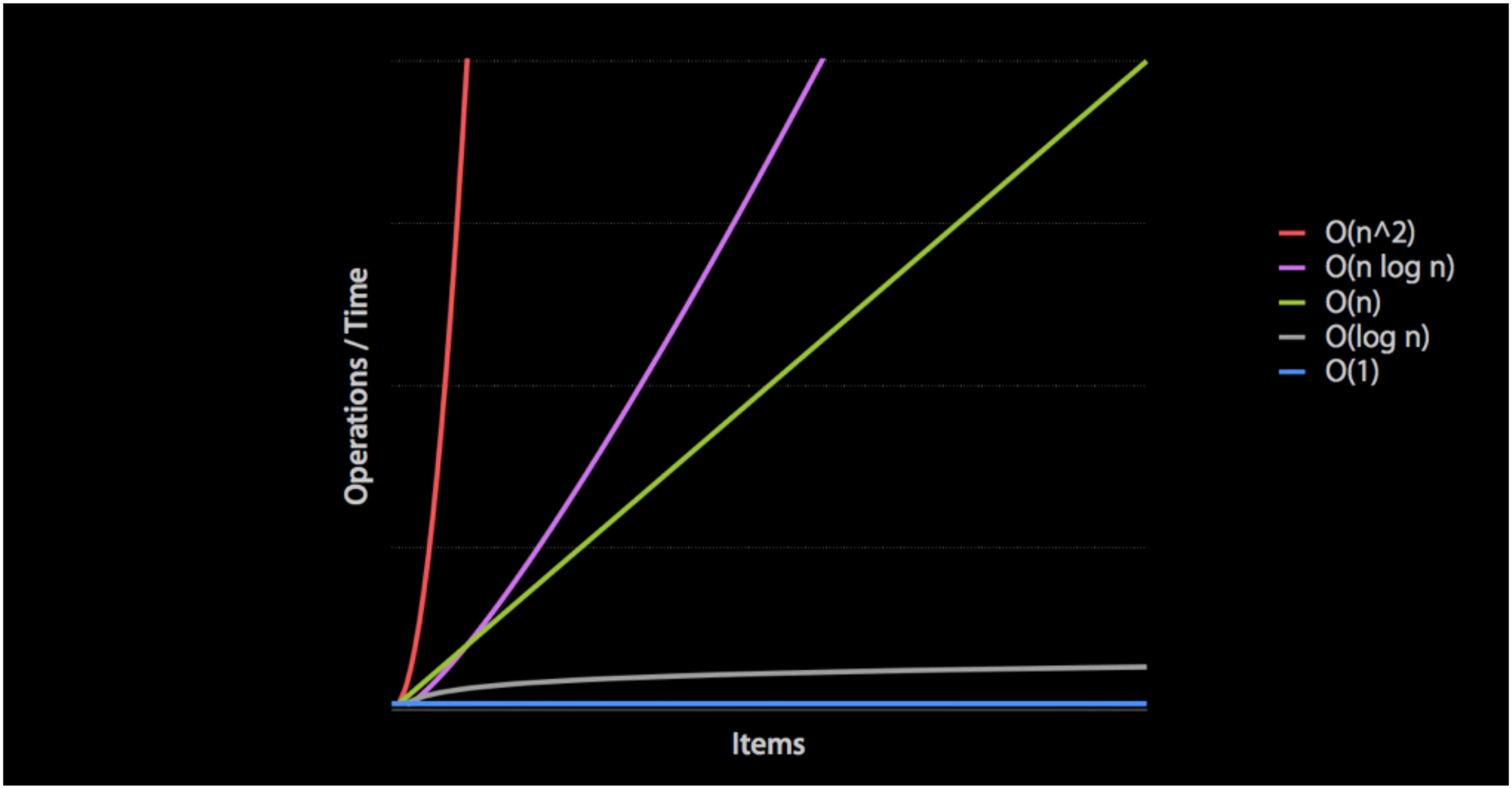
深入剖析iOS性能优化

什么影响性能?

如何检测?

时间复杂度



数组

- O(n) 会遍历的接口：containsObject:，indexOfObject*，removeObject:
- O(1) 栈顶栈底操作：objectAtIndex:，firstObject:，lastObject:，addObject:，removeLastObject:
- O(log n) 二分查找：indexOfObject:inSortedRange:options:usingComparator:

Dictionary 和 Set

- 无序没有重复元素，通过 hash table 进行快速的操作
- 添加删除和查找都是 O(1) 的

containsObject

不同实现

containsObject in 数组

APMCon

```
- (BOOL) containsObject: (id)anObject
{
  return ([self indexOfObject: anObject] != NSNotFound);
}

- (NSUInteger) indexOfObject: (id)anObject
{
  unsigned    c = [self count];

  if (c > 0 && anObject != nil)
    {
      unsigned      i;
      IMP get = [self methodForSelector: oaiSel];
      BOOL     (*eq)(id, SEL, id)
       = (BOOL (*)(id, SEL, id))[anObject methodForSelector: eqSel];

      for (i = 0; i < c; i++)
       if ((*eq)(anObject, eqSel, (*get)(self, oaiSel, i)) == YES)
         return i;
    }
  return NSNotFound;
}
```

containsObject in Set

```
- (BOOL) containsObject: (id)anObject
{
  return (([self member: anObject]) ? YES : NO);
}

- (id) member: (id)anObject
{
  if (anObject != nil)
    {
      GSIMapNode node = GSIMapNodeForKey(&map, (GSIMapKey)anObject);

      if (node != 0)
       {
         return node->key.obj;
       }
    }
  return nil;
}
```

GCD

- 通过 **dispatch_block_create_with_qos_class** 方法指定队列的 QoS 为 **QOS_CLASS_UTILITY**（>= iOS 8）。这个 QoS 系统会针对大的计算，I/O，网络以及复杂数据处理做**电量优化**。

Bend the knee
or burn

THREADS EXPLODE

爆炸和死锁

```
//线程爆炸
for (int i = 0; i < 999; i++) {
    dispatch_async(q, ^{...});
}
//会造成死锁
dispatch_barrier_sync(q,
^{});
```

避免方式一

使用串行队列

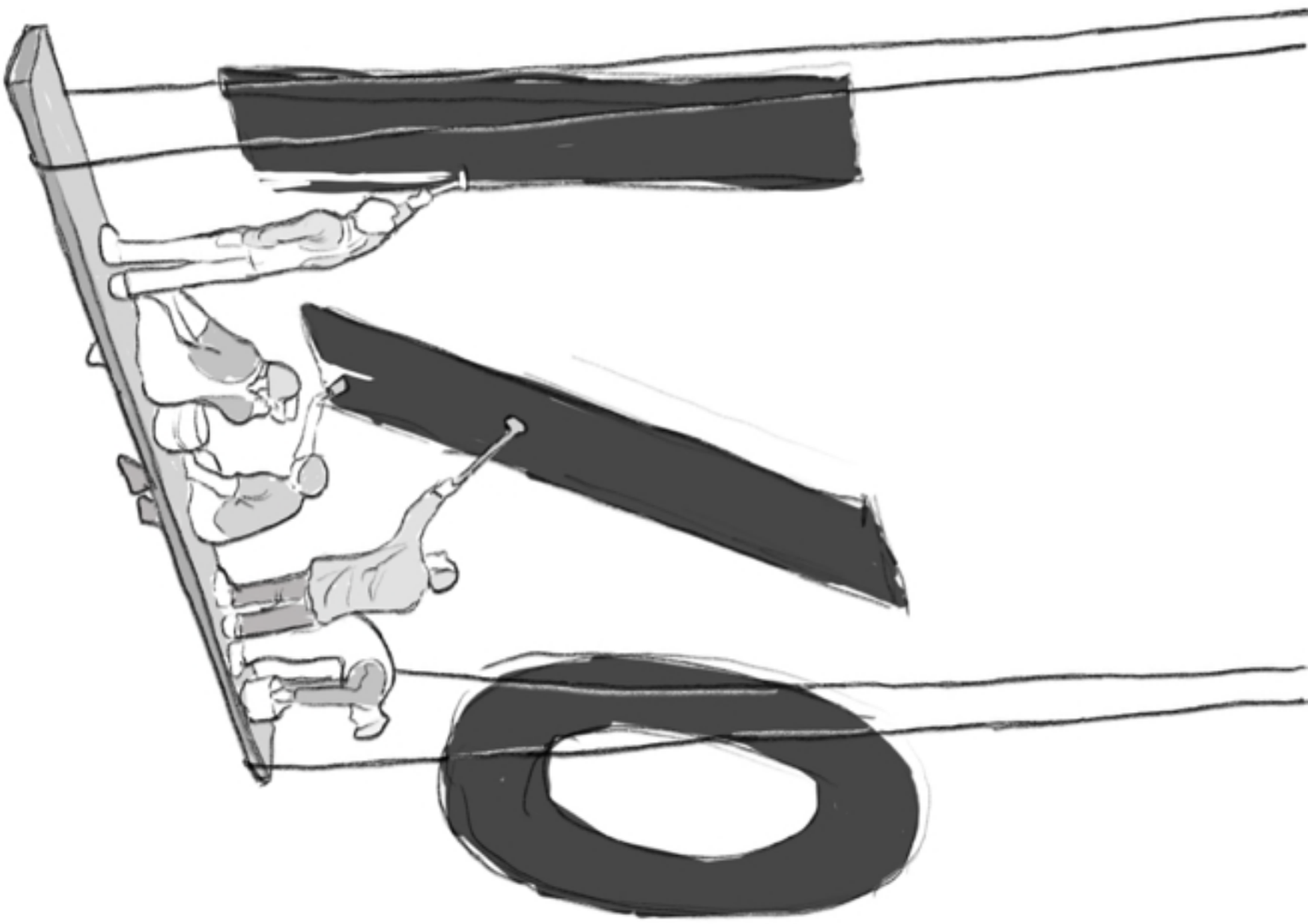
NSOperationQueues

NSOperationQueue.maxConcurrentOperationCount

```
dispatch_apply(999, q, ^(size_t i){...});
```

四

```
#define CONCURRENT_TASKS 4
sema = dispatch_semaphore_create(CONCURRENT_TASKS);
for (int i = 0; i < 999; i++){
    dispatch_async(q, ^{
        dispatch_semaphore_signal(sema);
    });
    dispatch_semaphore_wait(sema, DISPATCH_TIME_FOREVER);
}
```



- 将零碎的内容作为一个整体进行写入
- 使用合适的 I/O 操作 API
- 使用合适的线程
- 使用 NSCache 做缓存能够减少 I/O

NSCache

- Cache add
- Cache clean
- Callback

SD WebImage

```
-(UIImage *)imageFromMemory{
  return [self.memCach obj];
}

-(UIImage *)imageFromDisk{
  UIImage *image = [self im
  if (image){
    return image;
  }
  UIImage * diskImage = [self
  if (diskImage && self.shouk
    NSUInteger cost = SDCa
```

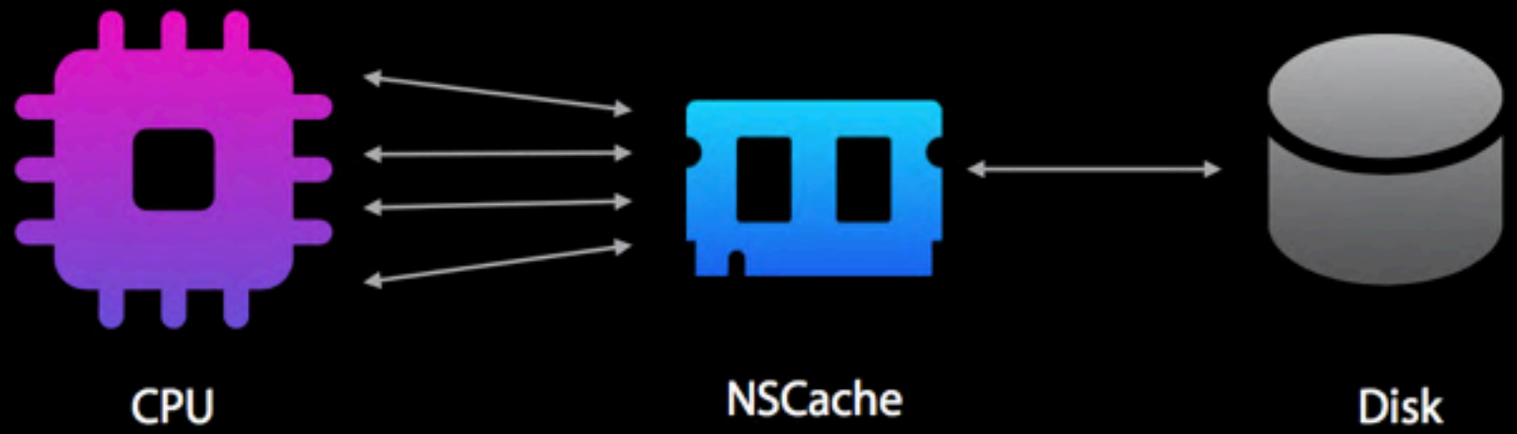
Caching

In-memory copy of data

Potential candidates

- Frequent writes
- Expensive reads

Memory and I/O tradeoffs



为什么不直接用字典?

- 自动清理系统占用内存
- NSCache 是线程安全
- - (void)cache:(NSCache *)cache willEvictObject:(id)obj; 缓存对象将被清理时的回调
- evictsObjectsWithDiscardedContent 可以控制是否清理

NSCache 里有个 NSMutableDictionary


```
@implementation NSCache
- (id) init
{
  if (nil == (self = [super init]))
    {
      return nil;
    }
  _objects = [NSMutableDictionary new];
  _accesses = [NSMutableArray new];
  return self;
}
```

还有些额外信息

```
@interface _GSCachedObject : NSObject
{
  @public
  id object; //cache 的值
  NSString *key; //设置 cache 的 key
  int accessCount; //保存访问次数，用于自动清理
  NSUInteger cost; //setObject:forKey:cost:
  BOOL isEvictable; //线程安全
}
@end
```

Cache 读取

APMCon

```
- (id) objectForKey: (id)key
{
  _GSCachedObject *obj = [_objects objectForKey: key];

  if (nil == obj)
    {
      return nil;
    }
  if (obj->isEvictable) //保证添加删除操作线程安全
    {
      // 将 obj 移到 access list 末端
      [_accesses removeObjectIdenticalTo: obj];
      [_accesses addObject: obj];
    }
  obj->accessCount++;
  _totalAccesses++;
  return obj->object;
}
```

Cache 添加

```
- (void) setObject: (id)obj forKey: (id)key cost: (NSUInteger)num
{
  _GSCachedObject *oldObject = [_objects objectForKey: key];
  _GSCachedObject *newObject;

  if (nil != oldObject)
    {
      [self removeObjectForKey: oldObject->key];
    }
  [self _evictObjectsToMakeSpaceForObjectWithCost: num];
  newObject = [_GSCachedObject new];
  // Retained here, released when obj is dealloc'd
  newObject->object = RETAIN(obj);
  newObject->key = RETAIN(key);
  newObject->cost = num;
  if ([obj conformsToProtocol: @protocol(NSDiscardableContent)])
    {
      newObject->isEvictable = YES;
      [_accesses addObject: newObject];
    }
  [_objects setObject: newObject forKey: key];
  RELEASE(newObject);
  _totalCost += num;
}
```

Cache 自动清理方法

何时自动清理的判断

```
// cost 在添加新 cache 值时指定的 cost
// _costLimit 是 totalCostLimit 属性值
if (_costLimit > 0 && _totalCost + cost > _costLimit){
    spaceNeeded = _totalCost + cost - _costLimit;
}
// 只有当 cost 大于人工限制时才会清理
// 或者 cost 设置为0不进行人工干预
if (count > 0 && (spaceNeeded > 0 || count >= _countLimit))
```

不要清理
经常访问的 objects


```
//_totalAccesses 所有的值的访问都会 +1
NSUInteger averageAccesses = (_totalAccesses / count * 0.2) + 1;
//accessCount 每次 obj 取值时会 +1
if (obj->accessCount < averageAccesses && obj->isEvictable)
```

清理前准备工作

```
NSUInteger cost = obj->cost;
obj->cost = 0;
// 不会被再次清除
obj->isEvictable = NO;
// 添加到 remove list 里
if (_evictsObjectsWithDiscardedContent)
{
    [evictedKeys addObject: obj->key];
}
_totalCost -= cost;
// 如果已经释放了足够空间就不用后面操作了
if (cost > spaceNeeded)
{
    break;
}
spaceNeeded -= cost;
```

清理时执行回调

```
- (void) removeObjectForKey: (id)key
{
    _GSCachedObject *obj = [_objects objectForKey: key];
    if (nil != obj)
    {
        [_delegate cache: self willEvictObject: obj->object];
        _totalAccesses -= obj->accessCount;
        [_objects removeObjectForKey: key];
        [_accesses removeObjectIdenticalTo: obj];
    }
}
```

NSCache 在 SDWebImage 的运用

APMCon

```
- (UIImage *)imageFromMemoryCacheForKey:(NSString *)key {
    return [self.memCache objectForKey:key];
}

- (UIImage *)imageFromDiskCacheForKey:(NSString *)key {

    // 检查 NSCache 里是否有
    UIImage *image = [self imageFromMemoryCacheForKey:key];
    if (image) {
        return image;
    }

    // 从磁盘里读
    UIImage *diskImage = [self diskImageForKey:key];
    if (diskImage && self.shouldCacheImagesInMemory) {
        NSUInteger cost = SDCacheCostForImage(diskImage);
        [self.memCache setObject:diskImage forKey:key cost:cost];
    }

    return diskImage;
}
```

控制 App 的 Wake 次数

- 唤起这个过程会有比较大的消耗
- 通知，VoIP，定位，蓝牙都会使设备从 Standby 状态唤起

定位方法的选择

- 连续的位置更新：startUpdatingLocation
- 延时有效定位：
  allowDeferredLocationUpdatesUntilTraveled: timeout:
- 重大位置变化：
  startMonitoringSignificantLocationChanges
- 区域监测：startMonitoringForRegion:(CLRegion *)

如何预防这些性能问题需要刻意预防么？

坚持几个原则

- 优化计算的复杂度从而减少 CPU 的使用
- 在应用响应交互的时候停止没必要的任务处理
- 设置合适的 QoS
- 将定时器任务合并，让 CPU 更多时候处于 idle 状态

如何检查?

监听主线程

- 用 CFRunLoopObserverCreate 创建一个观察者
- 接受 CFRunLoopActivity 的回调
- 用 CFRunLoopAddObserver 将观察者添加到 CFRunLoopGetMain() 主线程 Runloop 的 kCFRunLoopCommonModes 模式下进行观察
- 创建一个子线程来进行监控
- 根据两个 Runloop 的状态 BeforeSources 和 AfterWaiting 在区间时间是否能检测到来判断是否卡顿

如何打印堆栈信息

保存现场

```
2017-08-07 20:25:57.497 DecoupleDemo[35140:435526] monitor trigger
2017-08-07 20:25:57.506 DecoupleDemo[35140:435528] used 66 MB
Call 10 threads:
Stack of thread: 1027:
 CPU used: 1.0 percent
 user time: 469514 second
libsystem_kernel.dylib          0x1090edf46 __semwait_signal + 10
libsystem_c.dylib               0x108e9f038 usleep + 53
DecoupleDemo                    0x10515fd3e -[TestTableView configCell] + 814
DecoupleDemo                    0x105163779 -[testTableViewController
observeValueForKeyPath:ofObject:change:context:] + 601
Foundation                      0x10663185c NSKeyValueNotifyObserver + 351
Foundation                      0x106631113 NSKeyValueDidChange + 484
Foundation                      0x10670e90f -[NSObject(NSKeyValueObservingPrivate)
_changeValueForKeys:count:maybeOldValuesDict:usingBlock:] + 868
Foundation                      0x1065f6f8c -[NSObject(NSKeyValueObservingPrivate)
_changeValueForKey:key:key:usingBlock:] + 61
Foundation                      0x10665bee6 _NSSetObjectValueAndNotify + 262
DecoupleDemo                    0x10515b726 -[SMTableView
tableView:cellForRowAtIndexPath:] + 262
UIKit                           0x10764cab2 -[UITableView
_createPreparedCellForGlobalRow:withIndexPath:willDisplay:] + 750
UIKit                           0x10764ccf8 -[UITableView
_createPreparedCellForGlobalRow:willDisplay:] + 74
UIKit                           0x1076219e5 -[UITableView
_updateVisibleCellsNow:isRecursive:] + 3785
```

task_threads 取到所有的线程

```
thread_act_array_t threads; //int 组成的数组比如 thread[1] = 5635
mach_msg_type_number_t thread_count = 0; //mach_msg_type_number_t 是 int 类型
const task_t this_task = mach_task_self(); //int
//根据当前 task 获取所有线程
kern_return_t kr = task_threads(this_task, &threads, &thread_count);
```

thread_info
获取线程详细信息

APMCon

```
if (thread_info((thread_act_t)thread, THREAD_BASIC_INFO, (thread_info_t)threadInfo,
&threadInfoCount) == KERN_SUCCESS) {
    threadBasicInfo = (thread_basic_info_t)threadInfo;
    if (!(threadBasicInfo->flags & TH_FLAGS_IDLE)) {
        threadInfoSt.cpuUsage = threadBasicInfo->cpu_usage / 10;
        threadInfoSt.userTime = threadBasicInfo->system_time.microseconds;
    }
}
```

thread_get_state
获取线程里所有栈的信息

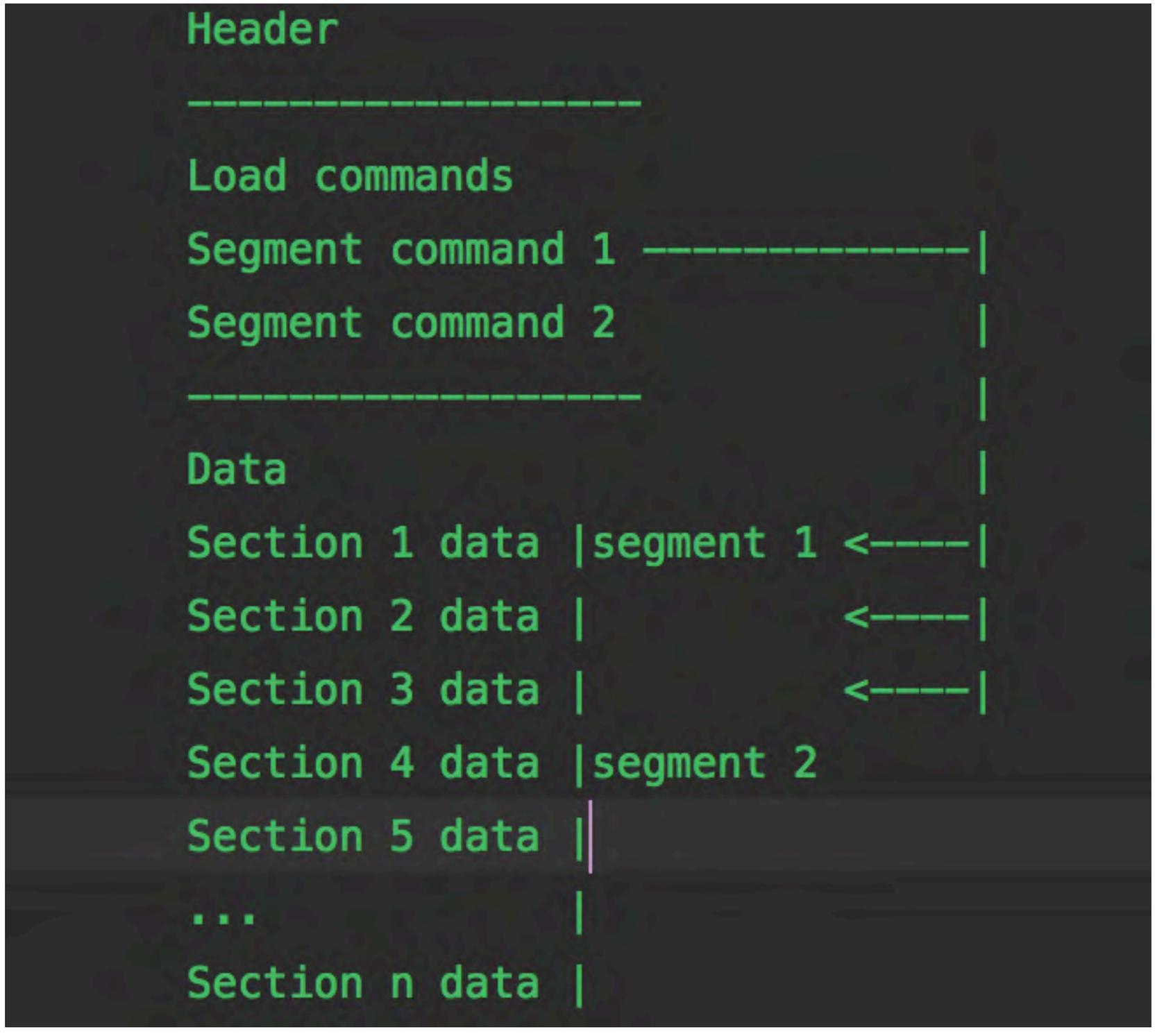

```
_STRUCT_MCONTEXT machineContext; //线程栈里所有的栈指针
//通过 thread_get_state 获取完整的 machineContext 信息，包含 thread 状态信息
mach_msg_type_number_t state_count = smThreadStateCountByCPU();
kern_return_t kr = thread_get_state(thread, smThreadStateByCPU(),
(thread_state_t)&machineContext.__ss, &state_count);
```

栈结构体保存栈数据

```
//为通用回溯设计结构支持栈地址由小到大，地址里存储上个栈指针的地址
typedef struct SMStackFrame {
    const struct SMStackFrame *const previous;
    const uintptr_t return_address;
} SMStackFrame;

SMStackFrame stackFrame = {0};
//通过栈基址指针获取当前栈帧地址
const uintptr_t framePointer = smMachStackBasePointerByCPU(&machineContext);
if (framePointer == 0 || smMemCopySafely((void *)framePointer, &stackFrame,
sizeof(stackFrame)) != KERN_SUCCESS) {
    return @"Fail frame pointer";
}
for (; i < 32; i++) {
    buffer[i] = stackFrame.return_address;
    if (buffer[i] == 0 || stackFrame.previous == 0 || smMemCopySafely(stackFrame.previous,
&stackFrame, sizeof(stackFrame)) != KERN_SUCCESS) {
        break;
    }
}
```

栈信息符号化


Header
Load commands
Segment command 1
Segment command 2
Data
Section 1 data
Section 2 data
Section 3 data
Section 4 data
Section 5 data
...
Section n data
segment 1
segment 2

获取 mach_header 和 slide
计算 ASLR 偏移量

APMCon

```
//根据 image 的序号获取 mach_header
const struct mach_header* machHeader = _dyld_get_image_header(idx);
//返回 image_index 索引的 image 的虚拟内存地址 slide 的数量，如果 image_index 超出范围返回0
//动态链接器加载 image 时，image 必须映射到未占用地址的进程的虚拟地址空间。动态链接器通过添加一个值到 image 的基地址来实现，这个值是虚拟内存 slide 数量
const uintptr_t imageVMAddressSlide = (uintptr_t)_dyld_get_image_vmaddr_slide(idx);
//https://en.wikipedia.org/wiki/Address_space_layout_randomization
const uintptr_t addressWithSlide = address - imageVMAddressSlide;
```

获取符号表的虚拟内存偏移量

```
//LC_SYMTAB 描述了 __LINKEDIT segment 内查找字符串和符号表的位置
if (loadCmd->cmd == LC_SYMTAB) {
    //获取字符串和符号表的虚拟内存偏移量。
    const struct symtab_command* symtabCmd = (struct symtab_command*)cmdPointer;
    const nlistByCPU* symbolTable = (nlistByCPU*)(segmentBase + symtabCmd->symoff);
    const uintptr_t stringTable = segmentBase + symtabCmd->stroff;
```

找到最匹配的符号地址

```
//给定的偏移量是文件偏移量，减去 __LINKEDIT segment 的文件偏移量获得字符串和符号表的虚拟内存偏移量
uintptr_t symbolBase = symbolTable[iSym].n_value;
uintptr_t currentDistance = addressWithSlide - symbolBase;
//寻找最小的距离 bestDistance，因为 addressWithSlide 是某个方法的指令地址，要大于这个方法的入口。
//离 addressWithSlide 越近的函数入口越匹配
if ((addressWithSlide >= symbolBase) && (currentDistance <= bestDistance)) {
    bestMatch = symbolTable + iSym;
    bestDistance = currentDistance;
}
```

获取更多信息

- 更细化的测量时间消耗，找到耗时方法
- 给优化定个目标，比如某场景响应操作在 100ms 内完成

如何获取到更多信息呢?

```
 1| 320.36|  -[testTableViewController tbView]
 2| 319.40|    -[TestTableView init]
 3| 114.54|      -[TestTableView buildForInit]
 4| 102.95|        -[TestTableView updateWithViewModel:]
 5|  94.26|          -[TestTableView judgeIfNeedRefresh]
 6|   2.02|            +[MJRefreshAutoNormalFooter footerWithRefreshingTarget:refreshingAction:]
 7|   1.95|              -[MJRefreshAutoNormalFooter init]
 8|   1.47|                -[MJRefreshAutoNormalFooter prepare]
 9|   1.61|                  -[UIScrollViewPanGestureRecognizer addObserver:forKeyPath:options:context:]
10|   4.13|                    -[UIImageView initWithImage:]
10|   7.06|                    +[NSBundle mj_arrowImage]
10|  52.40|                    -[UIActivityIndicatorView initWithActivityIndicatorStyle:]
10|  15.66|                    +[NSBundle mj_localizedStringForKey:]
10|   3.39|                    -[MJRefreshNormalHeader lastUpdatedTimeLabel]
 9|   1.05|                  -[UITableView addObserver:forKeyPath:options:context:]
10|   4.13|                    -[UIImageView initWithImage:]
10|   7.06|                    +[NSBundle mj_arrowImage]
10|  52.40|                    -[UIActivityIndicatorView initWithActivityIndicatorStyle:]
10|  15.66|                    +[NSBundle mj_localizedStringForKey:]
10|   3.39|                    -[MJRefreshNormalHeader lastUpdatedTimeLabel]
 9|  11.27|                  -[MJRefreshNormalHeader arrowView]
10|   4.13|                    -[UIImageView initWithImage:]
```

- hook objc_msgSend 方法能够获取所有被调用的方法，facebook 的 fishhook https://github.com/facebook/fishhook

- 记录深度就能够得到方法调用的树状结构，InspectiveC https://github.com/DavidGoldman/InspectiveC

- 通过执行前后时间的记录能够得到每个方法的耗时

获取方法调用树结构

设计两个结构体

CallRecord

```
typedef struct CallRecord_ {
  id obj;   //object_getClass 得到 Class
NSStringFromClass 得类名
  SEL _cmd; //通过 NSStringFromSelector 方
法能够得到方法名
  uintptr_t lr;
  int prevHitIndex;
  char isWatchHit;
} CallRecord;
```

ThreadCallStack

```
typedef struct ThreadCallStack_ {
  CallRecord *stack;
  int allocatedLength;
  int index; //index 记录方法树的深度
  ...
} ThreadCallStack;
```

存储读取

ThreadCallStack

```
static inline ThreadCallStack *
getThreadCallStack() {
  ThreadCallStack *cs = (ThreadCallStack
*)pthread_getspecific(threadKey); //读取
  if (cs == NULL) {
    cs = (ThreadCallStack
*)malloc(sizeof(ThreadCallStack));
    ...
    cs->stack = (CallRecord
*)calloc(DEFAULT_CALLSTACK_DEPTH,
sizeof(CallRecord)); //分配 CallRecord 默认空间
    ...
    pthread_setspecific(threadKey, cs); //保存数据
  }
  return cs;
}
```

记录方法调用深度

APMCon

```
//开始时
static inline void pushCallRecord(id obj, uintptr_t lr, SEL _cmd, ThreadCallStack *cs) {
  int nextIndex = (++cs->index); //增加深度
  if (nextIndex >= cs->allocatedLength) {
    cs->allocatedLength += CALLSTACK_DEPTH_INCREMENT;
    cs->stack = (CallRecord *)realloc(cs->stack, cs->allocatedLength * sizeof(CallRecord));
    cs->spacesStr = (char *)realloc(cs->spacesStr, cs->allocatedLength + 1);
    memset(cs->spacesStr, ' ', cs->allocatedLength);
    cs->spacesStr[cs->allocatedLength] = '\0';
  }
  CallRecord *newRecord = &cs->stack[nextIndex];
  newRecord->obj = obj;
  newRecord->_cmd = _cmd;
  newRecord->lr = lr;
  newRecord->isWatchHit = 0;
}
//结束时
static inline CallRecord * popCallRecord(ThreadCallStack *cs) {
  return &cs->stack[cs->index--]; //减少深度
}
```

objc_msgSend

前后插入执行方法

- 目的是在**调用前**和**调用后**分别加入 **pushCallRecord** 和 **popCallRecord**

- 不可能编写一个保留未知参数并跳转到 c 中任意函数指针的函数，那么这就需要用到**汇编**来做到

- 主要思路就是**先入栈参数**，**x0** 第一个参数是**传入对象**，**x1** 第二个参数是选择器 **_cmd**

- 然后**交换寄存器**中，将用于返回的寄存器 lr 移到 x1 里。先让 **pushCallRecord** 能够执行，再执行原始的 objc_msgSend，保存返回值，最后让 **popCallRecord** 能执行

Hook msgsend 方法

- dyld 是通过更新 Mach-O 二进制的 __DATA segment 特定的部分中的指针来邦定 lazy 和 non-lazy 符号
- 通过确认传递给 rebind_symbol 里每个符号名称更新的位置就可以找出对应替换来重新绑定这些符号

遍历 dyld

```
//首先是遍历 dyld 里的所有的 image，取出 image header 和 slide。注
第一次调用时主要注册 callback。
if (!_rebindings_head->next) {
    _dyld_register_func_for_add_image(_rebind_symbols_for_image);
} else {
    uint32_t c = _dyld_image_count();
    for (uint32_t i = 0; i < c; i++) {
        _rebind_symbols_for_image(_dyld_get_image_header(i),
_dyld_get_image_vmaddr_slide(i));
    }
}
```

找出符号表相关 Command

```
segment_command_t *cur_seg_cmd;
segment_command_t *linkedit_segment = NULL;
struct symtab_command* symtab_cmd = NULL;
struct dysymtab_command* dysymtab_cmd = NULL;

uintptr_t cur = (uintptr_t)header + sizeof(mach_header_t);
for (uint i = 0; i < header->ncmds; i++, cur += cur_seg_cmd->cmdsize) {
    cur_seg_cmd = (segment_command_t *)cur;
    if (cur_seg_cmd->cmd == LC_SEGMENT_ARCH_DEPENDENT) {
        if (strcmp(cur_seg_cmd->segname, SEG_LINKEDIT) == 0) {
            linkedit_segment = cur_seg_cmd;
        }
    } else if (cur_seg_cmd->cmd == LC_SYMTAB) {
        symtab_cmd = (struct symtab_command*)cur_seg_cmd;
    } else if (cur_seg_cmd->cmd == LC_DYSYMTAB) {
        dysymtab_cmd = (struct dysymtab_command*)cur_seg_cmd;
    }
}
```

获得 base 和 indirect 符号表

```
// 找到 base 符号表
uintptr_t linkedit_base = (uintptr_t)slide + linkedit_segment->vmaddr - linkedit_segment->fileoff;
nlist_t *symtab = (nlist_t *)(linkedit_base + symtab_cmd->symoff);
char *strtab = (char *)(linkedit_base + symtab_cmd->stroff);

// 找到 indirect 符号表(array of uint32_t indices into symbol table)
uint32_t *indirect_symtab = (uint32_t *)(linkedit_base + dysymtab_cmd->indirectsymoff);
```

进行方法替换

```
uint32_t *indirect_symbol_indices = indirect_symtab + section->reserved1;
void **indirect_symbol_bindings = (void **)((uintptr_t)slide + section->addr);
for (uint i = 0; i < section->size / sizeof(void *); i++) {
    uint32_t symtab_index = indirect_symbol_indices[i];
    ...
    uint32_t strtab_offset = symtab[symtab_index].n_un.n_strx;
    char *symbol_name = strtab + strtab_offset;
    ...
    struct rebindings_entry *cur = rebindings;
    while (cur) {
        for (uint j = 0; j < cur->rebindings_nel; j++) {
            if (strcmp(&symbol_name[1], cur->rebindings[j].name) == 0) {
                if (cur->rebindings[j].replaced != NULL &&
                    indirect_symbol_bindings[i] != cur->rebindings[j].replacement) {
                    *(cur->rebindings[j].replaced) = indirect_symbol_bindings[i];
                }
                indirect_symbol_bindings[i] = cur->rebindings[j].replacement;
                goto symbol_loop;
            }
        }
        cur = cur->next;
    }
symbol_loop:;
```

总结

- 时间复杂度
- GCD 优化
- I/O 优化
- 控制 Wake 次数
- 检测方法

Thanks

- Slides demo： https://github.com/ming1016/DecoupleDemo
- Slides 文章：https://ming1016.github.io/2017/06/20/deeply-ios-performance-optimization/
- 微博：@戴铭
- 博客：https://ming1016.github.io