was used. Pop its contents normally.
                // Pool pages remain allocated for re-use as usual.
                pop(coldPage()->begin());
            } else {
                // Pool was never used. Clear the placeholder.
                setHotPage(nil);
            }
            return;
        }

        page = pageForPointer(token);
        stop = (id *)token;
        if (*stop != POOL_BOUNDARY) {
```

```
            if (stop == page->begin()  &&  !page->parent) {
                // Start of coldest page may correctly not be POOL_BOUNDARY:
                // 1. top-level pool is popped, leaving the cold page in place
                // 2. an object is autoreleased with no pool
            } else {
                // 这是为了兼容旧的SDK，看来在新的SDK里面，token 可能的取值只有两个:
 (1) POOL_BOUNDARY, (2) page->begin() && !page->parent也就是第一个page
                // Error. For bincompat purposes this is not
                // fatal in executables built with old SDKs.
                return badPop(token);
            }
        }

        if (PrintPoolHiwat) printHiwat();

        page->releaseUntil(stop);  // 对token之前的object，每一个都调用
objc_release方法

        // memory: delete empty children
        if (DebugPoolAllocation  &&  page->empty()) {
            // special case: delete everything during page-per-pool debugging
            AutoreleasePoolPage *parent = page->parent;
            page->kill();
            setHotPage(parent);
        } else if (DebugMissingPools  &&  page->empty()  &&  !page->parent) {
            // special case: delete everything for pop(top)
            // when debugging missing autorelease pools
            page->kill();
            setHotPage(nil);
        }
        else if (page->child) {
            // hysteresis: keep one empty child if page is more than half full
            if (page->lessThanHalfFull()) {
                page->child->kill();
            }
            else if (page->child->child) {
                page->child->child->kill();
            }
        }
    }
```

在`Pop`中，会根据传入的`token`，调用`page->releaseUntil(stop)`方法，对每一个存储于`page`上的`object`调用`objc_release(obj)`方法。

之后，还会根据当前`page`的状态：`page->lessThanHalfFull()`或其他，来决定其`child`的处理方式：

1. 如果当前`page`存储的`object`已经`不满半页`，则将`page`的`child`释放
2. 如果当前`page`存储的`object`仍满半页，则保留一个`空的child`，并且将`空child`之后的所有

child都`释放掉`。

retain count

当我们需要获取对象的引用计数时，在ARC下可以调用如下方法：

```
CFGetRetainCount((__bridge CFTypeRef)(obj))
```

这是CF的方法调用，而在runtime中，我们可以调用NSObject的方法：

```
- (NSUInteger)retainCount OBJC_ARC_UNAVAILABLE;
```

通过注释，可以知道在ARC环境下，该方法是不可用的，但是不影响我们了解它的具体实现。

```
- (NSUInteger)retainCount {
    return ((id)self)->rootRetainCount();
}
```

方法里面讲`self`转为`id`类型，即`objc_object`类型，然后调用`objc_object`的`rootRetainCount()`方法。

```
inline uintptr_t objc_object::rootRetainCount()
{
    //case 1：  如果是tagged pointer，则直接返回this，因为tagged pointer是不需要引用计数的
    if (isTaggedPointer()) return (uintptr_t)this;

    // 将objcet对应的sidetable上锁
    sidetable_lock();
    isa_t bits = LoadExclusive(&isa.bits);
    ClearExclusive(&isa.bits);
    // case 2：  如果采用了优化的isa指针
    if (bits.nonpointer) {
        uintptr_t rc = 1 + bits.extra_rc; // 先读取isa.extra_rc
        if (bits.has_sidetable_rc) { // 如果使用了sideTable来存储retain count，还需要读取sidetable中的数据
            rc += sidetable_getExtraRC_nolock(); // 总引用计数= rc + sidetable
        }
        sidetable_unlock();
        return rc;
    }
    // case 3：如果没采用优化的isa指针，则直接返回sidetable中的值
    sidetable_unlock();
    return sidetable_retainCount();
}
```

获取`retain count`的方法很简单：

- 1. 判断`object`是否使用了`isa`优化
- 2. 如果使用了`isa`优化，先取出`1 + bits.extra_rc`
- 3. 再判断是否需要读取`side talbe（ if (bits.has_sidetable_rc)）`
- 4. 如果需要，则加上`side table`中存储的`retain count`
- 5. 如果没有使用`isa`优化，则直接读取`side table`中的`retain count`，并加`1`，作为引用计数。

还有一种特殊的情况是，如果`object pointer`是`tagged pointer`，则不参与任何操作。

release

当`object`需要引用计数减一时，会调用`release`方法。

```
objc_object::rootRelease()
{
    return rootRelease(true, false);
}

ALWAYS_INLINE bool
objc_object::rootRelease(bool performDealloc, bool handleUnderflow)
{
    if (isTaggedPointer()) return false;

    bool sideTableLocked = false;

    isa_t oldisa;
    isa_t newisa;

 retry:
    do {
        oldisa = LoadExclusive(&isa.bits);
        newisa = oldisa;
        if (slowpath(!newisa.nonpointer)) { // 慢路径 : 如果没有开启isa优化，则到
sidetable中引用计数减一
            ClearExclusive(&isa.bits); // 空方法
            if (sideTableLocked) sidetable_unlock();
            return sidetable_release(performDealloc);
        }
        // don't check newisa.fast_rr; we already called any RR overrides
        uintptr_t carry;
        newisa.bits = subc(newisa.bits, RC_ONE, 0, &carry);  // extra_rc--
        if (slowpath(carry)) { // 如果下溢出，则goto underflow
            // don't ClearExclusive()
            goto underflow;
        }
    } while (slowpath(!StoreReleaseExclusive(&isa.bits,
```

```
                                                oldisa.bits, newisa.bits))); // 修改isa bits(如果不成功，则进入while循环，再试一把，直到成功为止)

    if (slowpath(sideTableLocked)) sidetable_unlock();
    return false;  // 如果没有溢出，则在这里就会返回false（表明引用计数不等于0，没有dealloc)

    // 只有isa.extra_rc  -1 下溢出后，才会进入下面的代码。下溢出有两种情况:
    // 1. borrow from side table . isa.extra_rc 有从side table存储。这是假溢出，只需要将side table中的RC_HALF移回到isa.extra_rc即可。并返回false
    // 2. deallocate。 这种情况是真下溢出。此时isa.extra_rc < 0，且没有newisa.has_sidetable_rc 没有想side table 借位。说明object引用计数==0，(1) 设置newisa.deallocating = true;
    //  (2)触发object 的dealloc方法，(3)并返回true，表明对象deallocation
    //
        // Really deallocate.
        //    if (slowpath(newisa.deallocating)) {
        //        ClearExclusive(&isa.bits);
        //        if (sideTableLocked) sidetable_unlock();
        //        return overrelease_error();
        //        // does not actually return
        //    }
        //    newisa.deallocating = true;
        //    if (!StoreExclusive(&isa.bits, oldisa.bits, newisa.bits)) goto retry;
        //
        //    if (slowpath(sideTableLocked)) sidetable_unlock();
        //
        //    __sync_synchronize();
        //    if (performDealloc) {
        //        ((void(*)(objc_object *, SEL))objc_msgSend)(this, SEL_dealloc);
        //    }
        //    return true;
    //
    //

 underflow:
    // newisa.extra_rc-- underflowed: borrow from side table or deallocate

    // abandon newisa to undo the decrement
    newisa = oldisa;

    if (slowpath(newisa.has_sidetable_rc)) {  // 如果借用了 sideTable 做 rc，走这里
        if (!handleUnderflow) {
            ClearExclusive(&isa.bits);
            return rootRelease_underflow(performDealloc);
        }
```

```
        // Transfer retain count from side table to inline storage.

        if (!sideTableLocked) {
            ClearExclusive(&isa.bits); // ClearExclusive 是一个空函数
            sidetable_lock();
            sideTableLocked = true;
            // Need to start over to avoid a race against
            // the nonpointer -> raw pointer transition.
            goto retry;
        }

        // 如果extra_rc 减1后，其值carryout（小于0），则要处理side table，如果之前有在
side talbe中借位RC_HALF，则把这RC_HALF在拿回来到extrc_rc中，并保留side table剩下的值
        // Try to remove some retain counts from the side table.
        size_t borrowed = sidetable_subExtraRC_nolock(RC_HALF);

        // To avoid races, has_sidetable_rc must remain set
        // even if the side table count is now zero.

        if (borrowed > 0) {
            // Side table retain count decreased.
            // Try to add them to the inline count.
            newisa.extra_rc = borrowed - 1;  // redo the original decrement
too
            bool stored = StoreReleaseExclusive(&isa.bits,
                                                oldisa.bits, newisa.bits);
            if (!stored) {
                // Inline update failed.
                // Try it again right now. This prevents livelock on LL/SC
                // architectures where the side table access itself may have
                // dropped the reservation.
                isa_t oldisa2 = LoadExclusive(&isa.bits);
                isa_t newisa2 = oldisa2;
                if (newisa2.nonpointer) {
                    uintptr_t overflow;
                    newisa2.bits =
                        addc(newisa2.bits, RC_ONE * (borrowed-1), 0,
&overflow);
                    if (!overflow) {
                        stored = StoreReleaseExclusive(&isa.bits,
oldisa2.bits,
                                                       newisa2.bits);
                    }
                }
            }

            if (!stored) {
                // Inline update failed.
```

```
                // Put the retains back in the side table.
                sidetable_addExtraRC_nolock(borrowed); // 如果更新 isa extra_rc
失败，则把side table中的数再放回去 （好尴尬），然后再试一把
                goto retry;
            }

            // Decrement successful after borrowing from side table.
            // This decrement cannot be the deallocating decrement - the side
            // table lock and has_sidetable_rc bit ensure that if everyone
            // else tried to -release while we worked, the last one would
block.
            sidetable_unlock();
            return false;
        }
        else {
            // Side table is empty after all. Fall-through to the dealloc
path.
        }
    }

    // Really deallocate.

    if (slowpath(newisa.deallocating)) {
        ClearExclusive(&isa.bits);
        if (sideTableLocked) sidetable_unlock();
        return overrelease_error();
        // does not actually return
    }
    newisa.deallocating = true;
    if (!StoreExclusive(&isa.bits, oldisa.bits, newisa.bits)) goto retry;

    if (slowpath(sideTableLocked)) sidetable_unlock();

    __sync_synchronize();
    if (performDealloc) {
        ((void(*)(objc_object *, SEL))objc_msgSend)(this, SEL_dealloc);
    }
    return true;
}
```

这里的逻辑主要有两块：

- 1. 如果没有使用`isa.extra_rc`作引用计数，则调用`sidetable_release`，该方法会到`side table`中做计数减一，同时，会`check`计数是否为`0`，如果为`0`，则调用对象的`dealloc`方法。
- 2. 如果使用了`isa.extra_rc`作引用计数，则在`isa.extra_rc`中做引用计数减一,同时需要判断是否下溢出(`carry > 0`)

```
newisa.bits = subc(newisa.bits, RC_ONE, 0, &carry);  // extra_rc--
```

这里要注意处理下溢出的逻辑：

- 1. 首先，下溢出是针对`isa.extra_rc`来说的。也就是启用了`isa`优化引用计数才会走到`underflow:`代码段。
- 2. 造成`isa.extra_rc`下溢出其实有两个原因：`borrow from side table or deallocate`。要注意对这两个下溢出原因的不同处理。

dealloc

当对象引用计数为`0`时，会调用对象的`dealloc`方法，这在上面的`release`方法中，是通过

```
((void(*)(objc_object *, SEL))objc_msgSend)(this, SEL_dealloc);
```

来调用的。

我们来看一下`NSObject`的`dealloc`方法是怎样实现的：

```
- (void)dealloc {
    _objc_rootDealloc(self);
}

void _objc_rootDealloc(id obj)
{
    assert(obj);
    obj->rootDealloc();
}
```

```
inline void objc_object::rootDealloc()
{
    if (isTaggedPointer()) return;  // fixme necessary?

    if (fastpath(isa.nonpointer  &&
                 !isa.weakly_referenced  &&
                 !isa.has_assoc  &&
                 !isa.has_cxx_dtor  &&
                 !isa.has_sidetable_rc))
    {
        // 如果没有weak引用 & 没有关联对象 & 没有c++析构 & 没有side table借位
        // 就直接free
        assert(!sidetable_present());
        free(this);
    }
    else {
        object_dispose((id)this);
```

```
    }
}
```

```
object_dispose(id obj)
{
    if (!obj) return nil;

    objc_destructInstance(obj);    // step 1. 先调用runtime的
objc_destructInstance
    free(obj);  // step 2. free 掉这个obj

    return nil;
}
```

```
/***********************************************************************
* objc_destructInstance
* Destroys an instance without freeing memory.
* Calls C++ destructors.
* Calls ARC ivar cleanup.
* Removes associative references.
* Returns `obj`. Does nothing if `obj` is nil.
**********************************************************************/
void *objc_destructInstance(id obj)
{
    if (obj) {
        // Read all of the flags at once for performance.
        bool cxx = obj->hasCxxDtor();
        bool assoc = obj->hasAssociatedObjects();

        // This order is important.
        if (cxx) object_cxxDestruct(obj); // 调用C++析构函数
        if (assoc) _object_remove_assocations(obj); // 移除所有的关联对象，并将其自
身从Association Manager的map中移除
        obj->clearDeallocating(); // 清理相关的引用
    }

    return obj;
}
```

在对象`dealloc`的过程中，会根据当前对象`isa_t`的各个标志位，来做对应的清理工作，清理完毕后，会调用`free(obj)`来释放内存。

清理工作会在`objc_destructInstance`方法中进行，主要包括：

- 1. 如果有`C++析构`函数，调用`C++析构`
- 2. 如果有关联对象，调用`_object_remove_assocations(obj)`将关联在该对象的对象移除
- 3. 调用`obj->clearDeallocating()`方法，主要是 （1）将`weak`引用置为nil，并在`weak_table_t`中删除对象节点。 （2）如果有`side table`计数借位，则`side table`

中对应的节点移除

总结

本篇文章从`[[NSObject alloc] init]`方法说起，讲解了`alloc`，`init`背后的实现逻辑，以及OC中的所有权修饰符`__strong`，`__weak`。并讲述了`autoreleasepool`的背后实现。同时，分析了`retain`和`release`引用计数相关函数。

最终，我们分析了对象`dealloc`所做的清理工作。

iOS开发者都知道，当一个对象被释放时，所有对这个对象弱引用的指针都会释放并置为nil，那么系统是如何存储这些弱引用对象的呢？又是如何在一个对象释放时，将这些指向即将释放对象的弱引用的指针置为nil的呢？下面我们通过分析`SideTable`的结构来进一步了解内存管理的弱引用存储细节。

结构

在runtime中，有四个数据结构非常重要，分别是`SideTables`，`SideTable`，`weak_table_t`和`weak_entry_t`。它们和对象的引用计数，以及weak引用相关。

SideTables

下面我们看下SideTables的结构：

```
static StripedMap<SideTable>& SideTables() {
    return *reinterpret_cast<StripedMap<SideTable>*>(SideTableBuf);
}
```

`reinterpret_cast`，是C++里的强制类型转换符,我们看下`SideTableBuf`的定义。上面代码，我们看到StripedMap实际上返回的是一个SideTableBuf对象，那么我们来看下SideTableBuf对象：

```
//alignas 字节对齐
// SideTableBuf 静态全局变量
// sizeof(StripedMap<SideTable>) = 4096
//alignas (StripedMap<SideTable>) 是字节对齐的意思，表示让数组中每一个元素的起始位置对齐到4096的倍数
// 因此下面这句话可以翻译为 static uint8_t SideTableBuf[4096]
alignas(StripedMap<SideTable>) static uint8_t
    SideTableBuf[sizeof(StripedMap<SideTable>)];
```

`SideTableBuf`是一个外部不可见的静态内存区块，存储`StripedMap<SideTable>`对象。它是内存管理的基础。

我们接下来在看下`StripedMap`的结构

```
enum { CacheLineSize = 64 };
// StripedMap<T> 是一个模板类，根据传递的实际参数决定其中 array 成员存储的元素类型
```

```
// 能通过对象的地址，运算出 Hash 值，通过该 hash 值找到对应的 value
template<typename T>
class StripedMap {
#if TARGET_OS_IPHONE && !TARGET_OS_SIMULATOR
    enum { StripeCount = 8 };
#else
    enum { StripeCount = 64 };
#endif
    // PaddedT 为一个结构体
    struct PaddedT {
        T value alignas(CacheLineSize);
    };

    // array 中存放着8个sidetable
    PaddedT array[StripeCount];
    //取得p的哈希值，p就是实例对象的地址
    static unsigned int indexForPointer(const void *p) {
        uintptr_t addr = reinterpret_cast<uintptr_t>(p);
        // 这里根据对象的地址经过左移和异或操作 最终结果 模 8 得到一个0-7的值
        // 即对应该地址对应array中下标的sidetable中
        return ((addr >> 4) ^ (addr >> 9)) % StripeCount;
    }

 public:
    // 重写了[]方法 即通过下标获取数组中对应下标的值
    // array[index] = array[indexForPointer(p)].value
    T& operator[] (const void *p) {
        return array[indexForPointer(p)].value;
    }

    const T& operator[] (const void *p) const {
        return const_cast<StripedMap<T>>(this)[p];
    }
};
```

`StripedMap` 是一个以`void *`为hash key， T为vaule的hash 表。`StripedMap`的所有T类型数据都被封装到array中。

综上我们得出`SideTables`的机构实际是下图所示:

[图片上传失败...(image-e2c604-1636097462956)]

SideTable

下面来看下`sideTable`的结构

```
struct SideTable {
    // 保证原子操作的自旋锁
    spinlock_t slock;
```

```
    // 引用计数的 hash 表
    RefcountMap refcnts;
    // weak 引用全局 hash 表
    weak_table_t weak_table;

    SideTable() {
        memset(&weak_table, 0, sizeof(weak_table));
    }

    ~SideTable() {
        _objc_fatal("Do not delete SideTable.");
    }
};
```

上面是我们简化后的`SideTable`结构体，包含了:

- 保证原子属性的自旋锁`spinlock_t`
- 记录引用计数值的`RefcountMap`
- 用于存储对象弱引用的哈希表 `weak_table_t`

自旋锁(slock)我们这里就不做过多介绍了,我们先来看下RefcountMap，看下`RefcountMap`结构

```
// RefcountMap 是一个模板类
// key,DisguisedPtr<objc_object>类型
// value，size_t类型
// 是否清除为vlaue==0的数据，true
typedef objc::DenseMap<DisguisedPtr<objc_object>,size_t,true> RefcountMap;
```

DenseMap是llvm库中的类，是一个简单的二次探测哈希表，擅长支持小的键和值。`RefcountMap`是一个hash map，其key是obj的`DisguisedPtr<objc_object>`，而value，则是obj对象的`引用计数`,同时，这个map还有个加强版功能，当引用计数为0时，会自动将对象数据清除。

上面我们知道了，refcnts是用来存放引用计数的，那么我们如何获取一个对象的引用计数呢?

```
// 获取一个对象的retainCount
inline uintptr_t
objc_object::rootRetainCount()
{
    //优化指针 直接返回
    if (isTaggedPointer()) return (uintptr_t)this;
    //没优化则 到SideTable 读取
    sidetable_lock();
    //isa指针
    isa_t bits = LoadExclusive(&isa.bits);
    ClearExclusive(&isa.bits);//啥都没做
    if (bits.nonpointer) {//优化过 isa 指针
        uintptr_t rc = 1 + bits.extra_rc;//计数数量
```

```
        if (bits.has_sidetable_rc) {
        //bits.has_sidetable_rc标志位为1 表明有存放在sidetable中的引用计数
            //读取table的值 相加
            rc += sidetable_getExtraRC_nolock();
        }
        //解锁
        sidetable_unlock();
        return rc;
    }

    sidetable_unlock();
    //: 如果没采用优化的isa指针，则直接返回sidetable中的值
    return sidetable_retainCount();
}
```

从上面的代码我们可以得出:`retainCount = isa.extra_rc + sidetable_getExtraRC_nolock`,即引用计数=isa指针中存储的引用计数+sidetable中存储的引用计数

那么`sidetable_getExtraRC_nolock`是如何从sideTable中获取`retainCount`的呢？ 下面我们来看下这个方法的实现。

```
size_t
objc_object::sidetable_getExtraRC_nolock()
{
    //
    assert(isa.nonpointer);
    //key是 this，存储了每个对象的table
    SideTable& table = SideTables()[this];
    //找到 it 否则返回0
    RefcountMap::iterator it = table.refcnts.find(this);
    // 这里返回的it是RefcountMap类型 it == table.refcnts.end()
    // 表示在sidetable中没有找到this对应的引用计数则直接返回0
    if (it == table.refcnts.end()) return 0;
    // RefcountMap 结构的second值为引用计数值
    // DenseMap<DisguisedPtr<objc_object>,size_t,true> RefcountMap;
    else return it->second >> SIDE_TABLE_RC_SHIFT;
}
```

了解了SideTable的RefcountMap，下面我们接着看另外一个属性weak_table

weak_table

我们都知道weak_table是对象弱引用map，它记录了所有弱引用对象的集合。

我们先来看下`weak_table_t`的定义：

```
// 全局的弱引用表
struct weak_table_t {
    // hash数组，用来存储弱引用对象的相关信息weak_entry_t
    weak_entry_t *weak_entries;
    // hash数组中的元素个数
    size_t    num_entries;
    // hash数组长度-1，会参与hash计算。
    //（注意，这里是hash数组的长度，而不是元素个数。比如，数组长度可能是64，而元素个数仅存了
2个）
    uintptr_t mask;
    // 最大哈希偏移值
    uintptr_t max_hash_displacement;
};
```

`weak_entries`实质上是一个hash数组，数组中存储`weak_entry_t`类型的元素。weak_entry_t的定义如下

```
/**
 * The internal structure stored in the weak references table.
 * It maintains and stores
 * a hash set of weak references pointing to an object.
 * If out_of_line_ness != REFERRERS_OUT_OF_LINE then the set
 * is instead a small inline array.
 */
 //inline_referrers数组中可以存放元素的最大个数 如果超过了这个个数就会使用referrers 存放
#define WEAK_INLINE_COUNT 4
// out_of_line_ness field overlaps with the low two bits of
inline_referrers[1].
// inline_referrers[1] is a DisguisedPtr of a pointer-aligned address.
// The low two bits of a pointer-aligned DisguisedPtr will always be 0b00
// (disguised nil or 0x80..00) or 0b11 (any other address).
// Therefore out_of_line_ness == 0b10 is used to mark the out-of-line state.
// DisguisedPtr方法返回的hash值得最低2个字节应该是0b00或0b11，因此可以用
out_of_line_ness
// == 0b10来表明当前是否在使用数组或动态数组来保存引用该对象的列表。
#define REFERRERS_OUT_OF_LINE 2
struct weak_entry_t {
    // 被弱引用的对象
    DisguisedPtr<objc_object> referent;
    // 联合结构 两种结构共同占用一块内存空间 两种结构互斥
    union {
        // 弱引用 被弱引用对象的列表
        struct {
            // 弱引用该对象的对象列表的动态数组
            weak_referrer_t *referrers;
            // 是否使用动态数组标记位
            uintptr_t        out_of_line_ness : 2;
```

```
            // 动态数组中元素的个数
            uintptr_t        num_refs : PTR_MINUS_2;
            // 用于hash确定动态数组index，值实际上是动态数组空间长度-1（它和num_refs不
一样，
            // 这里是记录的是数组中位置的个数，而不是数组中实际存储的元素个数）。
            uintptr_t        mask;
            // 最大哈希偏移值
            uintptr_t        max_hash_displacement;
        };
        struct {
            // inline_referrers 数组 当不使用动态数组时使用 最大个数为4
            weak_referrer_t  inline_referrers[WEAK_INLINE_COUNT];
        };
    };
}
```

从上面的介绍我们可以总结`SideTables`和`SideTable`以及`weak_table_t`在层级上的关系图如下:

上图是从数据结构的角度来看弱引用的保存,下面我们来看下从垂直方向来看

从上面的总结中我们可以看到，弱引用的存储实际上一个三级的哈希表，通过一层层的索引找到或者存储对应的弱引用。那当向`weak_table_t`中插入或查找某个元素时是如何操作的呢？算法是什么样的呢?

weak_entry_for_referent

```
/ 在weak_table中查找所有弱引用referent的对象
static weak_entry_t *
weak_entry_for_referent(weak_table_t *weak_table, objc_object *referent)
{
    assert(referent);
    //获取这个weak_table_t中所有的弱引用对象
    weak_entry_t *weak_entries = weak_table->weak_entries;

    if (!weak_entries) return nil;
    //hash_pointer 哈希函数 传入的是 objc_object *key
    // weak_table->mask = weaktable的容量-1
    size_t begin = hash_pointer(referent) & weak_table->mask;
    size_t index = begin;
    // 哈希冲突次数
    size_t hash_displacement = 0;
    // 判断根据index获取到的弱引用对象数组中对应的weak_entry_t的弱引用对象是否为
    // 外部传入的对象
    while (weak_table->weak_entries[index].referent != referent) {
        // 开放地址法解决哈希冲突
```

```
        // & weak_table->mask 是为了在下一个地址仍然没有找到外部传入对象时回到第一个对比的位置
        index = (index+1) & weak_table->mask;
        if (index == begin)
            // 对比了所有数据 仍没有找到 直接报错
            bad_weak_table(weak_table->weak_entries);
        // 哈希冲突次数++
        hash_displacement++;
        // 最大哈希偏移值 表示已经遍历了数组中所有的元素
        // 没有找到那么直接返回nil
        if (hash_displacement > weak_table->max_hash_displacement) {
            return nil;
        }
    }
    // 直接返回被弱引用的对象
    return &weak_table->weak_entries[index];
}
```

上面就是根据对象地址获取所有弱引用该对象的的数组，基本逻辑都比较清晰，我们在遍历`weak_table->weak_entries`中的时候发现判断是否遍历完一遍的时候使用的方法

```
index = (index+1) & weak_table->mask;
```

假设当前数组长度8，下标分别是0-7，上面`weak_table->mask` = 7 = 0111。

| 下标 | 计算后结果 | | --- | --- | | index = 0 | index & mask = 0000 & 0111 = 0000 = 0 | | index = 1 | index & mask = 0001 & 0111 = 0001 = 1 | | index = 2 | index & mask = 0010 & 0111 = 0010 = 2 | | index = 3 | index & mask = 0011 & 0111 = 0011 = 3 | | index = 4 | index & mask = 0100 & 0111 = 0100 = 4 | | index = 5 | index & mask = 0101 & 0111 = 0101 = 5 | | index = 6 | index & mask = 0110 & 0111 = 0110 = 6 | | index = 7 | index & mask = 0111 & 0111 = 0111 = 7 | | index = 8 | index & mask = 1000 & 0111 = 0000 = 0 |

看完上面的计算相信大家都明白了这么做的真是意图了：

```
if (index == begin)
```

可以理解为：数组遍历完成，已经和数组中所有的元素做了对比。

随着某个对象被越来越多的对象弱引用，那么这个存放弱引用该对象的所有对象的数组也会越来越大。

hash表自动扩容

```
//weak_table_t扩容
// 参数 weak_table 要扩容的table new_size 目标大小
static void weak_resize(weak_table_t *weak_table, size_t new_size)
{
    //weak_table的容量
    size_t old_size = TABLE_SIZE(weak_table);
```

```
    // 取出weak_table中存放的所有实体
    weak_entry_t *old_entries = weak_table->weak_entries;
    // 新创建一个weak_entry_t类型的数组
    // 数组的大小是new_size * sizeof(weak_entry_t)
    weak_entry_t *new_entries = (weak_entry_t *)
        calloc(new_size, sizeof(weak_entry_t));

    // 重置weak_table的mask的值
    weak_table->mask = new_size - 1;
    // 将weak_table->weak_entries指向新创建的内存区域 注意 此时weak_table中没有任何数
据
    weak_table->weak_entries = new_entries;
    // 最大哈希偏移值重置为0
    weak_table->max_hash_displacement = 0;
    //weak_table 中存储实体个数为0
    weak_table->num_entries = 0;  // restored by weak_entry_insert below

    // 旧数据的搬迁
    if (old_entries) {
        weak_entry_t *entry;
        //old_entries看做数组中第一个元素的地址 由于数组是连续的存储空间 那么
old_entries + old_size = 数组最后一个元素的地址
        weak_entry_t *end = old_entries + old_size;
        // 遍历这些旧数据
        for (entry = old_entries; entry < end; entry++) {
            //weak_entry_t的referent(referent是指被弱引用的对象)
            if (entry->referent) {
                // 将旧数据搬移到新的结构中
                weak_entry_insert(weak_table, entry);
            }
        }
        // 释放所有的旧数据
        free(old_entries);
    }
}
```

从上面的代码中我们可以看到，哈希表的扩容主要分为下面几个步骤:

- 创建一个局部变量保存当前哈希表中保存的所有弱引用实体
- 新建一个容量是旧哈希表大小2倍的哈希表，同时重置`num_entries`、`max_hash_displacement`、`weak_entries`、`mask`
- 遍历之前保存的旧的数据 将数据按照顺序依次重新插入的新建的哈希表中
- 释放旧数据

我们看到将旧数据插入新数据的主要方法是`weak_entry_insert`，下面我们来仔细介绍下它:

weak_entry_insert

```
// 向指定的weak_table_t中插入某个对象
// weak_table_t 目标 table
// new_entry 被弱引用的对象
static void weak_entry_insert(weak_table_t *weak_table, weak_entry_t
*new_entry)
{
    // 取出weak_table中所有弱引用的对象
    weak_entry_t *weak_entries = weak_table->weak_entries;
    assert(weak_entries != nil);

    // 根据new_entry中被弱引用对象地址通过哈希算法 算出 弱引用new_entry->referent的对象
存放的index
    size_t begin = hash_pointer(new_entry->referent) & (weak_table->mask);
    size_t index = begin;
    size_t hash_displacement = 0;
    // weak_entries[index].referent 如果不为空 表示已经有
    while (weak_entries[index].referent != nil) {
        // 计算下一个要遍历的index
        index = (index+1) & weak_table->mask;
        // 遍历了所有元素发现weak_entries[index].referent 都不为nil
        if (index == begin)
            // 直接报错
            bad_weak_table(weak_entries);
        // 哈希冲突次数++
        hash_displacement++;
    }

    // 如果走到这里 表明index位置的元素referent=nil
    // 直接插入
    weak_entries[index] = *new_entry;
    // 实体个数++
    weak_table->num_entries++;

    // 最大哈希偏移值大于之前的记录
    if (hash_displacement > weak_table->max_hash_displacement) {
        // 更新最大哈希偏移值
        weak_table->max_hash_displacement = hash_displacement;
    }
}
```

插入操作也很简单，主要分为下面几个步骤:

- 取出哈希表中所有弱引用对象的数据
- 遍历第一步取出的所有数据，找到第一个空位置
- 将要插入的实体插入到这个位置，同时更新当前`weak_table`中弱引用实体个数
- 重置`weak_table`中最大哈希冲突次数的值

插入的主要逻辑实际上并不复杂，但是我们发现最后一步

```
// 如果本次哈希偏移值大于之前记录的最大偏移值 则更新
if (hash_displacement > weak_table->max_hash_displacement) {
    // 修改最大哈希偏移值
    weak_table->max_hash_displacement = hash_displacement;
}
```

通过上面的代码我们发现，假设`weak_table`的`weak_entries`最大容量为8,当前存放了3个被弱引用的对象且分别存放在下标为[0,1,2]中，同时要插入的对象`new_entry`不再`weak_entries`中，那么经过while循环，`hash_displacement = 3`。实际上如果在没有哈希冲突的情况下我们通过`hash_pointer`得到的index就应该是用来存放`new_entry`的，但是因为存在哈希冲突，所以后移了3位后才找到合适的位置来存放`new_entry`，因此`hash_displacement`也被理解为，本应存放的位置距离实际存放位置的差值。

综上，我们分析了哈希表中获取所有弱引用某个对象的对象数组，哈希表扩容方法，以及如何在哈希表中插入一个弱引用对象。

下面我们来看下新增和释放弱引用对象的方法

objc_initWeak

```
// 初始化一个weak 弱引用
// 参数location weak指针的地址  newObj weak指针指向的对象
id
objc_initWeak(id *location, id newObj)
{
    // 如果弱引用对象为空
    if (!newObj) {
        *location = nil;
        return nil;
    }
    // 调用storeWeak
    return storeWeak<DontHaveOld, DoHaveNew, DoCrashIfDeallocating>
        (location, (objc_object*)newObj);
}
```

- id *location： **weak指针的地址，即weak指针取地址： &weakObj 。它是一个指针的地址。之所以要存储指针的地址，是因为最后我们要讲**weak指针指向的内容置为nil，如果仅存储指针的话，是不能够完成这个功能的。
- id newObj： 所引用的对象。即例子中的obj 。

从上面我们看出`objc_initWeak`实际上是调用了`storeWeak`方法，且方法调用我们可以翻译为

```
storeWeak<false, true, true>
        (location, (objc_object*)newObj)
```

storeWeak

```
enum CrashIfDeallocating {
    DontCrashIfDeallocating = false, DoCrashIfDeallocating = true
};
template <HaveOld haveOld, HaveNew haveNew,
          CrashIfDeallocating crashIfDeallocating>
// HaveOld= true weak ptr之前是否已经指向了一个弱引用
// haveNew = true weak ptr是否需要指向一个新引用
// crashIfDeallocating = true 如果被弱引用的对象正在析构，此时再弱引用该对象，是否应该
crash
// crashIfDeallocating = false 将存储的数据置为nil
// *location 代表weak 指针的地址
// newObj 被weak引用的对象。
static id
storeWeak(id *location, objc_object *newObj)
{
    assert(haveOld  ||  haveNew);
    // 如果没有新值赋值 判断newObj 是否为空 否则断言
    if (!haveNew)
        assert(newObj == nil);

    Class previouslyInitializedClass = nil;
    id oldObj;

    SideTable *oldTable;
    SideTable *newTable;

 retry:
    // 如果weak ptr之前弱引用过一个obj，则将这个obj所对应的SideTable取出，赋值给
oldTable
    if (haveOld) {
        // 根据传入的地址获取到旧的值
        oldObj = *location;
        // 根据旧值的地址获取到旧值所存在的SideTable
        oldTable = &SideTables()[oldObj];
    } else {
        // 如果weak ptr之前没有弱引用过一个obj，则oldTable = nil
        oldTable = nil;
    }
    // 是否有新值 如果有
    if (haveNew) {
        // 如果weak ptr要weak引用一个新的obj，则将该obj对应的SideTable取出，赋值给
newTable
        newTable = &SideTables()[newObj];
    } else {
        // 如果weak ptr不需要引用一个新obj，则newTable = nil
        newTable = nil;
    }

    // 加锁管理一对 side tables，防止多线程中竞争冲突
```

```
    SideTable::lockTwo<haveOld, haveNew>(oldTable, newTable);

    // location 应该与 oldObj 保持一致，如果不同，说明当前的 location 已经处理过
oldObj 可是又被其他线程所修改
    if (haveOld  &&  *location != oldObj) {
        // 解锁后重试
        SideTable::unlockTwo<haveOld, haveNew>(oldTable, newTable);
        goto retry;
    }

    // 保证弱引用对象的 isa 都被初始化，防止弱引用和 +initialize 之间发生死锁，
    // 也就是避免 +initialize 中调用了 storeWeak 方法，而在 storeWeak 方法中
weak_register_no_lock
    // 方法中用到对象的 isa 还没有初始化完成的情况
    if (haveNew  &&  newObj) {
        Class cls = newObj->getIsa();
        // 如果cls还没有初始化，先初始化，再尝试设置weak
        if (cls != previouslyInitializedClass  &&
            !((objc_class *)cls)->isInitialized())
        {
            SideTable::unlockTwo<haveOld, haveNew>(oldTable, newTable);
            // 发送 +initialize 消息到未初始化的类
            _class_initialize(_class_getNonMetaClass(cls, (id)newObj));

            // 如果该类还没有初始化完成，例如在 +initialize 中调用了 storeWeak 方法，
            // 也就是会进入这里面，进而设置  previouslyInitializedClass  以在重试时识
别它
            // 这里记录一下previouslyInitializedClass， 防止改if分支再次进入
            previouslyInitializedClass = cls;
            // 重新获取一遍newObj，这时的newObj应该已经初始化过了
            goto retry;
        }
    }

    // 如果weak_ptr之前弱引用过别的对象oldObj，则调用weak_unregister_no_lock，在
oldObj的weak_entry_t中移除该weak_ptr地址
    if (haveOld) {
        weak_unregister_no_lock(&oldTable->weak_table, oldObj, location);
    }

    // 如果weak_ptr需要弱引用新的对象newObj
    if (haveNew) {
        // (1) 调用weak_register_no_lock方法，将weak ptr的地址记录到newObj对应的
weak_entry_t中
        newObj = (objc_object *)
            weak_register_no_lock(&newTable->weak_table, (id)newObj, location,
                                  crashIfDeallocating);

        // (2) 更新newObj的isa的weakly_referenced bit标志位
```

```
        if (newObj  &&  !newObj->isTaggedPointer()) {
            newObj->setWeaklyReferenced_nolock();
        }

        //  (3) *location 赋值，也就是将weak ptr直接指向了newObj。可以看到，这里并没有
将newObj的引用计数+1
        // 将weak ptr指向object
        *location = (id)newObj;
    }
    else {
        // No new value. The storage is not changed.
    }
    // 解锁，其他线程可以访问oldTable，newTable了
    SideTable::unlockTwo<haveOld, haveNew>(oldTable, newTable);
    // 返回newObj，此时的newObj与刚传入时相比，weakly-referenced bit位置1
    return (id)newObj;
}
```

storeWeak方法有点长，这也是weak引用的核心实现部分。其实核心也就实现了两个功能：

将weak指针的地址`location`存入到obj对应的weak_entry_t的数组（链表）中，用于在obj析构时，通过该数组（链表）找到所有其weak指针引用，并将指针指向的地址（`location`）置为nil。

如果启用了isa优化，则将obj的isa_t的`weakly_referenced`位置1。置位1的作用主要是为了标记obj被weak引用了，当dealloc时，runtime会根据`weakly_referenced`标志位来判断是否需要查找obj对应的weak_entry_t，并将引用置为nil。

上面的方法中，我们看到插入新值的方法为`weak_register_no_lock`,清除旧值的方法为`weak_unregister_no_lock`,下面我们来看下这两个方法：

weak_register_no_lock

```
/ 添加对某个对象的新的弱引用指针
// weak_table 目标被弱引用对象所存储的表
// referent_id 被所引用的对象
// referrer_id 要被添加的弱引用指针
// crashIfDeallocating 如果对象正在被释放时是否崩溃
id
weak_register_no_lock(weak_table_t *weak_table, id referent_id,
                      id *referrer_id, bool crashIfDeallocating)
{
    // 被弱引用的对象
    objc_object *referent = (objc_object *)referent_id;
    // 要添加的指向弱引用指针的对象
    objc_object **referrer = (objc_object **)referrer_id;

    // 如果referent为nil 或 referent 采用了TaggedPointer计数方式，直接返回，不做任何操
作
    if (!referent  ||  referent->isTaggedPointer()) return referent_id;
```

```
    // 确保被引用的对象可用（没有在析构，同时应该支持weak引用）
    bool deallocating;
    // referent 是否有自定义的释放方法
    if (!referent->ISA()->hasCustomRR()) {
        deallocating = referent->rootIsDeallocating();
    }
    else {
        // referent是否支持weak引用
        BOOL (*allowsWeakReference)(objc_object *, SEL) =
            (BOOL(*)(objc_object *, SEL))
            object_getMethodImplementation((id)referent,
                                           SEL_allowsWeakReference);
        // 如果referent不能够被weak引用，则直接返回nil
        if ((IMP)allowsWeakReference == _objc_msgForward) {
            return nil;
        }
        // 调用referent的SEL_allowsWeakReference方法来判断是否正在被释放
        deallocating =
            ! (*allowsWeakReference)(referent, SEL_allowsWeakReference);
    }

    // 正在析构的对象，不能够被弱引用
    if (deallocating) {
        // 判断是否需要崩溃 如果需要则崩溃
        if (crashIfDeallocating) {
            _objc_fatal("Cannot form weak reference to instance (%p) of "
                        "class %s. It is possible that this object was "
                        "over-released, or is in the process of
deallocation.",
                        (void*)referent, object_getClassName((id)referent));
        } else {
            return nil;
        }
    }

    // 对象没有被正在释放
    weak_entry_t *entry;
    // 在 weak_table中找到referent对应的weak_entry,并将referrer加入到weak_entry中
    // 如果能找到weak_entry,则讲referrer插入到weak_entry中
    if ((entry = weak_entry_for_referent(weak_table, referent))) {
        // 将referrer插入到weak_entry_t的引用数组中
        append_referrer(entry, referrer);
    }
    else {
        // 创建一个新的weak_entry_t ，并将referrer插入到weak_entry_t的引用数组中
        weak_entry_t new_entry(referent, referrer);
        // weak_table的weak_entry_t 数组是否需要动态增长，若需要，则会扩容一倍
        weak_grow_maybe(weak_table);
```

```
        // 将weak_entry_t插入到weak_table中
        weak_entry_insert(weak_table, &new_entry);
    }

    // Do not set *referrer. objc_storeWeak() requires that the
    // value not change.

    return referent_id;
}
```

上面方法主要功能是：添加对某个对象的新的弱引用指针

- 过滤掉`isTaggedPointer`和弱引用对象正在被释放这两种情况后(这里需要判断是否有自定义的释放方法)，然后根据`crashIfDeallocating`参数确定是崩溃还是返回nil
- 如果对象没有正在被释放，那么从`weak_table`中取出指向`referent`的弱引用指针实体，如果`weak_table`中存在指向`referent`的指针数组那么在这个数组中添加要新增的指针
- 如果`weak_table`没有找到指向`referent`的弱指针数组，那么新建一个`weak_entry_t`对象，将这个对象拆入到`weak_table`中(需要判断weak_table是否需要扩容)

下面我们来看下具体的插入方法：

append_referrer追加

```
// 在entry对象的弱引用数组中追加一个新的弱引用指针new_referrer
// entry 被弱引用的对象
// new_referrer 弱引用entry的指针
static void append_referrer(weak_entry_t *entry, objc_object **new_referrer)
{
    // 如果entry中弱引用指针没有超过了4个 表示弱引用指针存放在inline_referrers中
    // weak_entry 尚未使用动态数组
    if (! entry->out_of_line()) {
        // 遍历inline_referrers数组找到第一个为空的位置 将目标指针插入 尾部追加
        for (size_t i = 0; i < WEAK_INLINE_COUNT; i++) {
            if (entry->inline_referrers[i] == nil) {
                entry->inline_referrers[i] = new_referrer;
                return;
            }
        }

        // 如果entry中弱引用指针==4个
        // 新创建一个weak_referrer_t数组 大小为4(WEAK_INLINE_COUNT)
        // 如果inline_referrers的位置已经存满了，则要转型为referrers，做动态数组。
        weak_referrer_t *new_referrers = (weak_referrer_t *)
            calloc(WEAK_INLINE_COUNT, sizeof(weak_referrer_t));
        // 遍历inline_referrers 将数据放在新创建的临时数组中
        for (size_t i = 0; i < WEAK_INLINE_COUNT; i++) {
            new_referrers[i] = entry->inline_referrers[I];
        }
```

```
        // 弱引用指针的存储改为存放到entry->referrers(entry->inline_referrers ->
entry->referrers)
        entry->referrers = new_referrers;
        // 更新弱引用个数
        entry->num_refs = WEAK_INLINE_COUNT;
        //更新是否使用动态数组标记位
        entry->out_of_line_ness = REFERRERS_OUT_OF_LINE;
        // 更新mask和最大哈希偏移值
        entry->mask = WEAK_INLINE_COUNT-1;
        entry->max_hash_displacement = 0;
    }

    assert(entry->out_of_line());

    // 如果只想entry的弱引用个数大于4
    // 弱引用个数是否已超过数组容量的3/4
    if (entry->num_refs >= TABLE_SIZE(entry) * 3/4) {
        // 如果已超过 那么先扩容在插入
        return grow_refs_and_insert(entry, new_referrer);
    }

    // 如果不需要扩容，直接插入到weak_entry中
    // 注意，weak_entry是一个哈希表，key：w_hash_pointer(new_referrer) value:
new_referrer
    size_t begin = w_hash_pointer(new_referrer) & (entry->mask);
    size_t index = begin;
    size_t hash_displacement = 0;
    // 由低到高遍历entry->referrers 找到第一个空位置
    while (entry->referrers[index] != nil) {
        hash_displacement++;
        index = (index+1) & entry->mask;
        // 如果遍历了所有元素后都没有找到 那么报错
        if (index == begin)
            bad_weak_table(entry);
    }
    // 更新最大哈希偏移值
    if (hash_displacement > entry->max_hash_displacement) {
        entry->max_hash_displacement = hash_displacement;
    }
    // 将new_referrer插入到数组的第index个位置
    weak_referrer_t &ref = entry->referrers[index];
    ref = new_referrer;
    // 弱引用计个数+1
    entry->num_refs++;
}
```

插入的过程主要分下面三种情况:

- 如果`inline_referrers`没有存储满，直接存储到`inline_referrers`中

- 如果`inline_referrers`个数是4个了，在插入，就需要将`inline_referrers`拷贝到`referrers`，然后进入第三步。
- 如果`inline_referrers`存储满了，判断是否需要扩容，然后将数据存储到`referrers`中。

下面我们来看下扩容的方法:

grow_refs_and_insert

```
// entry 中存放弱引用指针数组 扩容
// weak_entry_t 要扩容的对象
// new_referrer 要插入的指向entry->referent弱引用指针
__attribute__((noinline, used))
static void grow_refs_and_insert(weak_entry_t *entry,
                                 objc_object **new_referrer)
{
    assert(entry->out_of_line());
    // 获取entry当前的大小
    size_t old_size = TABLE_SIZE(entry);
    // 新的大小为旧的大小的2倍
    size_t new_size = old_size ? old_size * 2 : 8;

    // 获取weak_entry_t中存储的弱引用指针个数
    size_t num_refs = entry->num_refs;
    //获取entry中旧的引用数组
    weak_referrer_t *old_refs = entry->referrers;
    // 更新entry->mask 这里是为了后续申请内存空间使用
    entry->mask = new_size - 1;

    // 创建一个新的entry->referrers数组
    // #define TABLE_SIZE(entry) (entry->mask ? entry->mask + 1 : 0)
    // TABLE_SIZE 获取的数组大小是 mask+1 = new_size
    entry->referrers = (weak_referrer_t *)
        calloc(TABLE_SIZE(entry), sizeof(weak_referrer_t));
    // 重置num_refs和max_hash_displacement
    entry->num_refs = 0;
    entry->max_hash_displacement = 0;

    // 将old_refs中的数据重新插入到新创建entry->referrers中
    for (size_t i = 0; i < old_size && num_refs > 0; i++) {
        if (old_refs[i] != nil) {
            append_referrer(entry, old_refs[I]);
            num_refs--;
        }
    }
    // 将new_referrer插入到扩容后的entry中
    append_referrer(entry, new_referrer);
    if (old_refs) free(old_refs);
}
```

看完了新增弱引用指针的操作，接下来我们看下如何删除弱引用指针即`weak_unregister_no_lock`

weak_unregister_no_lock

```
// 将 weak ptr地址 从obj的weak_entry_t中移除
// 参数weak_table 全局弱引用表
// referent_id 弱引用所指向的对象
// referrer_id 弱引用指针地址
void
weak_unregister_no_lock(weak_table_t *weak_table, id referent_id,
                        id *referrer_id)
{
    // 被弱引用的对象
    objc_object *referent = (objc_object *)referent_id;
    // 指向被弱引用对象的指针的地址
    objc_object **referrer = (objc_object **)referrer_id;

    weak_entry_t *entry;

    if (!referent) return;

    // 找到weak_table中指向被弱引用对象的所有指针 类型为 weak_entry_t
    if ((entry = weak_entry_for_referent(weak_table, referent))) {
        // 从数组中删除当前这个弱引用指针
        remove_referrer(entry, referrer);
        bool empty = true;
        // 弱引用referent对象的弱引用指针是否为空
        if (entry->out_of_line()  &&  entry->num_refs != 0) {
            empty = false;
        }
        else {
            // 如果referrer数组中为空 那么判断inline_referrers中是否为空 如果为空
empty=true
            for (size_t i = 0; i < WEAK_INLINE_COUNT; i++) {
                if (entry->inline_referrers[i]) {
                    empty = false;
                    break;
                }
            }
        }

        // 如果为空 则证明没有其他指针指向这个被所引用的对象
        if (empty) {
            // 将这个实体从weak_table中移除
            weak_entry_remove(weak_table, entry);
        }
    }

    // Do not set *referrer = nil. objc_storeWeak() requires that the
```

```
    // value not change.
}
```

`weak_unregister_no_lock`的实现逻辑比较简单,其实主要的操作为:

- 首先，它会在`weak_table`中找出`referent`对应的`weak_entry_t`
- 在`weak_entry_t`中移除`referrer`
- 移除元素后，判断此时`weak_entry_t`中是否还有元素 `（empty==true？）`
- 如果此时`weak_entry_t`已经没有元素了，则需要将`weak_entry_t`从`weak_table`中移除

而对于`remove_referrer`方法，我们来简单的看下他的实现:

remove_referrer

```
// 删除old_referrer集合中的referrers
// 参数 entry 被弱引用对象
// 参数 old_referrer 要删除的弱引用指针
static void remove_referrer(weak_entry_t *entry, objc_object **old_referrer)
{
    // 指向entry的弱引用指针不超过4个
    if (! entry->out_of_line()) {
        for (size_t i = 0; i < WEAK_INLINE_COUNT; i++) {
            // 遍历inline_referrers数组如果找到直接置空
            if (entry->inline_referrers[i] == old_referrer) {
                entry->inline_referrers[i] = nil;
                return;
            }
        }
        // 如果没有找到 则报错 弱引用指针小于4个且在inline_referrers中没有找到
        _objc_inform("Attempted to unregister unknown __weak variable "
                     "at %p. This is probably incorrect use of "
                     "objc_storeWeak() and objc_loadWeak(). "
                     "Break on objc_weak_error to debug.\n",
                     old_referrer);
        objc_weak_error();
        return;
    }

    // 哈希函数 判断这个旧的弱引用指针存放的位置
    size_t begin = w_hash_pointer(old_referrer) & (entry->mask);
    size_t index = begin;
    size_t hash_displacement = 0;
    // 遍历entry->referrers数组查找old_referrer
    while (entry->referrers[index] != old_referrer) {
        // 如果没有在指定index找到 那么取下一个位置的值比较
        index = (index+1) & entry->mask;
        // 如果找了一圈仍然没有找到 那么报错
        if (index == begin)
            bad_weak_table(entry);
```

```
            // 更新最大哈希偏移值
            hash_displacement++;
            // 如果最大哈希偏移值 超过了预定的限制 那么报错
            if (hash_displacement > entry->max_hash_displacement) {
                _objc_inform("Attempted to unregister unknown __weak variable "
                             "at %p. This is probably incorrect use of "
                             "objc_storeWeak() and objc_loadWeak(). "
                             "Break on objc_weak_error to debug.\n",
                             old_referrer);
                objc_weak_error();
                return;
            }
        }

        // 走到这一步说明在entry->referrers中的index位置找到了值为old_referrer的引用
        // 将数组的这个位置置空
        entry->referrers[index] = nil;
        // 弱引用个数-1
        entry->num_refs--;
    }
```

上面的描述也很简单，大概的流程为：

- 在`entry->inline_referrers`中一次查找值为`old_referrer`的指针 如果找到就清空如果没找到报错
- 在`entry->referrers`中查找值为`old_referrer`的指针，如果找到则置空同时`entry->num_refs`做-1操作(使用`inline_referrers`存储时不会更新`num_refs`值因此移除也不用-1)

我们在删除指向某个对象的某个弱引用指针之后，还会对存储指向该对象的弱引用指针数组做判空操作，如果发现数组为空，那表示目前没有弱引用指针指向这个对象，那我们需要将这个对象从`weak_table`中移除。下面我们来看下移除方法`weak_entry_remove`。

weak_entry_remove

```
//从weak_table中移除entry （指向entry的弱引用指针数为0）
static void weak_entry_remove(weak_table_t *weak_table, weak_entry_t *entry)
{
    // 如果弱引用指针超过4个(弱引用指针存放在entry->referrers中)
    if (entry->out_of_line())
        // 释放entry->referrers中所有数据
        free(entry->referrers);
    bzero(entry, sizeof(*entry));
    //num_entries-1
    weak_table->num_entries--;
    //weak_table是否需要锁绒
    weak_compact_maybe(weak_table);
}
```

上面方法的主要操作为:

- 将没有弱引用的对象从全局的`weak_table`中移除
- 减少`weak_table`中存储的弱引用对象个数
- 判断`weak_table`是否需要缩小容量

上面的所有就是当我们将一个obj作weak引用时，所发生的事情。那么，当obj释放时，所有weak引用它的指针又是如何自动设置为nil的呢？接下来我们来看一下obj释放时，所发生的事情。

Dealloc

当对象引用计数为0时，runtime会调用_objc_rootDealloc方法来析构对象，实现如下：

```
- (void)dealloc {
    _objc_rootDealloc(self);
}

void
_objc_rootDealloc(id obj)
{
    assert(obj);

    obj->rootDealloc();
}
```

`_objc_rootDealloc`又会调用objc_object的`rootDealloc`方法

rootDealloc

```
inline void
objc_object::rootDealloc()
{
//    判断object是否采用了Tagged Pointer计数，如果是，则不进行任何析构操作。
    if (isTaggedPointer()) return;  // fixme necessary?

    //接下来判断对象是否采用了优化的isa计数方式（isa.nonpointer）
    // 对象没有被weak引用!isa.weakly_referenced
    // 没有关联对象!isa.has_assoc
    // 没有自定义的C++析构方法!isa.has_cxx_dtor
    // 没有用到sideTable来做引用计数 !isa.has_sidetable_rc
    // 如果满足条件 则可以快速释放
    if (fastpath(isa.nonpointer  &&
                 !isa.weakly_referenced  &&
                 !isa.has_assoc  &&
                 !isa.has_cxx_dtor  &&
                 !isa.has_sidetable_rc))
    {
        assert(!sidetable_present());
        free(this);
    }
```

```
    else {
        // 慢速释放
        object_dispose((id)this);
    }
}
```

因此根据上面代码判断，如果obj被weak引用了，应该进入`object_dispose((id)this)`分支，下面我们来看下`object_dispose`方法：

object_dispose

```
id
object_dispose(id obj)
{
    if (!obj) return nil;
    // 析构obj
    objc_destructInstance(obj);
    // 释放内存
    free(obj);

    return nil;
}
```

析构obj主要是看`objc_destructInstance`方法，下面我们来看下这个方法的实现

objc_destructInstance

```
void *objc_destructInstance(id obj)
{
    if (obj) {
        // Read all of the flags at once for performance.
        //c++析构函数
        bool cxx = obj->hasCxxDtor();
        //关联函数
        bool assoc = obj->hasAssociatedObjects();

        // 如果有c++析构函数 则调用c++析构函数.
        if (cxx)
            object_cxxDestruct(obj);

        // 如果有关联对象则移除关联对象
        if (assoc)
            _object_remove_assocations(obj);
        // 清理相关的引用
        obj->clearDeallocating();
    }

    return obj;
```

```
}
```

清理相关引用方法主要是在`clearDeallocating`中实现的，下面我们再来看下这个方法：

clearDeallocating

```
//正在清除side table 和weakly referenced
inline void
objc_object::clearDeallocating()
{
    // obj是否采用了优化isa引用计数
    if (slowpath(!isa.nonpointer)) {
        //没有采用优化isa引用计数 清理obj存储在sideTable中的引用计数等信息
        sidetable_clearDeallocating();
    }
    // 启用了isa优化，则判断是否使用了sideTable
    // 使用的原因是因为做了weak引用（isa.weakly_referenced ） 或 使用了sideTable的辅助引用计数（isa.has_sidetable_rc）
    else if (slowpath(isa.weakly_referenced  ||  isa.has_sidetable_rc)) {
        // Slow path for non-pointer isa with weak refs and/or side table data.
        //释放weak 和引用计数
        clearDeallocating_slow();
    }

    assert(!sidetable_present());
}
```

这里的清理方法有两个分别为`sidetable_clearDeallocating`和`clearDeallocating_slow`，

我们先来看下`clearDeallocating_slow`：

clearDeallocating_slow

```
NEVER_INLINE void
objc_object::clearDeallocating_slow()
{
    assert(isa.nonpointer  &&  (isa.weakly_referenced ||
isa.has_sidetable_rc));

    // 在全局的SideTables中，以this指针为key，找到对应的SideTable
    SideTable& table = SideTables()[this];
    table.lock();
        //// 如果obj被弱引用
    if (isa.weakly_referenced) {
        //// 在SideTable的weak_table中对this进行清理工作
        weak_clear_no_lock(&table.weak_table, (id)this);
    }
```

```
    // 如果采用了SideTable做引用计数
    if (isa.has_sidetable_rc) {
        //在SideTable的引用计数中移除this
        table.refcnts.erase(this);
    }
    table.unlock();
}
```

这里调用了`weak_clear_no_lock`来做`weak_table`的清理工作，同时将所有weak引用该对象的ptr置为nil。

weak_clear_no_lock

```
//清理weak_table，同时将所有weak引用该对象的ptr置为nil
void
weak_clear_no_lock(weak_table_t *weak_table, id referent_id)
{
    objc_object *referent = (objc_object *)referent_id;

    // 找到referent在weak_table中对应的weak_entry_t
    weak_entry_t *entry = weak_entry_for_referent(weak_table, referent);

    if (entry == nil) {
        /// XXX shouldn't happen, but does with mismatched CF/objc
        //printf("XXX no entry for clear deallocating %p\n", referent);
        return;
    }

    // 找出weak引用referent的weak 指针地址数组以及数组长度
    weak_referrer_t *referrers;
    size_t count;
    // 是否使用动态数组
    if (entry->out_of_line()) {
        referrers = entry->referrers;
        count = TABLE_SIZE(entry);
    }
    else {
        referrers = entry->inline_referrers;
        count = WEAK_INLINE_COUNT;
    }

    // 遍历所有的所引用weak指针
    for (size_t i = 0; i < count; ++i) {
        // 取出每个weak ptr的地址
        objc_object **referrer = referrers[I];

        if (referrer) {
```

```
            // 如果weak ptr确实weak引用了referent，则将weak ptr设置为nil，这也就是为什么weak 指针会自动设置为nil的原因
            if (*referrer == referent) {
                *referrer = nil;
            }
            else if (*referrer) {
                // 如果所存储的weak ptr没有weak 引用referent，这可能是由于runtime代码的逻辑错误引起的，报错
                _objc_inform("__weak variable at %p holds %p instead of %p. "
                             "This is probably incorrect use of "
                             "objc_storeWeak() and objc_loadWeak(). "
                             "Break on objc_weak_error to debug.\n",
                             referrer, (void*)*referrer, (void*)referent);
                objc_weak_error();
            }
        }
    }
    // 由于referent要被释放了，因此referent的weak_entry_t也要移除出weak_table
    weak_entry_remove(weak_table, entry);
}
```

上面就是为什么当对象析构时，所有弱引用该对象的指针都会被设置为nil的原因。

总结

综上我们讲述了SideTable的结构，以及如何使用SideTable存储和清除对象和指向这些对象的指针地址。从而在侧面验证了弱引用的存储方式以及在对象释放时如何将弱引用的指针置空。读完这篇文章相信你对于SideTable结构和弱引用已经有了一个比较全面的认识。

在我们的App代码中，XCode会自动创建一个main.m文件，其中定义了main函数

这里的`main`函数是我们整个`App`的入口，它的调用时机甚至会早于`AppDelegate`中的`didFinishLaunching`回调。

因此我们会说，`main函数是我们App程序的入口点函数`。

那么，我们App所运行的第一个函数，真的是`main函数`吗？如果我们在XCode中设置符号断点`void _objc_init(void)`，则会发现，在进入`main函数`之前，其实系统还会调用`void _objc_init(void)`方法：

这里的`_objc_init`方法，实际上是`runtime`的入口函数。

```
void _objc_init(void)
```

也就是说，`在App的main函数之前，系统会首先对App的runtime运行环境，做了一系列的初始化操作。`

而这个`runtime入口函数`，又是被谁调用起来的呢？答案是`苹果的动态链接器dyld`（`the dynamic link editor`）。`dyld`是一个操作系统级的组件，它会负责iOS系统中每个App启动时的环境初始化以及动态库加载到内存等一系列操作。

在系统内核做好程序准备工作之后，交由dyld负责余下的工作。

在这里再重申一遍，runtime的入口函数是`_objc_init`，它是在main函数之前被dyld调用的。而`+load()`方法，则是在main函数前被`_objc_init`调用。今天，我们就来看一下，在main函数之前，runtime究竟做了哪些初始化工作。

Mach-O格式

在深入了解`_objc_init`的实现之前，我们需要先了解iOS系统中可执行文件的文件格式：`Mach-O格式`。关于Mach-O格式，我们在前面已介绍过。

我们可以在XCode 工程 product文件夹下找到·RuntimeEnter.app·工程，用·finder打开·所在目录，其实·RuntimeEnter.app·是一个压缩包，用鼠标右键选择·show Package Contents· ，可以看到下面有这些文件，其中和我们工程同名的可运行程序就是Mach-O格式的可运行文件：

_objc_init

我们在iOS App中设置符号断点·_objc_init·，则在App启动时（进入main函数之前），会进入如下调用堆栈：

可以看到，其底层是由dyld调用起来的。关于dyld我们不去多说，让我们看一下runtime中_objc_init的定义：

```
/**********************************************************************
* _objc_init
* Bootstrap initialization. Registers our image notifier with dyld.
* Called by libSystem BEFORE library initialization time
**********************************************************************/

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

    // fixme defer initialization until an objc-using image is found?
    environ_init();
    tls_init();
    static_init();
```

```
    lock_init();
    exception_init();

    _dyld_objc_notify_register(&map_images, load_images, unmap_image);
}
```

除去上面一堆init方法，我们重点关注

```
 _dyld_objc_notify_register(&map_images, load_images, unmap_image);
```

`_dyld_objc_notify_register`方法注册了对`dyld`中关于加载`images`的事件回调：

```
//
// Note: only for use by objc runtime
// Register handlers to be called when objc images are mapped, unmapped, and
initialized.
// Dyld will call back the "mapped" function with an array of images that
contain an objc-image-info section.
// Those images that are dylibs will have the ref-counts automatically bumped,
so objc will no longer need to
// call dlopen() on them to keep them from being unloaded.  During the call to
_dyld_objc_notify_register(),
// dyld will call the "mapped" function with already loaded objc images.
During any later dlopen() call,
// dyld will also call the "mapped" function.  Dyld will call the "init"
function when dyld would be called
// initializers in that image.  This is when objc calls any +load methods in
that image.
//
void _dyld_objc_notify_register(_dyld_objc_notify_mapped    mapped,
                                _dyld_objc_notify_init      init,
                                _dyld_objc_notify_unmapped  unmapped);
```

分别注册了那些事件呢？根据注释，我们可以知道，共注册了三个事件的回调：

- `_dyld_objc_notify_mapped`： OC 资源被加载映射到内存（+load()方法在此时被调用）
- `_dyld_objc_notify_init`： OC 资源被init时
- `_dyld_objc_notify_unmapped`： OC 资源被移除内存时

以上三个回调类型是用的函数指针，定义为

```
typedef void (*_dyld_objc_notify_mapped)(unsigned count, const char* const
paths[], const struct mach_header* const mh[]);
typedef void (*_dyld_objc_notify_init)(const char* path, const struct
mach_header* mh);
typedef void (*_dyld_objc_notify_unmapped)(const char* path, const struct
mach_header* mh);
```

_dyld_objc_notify_mapped

当OC资源被`dyld`加载到内存后，会调用回调`_dyld_objc_notify_mapped`。在runtime中，对应的函数是：

```
void map_images(unsigned count, const char * const paths[],
           const struct mach_header * const mhdrs[])
{
    rwlock_writer_t lock(runtimeLock);
    return map_images_nolock(count, paths, mhdrs);
}

void
map_images_nolock(unsigned mhCount, const char * const mhPaths[],
                  const struct mach_header * const mhdrs[])
{
    static bool firstTime = YES;
    header_info *hList[mhCount];
    uint32_t hCount;
    size_t selrefCount = 0;

    // Perform first-time initialization if necessary.
    // This function is called before ordinary library initializers.
    // fixme defer initialization until an objc-using image is found?
    if (firstTime) {
        preopt_init();
    }

    if (PrintImages) {
        _objc_inform("IMAGES: processing %u newly-mapped images...\n",
mhCount);
    }


    // Find all images with Objective-C metadata.
    hCount = 0;

    // Count classes. Size various table based on the total.
    int totalClasses = 0;
    int unoptimizedTotalClasses = 0;
    {
        uint32_t i = mhCount;
        while (i--) {
            const headerType *mhdr = (const headerType *)mhdrs[I];

            auto hi = addHeader(mhdr, mhPaths[i], totalClasses,
unoptimizedTotalClasses);
            if (!hi) {
```

```
                // no objc data in this entry
                continue;
            }

            if (mhdr->filetype == MH_EXECUTE) {
                // Size some data structures based on main executable's size
#if __OBJC2__
                size_t count;
                _getObjc2SelectorRefs(hi, &count);
                selrefCount += count;
                _getObjc2MessageRefs(hi, &count);
                selrefCount += count;
#else
                _getObjcSelectorRefs(hi, &selrefCount);
#endif

#if SUPPORT_GC_COMPAT
                // Halt if this is a GC app.
                if (shouldRejectGCApp(hi)) {
                    _objc_fatal_with_reason
                        (OBJC_EXIT_REASON_GC_NOT_SUPPORTED,
                         OS_REASON_FLAG_CONSISTENT_FAILURE,
                         "Objective-C garbage collection "
                         "is no longer supported.");
                }
#endif
            }

            hList[hCount++] = hi;

            if (PrintImages) {
                _objc_inform("IMAGES: loading image for %s%s%s%s%s\n",
                             hi->fname(),
                             mhdr->filetype == MH_BUNDLE ? " (bundle)" : "",
                             hi->info()->isReplacement() ? " (replacement)" :
"",
                             hi->info()->hasCategoryClassProperties() ? " (has
class properties)" : "",
                             hi->info()->optimizedByDyld()?"
(preoptimized)":"");
            }
        }
    }

    // Perform one-time runtime initialization that must be deferred until
    // the executable itself is found. This needs to be done before
    // further initialization.
    // (The executable may not be present in this infoList if the
    // executable does not contain Objective-C code but Objective-C
```

```
    // is dynamically loaded later.
    if (firstTime) {
        sel_init(selrefCount);
        arr_init();

#if SUPPORT_GC_COMPAT
        // Reject any GC images linked to the main executable.
        // We already rejected the app itself above.
        // Images loaded after launch will be rejected by dyld.

        for (uint32_t i = 0; i < hCount; i++) {
            auto hi = hList[I];
            auto mh = hi->mhdr();
            if (mh->filetype != MH_EXECUTE  &&  shouldRejectGCImage(mh)) {
                _objc_fatal_with_reason
                    (OBJC_EXIT_REASON_GC_NOT_SUPPORTED,
                     OS_REASON_FLAG_CONSISTENT_FAILURE,
                     "%s requires Objective-C garbage collection "
                     "which is no longer supported.", hi->fname());
            }
        }
#endif

#if TARGET_OS_OSX
        // Disable +initialize fork safety if the app is too old (< 10.13).
        // Disable +initialize fork safety if the app has a
        //   __DATA,__objc_fork_ok section.

        if (dyld_get_program_sdk_version() < DYLD_MACOSX_VERSION_10_13) {
            DisableInitializeForkSafety = true;
            if (PrintInitializing) {
                _objc_inform("INITIALIZE: disabling +initialize fork "
                             "safety enforcement because the app is "
                             "too old (SDK version " SDK_FORMAT ")",
                             FORMAT_SDK(dyld_get_program_sdk_version()));
            }
        }

        for (uint32_t i = 0; i < hCount; i++) {
            auto hi = hList[I];
            auto mh = hi->mhdr();
            if (mh->filetype != MH_EXECUTE) continue;
            unsigned long size;
            if (getsectiondata(hi->mhdr(), "__DATA", "__objc_fork_ok", &size))
{
                DisableInitializeForkSafety = true;
                if (PrintInitializing) {
                    _objc_inform("INITIALIZE: disabling +initialize fork "
                                 "safety enforcement because the app has "
```

```
                                    "a __DATA,__objc_fork_ok section");
                }
            }
            break;  // assume only one MH_EXECUTE image
        }
#endif

    }

    if (hCount > 0) {
        _read_images(hList, hCount, totalClasses, unoptimizedTotalClasses);
    }

    firstTime = NO;
}
```

`map_images`方法实质上会调用`map_images_nolock`方法。 而在`map_images_nolock`内部，又调用了`_read_images`方法，其核心是用来读取`Mach-O`格式文件的runtime相关的`section`信息，并转化为runtime内部的数据结构。

我们来看一下`_read_images` 方法的实现：

```
void _read_images(header_info **hList, uint32_t hCount, int totalClasses, int
unoptimizedTotalClasses)
{
    header_info *hi;
    uint32_t hIndex;
    size_t count;
    size_t I;
    Class *resolvedFutureClasses = nil;
    size_t resolvedFutureClassCount = 0;
    static bool doneOnce;
    TimeLogger ts(PrintImageTimes);

    runtimeLock.assertWriting();

#define EACH_HEADER \
    hIndex = 0;         \
    hIndex < hCount && (hi = hList[hIndex]); \
    hIndex++

    if (!doneOnce) {
        doneOnce = YES;

#if SUPPORT_NONPOINTER_ISA
        // Disable non-pointer isa under some conditions.

# if SUPPORT_INDEXED_ISA
        // Disable nonpointer isa if any image contains old Swift code
```

```
        for (EACH_HEADER) {
            if (hi->info()->containsSwift()  &&
                hi->info()->swiftVersion() < objc_image_info::SwiftVersion3)
            {
                DisableNonpointerIsa = true;
                if (PrintRawIsa) {
                    _objc_inform("RAW ISA: disabling non-pointer isa because "
                                 "the app or a framework contains Swift code "
                                 "older than Swift 3.0");
                }
                break;
            }
        }
# endif

# if TARGET_OS_OSX
        // Disable non-pointer isa if the app is too old
        // (linked before OS X 10.11)
        if (dyld_get_program_sdk_version() < DYLD_MACOSX_VERSION_10_11) {
            DisableNonpointerIsa = true;
            if (PrintRawIsa) {
                _objc_inform("RAW ISA: disabling non-pointer isa because "
                             "the app is too old (SDK version " SDK_FORMAT
")",
                             FORMAT_SDK(dyld_get_program_sdk_version()));
            }
        }

        // Disable non-pointer isa if the app has a __DATA,__objc_rawisa
section
        // New apps that load old extensions may need this.
        for (EACH_HEADER) {
            if (hi->mhdr()->filetype != MH_EXECUTE) continue;
            unsigned long size;
            if (getsectiondata(hi->mhdr(), "__DATA", "__objc_rawisa", &size))
{
                DisableNonpointerIsa = true;
                if (PrintRawIsa) {
                    _objc_inform("RAW ISA: disabling non-pointer isa because "
                                 "the app has a __DATA,__objc_rawisa
section");
                }
            }
            break;  // assume only one MH_EXECUTE image
        }
# endif

#endif
```

```
        if (DisableTaggedPointers) {
            disableTaggedPointers();
        }

        if (PrintConnecting) {
            _objc_inform("CLASS: found %d classes during launch",
totalClasses);
        }

        // namedClasses
        // Preoptimized classes don't go in this table.
        // 4/3 is NXMapTable's load factor
        int namedClassesSize =
            (isPreoptimized() ? unoptimizedTotalClasses : totalClasses) * 4 /
3;
        gdb_objc_realized_classes =
            NXCreateMapTable(NXStrValueMapPrototype, namedClassesSize);

        ts.log("IMAGE TIMES: first time tasks");
    }


    // Discover classes. Fix up unresolved future classes. Mark bundle
classes.

    for (EACH_HEADER) {
        if (! mustReadClasses(hi)) {
            // Image is sufficiently optimized that we need not call
readClass()
            continue;
        }

        bool headerIsBundle = hi->isBundle();
        bool headerIsPreoptimized = hi->isPreoptimized();

        classref_t *classlist = _getObjc2ClassList(hi, &count);
        for (i = 0; i < count; i++) {
            Class cls = (Class)classlist[I];
            Class newCls = readClass(cls, headerIsBundle,
headerIsPreoptimized);

            if (newCls != cls  &&  newCls) {
                // Class was moved but not deleted. Currently this occurs
                // only when the new class resolved a future class.
                // Non-lazily realize the class below.
                resolvedFutureClasses = (Class *)
                    realloc(resolvedFutureClasses,
                            (resolvedFutureClassCount+1) * sizeof(Class));
                resolvedFutureClasses[resolvedFutureClassCount++] = newCls;
```

```
            }
        }
    }

    ts.log("IMAGE TIMES: discover classes");

    // Fix up remapped classes
    // Class list and nonlazy class list remain unremapped.
    // Class refs and super refs are remapped for message dispatching.

    if (!noClassesRemapped()) {
        for (EACH_HEADER) {
            Class *classrefs = _getObjc2ClassRefs(hi, &count);
            for (i = 0; i < count; i++) {
                remapClassRef(&classrefs[I]);
            }
            // fixme why doesn't test future1 catch the absence of this?
            classrefs = _getObjc2SuperRefs(hi, &count);
            for (i = 0; i < count; i++) {
                remapClassRef(&classrefs[I]);
            }
        }
    }

    ts.log("IMAGE TIMES: remap classes");

    // Fix up @selector references
    static size_t UnfixedSelectors;
    sel_lock();
    for (EACH_HEADER) {
        if (hi->isPreoptimized()) continue;

        bool isBundle = hi->isBundle();
        SEL *sels = _getObjc2SelectorRefs(hi, &count);
        UnfixedSelectors += count;
        for (i = 0; i < count; i++) {
            const char *name = sel_cname(sels[i]);
            sels[i] = sel_registerNameNoLock(name, isBundle);
        }
    }
    sel_unlock();

    ts.log("IMAGE TIMES: fix up selector references");

#if SUPPORT_FIXUP
    // Fix up old objc_msgSend_fixup call sites
    for (EACH_HEADER) {
        message_ref_t *refs = _getObjc2MessageRefs(hi, &count);
        if (count == 0) continue;
```

```
        if (PrintVtables) {
            _objc_inform("VTABLES: repairing %zu unsupported vtable dispatch "
                         "call sites in %s", count, hi->fname());
        }
        for (i = 0; i < count; i++) {
            fixupMessageRef(refs+i);
        }
    }

    ts.log("IMAGE TIMES: fix up objc_msgSend_fixup");
#endif

    // Discover protocols. Fix up protocol refs.
    for (EACH_HEADER) {
        extern objc_class OBJC_CLASS_$_Protocol;
        Class cls = (Class)&OBJC_CLASS_$_Protocol;
        assert(cls);
        NXMapTable *protocol_map = protocols();
        bool isPreoptimized = hi->isPreoptimized();
        bool isBundle = hi->isBundle();

        protocol_t **protolist = _getObjc2ProtocolList(hi, &count);
        for (i = 0; i < count; i++) {
            readProtocol(protolist[i], cls, protocol_map,
                         isPreoptimized, isBundle);
        }
    }

    ts.log("IMAGE TIMES: discover protocols");

    // Fix up @protocol references
    // Preoptimized images may have the right
    // answer already but we don't know for sure.
    for (EACH_HEADER) {
        protocol_t **protolist = _getObjc2ProtocolRefs(hi, &count);
        for (i = 0; i < count; i++) {
            remapProtocolRef(&protolist[I]);
        }
    }

    ts.log("IMAGE TIMES: fix up @protocol references");

    // Realize non-lazy classes (for +load methods and static instances)
    for (EACH_HEADER) {
        classref_t *classlist =
            _getObjc2NonlazyClassList(hi, &count);
        for (i = 0; i < count; i++) {
            Class cls = remapClass(classlist[i]);
```

```
            if (!cls) continue;

            // hack for class __ARCLite__, which didn't get this above
#if TARGET_OS_SIMULATOR
            if (cls->cache._buckets == (void*)&_objc_empty_cache  &&
                (cls->cache._mask  ||  cls->cache._occupied))
            {
                cls->cache._mask = 0;
                cls->cache._occupied = 0;
            }
            if (cls->ISA()->cache._buckets == (void*)&_objc_empty_cache  &&
                (cls->ISA()->cache._mask  ||  cls->ISA()->cache._occupied))
            {
                cls->ISA()->cache._mask = 0;
                cls->ISA()->cache._occupied = 0;
            }
#endif

            realizeClass(cls);
        }
    }

    ts.log("IMAGE TIMES: realize non-lazy classes");

    // Realize newly-resolved future classes, in case CF manipulates them
    if (resolvedFutureClasses) {
        for (i = 0; i < resolvedFutureClassCount; i++) {
            realizeClass(resolvedFutureClasses[I]);
            resolvedFutureClasses[i]-
>setInstancesRequireRawIsa(false/*inherited*/);
        }
        free(resolvedFutureClasses);
    }

    ts.log("IMAGE TIMES: realize future classes");

    // Discover categories.
    for (EACH_HEADER) {
        category_t **catlist =
            _getObjc2CategoryList(hi, &count);
        bool hasClassProperties = hi->info()->hasCategoryClassProperties();

        for (i = 0; i < count; i++) {
            category_t *cat = catlist[I];
            Class cls = remapClass(cat->cls);

            if (!cls) {
                // Category's target class is missing (probably weak-linked).
                // Disavow any knowledge of this category.
```

```
                catlist[i] = nil;
                if (PrintConnecting) {
                    _objc_inform("CLASS: IGNORING category \?\?\?(%s) %p with
"
                                 "missing weak-linked target class",
                                 cat->name, cat);
                }
                continue;
            }

            // Process this category.
            // First, register the category with its target class.
            // Then, rebuild the class's method lists (etc) if
            // the class is realized.
            bool classExists = NO;
            if (cat->instanceMethods ||  cat->protocols
                ||  cat->instanceProperties)
            {
                addUnattachedCategoryForClass(cat, cls, hi);
                if (cls->isRealized()) {
                    remethodizeClass(cls);
                    classExists = YES;
                }
                if (PrintConnecting) {
                    _objc_inform("CLASS: found category -%s(%s) %s",
                                 cls->nameForLogging(), cat->name,
                                 classExists ? "on existing class" : "");
                }
            }

            if (cat->classMethods  ||  cat->protocols
                ||  (hasClassProperties && cat->_classProperties))
            {
                addUnattachedCategoryForClass(cat, cls->ISA(), hi);
                if (cls->ISA()->isRealized()) {
                    remethodizeClass(cls->ISA());
                }
                if (PrintConnecting) {
                    _objc_inform("CLASS: found category +%s(%s)",
                                 cls->nameForLogging(), cat->name);
                }
            }
        }
    }

    ts.log("IMAGE TIMES: discover categories");

    // Category discovery MUST BE LAST to avoid potential races
    // when other threads call the new category code before
```

```
    // this thread finishes its fixups.

    // +load handled by prepare_load_methods()

    if (DebugNonFragileIvars) {
        realizeAllClasses();
    }


    // Print preoptimization statistics
    if (PrintPreopt) {
        static unsigned int PreoptTotalMethodLists;
        static unsigned int PreoptOptimizedMethodLists;
        static unsigned int PreoptTotalClasses;
        static unsigned int PreoptOptimizedClasses;

        for (EACH_HEADER) {
            if (hi->isPreoptimized()) {
                _objc_inform("PREOPTIMIZATION: honoring preoptimized selectors
"
                             "in %s", hi->fname());
            }
            else if (hi->info()->optimizedByDyld()) {
                _objc_inform("PREOPTIMIZATION: IGNORING preoptimized selectors
"
                             "in %s", hi->fname());
            }

            classref_t *classlist = _getObjc2ClassList(hi, &count);
            for (i = 0; i < count; i++) {
                Class cls = remapClass(classlist[i]);
                if (!cls) continue;

                PreoptTotalClasses++;
                if (hi->isPreoptimized()) {
                    PreoptOptimizedClasses++;
                }

                const method_list_t *mlist;
                if ((mlist = ((class_ro_t *)cls->data())->baseMethods())) {
                    PreoptTotalMethodLists++;
                    if (mlist->isFixedUp()) {
                        PreoptOptimizedMethodLists++;
                    }
                }
                if ((mlist=((class_ro_t *)cls->ISA()->data())->baseMethods()))
{
                    PreoptTotalMethodLists++;
                    if (mlist->isFixedUp()) {
```

```
                        PreoptOptimizedMethodLists++;
                    }
                }
            }
        }

        _objc_inform("PREOPTIMIZATION: %zu selector references not "
                     "pre-optimized", UnfixedSelectors);
        _objc_inform("PREOPTIMIZATION: %u/%u (%.3g%%) method lists pre-
sorted",
                     PreoptOptimizedMethodLists, PreoptTotalMethodLists,
                     PreoptTotalMethodLists
                     ? 100.0*PreoptOptimizedMethodLists/PreoptTotalMethodLists
                     : 0.0);
        _objc_inform("PREOPTIMIZATION: %u/%u (%.3g%%) classes pre-registered",
                     PreoptOptimizedClasses, PreoptTotalClasses,
                     PreoptTotalClasses
                     ? 100.0*PreoptOptimizedClasses/PreoptTotalClasses
                     : 0.0);
        _objc_inform("PREOPTIMIZATION: %zu protocol references not "
                     "pre-optimized", UnfixedProtocolReferences);
    }

#undef EACH_HEADER
}
```

`_read_images` 方法写了很长，其实就是做了一件事，将`Mach-O`文件的`section`依次读取，并根据内容初始化`runtime的内存结构`。

根据注释，`_read_images` 方法主要做了下面这些事情：

- 1. 是否需要禁用`isa`优化。这里有三种情况：使用了swift 3.0前的swift代码。OSX版本早于10.11。在OSX系统下，Mach-O的DATA段明确指明了`__objc_rawisa`（不使用优化的isa）。
- 2. 在`__objc_classlist section`中读取`class list`
- 3. 在`__objc_classrefs section`中读取`class`引用的信息，并调用`remapClassRef`方法来处理。
- 4. 在`__objc_selrefs section`中读取`selector`的引用信息，并调用`sel_registerNameNoLock`方法处理。
- 5. 在`__objc_protolist section`中读取`cls`的`Protocol`信息，并调用`readProtocol`方法来读取`Protocol`信息。
- 6. 在`__objc_protorefs section`中读取`protocol`的`ref`信息，并调用`remapProtocolRef`方法来处理。
- 7. 在`__objc_nlclslist section`中读取`non-lazy class`信息，并调用`static Class realizeClass(Class cls)`方法来实现这些`class`。`realizeClass`方法核心是初始化`objc_class`数据结构，赋予初始值。
- 8. 在`__objc_catlist section`中读取`category`信息，并调

用`addUnattachedCategoryForClass`方法来为类或元类添加对应的方法，属性和协议。关于`Category`，我们在前面的文章中，已经有过相关的讨论。

OK，以上就是在`dyld`将`image map`到`内存`后，runtime所做的事情：**根据Mach-O相关section中的信息，来初始化runtim的内存结构。**

_dyld_objc_notify_init

当`dyld`要`init image`的时候，会产生`_dyld_objc_notify_init`通知。在runime中, 是通过`load_images`方法做回调响应的。

```
void load_images(const char *path __unused, const struct mach_header *mh)
{
    // Return without taking locks if there are no +load methods here.
    if (!hasLoadMethods((const headerType *)mh)) return;

    recursive_mutex_locker_t lock(loadMethodLock);

    // Discover load methods
    {
        rwlock_writer_t lock2(runtimeLock);
        prepare_load_methods((const headerType *)mh);
    }

    // Call +load methods (without runtimeLock - re-entrant)
    call_load_methods();
}
```

在load_images方法其实就是干了一件事，调用Class的`+load()`方法。

runtime调用`+load()`方法分为两步走：

- 1. Discover load methods
- 2. Call +load methods

Discover load methods

在`Discover load methods`中，会调用`prepare_load_methods`来处理`+load`方法：

```
void prepare_load_methods(const headerType *mhdr)
{
    size_t count, I;

    runtimeLock.assertWriting();

    // 添加 class 到 call +load 队列(super class 会在 subclass之前)
    classref_t *classlist =
```

```
        _getObjc2NonlazyClassList(mhdr, &count);
    for (i = 0; i < count; i++) {
        schedule_class_load(remapClass(classlist[i]));
    }

    // 添加 category到call +load 队列
    category_t **categorylist = _getObjc2NonlazyCategoryList(mhdr, &count);
    for (i = 0; i < count; i++) {
        category_t *cat = categorylist[I];
        Class cls = remapClass(cat->cls);
        if (!cls) continue;  // category for ignored weak-linked class
        realizeClass(cls);
        assert(cls->ISA()->isRealized());
        add_category_to_loadable_list(cat);
    }
}
```

`prepare_load_methods` 逻辑分为两个部分：

1. 调用schedule_class_load 来组织class的+load 方法
2. 调用add_category_to_loadable_list来组织category的+load方法

`schedule_class_load`方法实现如下：

```
static void schedule_class_load(Class cls)
{
    if (!cls) return;
    assert(cls->isRealized());  // _read_images should realize

    if (cls->data()->flags & RW_LOADED) return;

    // Ensure superclass-first ordering
    schedule_class_load(cls->superclass);

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

这是一个递归调用，会先把`superclass`用`add_class_to_loadable_list`方法到`loadable class list`中：

```
void add_class_to_loadable_list(Class cls)
{
    IMP method;

    loadMethodLock.assertLocked();

    method = cls->getLoadMethod();
    if (!method) return;  // Don't bother if cls has no +load method
```

```
    if (PrintLoading) {
        _objc_inform("LOAD: class '%s' scheduled for +load",
                     cls->nameForLogging());
    }

    if (loadable_classes_used == loadable_classes_allocated) {
        loadable_classes_allocated = loadable_classes_allocated*2 + 16;
        loadable_classes = (struct loadable_class *)
            realloc(loadable_classes,
                                loadable_classes_allocated *
                                sizeof(struct loadable_class));
    }

    loadable_classes[loadable_classes_used].cls = cls;
    loadable_classes[loadable_classes_used].method = method;
    loadable_classes_used++;
}
```

```
static struct loadable_class *loadable_classes = nil;
```

从上面的方法可以看出，每一个定义了+load的类，都会被放到loadable_classes中。

因此，+load方法并不存在子类重写父类之说。而且父类的+load方法会先于子类调用。

在来看一下`add_category_to_loadable_list`方法：

```
void add_category_to_loadable_list(Category cat)
{
    IMP method;

    loadMethodLock.assertLocked();

    method = _category_getLoadMethod(cat);

    // Don't bother if cat has no +load method
    if (!method) return;

    if (PrintLoading) {
        _objc_inform("LOAD: category '%s(%s)' scheduled for +load",
                     _category_getClassName(cat), _category_getName(cat));
    }

    if (loadable_categories_used == loadable_categories_allocated) {
        loadable_categories_allocated = loadable_categories_allocated*2 + 16;
        loadable_categories = (struct loadable_category *)
            realloc(loadable_categories,
                                loadable_categories_allocated *
                                sizeof(struct loadable_category));
```

```
    }

    loadable_categories[loadable_categories_used].cat = cat;
    loadable_categories[loadable_categories_used].method = method;
    loadable_categories_used++;
}
```

```
static struct loadable_category *loadable_categories = nil;
```

在`add_category_to_loadable_list`方法中，会将所有定义了`+load`方法的`category`都放到`loadable_categories`队列中。

Call +load methods

将定义了`+load`方法的`class`和`category`分别放到`loadable_classes`和`loadable_categories`队列后，runtime就会依次读取队列中的`class`和`category`，并为之调用`+load`方法：

```
void call_load_methods(void)
{
    static bool loading = NO;
    bool more_categories;

    loadMethodLock.assertLocked();

    // Re-entrant calls do nothing; the outermost call will finish the job.
    if (loading) return;
    loading = YES;

    void *pool = objc_autoreleasePoolPush();

    do {
        // 1. Repeatedly call class +loads until there aren't any more
        while (loadable_classes_used > 0) {
            call_class_loads();
        }

        // 2. Call category +loads ONCE
        more_categories = call_category_loads();

        // 3. Run more +loads if there are classes OR more untried categories
    } while (loadable_classes_used > 0  ||  more_categories);

    objc_autoreleasePoolPop(pool);

    loading = NO;
}
```

从上面的`call_load_methods`方法可以看出：

`super class`的`+load`方法调用时机是先于`sub class`的 `class`的`+load`方法调用时机是先于`category`的

_dyld_objc_notify_unmapped

当`dyld`要将`image`移除内存时，会发送`_dyld_objc_notify_unmapped`通知。在runtime中，是用`unmap_image`方法来响应的。

```
void
unmap_image(const char *path __unused, const struct mach_header *mh)
{
    recursive_mutex_locker_t lock(loadMethodLock);
    rwlock_writer_t lock2(runtimeLock);
    unmap_image_nolock(mh);
}
```

```
void unmap_image_nolock(const struct mach_header *mh)
{
    if (PrintImages) {
        _objc_inform("IMAGES: processing 1 newly-unmapped image...\n");
    }

    header_info *hi;

    // Find the runtime's header_info struct for the image
    for (hi = FirstHeader; hi != NULL; hi = hi->getNext()) {
        if (hi->mhdr() == (const headerType *)mh) {
            break;
        }
    }

    if (!hi) return;

    if (PrintImages) {
        _objc_inform("IMAGES: unloading image for %s%s%s\n",
                     hi->fname(),
                     hi->mhdr()->filetype == MH_BUNDLE ? " (bundle)" : "",
                     hi->info()->isReplacement() ? " (replacement)" : "");
    }

    _unload_image(hi);

    // Remove header_info from header list
    removeHeader(hi);
    free(hi);
```

```
}
```

主要是做了`header`信息的移除。

总结

在本篇文章中，我们知道了`dyld`在main()函数之前，会调用`runtime`的`_objc_init`方法。`_objc_init`是`runtime`的入口函数，它会根据`Mach-O`文件中相关的`section`信息来初始化`runtime`内存空间。比如，加载`class`，`protocol`，以及附加`category`到`class`，调用`+load`方法等。

当然，在`main()`函数前，`dyld`除了调用`_objc_init`外，还会做许多其他的操作。如将动态链接库加载入内存。但这就不属于runtime的范畴了，我们不去深究。

当`dyld`将我们App的运行环境都准备好后，`dyld`会清理现场，将调用栈回归，调用`main()`函数，这时候，我们的App就算启动了：

```
int main(int argc, char * argv[]) {
    @autoreleasepool {
        return UIApplicationMain(argc, argv, nil,
NSStringFromClass([AppDelegate class]));
    }
}
```

在`main()`函数被调用前，系统其实已经为我们做了很多的准备工作。就像sunnyxx在其博客中说的：

孤独的 main 函数，看上去是程序的开始，却是一段精彩的终结

自动触发KVO

在平日代码中，我们通过KVO来监视实例某个属性的变化。 比如，我们要监视Student 的 age属性，可以这么做：

```
@interface Student : NSObject
@property(nonatomic, strong) NSString *name;
@end

@interface ViewController ()

@end

@implementation ViewController
- (void)viewDidLoad {
    [super viewDidLoad];
    Student *std = [Student new];
    std.name = @"Tom";
        [std addObserver:self forKeyPath:@"name"
```

```
options:NSKeyValueObservingOptionNew|NSKeyValueObservingOptionOld context:nil];
 }

- (void)observeValueForKeyPath:(NSString *)keyPath ofObject:(id)object change:
(NSDictionary<NSKeyValueChangeKey,id> *)change context:(void *)context {
        ...
}
@end
```

我们使用KVO需要遵循以下步骤：

- 1. 调用addObserver:forKeyPath:options:context: 方法来注册观察者，观察者可以接收到KeyPath对应属性的修改通知
- 2. 当观察的属性发生变化时，系统会在observeValueForKeyPath:ofObject:change:context:方法中回调观察者
- 3. 当观察者不需要监听变化是，需要调用removeObserver:forKeyPath:将KVO移除。需要注意的是，在观察者被释放前，必须要调用removeObserver:forKeyPath:将其移除，否则会crash。同时，作为被观察者，当其自身dealloc时，必须清空他的观察者，否则会crash。

手动触发KVO

当我们设置了观察者后，当被观察的keyPath对应的setter方法调用后，则会自动的触发KVO的回调函数。那么，有时候我们想要控制这种自动触发的机制，该怎么办呢？你可以重写如下方法：

```
+ (BOOL)automaticallyNotifiesObserversForKey:(NSString *)theKey {
    BOOL automatic = NO;
    if ([theKey isEqualToString:@"balance"]) {
        automatic = NO;
    }
    else {
        automatic = [super automaticallyNotifiesObserversForKey:theKey];
    }
    return automatic;
}
```

`automaticallyNotifiesObserversForKey`方法声明在NSObject的Category `NSObject(NSKeyValueObservingCustomization)`中。

除了在setter方法中，有时候我们想主动触发一下KVO，该怎么办呢？ 那就需要使用

```
willChangeValueForKey:
didChangeValueForKey:
```

来通知系统Key Value发生了改变。如：

```
- (void)updateName:(NSString *)name {
        [self willChangeVauleForKey:@"name"];
        _name = name;
        [self didChangeVauleForKey:@"name"];
}
```

KVO实现机制

那么，KVO背后是如何实现的呢？在苹果的官方文档上，有一个笼统的描述：

> Automatic key-value observing is implemented using a technique called isa-swizzling. The isa pointer, as the name suggests, points to the object's class which maintains a dispatch table. This dispatch table essentially contains pointers to the methods the class implements, among other data. When an observer is registered for an attribute of an object the isa pointer of the observed object is modified, pointing to an intermediate class rather than at the true class. As a result the value of the isa pointer does not necessarily reflect the actual class of the instance. You should never rely on the isa pointer to determine class membership. Instead, you should use the class method to determine the class of an object instance.

主要说了两件事：

- 1. KVO是基于`isa-swizzling`技术实现的。`isa-swizzling`会将被观察对象的`isa`指针进行替换。
- 2. 因为在实现`KVO`时，系统会替换掉被观察对象的`isa`指针，因此，不要使用`isa`指针来判断类的关系，而应该使用`class`方法。

为什么要替换掉`isa`指针？文档中说的很清楚，因为isa指针会指向类实例对应的类的方法列表，而替换掉了isa指针，相当于替换掉了类的方法列表。

那么为啥要替换类的方法列表呢？又是怎么替换的呢？文档到这里戛然而止，没有细说。

下面，我们就用代码实验的方式，来窥探一下KVO的实现机制。

准备如下代码：

```
@interface Student : NSObject
@property(nonatomic, strong) NSString *name;
@property(nonatomic, strong) NSMutableArray *friends;
@end

@implementation Student
- (void)showObjectInfo {
    NSLog(@"Object instance address is %p, Object isa content is %p", self, *
((void **)(__bridge void *)self));
}

@end
```

我们在`Student`类中定义了方法`- (void)showObjectInfo`，主要是用来打印`Student`实例的地址，以及`Student`的`isa`指针中的内容。这可以用来研究系统是如何做`isa-swizzling`操作的。

然后准备下面的方法，来打印类的方法列表：

```
static NSArray * ClassMethodNames(Class c)
{
    NSMutableArray * array = [NSMutableArray array];
    unsigned int methodCount = 0;
    Method * methodList = class_copyMethodList(c, &methodCount);
    unsigned int i;
    for(i = 0; i < methodCount; i++) {
        [array addObject: NSStringFromSelector(method_getName(methodList[i]))];
    }
    free(methodList);
    return array;
}
```

运行如下代码

```
- (void)viewDidLoad {
    [super viewDidLoad];
    Student *std = [Student new];
    // 1. 初始值
    std.name = @"Tom";
    NSLog(@"std->isa:%@", object_getClass(std));
    NSLog(@"std class:%@", [std class]);
    NSLog(@"ClassMethodNames:%@", ClassMethodNames(object_getClass(std)));
    [std showObjectInfo];

    // 2. 添加KVO
    [std addObserver:self forKeyPath:@"name"
options:NSKeyValueObservingOptionNew|NSKeyValueObservingOptionOld context:nil];
    [std addObserver:self forKeyPath:@"friends"
options:NSKeyValueObservingOptionNew|NSKeyValueObservingOptionOld context:nil];
    NSLog(@"std->isa:%@", object_getClass(std));
    NSLog(@"std class:%@", [std class]);
    NSLog(@"ClassMethodNames:%@", ClassMethodNames(object_getClass(std)));
    [std showObjectInfo];
    std.name = @"Jack";

    // 3. 移除KVO
    [std removeObserver:self forKeyPath:@"name"];
    [std removeObserver:self forKeyPath:@"friends"];
    NSLog(@"std->isa:%@", object_getClass(std));
    NSLog(@"std class:%@", [std class]);
    NSLog(@"ClassMethodNames:%@", ClassMethodNames(object_getClass(std)));
    [std showObjectInfo];
}
```

输出为：

```
// 1. 初始值
std->isa:Student
std class:Student
ClassMethodNames:(
    showObjectInfo,
    "setFriends:",
    friends,
    ".cxx_destruct",
    "setName:",
    name
)
Object address is 0x28194fe80, Object isa content is 0x1a1008090cd
```

```
// 2. 添加KVO
std->isa:NSKVONotifying_Student
std class:Student
ClassMethodNames:(
    "setFriends:",
    "setName:",
    class,
    dealloc,
    "_isKVOA"
)
```

```
Object address is 0x28194fe80, Object isa content is 0x1a282b5bf05
```

```
// 3. 移除KVO
std->isa:Student
std class:Student
ClassMethodNames:(
    showObjectInfo,
    "setFriends:",
    friends,
    ".cxx_destruct",
    "setName:",
    name
)
```

```
Object address is 0x28194fe80, Object isa content is 0x1a1008090cd
```

通过观察`添加KVO前`、`添加KVO后`，`移除KVO后`这三个实际的`Object`地址信息可以知道，`Object`的地址并没有改变，但是其`isa`指针中的内容，却经历了如下变化：`0x1a1008090cd`->`0x1a282b5bf05`->`0x1a1008090cd`。对应的，通过`object_getClass(std)`方法来输出std的类型是：`Student`->`NSKVONotifying_Student`->`Student`

这就是所谓的`isa-swizzling`，当KVO时，系统会将被观察对象的`isa指针`内容做替换，让其指向新的类`NSKVONotifying_Student`，而在移除KVO后，系统又会将`isa指针内容还原`。

那么，`NSKVONotifying_Student`这个类又是什么样的呢？通过打印其方法列表，可以知道，`NSKVONotifing_Stdent`定义或重写了如下方法：

```
ClassMethodNames:(
    "setFriends:",
    "setName:",
    class,
    dealloc,
    "_isKVOA"
)
```

可以看到，系统新生成的类重写了我们KVO的属性`Friends`和`Name`的`set方法`。

同时，还重写了`class`方法。通过runtime的源码可以知道，class方法实际是调用了`object_getClass`方法：

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

而在`object_getClass`方法中，会输出实例的isa指向的类：

```
Class object_getClass(id obj)
{
    if (obj) return obj->getIsa();
    else return Nil;
}
```

按说`[std class]`和`object_getClass(std)`的输出应该一致，但是系统会在KVO的时候，悄悄改写实例的`class`方法。这也就是为什么，当使用`[std class]`方法打印实例的类时，会输出`Student`而不是实际的`NSKVONotifing_Student`。

然后系统还重写了`dealloc`方法，估计是为了在实例销毁时，做一些检查及清理工作。

最后，添加了`_isKVOA`方法，这估计是系统为了识别是`KVO类`而添加的。

这里，细心的同学会发现，在KVO之前，`Student`的方法列表里面是包含属性的`get`方法，`showObjectInfo`方法以及`.cxx_destruct`这些方法的。而当系统将`Student`替换为`NSKVONotifing_Student`后，这些方法那里去了呢？如果这些方法没有在`NSKVONotifing_Student`再实现一遍的话，那当KVO后，我们再调用属性的`get`方法、`showObjectInfo`方法岂不是会`crash`？

但平日的编程实践告诉我们，并不会crash。那这些方法都去那里了呢？让我们来看一下`NSKVONotifing_Student`的父类是什么：

```
// 2. 添加KVO
...
Class objectRuntimeClass = object_getClass(std);
Class superClass = class_getSuperclass(objectRuntimeClass);
NSLog(@"super class is %@", superClass);
```

输出为：

```
super class is Student
```

哈哈，很有意思吧，原来`NSKVONotifing_Student`的父类竟然是`Student`。那根据OC的消息实现机制，当在`NSKVONotifing_Student`中没有找到方法实现时，会自动到其父类`Student`中寻找相应的实现。因此，在`NSKVONotifing_Student`中，仅仅需要定义或重写`KVO`相关的方法即可，至于`Student`中定义的其他方法，则会在消息机制中在被自动找到。

以上，便是KVO的isa-swizzling技术的大体实现流程。让我们总结一下：

- 1. 当类实例被`KVO`后，系统会替换实例的`isa`指针内容。让其指向`NSKVONotifing_XX`类型的新类。
- 2. 在`NSKVONotifing_XX`类中，会：重写`KVO`属性的`set`方法，支持`KVO`。重写`class`方法，来伪装自己仍然是`XX类`。添加`_isKVOA`方法，来说明自己是一个`KVO`类。重写`dealloc`方法，让实例下析构时，好做一些检查和清理工作
- 3. 为了让用户在`KVO isa-swizzling`后，仍然能够调用原始`XX类`中的方法，系统还会将`NSKVONotifing_XX`类设置为`原始XX类的子类`。
- 4. 当移除`KVO`后，系统会将`isa指针`中的`内容复原`。

手动实现KVO

既然知道了KVO背后的实现原理，我们能不能利用runtime方法，模拟实现一下KVO呢？ 当然可以，下来看下效果：

```
#import "ViewController.h"
#import "NSObject+KVOBlock.h"
#import <objc/runtime.h>
@implementation Student
@end
@interface ViewController ()
```

```
@end

@implementation ViewController

- (void)viewDidLoad {
    [super viewDidLoad];
    Student *std = [Student new];
    // 直接用block回调来接受 KVO
    [std sw_addObserver:self forKeyPath:@"name" callback:^(id  _Nonnull
observedObject, NSString * _Nonnull observedKeyPath, id  _Nonnull oldValue, id
_Nonnull newValue) {
        NSLog(@"old value is %@, new vaule is %@", oldValue, newValue);
    }];

    std.name = @"Hello";
    std.name = @"Lilhy";
    NSLog(@"class is %@, object_class is %@", [std class],
object_getClass(std));
    [std sw_removeObserver:self forKeyPath:@"name"];
    NSLog(@"class is %@, object_class is %@", [std class],
object_getClass(std));

}
@end
```

为了模拟的和系统KVO实现类似，我们也改写了`class`方法，在KVO移除前后，打印`std`的类信息为：

```
class is Student, object_class is sw_KVONotifing_Student
// 移除KVO后
class is Student, object_class is Student
```

在这里我手动实现了`KVO`，并通过`Block`的方式来接受`KVO`的回调信息。接下来我们就一步步的分析是如何做到的。我们应该重点观察所使用到的runtime方法。

首先，我们新建一个`NSObject`的分类`NSObject (KVOBlock)`，并声明如下方法：

```
typedef void(^sw_KVOObserverBlock)(id observedObject, NSString *observedKeyPath,
id oldValue, id newValue);

@interface NSObject (KVOBlock)
- (void)sw_addObserver:(NSObject *)observer
            forKeyPath:(NSString *)keyPath
              callback:(sw_KVOObserverBlock)callback;

- (void)sw_removeObserver:(NSObject *)observer
               forKeyPath:(NSString *)keyPath;

@end
```

在关键的`sw_addObserver:forKeyPath:callback:`中，是这么实现的：

```
static void *const sw_KVOObserverAssociatedKey = (void
*)&sw_KVOObserverAssociatedKey;
static NSString *sw_KVOClassPrefix = @"sw_KVONotifing_";

- (void)sw_addObserver:(NSObject *)observer
            forKeyPath:(NSString *)keyPath
              callback:(sw_KVOObserverBlock)callback {
    // 1. 通过keyPath获取当前类对应的setter方法，如果获取不到，说明setter 方法即不存在与
KVO类，也不存在与原始类，这总情况正常情况下是不会发生的，触发Exception
    NSString *setterString = sw_setterByGetter(keyPath);
    SEL setterSEL = NSSelectorFromString(setterString);
    Method method = class_getInstanceMethod(object_getClass(self), setterSEL);

    if (method) {
        // 2. 查看当前实例对应的类是否是KVO类，如果不是，则生成对应的KVO类，并设置当前实例对
应的class是KVO类
        Class objectClass = object_getClass(self);
        NSString *objectClassName = NSStringFromClass(objectClass);
        if (![objectClassName hasPrefix:sw_KVOClassPrefix]) {
            Class kvoClass = [self
makeKvoClassWithOriginalClassName:objectClassName]; // 为原始类创建KVO类
            object_setClass(self, kvoClass); // 将当前实例的类设置为KVO类
        }

        // 3. 在KVO类中查找是否重写过keyPath 对应的setter方法，如果没有，则添加setter方法
到KVO类中
        // 注意，此时object_getClass(self)获取到的class应该是KVO class
        if (![self hasMethodWithMethodName:setterString]) {
            class_addMethod(object_getClass(self),
NSSelectorFromString(setterString), (IMP)sw_kvoSetter,
method_getTypeEncoding(method));
        }
```

```
        // 4. 注册Observer
        NSMutableArray<SWKVOObserverItem *> *observerArray =
objc_getAssociatedObject(self, sw_KVOObserverAssociatedKey);
        if (observerArray == nil) {
            observerArray = [NSMutableArray new];
            objc_setAssociatedObject(self, sw_KVOObserverAssociatedKey,
observerArray, OBJC_ASSOCIATION_RETAIN_NONATOMIC);
        }
        SWKVOObserverItem *item = [SWKVOObserverItem new];
        item.keyPath = keyPath;
        item.observer = observer;
        item.callback = callback;
        [observerArray addObject:item];


    }else {
        NSString *exceptionReason = [NSString stringWithFormat:@"%@ Class %@
setter SEL not found.", NSStringFromClass([self class]), keyPath];
        NSException *exception = [NSException
exceptionWithName:@"NotExistKeyExceptionName" reason:exceptionReason
userInfo:nil];
        [exception raise];
    }
}
```

上面的函数重点是：

- 1. 调用makeKvoClassWithOriginalClassName方法来生成原始类对应的KVO类
- 2. 利用class_addMethod方法，为KVO类添加改写的setter实现

完成了上面两点，一个手工的KVO实现基本就完成了。另一个需要注意的是，如何存储`observer`。在这里是通过一个`MutableArray`数组，当做`Associated object`来存储到类实例中的。

可以看出来，这里的重点在于如何创建原始类对应的KVO类：

```
- (Class)makeKvoClassWithOriginalClassName:(NSString *)originalClassName {
    // 1. 检查KVO类是否已经存在，如果存在，直接返回
    NSString *kvoClassName = [NSString stringWithFormat:@"%@%@",
sw_KVOClassPrefix, originalClassName];
    Class kvoClass = objc_getClass(kvoClassName.UTF8String);
    if (kvoClass) {
        return kvoClass;
    }

    // 2. 创建KVO类，并将原始class设置为KVO类的super class
    kvoClass = objc_allocateClassPair(object_getClass(self),
kvoClassName.UTF8String, 0);
    objc_registerClassPair(kvoClass);

    // 3. 重写KVO类的class方法，使其指向我们自定义的IMP,实现KVO class的‘伪装’
```

```
    Method classMethod = class_getInstanceMethod(object_getClass(self),
@selector(class));
    const char* types = method_getTypeEncoding(classMethod);
    class_addMethod(kvoClass, @selector(class), (IMP)sw_class, types);
    return kvoClass;
}
```

其实实现也不难，调用了runtime的方法

- 1. objc_allocateClassPair(object_getClass(self), kvoClassName.UTF8String, 0) 动态生成新的KVO类，并设置KVO类的super class是原始类
- 2. 注册KVO类 : objc_registerClassPair(kvoClass)
- 3. 为了实现KVO伪装成原始类，还为KVO类添加了我们自己重写的class方法：

```
  Method classMethod = class_getInstanceMethod(object_getClass(self),
@selector(class));
    const char* types = method_getTypeEncoding(classMethod);
    class_addMethod(kvoClass, @selector(class), (IMP)sw_class, types);
```

```
// 自定义的class方法实现
static Class sw_class(id self, SEL selector) {
    return class_getSuperclass(object_getClass(self));  // 因为我们将原始类设置为
了KVO类的super class，所以直接返回KVO类的super class即可得到原始类Class
}
```

那么当我们需要移除Observer时，需要调用sw_removeObserver:forKeyPath: 方法：

```
- (void)sw_removeObserver:(NSObject *)observer  forKeyPath:(NSString *)keyPath
{
    NSMutableArray<SWKVOObserverItem *> *observerArray =
objc_getAssociatedObject(self, sw_KVOObserverAssociatedKey);
    [observerArray enumerateObjectsUsingBlock:^(SWKVOObserverItem * _Nonnull
obj, NSUInteger idx, BOOL * _Nonnull stop) {
        if (obj.observer == observer && [obj.keyPath isEqualToString:keyPath])
{
            [observerArray removeObject:obj];
        }
    }];

    if (observerArray.count == 0) { // 如果已经没有了observer，则把isa复原，销毁临时
的KVO类
        Class originalClass = [self class];
        Class kvoClass = object_getClass(self);
        object_setClass(self, originalClass);
        objc_disposeClassPair(kvoClass);
    }
}
```

注意，这里当`Observer`数组为空时，我们会将当前实例的所属类复原成原始类，并`dispose`掉生成的`KVO类`。

好了，`runtime`系列我们终于走到了大结局。为了验证大家对前面几篇文章的理解，我特意设计了如下题目，来考察大家（笑）。如果一时想不到答案也没有关系，解答中都有详细的分析。有些内容我们之前可能提到了一点，而这些题目则是对之前内容的深化和扩展，特别是关于OC对象内存分配相关的知识。

大家沉下心来分析下面的题目，相信会对OC背后的内存结构，消息机制有更进一步的提高。

1. 下面代码的输出结果?

```
        BOOL res1 = [[NSObject class] isKindOfClass:[NSObject class]];
        BOOL res2 = [[NSObject class] isMemberOfClass:[NSObject class]];
        BOOL res3 = [[Son class] isKindOfClass:[Son class]];
        BOOL res4 = [[Son class] isMemberOfClass:[Son class]];
        NSLog(@"%d %d %d %d", res1, res2, res3, res4);
```

解答：这道题的关键在于知道，`isKindOfClass`,`isMemberOfClass`和`class`方法均有两个版本：实例方法 和 类方法两个版本。 也意味着，类实例和类调用这三个方法的实现，是不一样的。我们来看它们的具体实现：

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

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

Class object_getClass(id obj)
{
    if (obj) return obj->getIsa();
    else return Nil;
}

inline Class
objc_object::getIsa()
{
    if (!isTaggedPointer()) return ISA();

    uintptr_t ptr = (uintptr_t)this;
    if (isExtTaggedPointer()) {
        uintptr_t slot =
            (ptr >> _OBJC_TAG_EXT_SLOT_SHIFT) & _OBJC_TAG_EXT_SLOT_MASK;
        return objc_tag_ext_classes[slot];
    } else {
```

```
        uintptr_t slot =
            (ptr >> _OBJC_TAG_SLOT_SHIFT) & _OBJC_TAG_SLOT_MASK;
        return objc_tag_classes[slot];
    }
}
```

```
+ (BOOL)isKindOfClass:(Class)cls {
    for (Class tcls = object_getClass((id)self); tcls; tcls = tcls->superclass)
{
        if (tcls == cls) return YES;
    }
    return NO;
}

- (BOOL)isKindOfClass:(Class)cls {
    for (Class tcls = [self class]; tcls; tcls = tcls->superclass) {
        if (tcls == cls) return YES;
    }
    return NO;
}
```

```
+ (BOOL)isMemberOfClass:(Class)cls {
    return object_getClass((id)self) == cls;
}

- (BOOL)isMemberOfClass:(Class)cls {
    return [self class] == cls;
}
```

这里需要注意的是，对于`+ (BOOL)isKindOfClass:(Class)cls`和`+ (BOOL)isMemberOfClass:(Class)cls`，其实现里面会再去取cls所属的class:

```
Class tcls = object_getClass((id)self)
```

对于类来说，这时候tcls是元类。

知道了runtime的底层实现，再结合下面这张经典的图，就可以给出答案：

答案会输出

```
1 0 0 0
```

2. 下面代码会输出什么？编译错误？运行时crash？

```
@interface NSObject (Sark)
```

```
+ (void)foo;
@end

@implementation NSObject (Sark)
- (void)foo {
    NSLog(@"AAAA");
}
@end

int main(int argc, const char * argv[]) {
    @autoreleasepool {
        [NSObject foo];
        [[NSObject new] foo];
    }
    return 0;
}
```

解答：

根据上面的图以及类方法会调用会在类的元类中查找`IMP`。 当调用`[NSObject foo]`时，会到`NSObject meta class`即`Root meta class`中查找是否有对应的`foo`实现。没有，则会到`Root meta class`的`super class`，也就是`NSObject class`中寻找对应的实现。而在`NSObject class`中，我们定义了`foo`的实现（不必区分是+还是-实现），因此，`[NSObject foo]`会调用`NSLog(@"AAAA");`实现。

当调用`[[NSObject new] foo]`时，则会到`NSObject`中寻找对应的实现。这个自然也会调用`NSLog(@"AAAA");`但是，在`Xcode`中做了一个检查，当在头文件中没有声明对应的方法时，会报错。而这里只有`+ foo`,却没有`- foo`，因此，会报错。但反过来，如果只定义了`- foo`，确可以用类方法形式调用`[NSObject foo]`的。

3. 试分析如下代码

```
#import <UIKit/UIKit.h>
@interface Student : NSObject
@property(nonatomic, strong) NSString *name;
@property(nonatomic, strong) NSNumber *age;
@end

@interface Sark : NSObject
@property(nonatomic, strong) NSString *name;
@property(nonatomic, assign) double fNum;
@property(nonatomic, strong) Student *myStudent;
@property(nonatomic, strong) NSNumber *age;

- (void)speak;
@end

@interface ViewController : UIViewController
```

```
@end
```

```
#import "ViewController.h"

@implementation Student



@end


@implementation Sark
- (instancetype)init {
    if (self = [super init]) {
        _name = @"Lily";
    }
    return self;
}
- (void)speak {
    NSLog(@"Instance address is %p", self);    //  此处如果输出 Instance address is 0x280589860
    // 那么以下该输出什么？？？
    NSLog(@"name address is %p", &_name);
    NSLog(@"fNum address is %p", &_fNum);
    NSLog(@"myStudent address is %p", &_myStudent);
    NSLog(@"age address is %p", &_age);
}
@end

@interface ViewController ()

@end

@implementation ViewController

- (void)viewDidLoad {
    [super viewDidLoad];
    Sark *sark = [Sark new];
    Sark *sark2 = [Sark new];
    NSLog(@"sark 地址=%p", &sark); //  此处如果输出 sark 地址=0x16f60d348
    NSLog(@"sark2 地址=%p", &sark2); // 那么 sark2 地址=??
    [sark speak];
    [sark2 speak];
}
```

解答：

这个问题实质上是考察OC内存分配时，栈和堆的区别，以及OC类实例的内存布局。（1）对于临时变量，如sark2指针，OC会分配到栈上，而OC中的类实例则是分配在堆上的。（2）在栈上的地址是从高向低分配的，在堆上的地址是从低到高分配的。（3）OC类实例的内存布局是线性布局，会按照实例变量的声明顺序，依次排列。这里的实例变量都是指针类型，64位操作系统中，指针占用8个字节。

请结合注释和结果分析本题：

```
#import "ViewController.h"

@implementation Student

@end


@implementation Sark
- (instancetype)init {
    if (self = [super init]) {
        _name = @"Lily";
    }
    return self;
}
- (void)speak {
    // OC的对象 内存在堆上
    // 堆： 由低向高分配内存
    NSLog(@"Instance address is %p", self);
    NSLog(@"name address is %p", &_name);
    NSLog(@"fNum address is %p", &_fNum);
    NSLog(@"myStudent address is %p", &_myStudent);
    NSLog(@"age address is %p", &_age);

}
@end

@interface ViewController ()

@end

@implementation ViewController

- (void)viewDidLoad {
    [super viewDidLoad];
    // 对象指针 分配在栈上
    // 栈： 由高向低分配内存
    NSLog(@"对象指针 在栈上分配内存----------------------");
    Sark *sark = [Sark new];
    Sark *sark2 = [Sark new];
    NSLog(@"sark 地址=%p", &sark); // 栈上地址
    NSLog(@"sark2 地址=%p", &sark2); // 栈上地址
    NSLog(@"");
```

```
    NSLog(@"对象 在堆上分配内存--------------------");
    NSLog(@"sark--------------------");
    [sark speak];
    NSLog(@"");
    NSLog(@"sark2--------------------");
    [sark2 speak];
}

@end
```

打印结果如下：

4. 下面的代码会：`compile error/runtime crash/`正常运行?

```
@interface Sark : NSObject
@property(nonatomic, strong) NSString *name;
- (void)showYourName;
@end

@implementation Sark
- (void)showYourName {
    NSLog(@"My name is %@", self.name);
}
@end


@interface ViewController : UIViewController
@end

@implementation ViewController
- (void)viewDidLoad {
    [super viewDidLoad];
    Class cls = [Sark class];
    void* obj = &cls;
    [(__bridge id)obj showYourName];
}
@end
```

解答： 这个问题首先，来判断是否会`crash`。核心是分析

```
[(__bridge id)obj showYourName];
```

是否会`crash`。

这里如果`obj`是`Sark`的一个实例，则不会`crash`。但此时`obj`是

```
    void* obj = &cls;
```

也就是`Sark Class`的一个指针。根据`Class`在runtime中的定义:

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

obj定义是

```
 void* obj = &(objc_class *);
```

即

```
 obj = objc_class **;
```

而我们生成一个Sark对象则是:

```
Sark *sarkObj = [Sark new];
```

我们将Sark用NSobject替换:

```
Sark *sarkObj = [NSObject new];
```

即

```
Sark *sarkObj = NSObject *;
```

而NSObject定义是:

```
@interface NSObject <NSObject> {
#pragma clang diagnostic push
#pragma clang diagnostic ignored "-Wobjc-interface-ivars"
    Class isa  OBJC_ISA_AVAILABILITY;
#pragma clang diagnostic pop
}
```

可以用`Class isa`来替代`NSObject`:

```
Sark *sarkObj = Class *;
```

即

```
Sark *sarkObj = objc_class**;
```

根据

```
obj = objc_class **;
```

和

```
Sark *sarkObj = objc_class**;
```

可知

```
obj == sarkObj
```

很奇妙！这里涉及到OC中对象的本质：

> Objc中的对象是一个指向objc_class类型的指针，即 id obj = objc_object * ， 而对象的实例变量地址 void *ivar = obj + offset(N)

为了验证结果，我们来做个试验，有如下代码：

```
@interface Sark : NSObject
@property(nonatomic, strong) NSString *name;
@property(nonatomic, strong) NSNumber *age;

- (void)speak;
@end

@implementation Sark
- (instancetype)init {
    if (self = [super init]) {
        _name = @"Lily";
    }
    return self;
}
- (void)speak {
    // OC的对象 内存在堆上
    // 堆： 由低向高分配内存
    NSLog(@"Instance address is %p", self);    // self表示当前实例的this指针，这里应该输出实例的地址（堆地址）
    NSLog(@"name address is %p", &_name);  //name 实例变量地址
    NSLog(@"age address is %p", &_age);   // age实例变量地址

}
@end
```

```
- (void)viewDidLoad {
    [super viewDidLoad];
    // 对象指针 分配在栈上
    // 栈:  由高向低分配内存
    id sark = [Sark new];
    NSLog(@"&sark = %p", &sark);    // 按照冰霜的结论，此处应该输出实例的地址 ("%p", self)
    NSLog(@"sark = %p", sark);     // 按照本文结论，此处应该输出实例的地址 ("%p", self)
    [sark speak];
}
```

输出结果为：

可以看到，只有按照本文中的方式

```
NSLog(@"sark = %p", sark);
```

输出了实例的地址`0x281b00840`，并且也符合结论：

> 对象的实例变量地址 void *ivar = obj + offset(N)

此时`N=8 bytes`，因为这里的实例变量都是指针类型，而指针在`64位机器`中占据`8个bytes`。

为了更好的了解上面关于OC实例变量地址的问题，我们来看一下，当我们调用方法

```
- (void)viewDidLoad {
    ...
    id sark = [Sark new];
    ...
}
```

系统是怎么分配内存的。

我们知道，在函数中生成的变量，被称作“临时变量”，“临时变量”内存是分配在栈上的。在栈上的变量是不需要我们手动管理内存的，系统会自动释放栈上的变量。在这里id类型的sark，就是一个栈变量。sark的实质是一个指针，它指向Sark类实例。

而在等号的右边，我们调用了[Sark new]，这会生成一个Sark 类实例对象。而在OC中，类实例对象是在堆内存上创建的。在堆上的变量，是需要我们手动管理内存的。而这里为什么没有内存管理相关的代码呢？这是因为ARC的存在，编译器会自动在我们代码的合适位置插入retain和release代码。但仍要明确，OC类实例对象是分配在堆上的。

```
- (void)viewDidLoad {
    ...
    id sark = [Sark new];
    ...
}
```

可以理解为：

> 在等号右边生成了一个堆上的实例对象，并会把堆地址返回给等号左边，id sark这个指针也就存储了这个堆地址。

在图中，我们还看到了`self`，其本质也是一个`指向实例的指针`，当我们调用`self`方法的时候，系统会在`栈`上为我们生成一个`指向Sark实例`的`指针`，指向`Sark实例的堆地址`。