```
RAM: 运行内存,CPU可以直接访问,读写速度非常快,但是不能掉电存储
                                     速度慢一点,需要定期的刷新(充电),我们常说的内存条就
                       1. 动态DRAM
                                     是指它,价格会稍低一点,手机中的运行内存也是指它
               RAM
                                     速度快,我们常说的一级缓存,二级缓存就是指它,当然价
                       2. 静态SRAM
                                     格高一点
                       RAM类型不具备掉电存储能力(即一停止供电数据
                       全没了,从新上电后全是乱码,所以需要初始化)
                                                             app程序一般存放于ROM中
                       ROM: 存储性内存,可以掉电存储,例如SD卡、Flash(机械磁盘也可以简单的理解为ROM)
                                                   需要文件系统/驱动程序(嵌入
                       手机里面使用的ROM基本都是
               ROM
                                                   式中的EMC)将其读到RAM里
                       NandFlash, CPU不可以直接访问
                                                   面,CPU才可以访问
                       地址从 高->低: 栈区->堆区->全局区->常量区->代码区
                                                高地址
                                                由系统管理,会自动存一些局部变量,函数
                                                                                 局部变量,递归,函数循环调
                                栈区 (stack):
                                                跳转跳转时现场保护(寄存器值保存于恢复)
                                                                                 用等等
                                                非手动作用区域
                                                程序员管理,比如alloc申请内存,free释放内存
                                堆区(heap):
                                                      全局变量和静态变量的存储区域
                                                       1. bss段
                                                                 未初始化的全局变量和静态变量
                                                      2. 数据段
                                                                  已经初始化的全局变量和静态变量
                                全局区(静态区 static)
                                                      已初始化 (int a = 10;) 的全局变量和静态变量在一块区域, 未初始
               内存分区(RAM)
                                                      化(int a;)的全局变量和未初始化的静态变量在相邻的另一块区域
                                                      程序员创建, 由系统系统释放
                                            常量字符串就是放在这里的,
                                常量区:
                                            还有const常量
                                            低地址
                                                               实际上不是一次性把数据复制
                                           app程序会拷贝到这里
                                                               到真实的物理内存中, 而是一
                                代码区:
                                                               开始在虚拟内存中
                                           系统把ROM里的APP部分加载代码(比如动态库,类方法属性等等)
                                           copy到RAM的代码区,这样运行时CPU就能直接访问相关代码了
                                 物理内存 (RAM)
                                                  真实的物理内存一般较小,比如2G
                                                    在copy代码时不是真的一下全部复制到RAM上,而是在ROM硬盘上规划出一个大小
                                                    为4G(32位的=2的32次方)空间当做虚拟内存,因为CPU并不能直接访问ROM,
               虚拟内存与物理内存
                                                     所有虚拟内存地址都对应页表项(建立好虚拟内存和磁盘文件之间的映射)
                                 虚拟内存 (ROM上)
                                                    CPU访问到虚拟页有效位为1,说明已经在物理内存中,如果是0:
                                                    找到对应磁盘位置,再复制到ARM内存中(达到一个过渡作用)
                       每个对象都有一个isa,它不再是简单的一个指针,64位操作系
                       统后,isa优化成了共用体(存储多种信息,包括引用计数)
                                  define ISA_BITFIELD
                                  uintptr_t nonpointer : 1; //指针是否优化过
                                  uintptr_t has_assoc : 1; //是否有设置过关联对象,如果没有,释放时会更快
                                  uintptr_t has_cxx_dtor : 1; //是否有C++的析构函数 (.cxx_destruct) , 如果没有,释放时会更快
                       isa定义:
                                  uintptr_t shiftcls : 33; //存储着Class、Meta-Class对象的内存地址信息
                                               : 6; //用于在调试时分辨对象是否未完成初始化
                                  uintptr_t magic
               isa
                                  uintptr_t weakly_referenced: 1; //是否有被弱引用指向过,如果没有,释放时会更快
                                  uintptr_t deallocating : 1; //对象是否正在释放
                                  uintptr_t has_sidetable_rc:1; //引用计数器是否过大无法存储在isa中
                                                : 19 //里面存储的值是引用计数器减1
                                  uintptr t extra rc
                       通过上面isa结构发现,内部一共64位,分别记录不同信
                                                                 系统也是根据相应状态进行下
                       息,isa记录者关联,析构,类信息(33位),弱引用
                                                                 一步处理
                       释放状态,引用计数 (1+19位)等等,不是简单的指针
                       引用计数超出范围时,部分计数移动到引用计数表(SideTables)里存储
                                     SideTables是全局散列
                                                          每个结构体包含一个引用计
                                                                                  项目一共64个引用计数
                                     表,包含64个结构体
                                                          数表和一个弱引用计数表
                                                                                  表和64个弱引用计数表
                                          SideTables散列表
                                                                引用计数表
                                          SideTable结构体1
                                            自旋锁1
                                            引用计数表1
                                            弱引用表1_
                                          SideTable结构体2
                                            自旋锁2
                                            引用计数表2
                                            弱引用表2
                                                                弱引用表
                                          SideTable结构体64
               引用计数表 (SideTables)
                                            自旋锁64
                                           引用计数表64
                                            弱引用表64
                                                    也是散列表(哈希)
                                     引用计数表?
                                                                   Key 是对象地址
                                                    存储的是字典:
                                                                   value 是计数值
                                                     也是散列表(哈希)
                                     弱引用计数表?
                                                                    对象地址
                                                     存储的是结构体
                                                                    对象的弱引用指针数组
                                                64位开始: 当NSNumber, NSString, NSData等值范围较小时存储Tagged
                                                Pointer 类型
                                                大小: 8个字节存储就可以了
                                                类型: tag类型 + data数据 标识
                       小对象(Tagged Pointer)
                                                释放:随着指针释放
                                                扩容: 当值超出范围后, 变为普通对象类型
                                                方法:调用方法时也是有区别的,不像正常msgsend流程
                                                                               0 释放
                                   iOS普通对象是通过isa(引用计数标记方式)来管理内存的
                                                                               > 0 持有
                                                         19位存储引用计数: 范围0~255
                                                         1位标记是否超出范围: 0, 1
                                    上面isa结构显示引用计数
                                                         假如标记=1,说明超出,此时就把引用计数设置为128,另外的128
                                                         个计数挪到引用计数表(SideTables)里,下次计数加一,还是在
                                                         isa里增加,直到溢出,如此循环
iOS内存管理
                                   引用计数最终调用objc的retain和release来操作:
               对象
                                   inline id
                                   objc_object::rootRetain()
                                     assert(!UseGC);
                       普通对象:
                                     //是小对象,不使用引用计数,直接返回,免去相关内存管理,性能高了很多
                                     if (isTaggedPointer()) return (id)this;
                                     return sidetable_retain(); //普通对象,计数+1
                                   inline bool
                                   objc_object::rootRelease()
                                     assert(!UseGC);
                                     if (isTaggedPointer()) return false; //释放
                                     return sidetable release(true); //释放
                                   当进行alloc, new, retain等引用计数+1
                                   当进行release, autorelease时都会进行计数-1或者dealloc, 清空
                         ARC是相对MRC来说的
                                 程序员手动管理引用计数,一一对应,retain了一次,最终都要
                         MRC
                                 release一次,这样才能释放
                                     系统帮我们管理了部分引用计数,比如使用了alloc或者retain,无需
                                     手动release,超出作用域,或者最后再dealloc里会进行释放,也就是
                                     系统帮我们在合适的地方进行了release
                                                弱引用,记录在弱引用表里的对象的弱引用指针数组中
               ARC?
                                                Block内部常使用,避免循环引用
                                       _weak
                         ARC干了啥
                                                计数=0时,系统自动把弱引用设nil,并清空所有的关联
                                                                作用域内系统会retain计数+1
                                                作用域强引用:
                                                                超出作用域后,会release计数-1
                                       _strong
                                                Block内部常使用,避免循环引用,同时避免weak修饰对象释
                                                放导致问题,延迟对象释放,等block执行完毕后再释放
                                   也只是对set、get进行加锁,其他方法依然不是线程安全的
                        原子性:
                                   因此,通常情况下还是要进行
               atomic
                                   额外加锁
                          -般使用nonatomic
                                          自己根据需要加锁
                       对内存数据可读read可写write能力
               读写
                                       Alloc的调用本质
                                                Alloc
                                      callAlloc(Class, bool
                                         bool)
                                                                 NO
                                                         canAllocFast
                                               allocWithZo
                                                                 返回错误
                                      allocWithZone的调用本
                                               allocWithZone
                                      _objc_rootAllocWithZo
                                                           YES
                                                         初始化isa
               内存管理-对象生成alloc
                                                         返回objc
                                                                 class_createInsta
                                                _OBJC2
                                                         zone == nil
                                                                  nceFromZone
                                                                        NO
                                                                   objc?
                                                         YES
                                                                    YES
                                                                   返回objc
                                                                         返回错误
                                              class_createInstanc
                                                objc?
                                                  YES
                                                      dealloc
                                                     TaggedPointer
                                                                返回
                                                        NO
                                                                    释放分配的内
                                                      isa相关判断
               内存管理-对象销毁dealloc
                                                        NO
                                                     object_dispose
                                               free
                                                             object_dispose
                                                   object_cxxDestruc
                                                                      清空引用计数表并清
                                                             删除关联对象
                                                                        除弱引用表
                                                  会执行cxx_destruct析构
                                                  函数,在ARC下对象的释放
                                                  也是通过析构函数完成
                                    1. 调用 -release : 引用计数变为零
                                       * 对象正在被销毁, 生命周期即将结束。
                                       * 不能再有新的 __weak 弱引用, 否则将指向 nil.
                                       * 调用 [self dealloc]
                                    2. 父类 调用 -dealloc
                                       * 继承关系中最底层的父类 在调用 -dealloc
                                       * 如果是 MRC 代码 则会手动释放实例变量们(iVars)
                                       * 继承关系中每一层的父类 都在调用 -dealloc
                                    3. NSObject 调 -dealloc
                                       * 只做一件事: 调用 Objective-C runtime 中的 object_dispose() 方法
                                    4. 调用 object_dispose()
                                       * 为 C++ 的实例变量们 (iVars) 调用 destructors
                                       * 为 ARC 状态下的 实例变量们(iVars) 调用 -release
                                       * 解除所有使用 runtime Associate方法关联的对象
                                       * 解除所有 __weak 引用
                                       * 调用 free()
```