iOS内存管理及优化

CHR移动端分享

李仲禹

引子

- 桌面操作系统中很少有应用因为使用内存过多而被Kill掉，为什么iOS会呢?
- 虚拟内存为何物？为什么有时它能超过物理总内存？虚拟内存占用过高会引来内存警告吗?
- Allocations中的Dirty Size和Resident Size分别指什么？All Heap & AnonymousVM是什么?
- iOS的内存管理机制是什么样的？它是基于什么原则来Kill掉进程的?
- 内存有分类吗？什么类型的内存可以回收?
- 我们了解自己的程序吗？什么地方占用内存多，什么地方可以优化？如何避免内存峰值过高?

目录

- 程序员对内存的关注点
- 基本概念及原理
- iOS内存管理
- 分析工具
- 最佳实践

程序员对内存的关注点

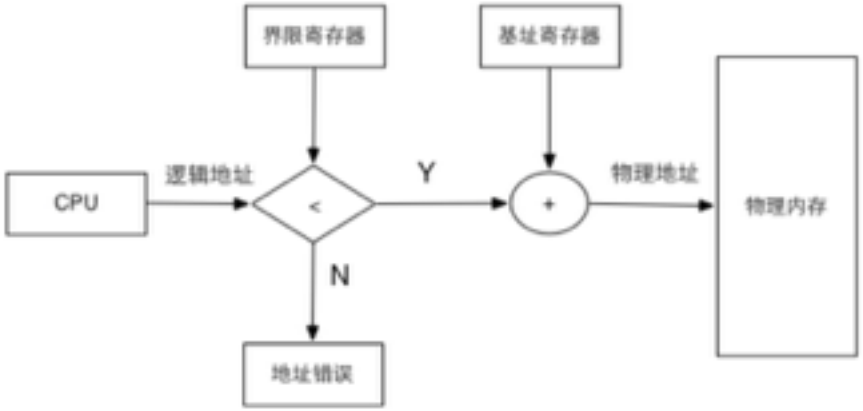
- 正确使用
  - 非法访问
  - 内存泄漏
- 高效使用
  - 降低内存峰值
  - 处理内存警告
  - Cache

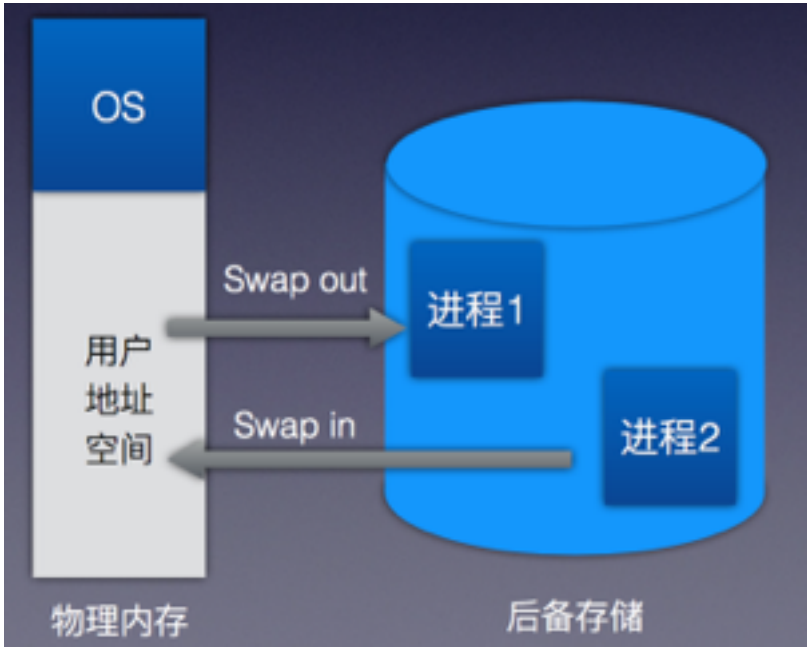
基本概念及原理

- 无内存抽象的问题：
  - 无内存保护
  - 有限的可分配空间
  - 内存利用率低
- 要解决的问题：
  - 存储保护
  - 内存扩充
  - 减少内存碎片

基本概念和原理

- 内存抽象
  - 逻辑地址 VS 物理地址
  - Swap

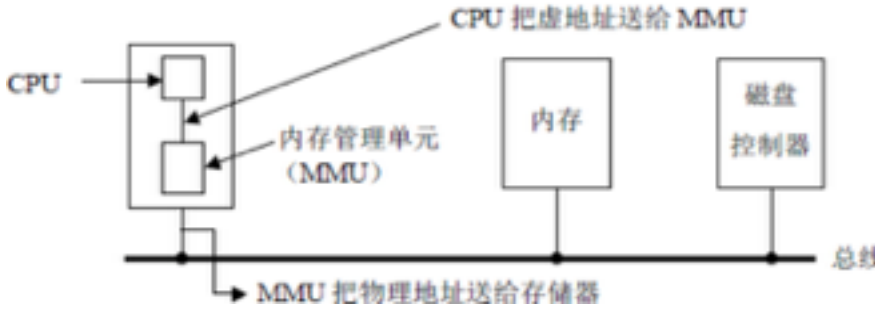


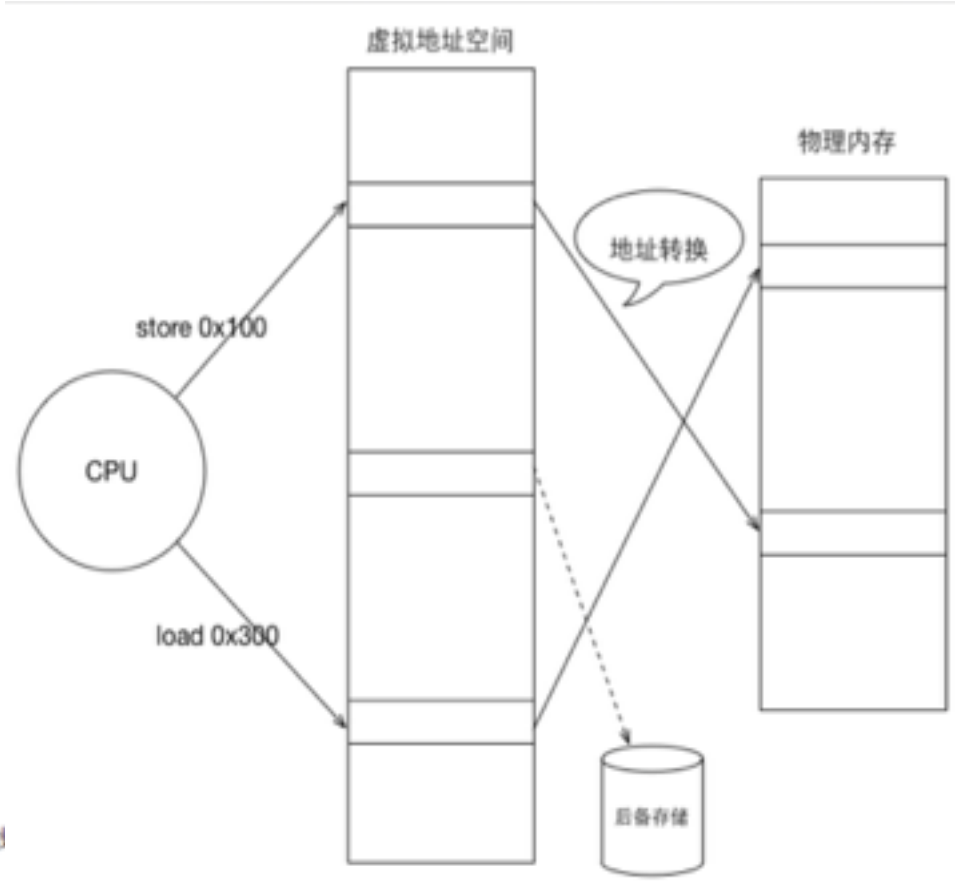


基本概念和原理

虚拟内存

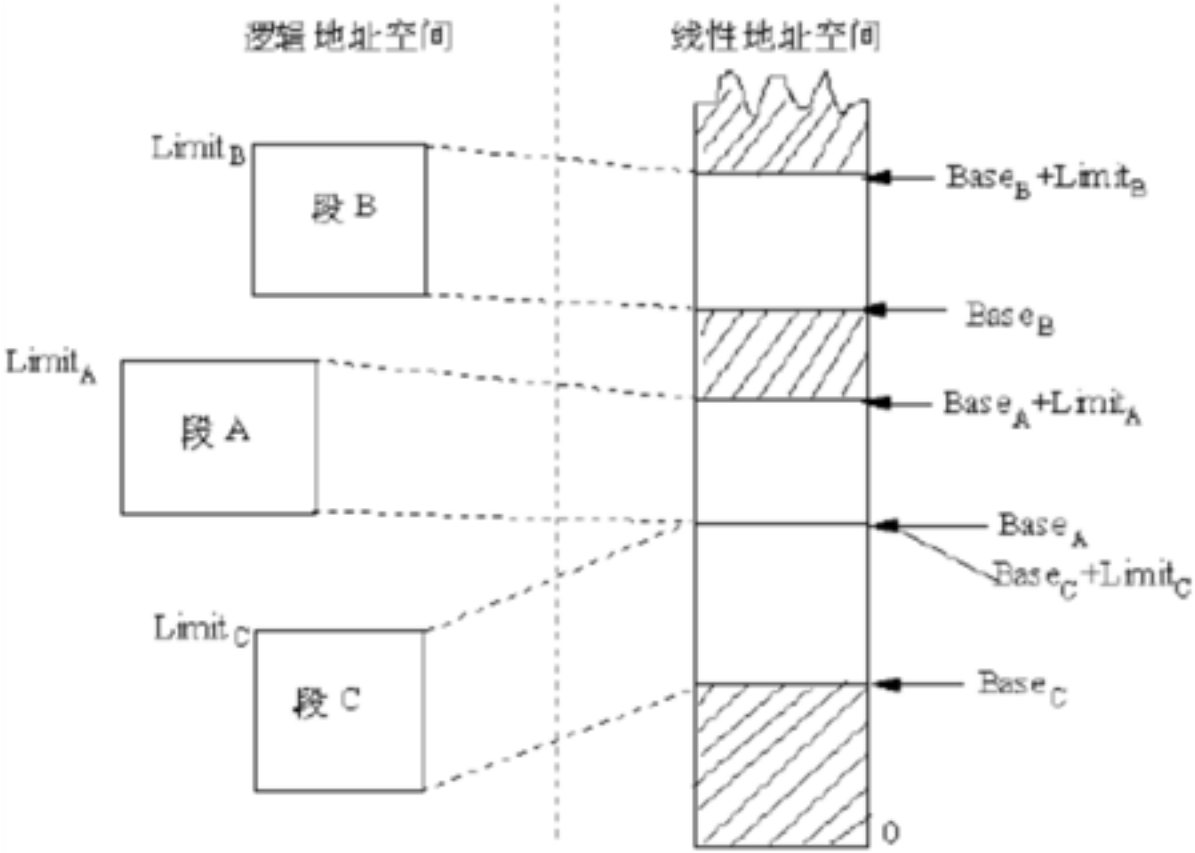
- 虚拟地址->物理地址，地址转换由CPU内部的内存管理单元（MMU）处理
- 巨大的虚拟地址空间（32位4GB，64位16GGB）





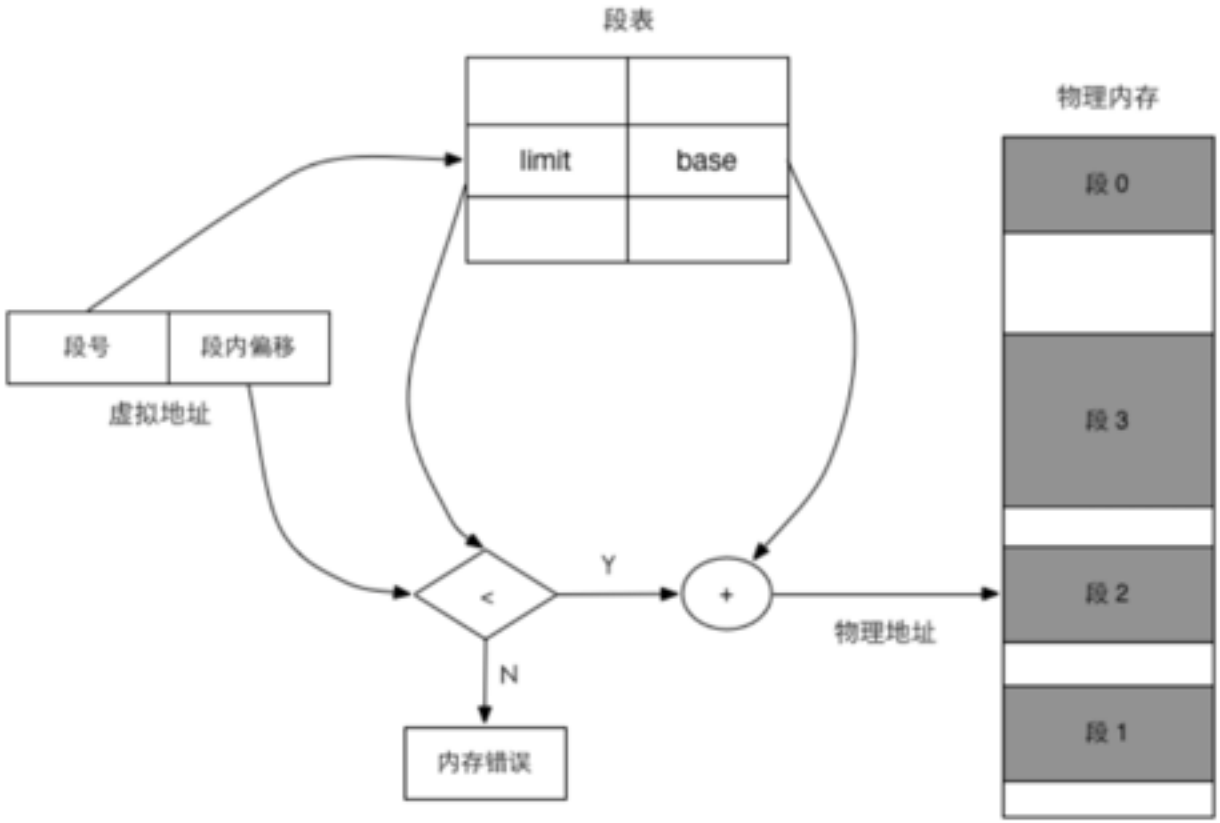
基本概念和原理

段式虚拟内存



基本概念和原理

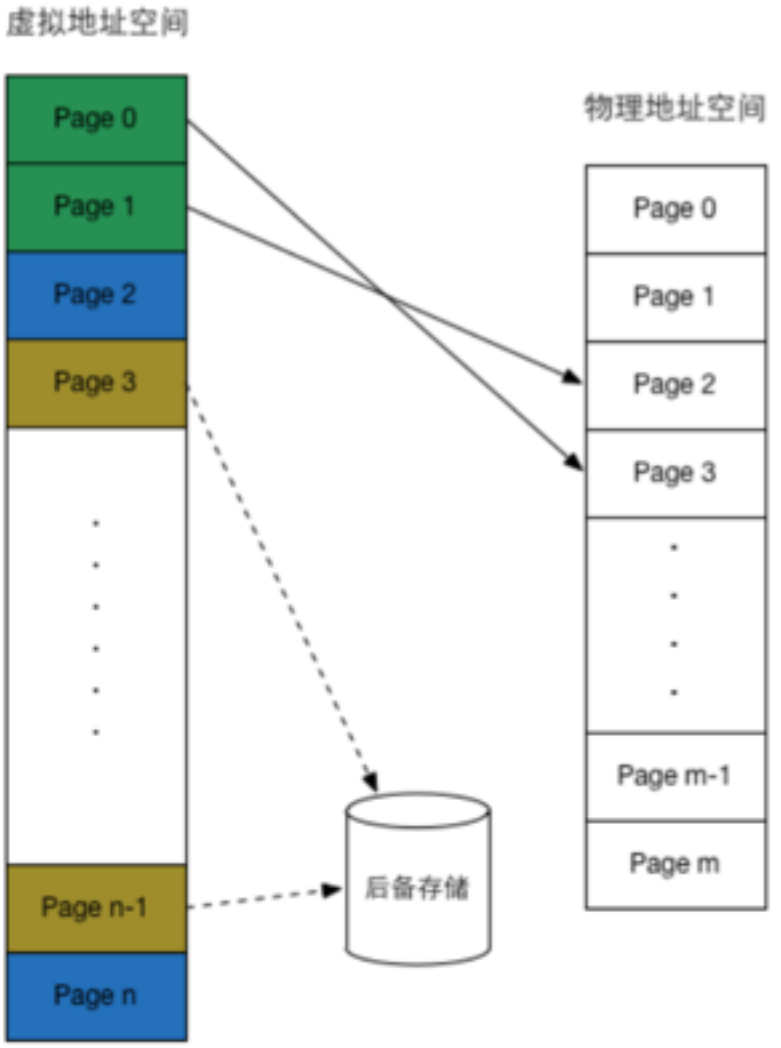
段式虚拟内存



基本概念和原理

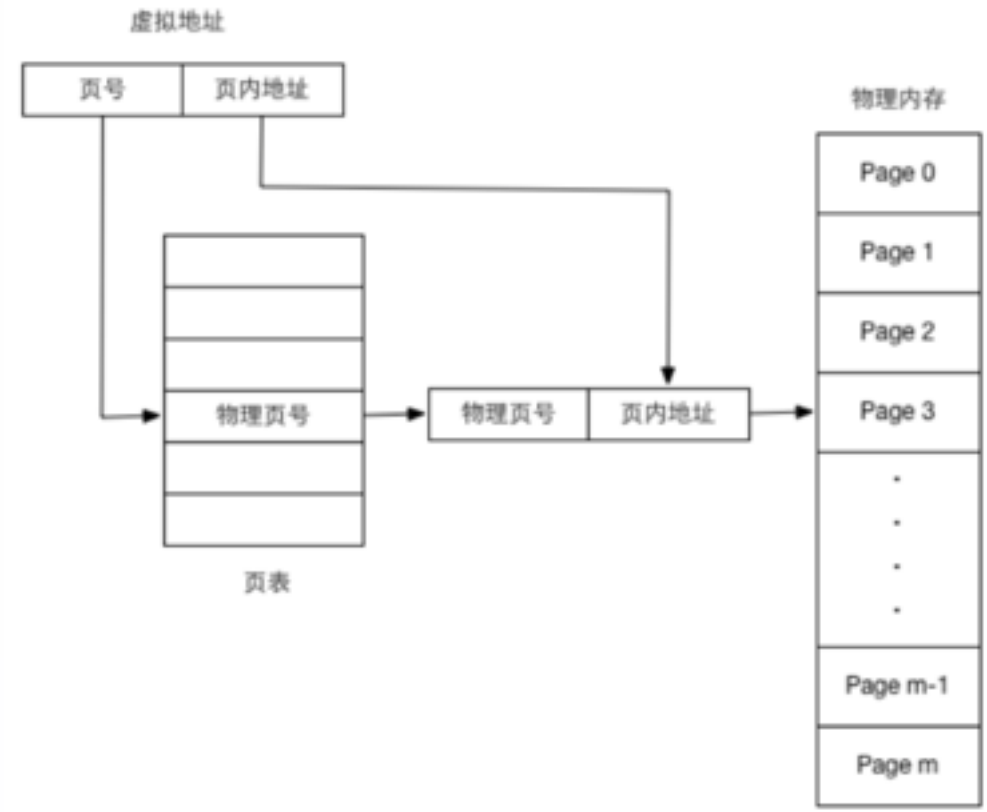
页式虚拟内存

- 解决外部碎片
- 更小的分配及置换单位，离散分部
- Page fault
- Page in & Page out



基本概念和原理

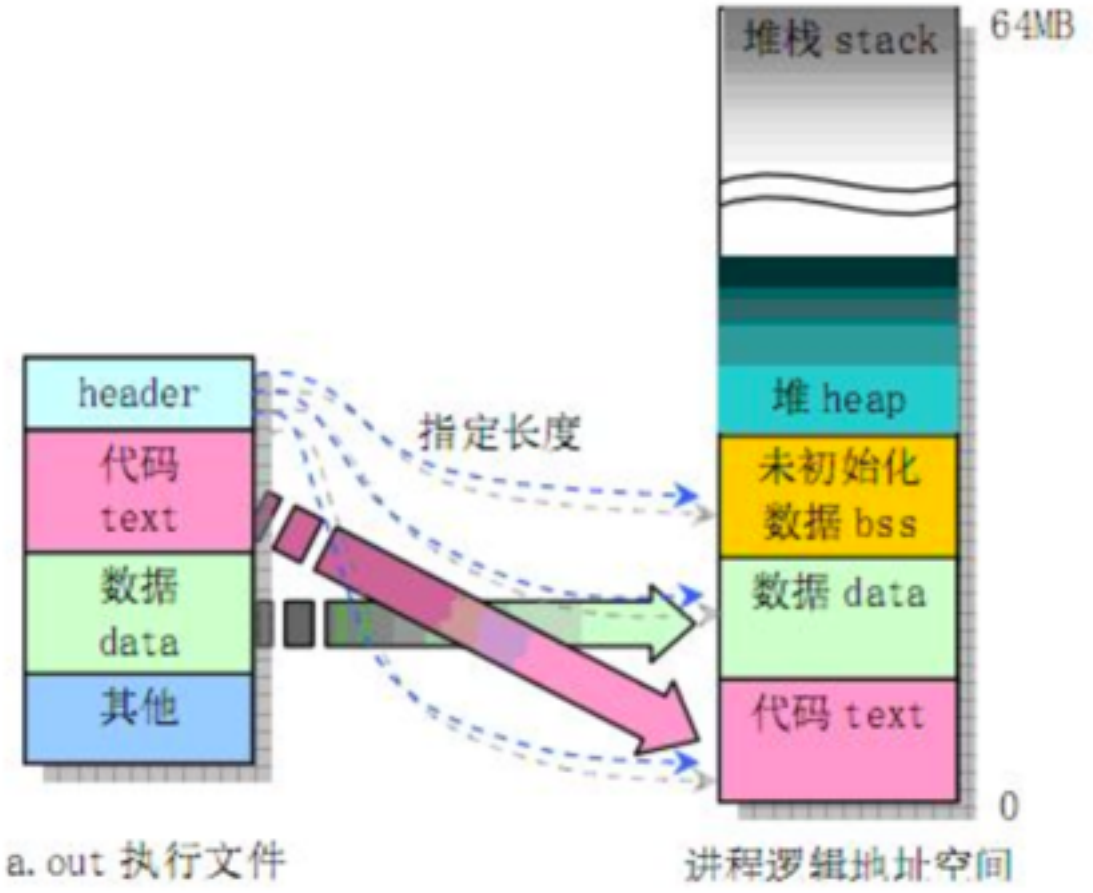
页式虚拟内存



基本概念和原理

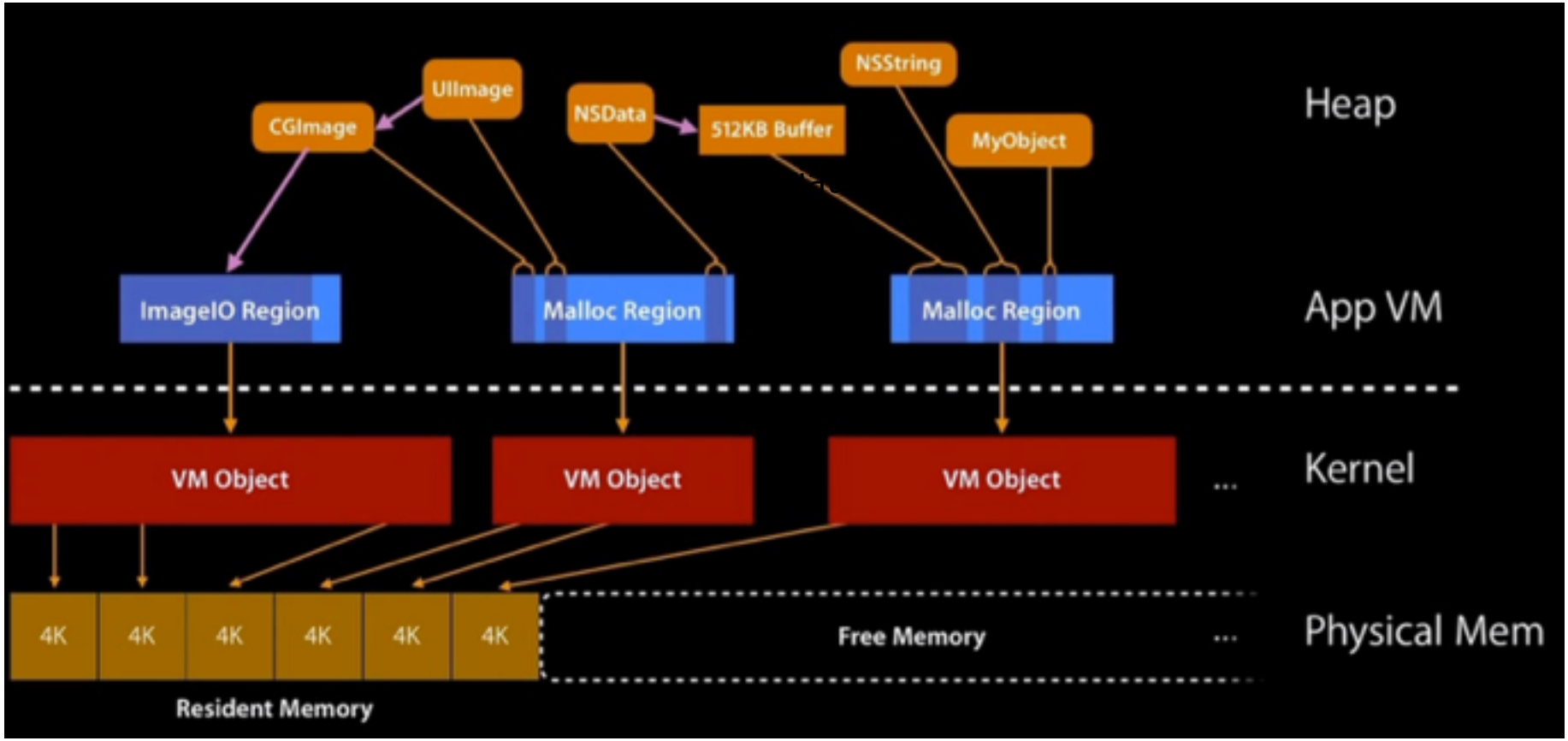
iOS的内存段

- __PAGEZERO
- __TEXT
- __DATA
- __MALLOC_TINY
- __MALLOC_SMALL
- __MALLOC_LARGE



iOS内存管理

- Virtual Memory



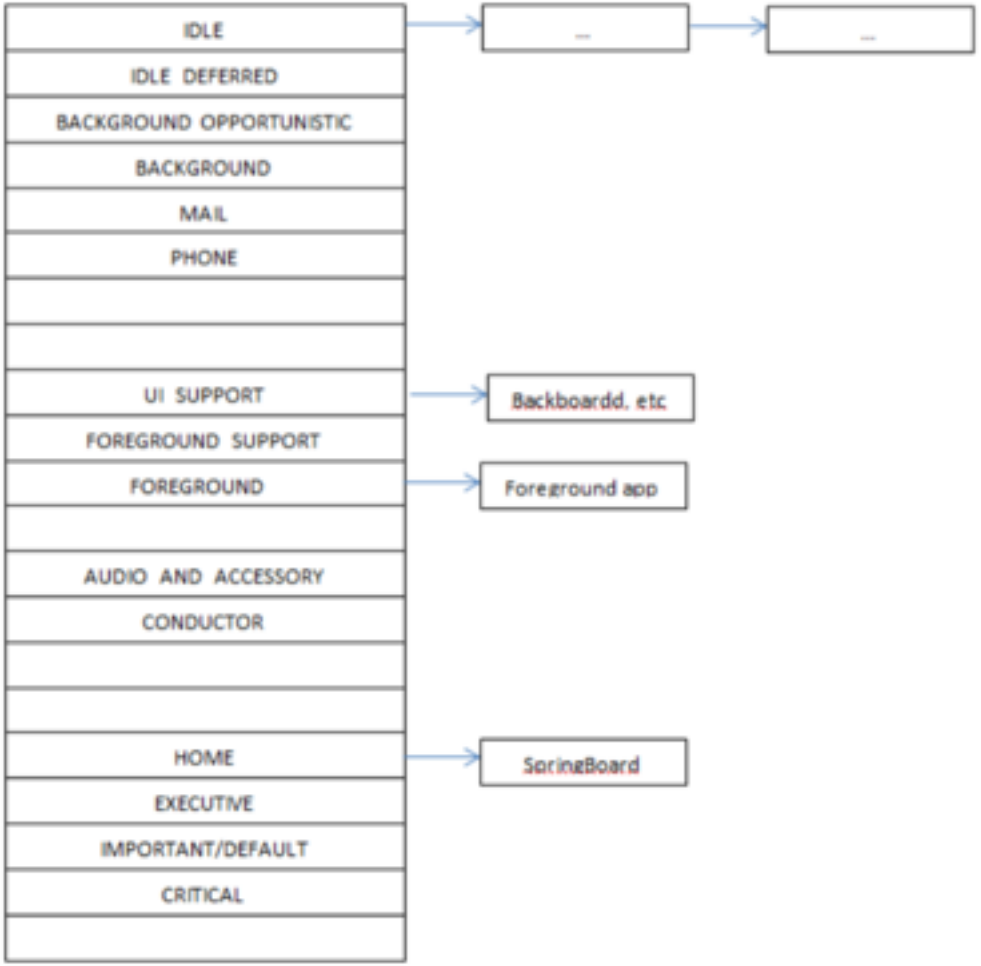
iOS内存管理

- 无Swap机制
  - 移动设备的闪存容量有限
  - 闪存的写次数有限，频繁写会降低寿命

- 思考
  - 代码是要加载到内存执行的，如果没有Swap机制那代码很大的程序岂不是很占内存?

iOS内存管理

- 低内存处理机制Jetsam



iOS内存管理

- UIKit提供三种通知方式
  - [UIApplicationDelegate applicationDidReceiveMemoryWarning:]
  - [UIViewController didReceiveMemoryWarning:]
  - UIApplicationDidReceiveMemoryWarningNotification
- 内存警告消息来自主线程，应避免主线程这时候卡顿或分配过大内存或者快速分配
- 如果App因为内存警告被Kill掉，会生成LowMemory***.log

iOS内存管理

内存分类

- Clean Memory: 在闪存中有备份，能再次读取重建
  - Code, Framework, memory-mapped files
- Dirty Memory: 所有非Clean Memory，系统无法回收
  - Heap allocations, decompressed images, caches

iOS内存管理

Clean Memory & Dirty Memory

```
- (void)cleanMemoryOrDirtyMemory {
        NSString str1 = [NSString stringWithString:@”Welcome!”];
        NSString str2 = @”Welcome!”;
        char *buf = malloc(100 * 1024 * 1024);
        for (int i = 0; i < 50 * 1024 * 1024; i++) {
                buf[i] = rand();
        }
}
```

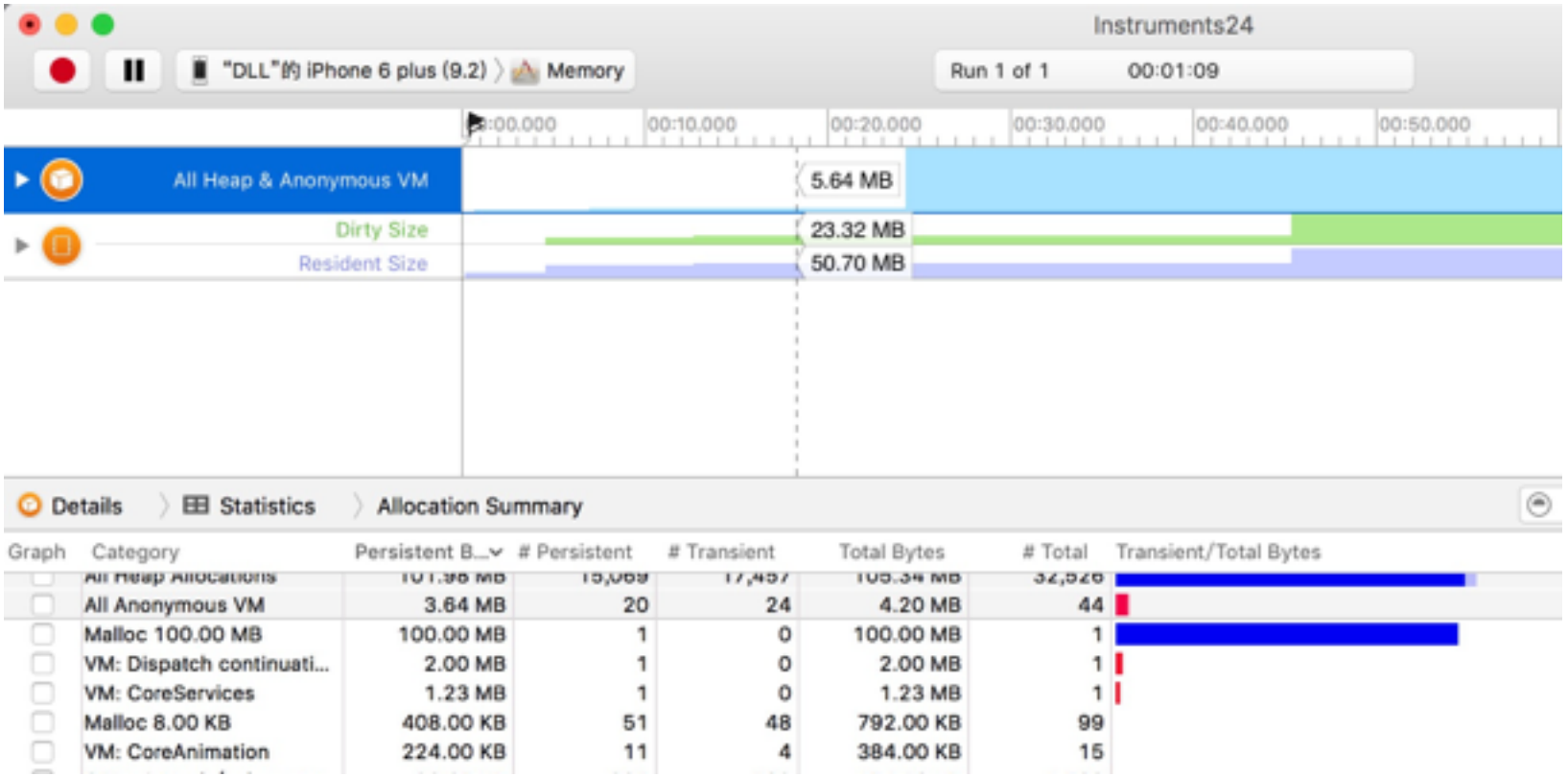
iOS内存管理

Dirty & Resident & Virtual Memory

- 虚拟内存层面
  - Virtual Memory = Clean Memory + Dirty Memory
- 物理内存层面
  - Resident Memory = Clean Memory(Loaded in Physical Memory) + Dirty Memory

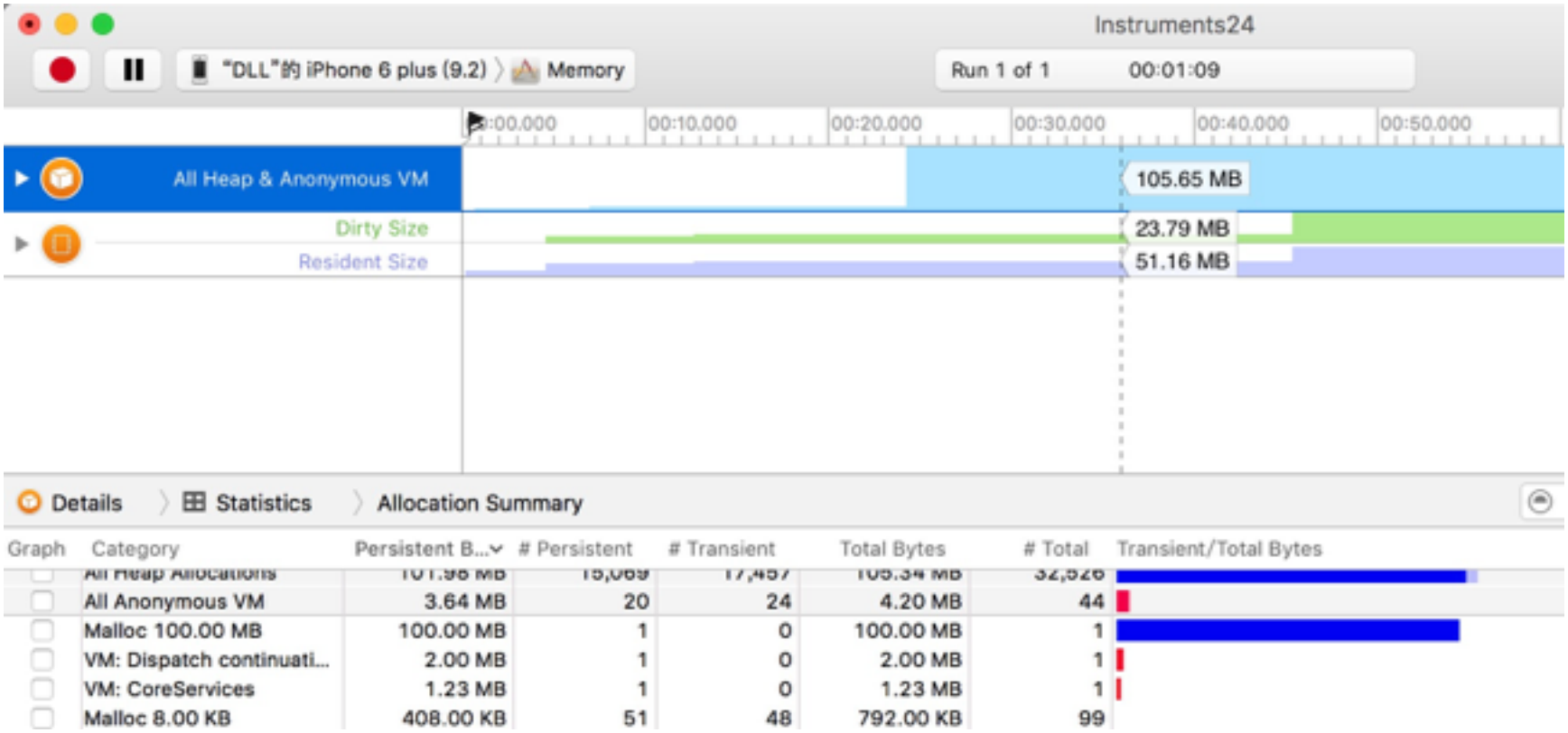
分析工具

Allocations



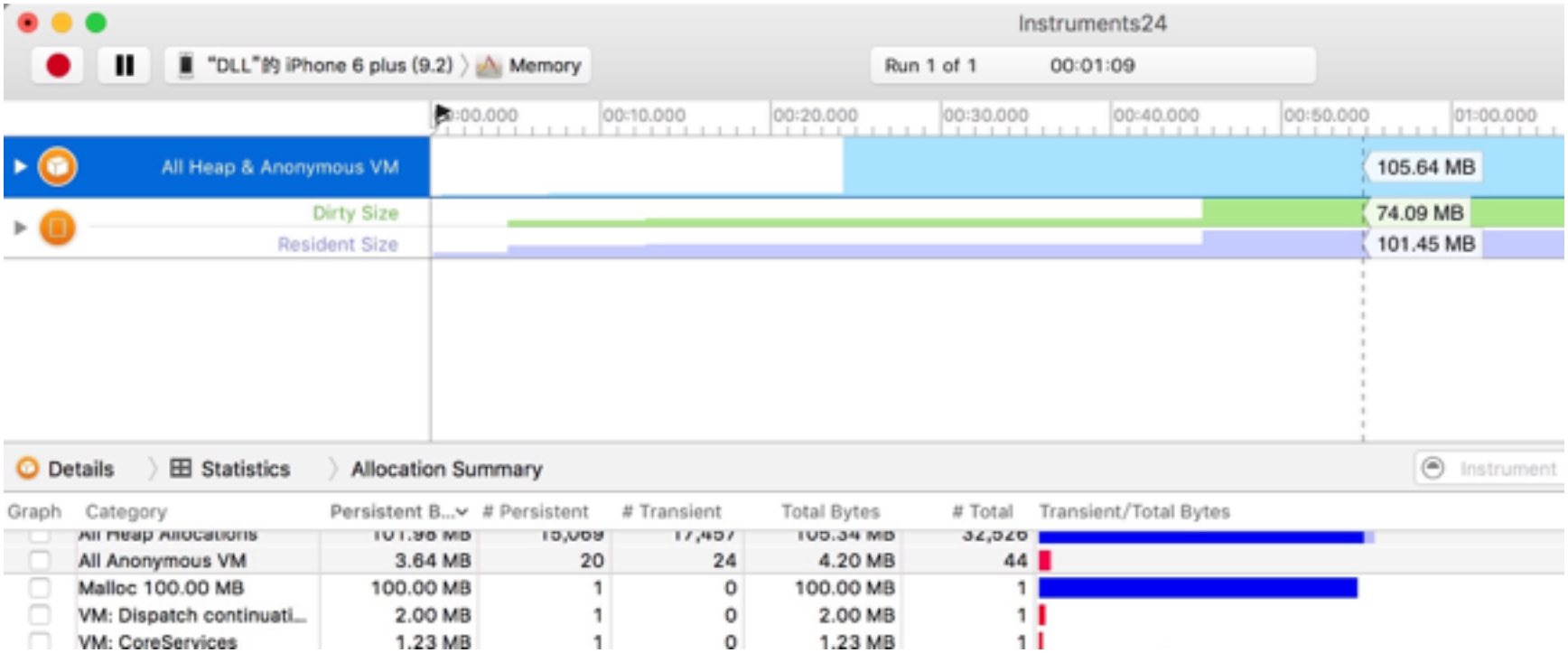
分析工具

char *buf = malloc(100 * 1024 * 1024);



分析工具

```
for (int i = 0; i < 50 * 1024 * 1024; i++) {
        buf[i] = rand();
}
```

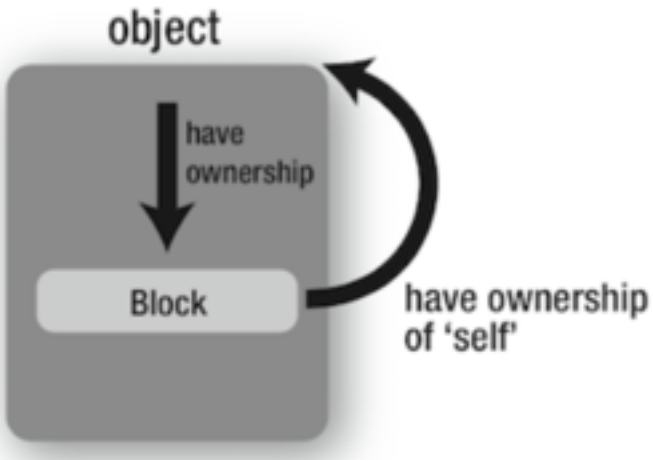


最佳实践

Weak Strong Dance

```
Person *person = [Person new];
person.doSomething = ^{
    [person sayHi];
};
```

```
Person *person = [Person new];
__weak __typeof(person) weakPerson = person;
person.doSomething = ^{
    __typeof(person) strongPerson = weakPerson;
    [strongPerson sayHi];
};
```




```
Person *person = [Person new];
DefineWeakVarBeforeBlock(person);
person.doSomething = ^{
    DefineStrongVarInBlock(person);
    [person sayHi];
};
```

Figure *Circular reference with a Block member variable*

最佳实践

DeallocBlockExecutor

```
- (void)setThemeMap:(NSDictionary *)themeMap {
    objc_setAssociatedObject(self, &kUIView_ThemeMap, themeMap, OBJC_ASSOCIATION_RETAIN_NONATOMIC);
    if (themeMap) {
        // 需要在dealloc时注销observer
        if (objc_getAssociatedObject(self, &kUIView_DeallocHelper) == nil) {
            __unsafe_unretained typeof(self) weakSelf = self;
            id deallocHelper = [self addDeallocBlock:^{
                [[NSNotificationCenter defaultCenter] removeObserver:weakSelf];
            }];
            objc_setAssociatedObject(self, &kUIView_DeallocHelper, deallocHelper,
                OBJC_ASSOCIATION_RETAIN_NONATOMIC);
        }
        [[NSNotificationCenter defaultCenter] removeObserver:self forKeyPath:kThemeDidChangeNotification];
        [[NSNotificationCenter defaultCenter] addObserver:self selector:@selector(themeDidChange) name:
            kThemeDidChangeNotification object:nil];
        [self themeDidChange];
    } else {
        [[NSNotificationCenter defaultCenter] removeObserver:self name:kThemeDidChangeNotification object:nil];
    }
}
```

最佳实践

降低内存峰值

- Lazy Allocation
- alloca VS malloc
- calloc VS malloc + memset
- AutoreleasePool
- imageNamed VS imageWithContentOfFile
- NSData VS fileMapping

最佳实践

```
+ (instancetype)defaultWindow
{
    static DLLMaskWindow *__defaultWindow;
    static dispatch_once_t __token = 0;
    dispatch_once(&__token, ^{
        __defaultWindow = [[DLLMaskWindow alloc] init];
    });
    return __defaultWindow;
}
```

- 直到使用的时候才分配
- 线程安全
- 不仅是分配对象，还有资源文件读取

栈内存分配alloca(size_t)

- 栈分配仅仅修改栈指针寄存器，比malloc遍历并修改空闲列表要快得多
- 栈内存一般都已经在物理内存中，不用担心页错误
- 函数返回的时候栈分配的空间会自动释放
- 但仅适合于小空间的分配，并且函数嵌套不宜过深

calloc VS malloc + memset

- 分配内存并初始化
- 立即分配虚拟内存并设置清0标记位，但不分配物理内存
- 只有相应的虚拟地址空间被读写操作的时候才需要分配相应的物理内存页并初始化

NSAutoreleasePool

- 基于引用计数，Pool执行drain方法会release所有该Pool中的autorelease对象
- 可以同时嵌套多个AutoreleasePool
- 每个线程没有默认的AutoreleasePool，需要手动创建，避免内存泄漏
- 在一段内存分配频繁的代码中适当嵌套AutoreleasePool有利于降低整体内存峰值

图片读取

- imageNamed
  - 使用系统缓存，适用于频繁使用的小图片
- imageWithContentOfFile
  - 不带缓存机制，适用于大图片，用完就释放

NSData & 内存映射文件

```
[NSData dataWithContentsOfFile:path];

[NSData dataWithContentsOfFile:path
options:NSDataReadingMappedIfSafe error:&error];
```

- 映射文件到虚拟内存，只有读取操作的时候才会读取相应页的内容到物理内存页中
- 大文件建议采用内存映射的方式

NSCache & NSPurgeableData

- NSCache
  - 2种界限条件：totalCostLimit & countLimit
  - 类NSMutableDictionary, setObject:forKey:cost:
  - evictsObjectWithDiscardContent & <NSDicardableContent>
  - 最好监听内存警告消息并移除所有Cache

- NSPurgeableData
  - 当系统处于低内存的时候会自动移除
  - 适用于大数据

内存警告处理

- 尽可能释放多资源，尤其图片等占内存多的资源，等需要用的时候再重建
- 单例对象不要创建之后就一直持有数据，在内存紧张的时候释放掉
- iOS6之后系统内存紧张会自动释放CALayer的CABackingStore对象，需要使用的时候再调drawRect:来构建，所以viewController不会再unloadView