# IOS内存管理

* 内存概述
  + 内存是用来存啥的？
  + 内存布局
  + 哈希表
  + 垃圾回收（GC）
* IOS内存管理机制
  + MRC & ARC
  + TaggedPointer & NONPOINTER\_ISA
  + 引用计数表 & 弱引用表
  + 自动释放池
* 循环引用
  + 分类
  + 如何破除循环引用
  + 循环引用实例
* 内存概述
  + 内存是用来存啥的呢？
    - 答案：指令+数据！
  + 内存布局：对于程序运行过程中的内存使用，堆和栈一般是相向扩展的



内核区：命令行参数，环境变量等。

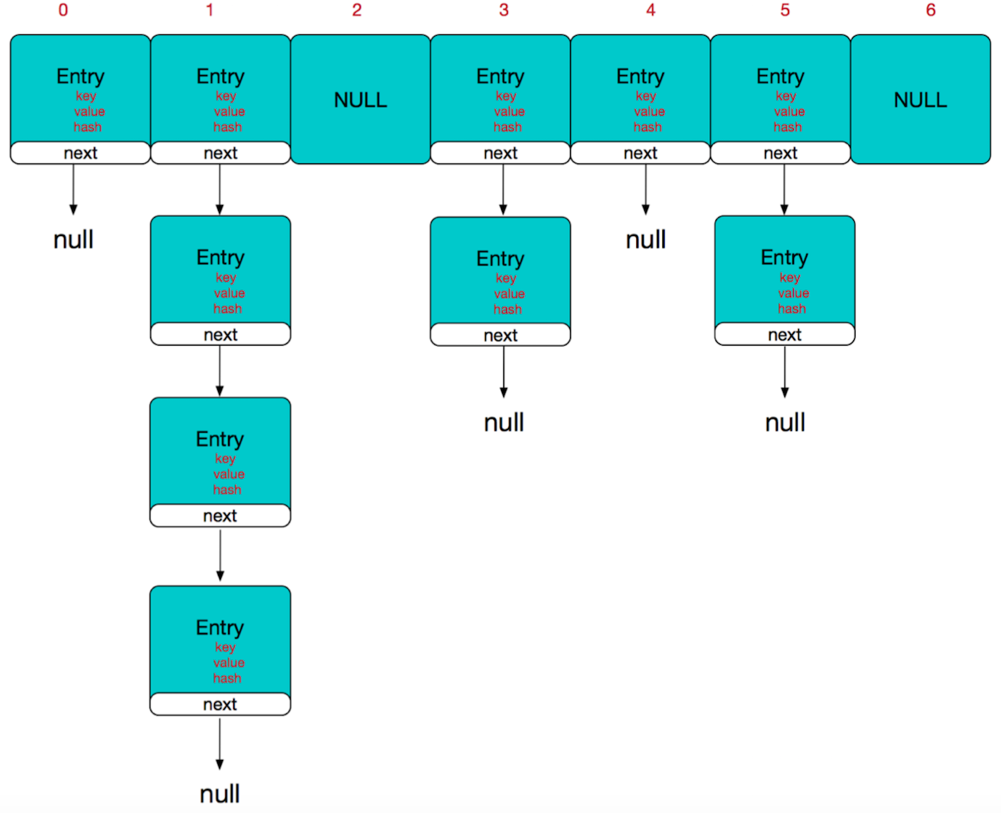
栈：编译的时候能确定好的。函数内部使用的变量、函数的参数以及返回值；由编译器自动分配和释放；后进先出；从高往低分配。

堆：编译时不能提前确定，通常堆中的对象都是以指针来访问的，指针从线程栈中来，但不独属于某个线程。它的大小并不固定，可动态扩张或缩减。一般由程序员分配和释放。动态调用malloc和free来分配和释放内存；从低往高分配。

数据：.BSS(未初始化的全局变量、静态变量，系统初始化为0)；RW data(已初始化的全局变量、静态变量)；RO data(常量) 。

代码：程序编译后的机器码；这部分区域的大小在程序执行前就已经确定，而且内存区域属于只读。

总结：代码段（Code）、只读数据段（RO data）、读写数据段（RW Data）、未初始化数据段（BSS）属于静态区域。代码段 和 已初始化的数据段(RO/RW data) 都在可运行文件里，由系统从可运行文件里载入；而BSS段不在可运行文件里，由系统初始化。而堆和栈作为动态区域, 在程序运行的过程中分配和释放。 (也就是说，一个可执行程序分为映像和运行两种状态。在编译链接后形成的映像中，将只包含代码段（text）、只读数据段（RO data）和读写数据段（RW data）。在程序运行之前加载的过程中，将动态生成未初始化数据段（BSS），在程序运行时将动态生成堆（Heap）和栈（Stack）区域。）

* + 哈希表（也叫散列表）
    - 哈希表的主干是数组。比如我们要新增或查找某个元素，可以使用哈希函数（这个哈希函数的设计直接影响到哈希表的优劣）根据该元素的关键字直接映射到数组中的某个位置，通过这个位置下标一次定位就可以完成操作。
    - 哈希冲突：对两个或多个元素进行哈希计算得到同一个存储地址。再好的函数也不能保证计算得到的内存地址绝对不会发生冲突。哈希表采用了链地址法（链表）来解决哈希冲突。
    - 哈希表的整体结构如下：
    - 
  + 垃圾回收（GC）
    - GC 将内存中的对象主要分成两个区域：Young 区和 Old 区。对象先在 Young 区被创建，然后如果经过一段时间还存活着，则被移动到 Old 区。
    - Young 区的对象因为大部分生命期都很短，每次回收之后只有少部分能够存活，所以采用的算法叫 Copying 算法，简单说来就是直接把活着的对象复制到另一个地方。Young 区内部又分成了三块区域：Eden 区 , From 区 , To 区。每次执行 Copying 算法时，即将存活的对象从 Eden 区和 From 区复制到 To 区，然后交换 From 区和 To 区的名字（即 From 区变成 To 区，To 区变成 From 区）。
    - Old 区的对象因为都是存活下来的老司机了，所以如果用 Copying 算法的话，很可能 90% 的对象都得复制一遍了，不划算。所以 Old 区的回收算法叫 Mark-Sweep 算法。简单来说，就是只是把不用的对象先标记（Mark）出来，然后回收（Sweep），活着的对象就不动它了。因为大部分对象都活着，所以回收下来的对象并不多。但是这个算法会有一个问题：它会产生内存碎片，所以它一般还会带有整理内存碎片的逻辑，在算法中叫做 Compact，其实就是把对象插到这些空的位置里。
    - 如何找出需要回收的垃圾对象？为了避免 ARC 解决不了的循环引用问题，GC 引入了一个叫做「可达性」的概念，应用这个概念，即使是有循环引用的垃圾对象，也可以被回收掉。当 GC 工作时，GC 认为当前的一些对象是有效的，这些对象包括：全局变量，栈里面的变量等，然后 GC 从这些变量出发，去标记这些变量「可达」的其它变量，这个标记是一个递归的过程，最后就像从树根的内存对象开始，把所有的树枝和树叶都记成可达的了。那除了这些「可达」的变量，别的变量就都需要被回收了。
* IOS内存管理机制
  + MRC & ARC
    - MRC
      * alloc：用来分配一个对象的内存空间
      * retain：引用计数加1
      * release：引用计数减1
      * retainCount：获取当前对象的引用计数值
      * autorelease：当前对象会在NSAutoreleasePool结束的时候调用它的release操作，进行引用计数减1
      * dealloc：显式调用[super dealloc]
      * copy：
      * mutablleCopy:
    - ARC：由LLVM和Runtime共同协作来进行自动引用计数的。
      * 禁止手动调用retain、release、retainCount、autorelease，可以重写某个对象的dealloc方法，但不能显式调用[super dealloc]。
      * 使用 @autoreleasepool 块替代 NSAutoreleasePool。
      * 同时使用ObjC指针和CFTypeRef指向的对象时，由于CoreFoundation框架不支持ARC，所以除了转换类型，还需指定内存管理所有权的改变：
        + \_\_bridge 只声明类型转变，不做内存管理规则的转变。如以下例子，依然要用 Objective-C 类型的 ARC 来管理 s1，你不能用 CFRelease() 去释放 s1。

CFStringRef s1 = (\_\_bridge CFStringRef) [[NSString alloc] initWithFormat:@"Hello, %@!", name];

* + - * + \_\_bridge\_retained（或CFBridgingRetain） 表示将指针类型转变的同时，将内存管理的责任由原来的 Objective-C 交给Core Foundation 来处理，也就是，将 ARC 转变为 MRC。如下代码，我们在第二行做了转化，这时内存管理规则由 ARC 变为了 MRC，我们需要手动的来管理 s2 的内存，而对于 s1，我们即使将其置为 nil，也不能释放内存。

NSString \*s1 = [[NSString alloc] initWithFormat:@"Hello, %@!", name];

CFStringRef s2 = (\_\_bridge\_retained CFStringRef)s1;

// or CFStringRef s2 = (CFStringRef)CFBridgingRetain(s1);

// do something with s2

//...

CFRelease(s2); // 注意要在使用结束后加这个

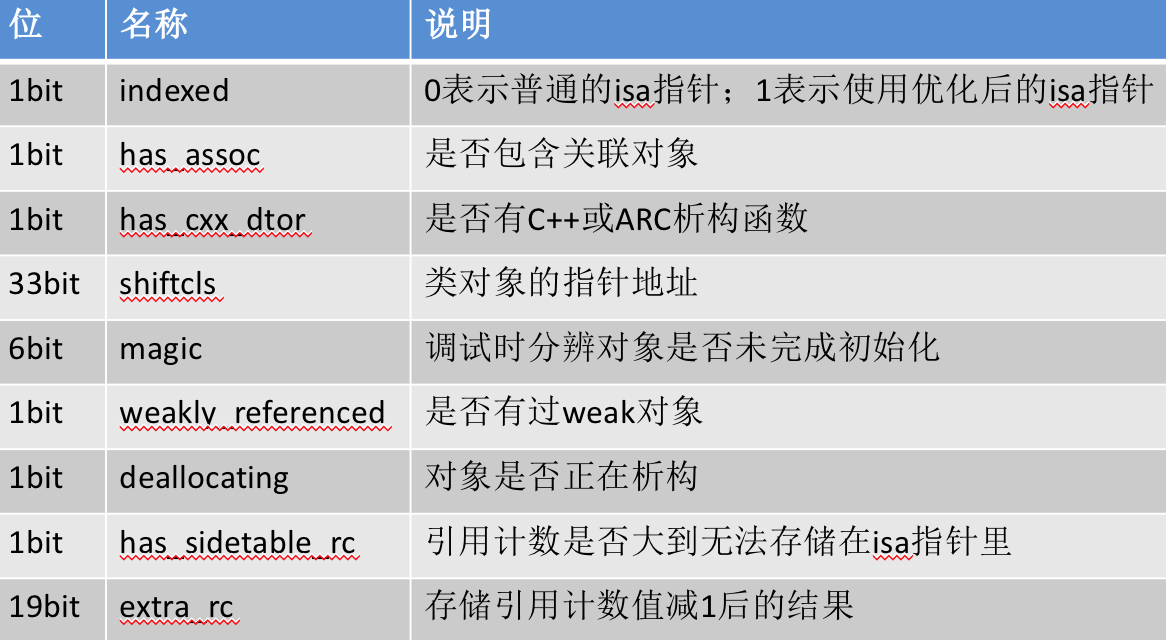
* + - * + \_\_bridge\_transfer（或CFBridgingRelease） 这个修饰符和函数的功能和上面那个 \_\_bridge\_retained 相反，它表示将管理的责任由 Core Foundation 转交给 Objective-C，即将管理方式由 MRC 转变为 ARC。例如，这里我们将 result 的管理责任交给了 ARC 来处理，我们就不需要再显式地将 CFRelease() 了。

CFStringRef result = CFURLCreateStringByAddingPercentEscapes(. . .);

NSString \*s = (\_\_bridge\_transfer NSString \*)result;

//or NSString \*s = (NSString \*)CFBridgingRelease(result);

return s;

* + - * ARC下新增的所有权修饰符：
        + \_\_strong 表示强引用，对应定义property时用strong。对象类型默认都是strong。
        + \_\_weak 表示弱引用，对应定义property时用weak。对象被释放时，所有指向它的弱引用都会被置为nil，这样可以防止野指针。因为weak不会引起对象的引用计数变化，因此，该对象在运行过程中很有可能会被释放。所以，需要将对象注册到自动释放池中并在自动释放池销毁时释放对象占用的内存。如果大量使用weak的话，在我们去访问weak修饰的对象时，会有大量对象注册到自动释放池,这会影响程序的性能。推荐方案 ： 要访问weak修饰的变量时，先将其赋给一个strong变量，然后进行访问。
      * ARC相对于GC的优点：
        + ARC 工作在编译期，在运行时没有额外开销。
        + ARC 的内存回收是平稳进行的，对象不被使用时会立即被回收。而 GC 的内存回收是一阵一阵的，回收时需要暂停程序，会有一定的卡顿。
      * ARC相对于GC的缺点：
        + GC 真的是太简单了，基本上完全不用处理内存管理问题，而 ARC 还是需要处理类似循环引用这种内存管理问题。
        + GC 一类的语言相对来说学习起来更简单。
  + TaggedPointer & NONPOINTER\_ISA
    - 32位机上指针占4个字节，所以一个对象包含两个指针，占8个字节；64机上指针占8个字节，所以表示同一个对象需要16个字节，翻倍了，其实本来8个字节就够了。因此，苹果引入了Tagged Pointer和NONPOINTER\_ISA来实现64位机上的内存优化。
    - TaggedPointer类型的对象，它的isa指针为空，不会指向另一块存储空间，因为它的值包含在这个指针本身里面了。对于一些小对象使用TaggedPointer这种内存管理方案。比如NSNumber对象， 11位长度以下的NSString对象。
    - NONPOINTER\_ISA在64位机上，对象的isa区域不再只是一个指向另一块存储空间的指针。还包含了更多信息，比如引用计数，析构状态，被其他 weak 变量引用情况等。如果引用计数超过了当前指针所能表示的范围，Runtime 会使用一张散列表来管理用计数。

struct {//objc-private.h

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)

};

* + - 代码分析内存：
      * 常量字符串: isa + ? + 内容地址 （常量的isa不遵循以上nonpointer\_isa规则，可能是因为常量保存在常量区，而不是堆区，引用计数无穷大）;copy后，跟被copy对象一样，同一个常量字符串；mutableCopy后，变成一个分配了新地址的mutableString。
      * taggedPointerString: 直接存放内容，0xa000000006362613（没有isa）；copy后，跟被copy对象一样，同一个taggedPointerString；mutableCopy后，变成一个分配了新地址的mutableString。
      * NSString：isa + ?（似乎引用计数存放在该字节高32位里，而不是isa里） + 内容（内容里最低字节存放字符长度）;copy后，跟被copy对象一样；mutableCopy后，变成一个分配了新地址的mutableString。
      * mutableString: isa + ?（似乎引用计数存放在该字节高32位里，而不是isa里） + 内容地址（内容里最低字节存放字符长度）；copy后，变成一个taggedPointerString（11位以下长度）或NSString（11位以上长度）；mutableCopy后，变成另外一个mutableString。appendString时，可能会涉及扩容问题，此时会分配新地址存放内容，并销毁旧地址，同时它的存储区的内容地址会变成新分配的地址。
      * NSArray: isa + count + 内容（这个内容是数组里所有对象的地址的顺序排列）；copy后，跟被copy的对象指向的是同一个地址；mutableCopy后，生成一个新的NSMutableArray，但其内部所有对象的地址并未改变。
      * NSMutableArray: isa + ? + 内容地址（在内容地址里存放的是数组里所有对象的地址的顺序排列）; copy后，变成了一个分配了新地址的NSArray，但其内部所有对象的地址并未改变；mutableCopy后，生成以一个新的堆地址来存放跟被拷贝对象一样的内容（isa + ? + 内容地址），但是再给该mutableArray里添加对象的时候，其中存放的“isa + ？ + 内容地址”会发生变化。无论是被mutableCopy的对象，还是mutableCopy后的对象，在添加对象的时候，其内容地址都有可能发生变化，这可能涉及到扩容的问题，添加对象时，系统开辟另一块存储空间，将之前的所有对象拷贝过去，再添加新对象，同时销毁之前的存储空间。
      * SingleEntryDictionary: isa + value + key；copy后，跟被copy的对象指向的是同一个地址；mutableCopy后，生成一个新的NSMutableDictionary，但其内部所有key，value的地址并未改变。
      * NSDictionary: isa + ？（似乎低32位存放的是键值对个数）+ key + value + key + value +…(三个以上个数似乎不是这样的排列); copy后，跟被copy的对象指向的是同一个地址；mutableCopy后，生成一个新的NSMutableDictionary，但其内部所有key，value的地址并未改变。
      * NSMutableDictionary: isa + 内容地址(内容里的键值对地址的排列规则还待进一步验证) + ？（似乎包含了键值对个数）；copy后，重新分配了新的地址存放新的isa + 内容地址（内容地址跟被拷贝前的一样）；mutableCopy后，生成以一个新的堆地址来存放跟被拷贝对象一样的内容（isa + 内容地址 + ？），但是再给该mutableDictionary里添加键值对的时候，其中存放的“isa + 内容地址 + ？”会发生变化。
  + 引用计数表 & 弱引用表
    - SideTables包括了多个SideTable，在不同系统架构中SideTable的个数是不同的；SideTables是哈希表，可以通过一个对象的指针来找到具体的引用计数表或弱引用表在哪一个具体的SideTable中。
    - 为什么用多个SideTable？ 如果只有一个table，意味着内存中分配的所有对象都要在一个表中操作，因为多个线程可能同时操作这个表，所以就要对这个表加锁，如果并发操作这个表的线程有成千上万个，就会产生效率问题。所以系统引入了分离锁这样一个技术方案，把大表拆成多个小表来进行操作，分别对小表加锁，从而提升效率。
    - 自旋锁：
      * 自旋锁：是“忙等”的锁。由于自旋时不释放CPU，因而持有自旋锁的线程应该尽快释放自旋锁，否则其他等待该自旋锁的线程会一直自旋，从而浪费CPU时间。
      * 自旋锁适用于那些仅需要阻塞很短时间的场景。
      * 自旋锁主要用来防止多处理器中并发访问临界区，防止内核抢占造成的竞争。如果持有自旋锁的代码sleep了就可能导致整个系统挂起。
      * 自旋锁与互斥锁的比较
        + 使用任何锁都需要消耗系统资源（内存资源和CPU时间），这种资源消耗可以分为两类：

建立锁所需要的资源

当线程被阻塞时所需要的资源

* + - * + 对于自旋锁来说，它只需要消耗很少的资源来建立锁；随后当线程被阻塞时，它就会一直重复检查看锁是否可用了，也就是说当自旋锁处于等待状态时它会一直消耗CPU时间。
        + 而对于另一种常见的互斥锁来说，与自旋锁相比它需要消耗大量的系统资源来建立锁；随后当线程被阻塞时，线程的调度状态被修改，并且线程被加入等待线程队列；最后当锁可用时，在获取锁之前，线程会被从等待队列取出并更改其调度状态；但是在线程被阻塞期间，它不消耗CPU资源。
        + 自旋锁和互斥锁适用于不同的场景。互斥锁适用于那些可能会阻塞很长时间的场景。举个例子：如果只有一行代码需要加锁，那么使用自旋锁耗费的时间会比互斥锁少；而如果有一个需要跑10次的循环需要加锁，那么互斥锁应该会是更好的选择。
    - 引用计数表：
      * alloc: 通过一系列函数调用，最终直接调用calloc，不会进行引用计数加1,见retainCount讲解。//NSObject.mm alloc
      * retain: 判断是否nonpointer，如果不是，直接去操作引用计数表，如果是，先看isa引用计数位加1后是否越界，如果越界，留下一半引用计数在isa的引用计数位，其他复制到引用计数表去。//NSObject.mm retain
      * release: 与retain相反的过程。//NSObject.mm release
      * retainCount: 是nonpointer，isa引用计数位+引用计数表引用计数位+1；否则，引用计数表引用计数位+1。都要加1，所以说alloc时引用计数并没有加1。//NSObject.mm retainCount
      * dealloc: nonpointer\_isa,weakly\_referenced,has\_assoc,has\_cxx\_dtor,has\_sidetable\_rc 这些都为false的时候，才可以直接free，如果has\_cxx\_dtor，需要destruct，如果hasAssociatedObjects，需要移除关联对象，不是nonpointer，直接从弱引用表移除弱引用和擦除引用计数表里的引用计数，是nonpointer，需要再判断weakly\_referenced和has\_sidetable\_rc来决定要不要移除弱引用，要不要擦除引用计数表。//NSObject.mm dealloc
      * 引用计数表，使用哈希表来实现，它的插入和获取都是通过同一个哈希算法来得到一个size\_t类型的值的，避免了for循环的使用，从而保证了操作效率。哈希算法找到的位置，实际上是一个unsighed long 型的变量。
      * 
    - 弱引用表：
      * \_\_weak的对象通过调用initWeak和storeWeak来添加到弱引用表。//NSObject.mm storeWeak
      * 如何将weak对象设为nil：dealloc -> weak\_clear\_no\_lock(从弱引用表中取出所有该对象的weak\_entry\_t对象，weak\_entry\_t里存放了weak\_referrer\_t数组，就是所有的弱引用，全部设为nil，然后再移除weak\_entry\_t对象)
      * 弱引用表，也是一个哈希表。对象指针通过一个哈希函数的运算，得到一个weak\_entry\_t， weak\_entry\_t 的结构和weak\_table\_t有一点点类似，基本也是一个HashTable的实现。
      * weak\_entry\_t有一个巧妙的设计，即如果一个对象对应的弱引用数目较少的话(<=WEAK\_INLINE\_COUNT，runtime把这个值设置为4)，则其弱引用会被依次保存到一个inline数组里。这个inline数组的内存会在weak\_entry\_t初始化的时候一并分配好，而不是需要用到的时候再去申请新的内存空间，从而达到提到运行效率的目的。
      * 参考地址： https://www.jianshu.com/p/045294e1f062
    - 代码实现：

static StripedMap<SideTable>& SideTables() {//NSObject.mm

return \*reinterpret\_cast<StripedMap<SideTable>\*>(SideTableBuf);

}

template<typename T>//objc-private.h

class StripedMap {

enum { CacheLineSize = 64 };

#if TARGET\_OS\_EMBEDDED

enum { StripeCount = 8 };

#else

enum { StripeCount = 64 };

#endif

struct PaddedT {

T value alignas(CacheLineSize);

};

PaddedT array[StripeCount];

static unsigned int indexForPointer(const void \*p) {

uintptr\_t addr = reinterpret\_cast<uintptr\_t>(p);

return ((addr >> 4) ^ (addr >> 9)) % StripeCount;

}

…}

struct SideTable {

spinlock\_t slock;

RefcountMap refcnts;

weak\_table\_t weak\_table;

…}

typedef objc::DenseMap<DisguisedPtr<objc\_object>,size\_t,true> RefcountMap;

struct weak\_table\_t {//objc-weak.h

weak\_entry\_t \*weak\_entries;

size\_t num\_entries;

uintptr\_t mask;

uintptr\_t max\_hash\_displacement;

};

struct weak\_entry\_t {

DisguisedPtr<objc\_object> referent;

union {

struct {

weak\_referrer\_t \*referrers;

uintptr\_t out\_of\_line\_ness : 2;

uintptr\_t num\_refs : PTR\_MINUS\_2;

uintptr\_t mask;

uintptr\_t max\_hash\_displacement;

};

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;

}

}

};

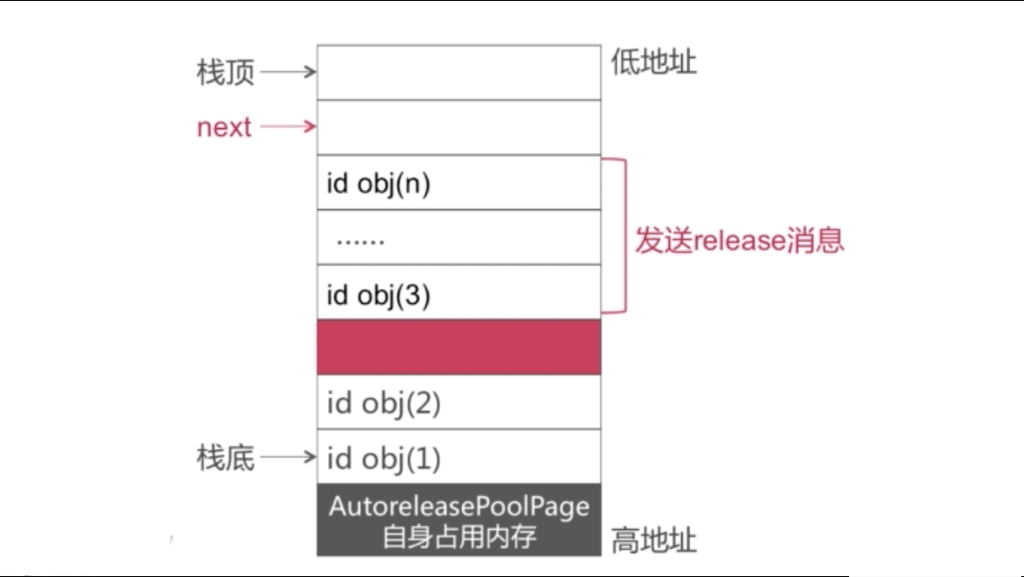
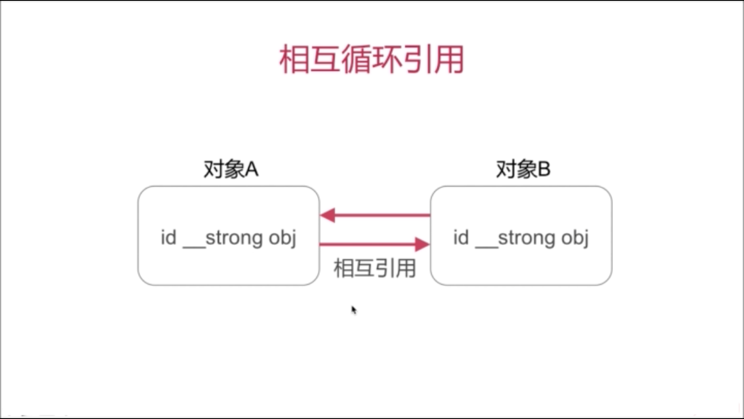
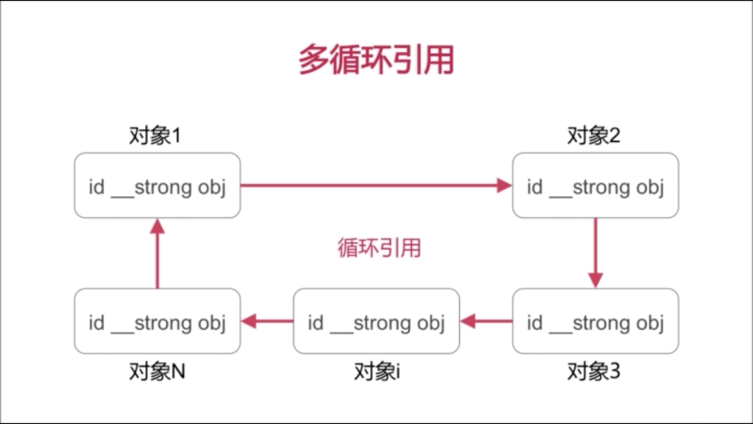
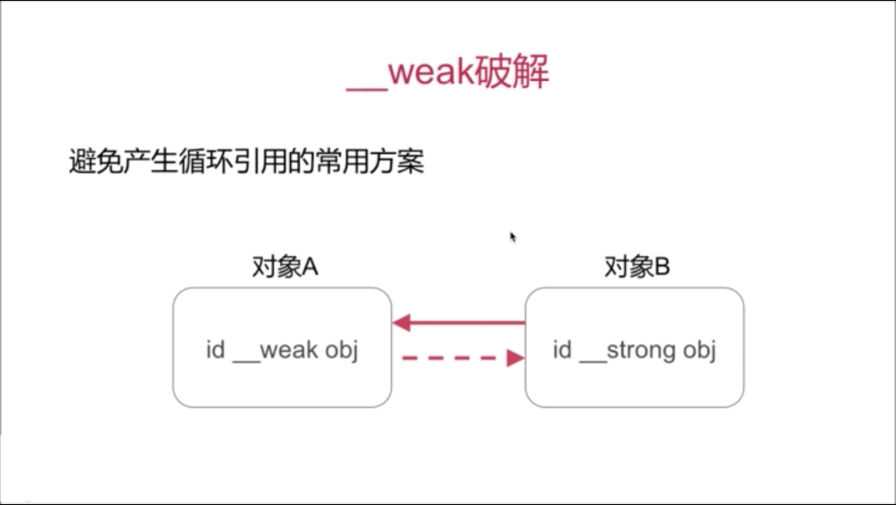
typedef DisguisedPtr<objc\_object \*> weak\_referrer\_t;

* + 自动释放池
    - Main函数自动添加了@autoreleasepool{}; 很多次循环的内部最好自己添加@autoreleasepool{},以及时释放其内部的临时对象。
    - 实现原理
      * 以栈为节点通过双向链表的形式组合而成的。

id \*next;//NSObject.mm AutoreleasePoolPage

pthread\_t const thread;

AutoreleasePoolPage \* const parent;

* + - * AutoreleasePoolPage \*child;
        + objc\_autoreleasePoolPush (插入哨兵对象)
        + objc\_autoreleasePoolPop (批量的释放操作)
    - 
    - 总结
      * 在当次runloop将要结束的时候调用AutoreleasePoolPage::pop()。
      * 多层嵌套就是多次插入哨兵对象。
      * 在for循环中alloc图片数据等内存消耗较大的场景手动插入autoreleasePool。
* 循环引用
  + 分类
    - 自循环引用 （block）
      * 
    - 相互循环引用（delegate）
      * 
    - 多循环引用
      * 
  + 如何破除循环引用
    - 使用\_\_weak所有权修饰符
      * 
    - 使用\_\_block
      * MRC下，\_\_block修饰对象不会增加其引用计数，避免了循环引用。
      * ARC下，\_\_block修饰对象会被强引用，无法避免循环引用，需手动解环。
        + @implementation TestObj
        + - (id)init {
        + self = [super init];
        + \_\_block id tmp = self;
        + self.attributBlock = ^{
        + NSLog(@"Self = %@",tmp);
        + tmp = nil;
        + };
        + }
        + - (void)execBlock {
        + self.attributBlock();
        + }
        + @end
        + // 使用类
        + id obj = [[TestObj alloc] init];
        + [obj execBlock]; // 如果不调用此方法，tmp 永远不会置 nil，内存泄露会一直在
    - 使用\_\_unsafe\_unretained
      * 修饰对象不会增加其引用计数，避免了循环引用。
      * 如果被修饰对象在某一时机被释放，会产生悬垂指针（导致内存泄漏）。
  + 循环引用示例
    - NSTimer
      * 例子
    - delegate
    - block
      * 类型：
        + \_NSConcreteGlobalBlock

存储在已初始化数据区

Copy后，什么也不做

* + - * + \_NSConcreteStackBlock

存储在栈区，作用域结束会被销毁

Copy后，存放到堆上，作用域结束后栈上的block会被销毁，堆上的仍然存在

* + - * + \_NSConcreteMallocBlock

存储在堆区

Copy后，增加引用计数

* + - * 例子