iOS 底层解析 weak 的实现原理（包含 weak 对象的初始化，引用，释放的分析）

原文

很少有人知道 weak 表其实是一个 hash（哈希）表，Key 是所指对象的地址，Value 是 weak 指针的地址数组。更多人的人只是知道 weak 是弱引用，所引用对象的计数器不会加一，并在引用对象被释放的时候自动被设置为 nil。通常用于解决循环引用问题。但现在单知道这些已经不足以应对面试了，好多公司会问 weak 的原理。weak 的原理是什么呢？下面就分析一下 weak 的工作原理（只是自己对这个问题好奇，学习过程中的笔记，希望对读者也有所帮助）。

weak 实现原理的概括

Runtime 维护了一个 weak 表，用于存储指向某个对象的所有 weak 指针。weak 表其实是一个 hash（哈希）表，Key 是所指对象的地址，Value 是 weak 指针的地址（这个地址的值是所指对象的地址）数组。

weak 的实现原理可以概括一下三步：

1、初始化时：runtime 会调用 objc_initWeak 函数，初始化一个新的 weak 指针指向对象的地址。

2、添加引用时：objc_initWeak 函数会调用 objc_storeWeak() 函数，objc_storeWeak() 的作用是更新指针指向，创建对应的弱引用表。

3、释放时，调用 clearDeallocating 函数。clearDeallocating 函数首先根据对象地址获取所有 weak 指针地址的数组，然后遍历这个数组把其中的数据设为 nil，最后把这个 entry 从 weak 表中删除，最后清理对象的记录。

下面将开始详细介绍每一步：

1、初始化时：runtime 会调用 objc_initWeak 函数，objc_initWeak 函数会初始化一个新的 weak 指针指向对象的地址。

示例代码：

```
{
    NSObject *obj = [[NSObject alloc] init];
    id __weak obj1 = obj;
}
```

当我们初始化一个 weak 变量时，runtime 会调用 NSObject.mm 中的 objc_initWeak 函数。这个函数在 Clang 中的声明如下：

```
id objc_initWeak(id *object, id value);
```

而对于 objc_initWeak() 方法的实现

```
id objc_initWeak(id *location, id newObj) {
// 查看对象实例是否有效
// 无效对象直接导致指针释放
    if (!newObj) {
        *location = nil;
```

```
            return nil;
        }
        // 这里传递了三个 bool 数值
        // 使用 template 进行常量参数传递是为了优化性能
        return storeWeakfalse/*old*/, true/*new*/, true/*cras
        (location, (objc_object*)newObj);
    }
```

可以看出，这个函数仅仅是一个深层函数的调用入口，而一般的入口函数中，都会做一些简单的判断（例如 objc_msgSend 中的缓存判断），这里判断了其指针指向的类对象是否有效，无效直接释放，不再往深层调用函数。否则，object 将被注册为一个指向 value 的__weak 对象。而这事应该是 objc_storeWeak 函数干的。

注意：objc_initWeak 函数有一个前提条件：就是 object 必须是一个没有被注册为__weak 对象的有效指针。而 value 则可以是 null，或者指向一个有效的对象。

2、添加引用时：objc_initWeak 函数会调用 objc_storeWeak() 函数，objc_storeWeak() 的作用是更新指针指向，创建对应的弱引用表。

objc_storeWeak 的函数声明如下：

```
id objc_storeWeak(id *location, id value);
```

objc_storeWeak() 的具体实现如下：

```
// HaveOld:      true - 变量有值
//              false - 需要被及时清理，当前值可能为 nil
// HaveNew:      true - 需要被分配的新值，当前值可能为 nil
//              false - 不需要分配新值
// CrashIfDeallocating: true - 说明 newObj 已经释放或者 new
//              false - 用 nil 替代存储
template bool HaveOld, bool HaveNew, bool CrashIfDealloca
static id storeWeak(id *location, objc_object *newObj) {
    // 该过程用来更新弱引用指针的指向
    // 初始化 previouslyInitializedClass 指针
    Class previouslyInitializedClass = nil;
    id oldObj;
    // 声明两个 SideTable
    // ① 新旧散列创建
    SideTable *oldTable;
    SideTable *newTable;
    // 获得新值和旧值的锁存位置（用地址作为唯一标示）
    // 通过地址来建立索引标志，防止桶重复
    // 下面指向的操作会改变旧值
retry:
```

```
if (HaveOld) {
    // 更改指针，获得以 oldObj 为索引所存储的值地址
    oldObj = *location;
    oldTable = &SideTables()[oldObj];
} else {
    oldTable = nil;
}
if (HaveNew) {
    // 更改新值指针，获得以 newObj 为索引所存储的值地址
    newTable = &SideTables()[newObj];
} else {
    newTable = nil;
}
// 加锁操作，防止多线程中竞争冲突
SideTable::lockTwoHaveOld, HaveNew>(oldTable, newTabl
// 避免线程冲突重处理
// location 应该与 oldObj 保持一致，如果不同，说明当前的 lo
if (HaveOld  &&  *location != oldObj) {
    SideTable::unlockTwoHaveOld, HaveNew>(oldTable, r
    goto retry;
}
// 防止弱引用间死锁
// 并且通过 +initialize 初始化构造器保证所有弱引用的 isa 非
if (HaveNew  &&  newObj) {
    // 获得新对象的 isa 指针
    Class cls = newObj->getIsa();
    // 判断 isa 非空且已经初始化
    if (cls != previouslyInitializedClass  &&
        !((objc_class *)cls)->isInitialized()) {
        // 解锁
        SideTable::unlockTwoHaveOld, HaveNew>(oldTabl
        // 对其 isa 指针进行初始化
        _class_initialize(_class_getNonMetaClass(cls,
        // 如果该类已经完成执行 +initialize 方法是最理想情
        // 如果该类 +initialize 在线程中
        // 例如 +initialize 正在调用 storeWeak 方法
        // 需要手动对其增加保护策略，并设置 previouslyInit
        previouslyInitializedClass = cls;
        // 重新尝试
        goto retry;
    }
}
// ② 清除旧值
if (HaveOld) {
    weak_unregister_no_lock(&oldTable->weak_table, ol
}
// ③ 分配新值
if (HaveNew) {
    newObj = (objc_object *)weak_register_no_lock(&ne
                                                  (ic
                                                  Cra
    // 如果弱引用被释放 weak_register_no_lock 方法返回 ni
    // 在引用计数表中设置若引用标记位
    if (newObj  &&  !newObj->isTaggedPointer()) {
        // 弱引用位初始化操作
```

```
                // 引用计数那张散列表的weak引用对象的引用计数中标识为
                newObj->setWeaklyReferenced_nolock();
            }
            // 之前不要设置 location 对象，这里需要更改指针指向
            *location = (id)newObj;
        }
        else {
            // 没有新值，则无需更改
        }
        SideTable::unlockTwoHaveOld, HaveNew>(oldTable, newTa
        return (id)newObj;
    }
```

撇开源码中各种锁操作，来看看这段代码都做了些什么。

1)、SideTable

SideTable 这个结构体，我给他起名引用计数和弱引用依赖表，因为它主要用于管理对象的引用计数和 weak 表。在 NSObject.mm 中声明其数据结构：

```
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;
}
```

对于 slock 和 refcnts 两个成员不用多说，第一个是为了防止竞争选择的自旋锁，第二个是协助对象的 isa 指针的 extra_rc 共同引用计数的变量（对于对象结果，在今后的文中提到）。这里主要看 weak 全局 hash 表的结构与作用。

2)、weak 表

weak 表是一个弱引用表，实现为一个 weak_table_t 结构体，存储了某个对象相关的的所有的弱引用信息。其定义如下 (具体定义在 objc-weak.h 中)：

```
    struct weak_table_t {
     // 保存了所有指向指定对象的 weak 指针
     weak_entry_t *weak_entries;
     // 存储空间
     size_t    num_entries;
     // 参与判断引用计数辅助量
     uintptr_t mask;
```

```
    // hash key 最大偏移值
    uintptr_t max_hash_displacement;
};
```

这是一个全局弱引用 hash 表。使用不定类型对象的地址作为 key ，用 weak_entry_t 类型结构体对象作为 value 。其中的 weak_entries 成员，从字面意思上看，即为弱引用表入口。其实现也是这样的。

其中 weak_entry_t 是存储在弱引用表中的一个内部结构体，它负责维护和存储指向一个对象的所有弱引用 hash 表。其定义如下：

```
typedef objc_object ** weak_referrer_t;
struct weak_entry_t {
    DisguisedPtrobjc_object> referent;
    union {
        struct {
            weak_referrer_t *referrers;
            uintptr_t        out_of_line : 1;
            uintptr_t        num_refs : PTR_MINUS_1;
            uintptr_t        mask;
            uintptr_t        max_hash_displacement;
        };
        struct {
            // out_of_line=0 is LSB of one of these (don'
            weak_referrer_t  inline_referrers[WEAK_INLINE
        };
    }
}
```

在 weak_entry_t 的结构中，DisguisedPtr referent 是对泛型对象的指针做了一个封装，通过这个泛型类来解决内存泄漏的问题。从注释中写 out_of_line 成员为最低有效位，当其为 0 的时候， weak_referrer_t 成员将扩展为多行静态 hash table。其实其中的 weak_referrer_t 是二维 objc_object 的别名，通过一个二维指针地址偏移，用下标作为 hash 的 key，做成了一个弱引用散列。

那么在有效位未生效的时候，out_of_line 、 num_refs、 mask 、 max_hash_displacement 有什么作用？以下是笔者自身的猜测：

out_of_line：最低有效位，也是标志位。当标志位 0 时，增加引用表指针纬度。

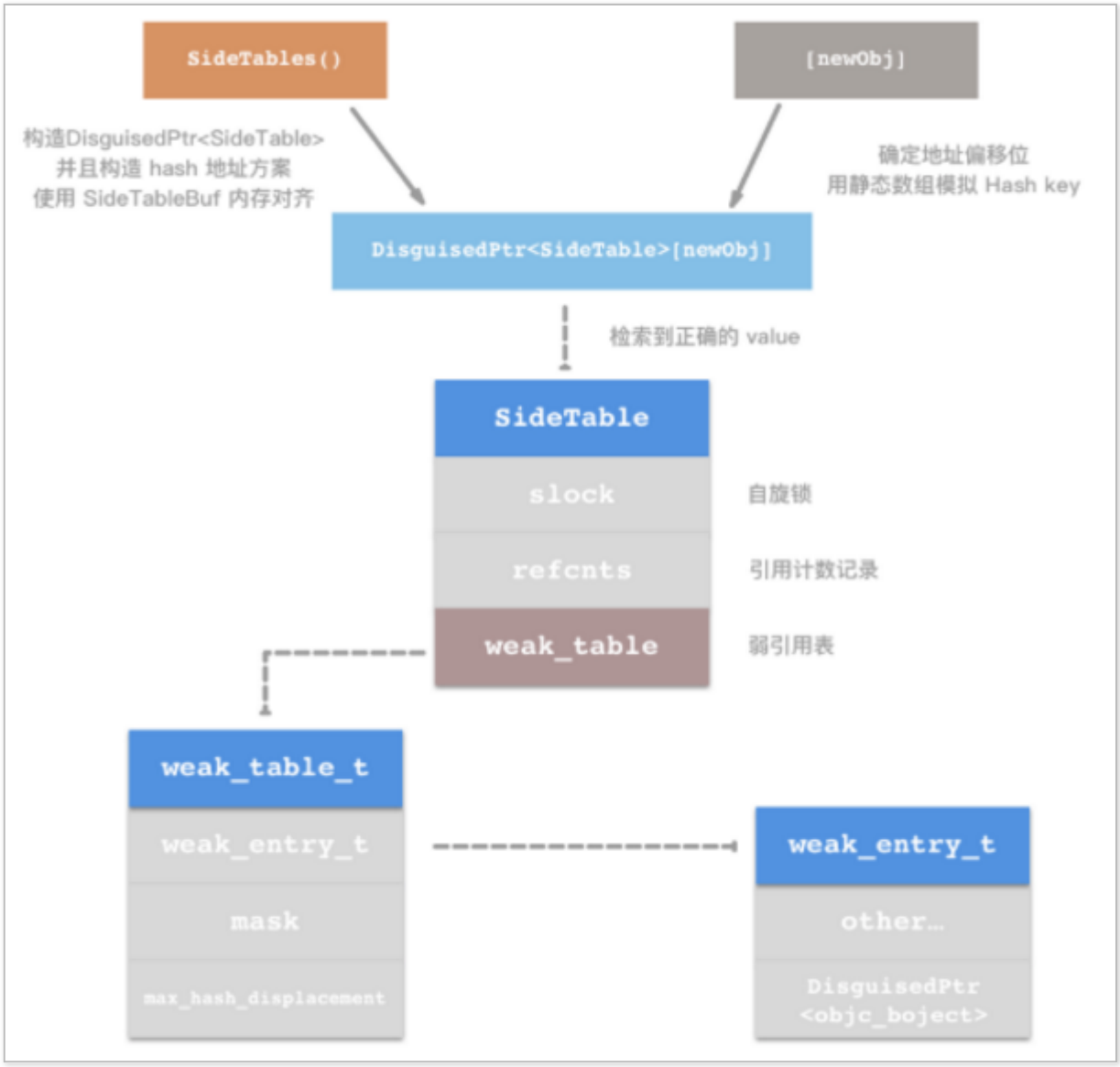
num_refs：引用数值。这里记录弱引用表中引用有效数字，因为弱引用表使用的是静态 hash 结构，所以需要使用变量来记录数目。

mask：计数辅助量。

max_hash_displacement：hash 元素上限阀值。

其实 out_of_line 的值通常情况下是等于零的，所以弱引用表总是一个 objc_objective 指针二维数组。一维 objc_objective 指针可构成一张弱引用散列表，通过第三纬度实现了多张散列表，并且表数量为 WEAK_INLINE_COUNT 。

总结一下 StripedMap[]： StripedMap 是一个模板类，在这个类中有一个 array 成员，用来存储 PaddedT 对象，并且其中对于 [] 符的重载定义中，会返回这个 PaddedT 的 value 成员，这个 value 就是我们传入的 T 泛型成员，也就是 SideTable 对象。在 array 的下标中，这里使用了 indexForPointer 方法通过位运算计算下标，实现了静态的 Hash Table。而在 weak_table 中，其成员 weak_entry 会将传入对象的地址加以封装起来，并且其中也有访问全局弱引用表的入口。



旧对象解除注册操作 weak_unregister_no_lock

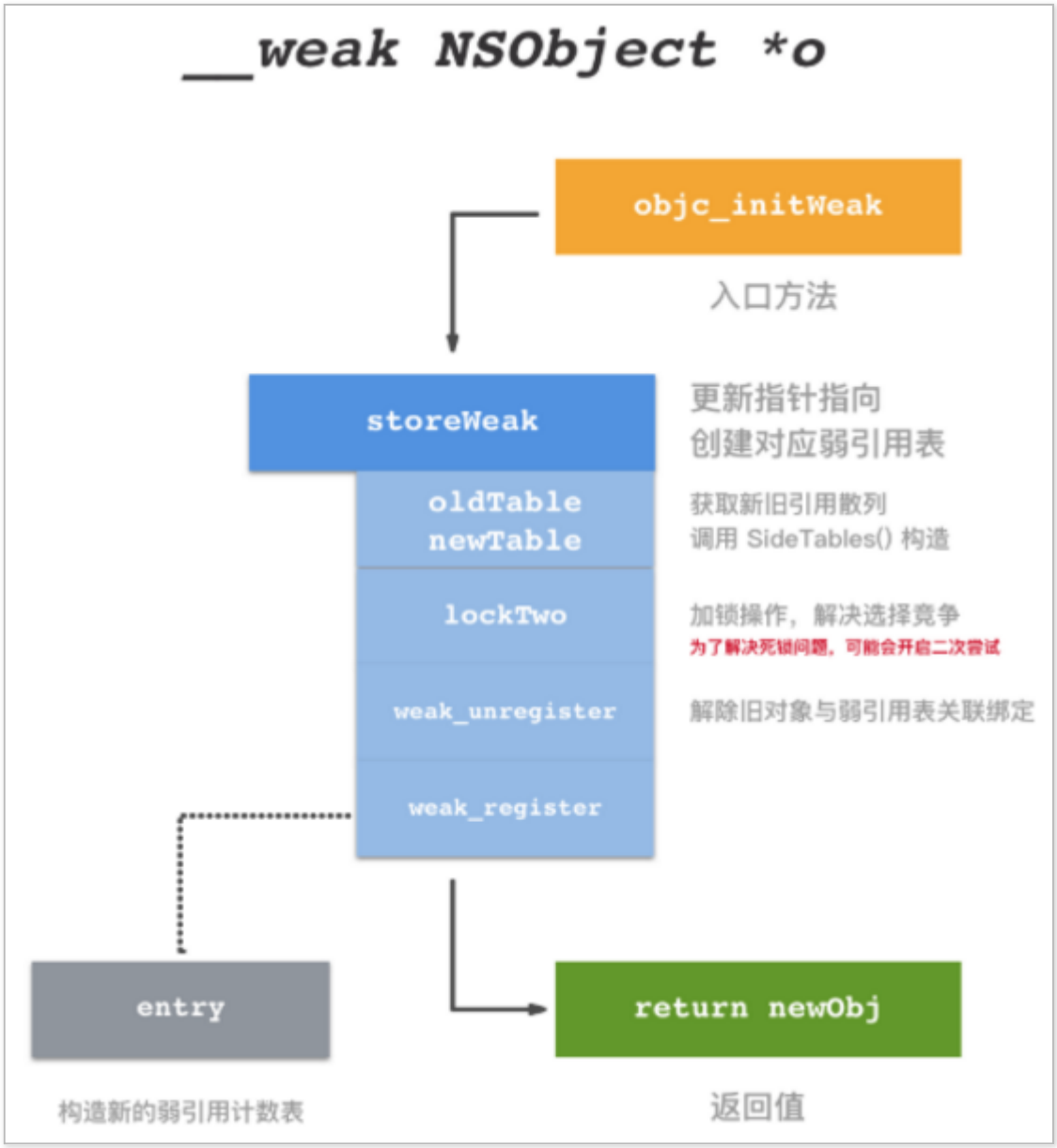
该方法主要作用是将旧对象在 weak_table 中接触 weak 指针的对应绑定。根据函数名，称之为解除注册操作。从源码中，可以知道其功能就是从 weak_table 中接触 weak 指针的绑定。而其中的遍历查询，就是针对于 weak_entry 中的多张弱引用散列表。

新对象添加注册操作 weak_register_no_lock

这一步与上一步相反，通过 weak_register_no_lock 函数把心的对象进行注册操作，完成与对应的弱引用表进行绑定操作。

初始化弱引用对象流程一览

弱引用的初始化，从上文的分析中可以看出，主要的操作部分就在弱引用表的取键、查询散列、创建弱引用表等操作，可以总结出如下的流程图：



这个图中省略了很多情况的判断，但是当声明一个 weak 会调用上图中的这些方法。当然， storeWeak 方法不仅仅用在 weak 的声明中，在 class 内部的操作中也会常常通过该方法来对 weak 对象进行操作。

3、释放时，调用 clearDeallocating 函数。clearDeallocating 函数首先根据对象地址获取所有 weak 指针地址的数组，然后遍历这个数组把其中的数据设为 nil，最后把这个 entry 从 weak 表中删除，最后清理对象的记录。

当 weak 引用指向的对象被释放时，又是如何去处理 weak 指针的呢？当释放对象时，其基本流程如下：

1、调用 objc_release

2、因为对象的引用计数为 0，所以执行 dealloc

3、在 dealloc 中，调用了_objc_rootDealloc 函数

4、在_objc_rootDealloc 中，调用了 object_dispose 函数

5、调用 objc_destructInstance

6、最后调用 objc_clear_deallocating

重点看对象被释放时调用的 objc_clear_deallocating 函数。该函数实现如下：

```
void  objc_clear_deallocating(id obj)
{
    assert(obj);
    assert(!UseGC);
    if (obj->isTaggedPointer()) return;
   obj->clearDeallocating();
}
```

也就是调用了 clearDeallocating，继续追踪可以发现，它最终是使用了迭代器来取 weak 表的 value，然后调用 weak_clear_no_lock, 然后查找对应的 value，将该 weak 指针置空，weak_clear_no_lock 函数的实现如下：

```
/**
 * Called by dealloc; nils out all weak pointers that poi
 * provided object so that they can no longer be used.
 *
 * @param weak_table
 * @param referent The object being deallocated.
 */
void weak_clear_no_lock(weak_table_t *weak_table, id refe
{
    objc_object *referent = (objc_object *)referent_id;
    weak_entry_t *entry = weak_entry_for_referent(weak_ta
    if (entry == nil) {
        /// XXX shouldn't happen, but does with mismatche
        //printf("XXX no entry for clear deallocating %p\
        return;
    }
    // zero out references
    weak_referrer_t *referrers;
    size_t count;

    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];
            if (referrer) {
                if (*referrer == referent) {
                    *referrer = nil;
                }
                else if (*referrer) {
                    _objc_inform("__weak variable at %p holds
                                 "This is probably incorrect
                                 "objc_storeWeak() and objc_l
                                 "Break on objc_weak_error to
                                 referrer, (void*)*referrer,
                    objc_weak_error();
                }
            }
        }
        weak_entry_remove(weak_table, entry);
    }
```

objc_clear_deallocating 该函数的动作如下：

1、从 weak 表中获取废弃对象的地址为键值的记录

2、将包含在记录中的所有附有 weak 修饰符变量的地址，赋值为 nil

3、将 weak 表中该记录删除

4、从引用计数表中删除废弃对象的地址为键值的记录

看了 objc-weak.mm 的源码就明白了：其实 Weak 表是一个 hash（哈希）表，然后里面的 key 是指向对象的地址，Value 是 Weak 指针的地址的数组。

补充：.m 和. mm 的区别

.m：源代码文件，这个典型的源代码文件扩展名，可以包含 OC 和 C 代码。

.mm：源代码文件，带有这种扩展名的源代码文件，除了可以包含 OC 和 C 代码之外，还可以包含 C++ 代码。仅在你的 OC 代码中确实需要使用 C++ 类或者特性的时候才用这种扩展名。

参考资料：

- weak 弱引用的实现方式
- weak 的生命周期：具体实现方法

全文完