Weak 底层原理

参考并感谢：
[iOS weak实现流程]
[iOS 底层解析weak的实现原理]
[iOS weak 原理]

weak关键字基础常识

`weak` 关键字修饰的对象指针是弱引用，被引用对象的引用计数不会+1,并且引用对象被释放时会被自动置为nil（`assign` 不会自动置为nil）

`weak` 和 `asign` 区别

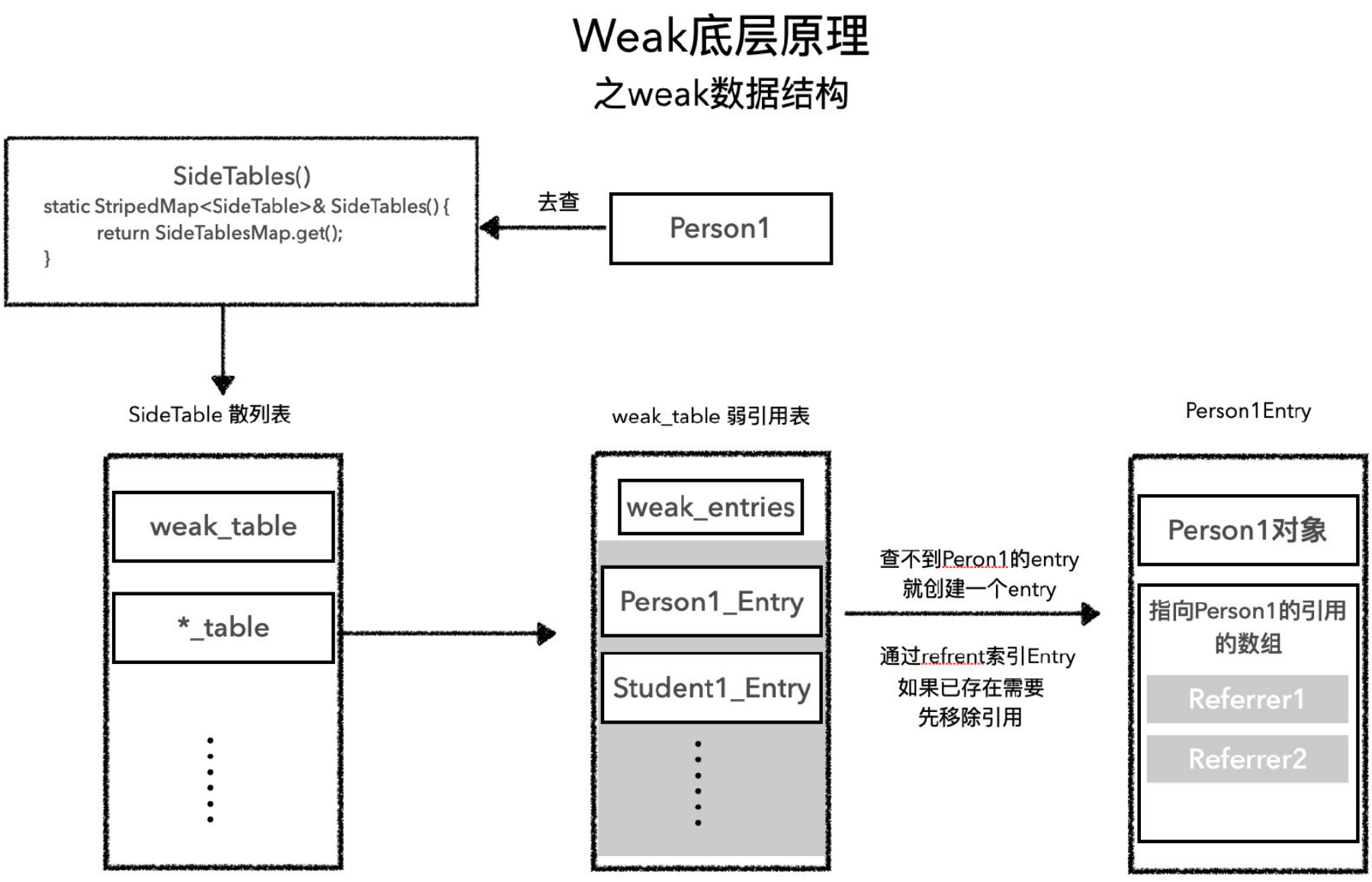
- 修饰类型

`assign` 既可以修饰基本数据类型 `int` `long` `double` 等，也可以修饰OC的继承于 `NSObject` 的对象类型。
`weak` 只能修饰继承于 `NSObject` 的对象类型，不能用于修饰基本数据类型。

- 对象释放

`weak` 修饰的对象释放时，指针会自动置为 `nil` ,不会产生野指针。
`assign` 修饰的对象释放时，不会自动置为 `nil` ，访问已经释放的对象时候会产生野指针，会导致 `EXC_BAD_ACCESS` 错误

weak实现相关数据结构



SideTables()

SideTables()是一个静态函数，代码如下：

```
static objc::ExplicitInit<StripedMap<SideTable>> SideTablesMap;

static StripedMap<SideTable>& SideTables() {
    return SideTablesMap.get();
}
```

函数体里面调用了一个全局的静态变量SideTablesMap的 get()方法，这个静态变量保存了所有的SideTable，是objc命名空间下的一个ExplicitInit类，它里面实现了get()方法，如下：

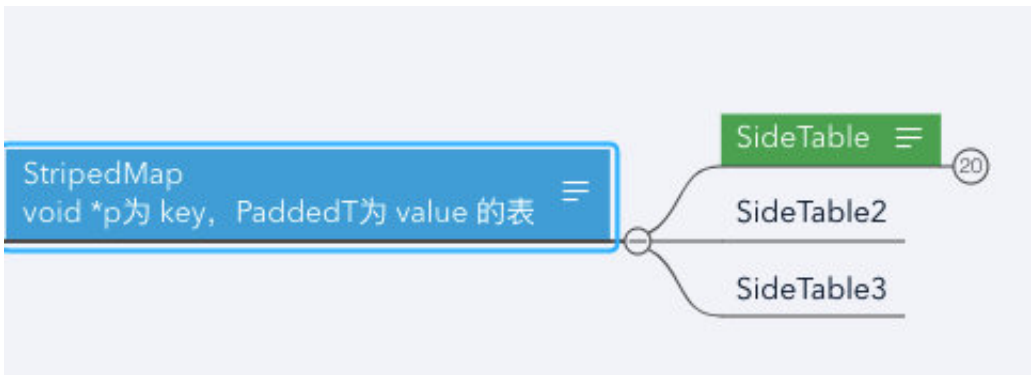
```
Type &get() {
        return *reinterpret_cast<Type *>(_storage);
}
```

这个get()方法其实返回的就是StripedMap类的实例。

StripedMap

StripedMap 是一个以void *p为 key，PaddedT为 value 的表:

```
template<typename T>
class StripedMap {
#if TARGET_OS_IPHONE && !TARGET_OS_SIMULATOR
    enum { StripeCount = 8 }; // 真机下StripedMap存储的 SideTable 数量为 8
#else
    enum { StripeCount = 64 }; // 模拟器下为 64
#endif

    // 对于SideTable的封装
    struct PaddedT {
        T value alignas(CacheLineSize);
    };

    // 存储 PaddedT 的散列表
    PaddedT array[StripeCount];

    // 散列函数，通过对象地址计算出对应 PaddedT在数组中的下标
    static unsigned int indexForPointer(const void *p) {
        uintptr_t addr = reinterpret_cast<uintptr_t>(p);
        return ((addr >> 4) ^ (addr >> 9)) % StripeCount;
    }

 public:
    // 取值操作，重写了[ ]方法，上面提到的&SideTables()[oldObj]会调用到这个方法
    T& operator[] (const void *p) {
        return array[indexForPointer(p)].value;
    }
    const T& operator[] (const void *p) const {
        return const_cast<StripedMap<T>>(this)[p];
    }

   // ... 省略一些方法
};
```

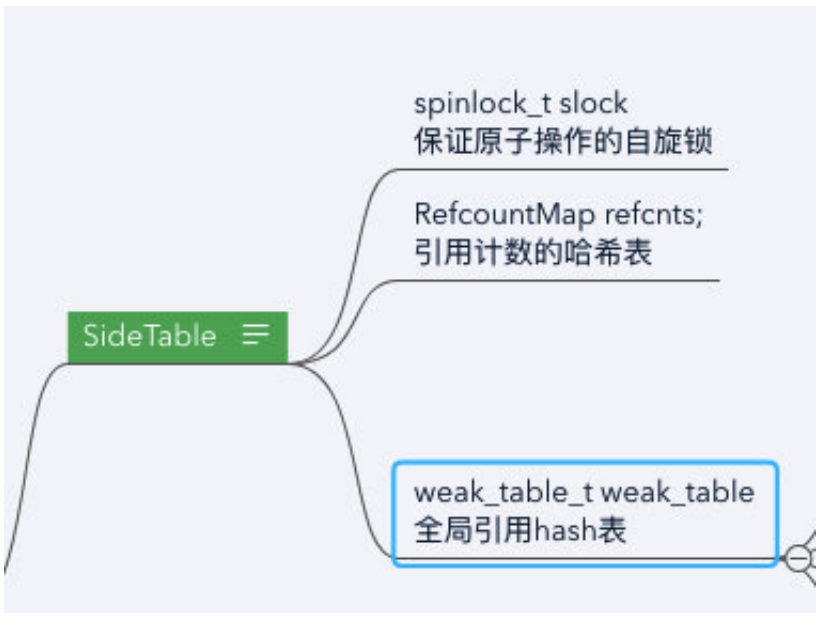


SideTable

StripedMap其内部有一个哈希表，表中存储的是 PaddedT 结构体，结构体的 value 就是 SideTable，其源码如下：

```
struct SideTable {
    spinlock_t slock; // 保证原子操作的自旋锁
    RefcountMap refcnts; // 引用计数的 hash 表
```

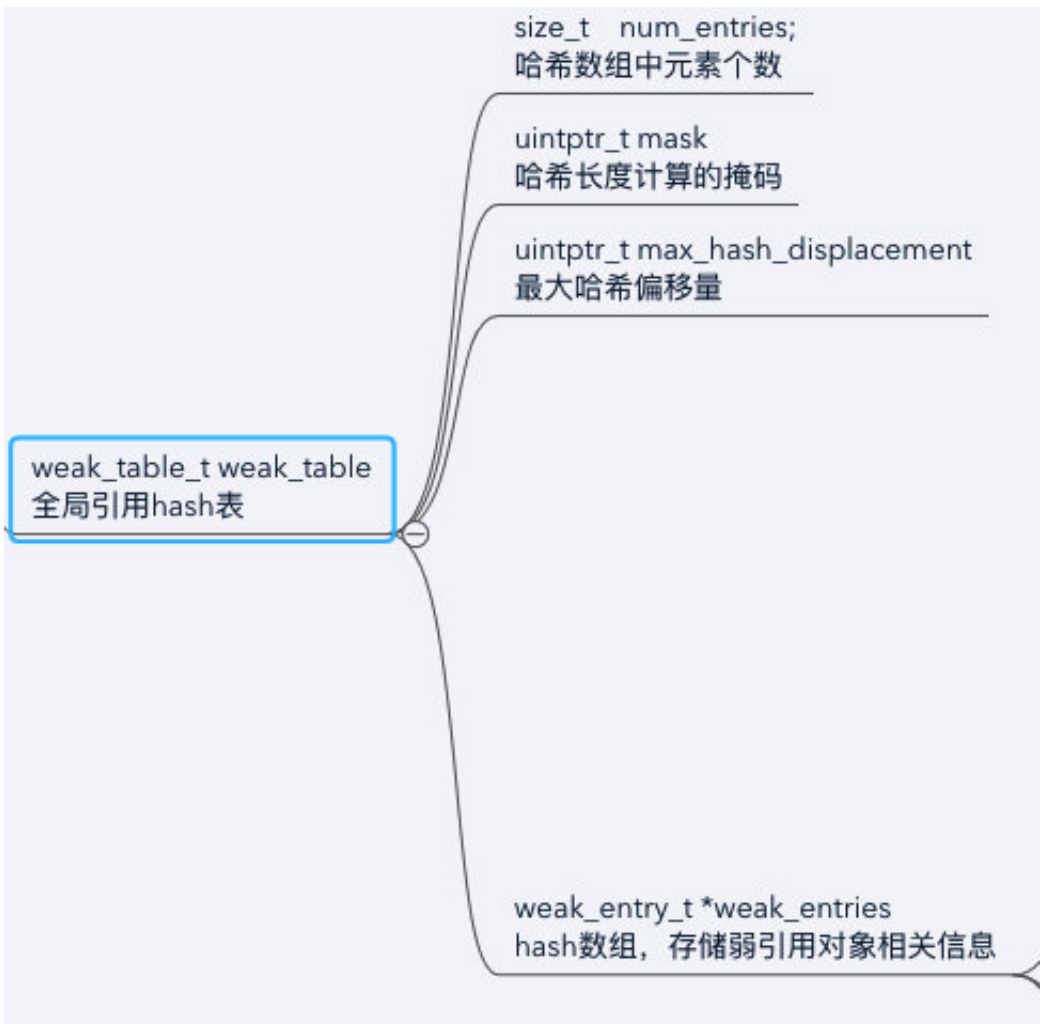
```
    weak_table_t weak_table;  // weak 引用全局 hash 表
    // 构造函数
    SideTable() {
        memset(&weak_table, 0, sizeof(weak_table));
    }
    // 析构函数
    ~SideTable() {
        _objc_fatal("Do not delete SideTable.");
    }
};
```



weak_table_t

SideTable 结构体中有一个weak_table_t结构体类型的成员变量，源码实现：

```
// 全局弱引用表
struct weak_table_t {
    weak_entry_t *weak_entries; // hash 数组，用来存储弱引用对象相关信息的 weak_entry_t
    size_t    num_entries; // hash数组中元素的个数
    uintptr_t mask; // hash 数组的长度（并不是实际的存储个数）-1，主要用来参与哈希函数
    uintptr_t max_hash_displacement; // 最大哈希偏移值
};
```



weak_entry_t

```
struct weak_entry_t {
    DisguisedPtr<objc_object> referent; // 弱引用的对象
    union {
        // 弱引用数量大于 4 个用到的结构体
        struct {
            weak_referrer_t *referrers; // 存储指向该对象的弱引用
            uintptr_t        out_of_line_ness : 2;
            uintptr_t        num_refs : PTR_MINUS_2;
            uintptr_t        mask;
            uintptr_t        max_hash_displacement;
        };
        // 弱引用数量不大于 4 个用到的结构体
        struct {
            // out_of_line_ness field is low bits of inline_referrers[1]
            weak_referrer_t  inline_referrers[WEAK_INLINE_COUNT];  // 存储指向该对象的弱引用数组
        };
    };

    // 判断是否用的是 referrers 来存储弱引用指针
    bool out_of_line() {
        return (out_of_line_ness == REFERRERS_OUT_OF_LINE);
    }
    // 覆盖老数据
    weak_entry_t& operator=(const weak_entry_t& other) {
        memcpy(this, &other, sizeof(other));
        return *this;
    }
    // 构造方法
    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;
        }
    }
};
```

weak_entry_t内部优化设计：

- 定长数组优化内存

> *一个对象的弱引用数量 小于等于 WEAK_INLINE_COUNT 个的话，runtime会创建一个长度为4的定长数组。弱引用会*

> *被一次保存到inline数组里。inline数组内存毁在weak_entry_t初始化的时候分配好。*

- 联合体union中的两个struct共享内存的

> *联合体union中的两个struct共享内存的.如果不使用inline数组，而直接使用HashTable的方式来实现，那么num_refs，mask和max_hash_displacement这些变量都需要单独的存储空间，会使用更多的内存。综上，使用inline数组在节约一定内存空间的同时还相对提高了运行效率*

对weak_entry_t主要操作的函数是grow_refs_and_insert和append_referrer

append_referrer

为一个对象保存新的弱引用。

```
static void append_referrer(weak_entry_t *entry, objc_object **new_referrer)
{
    if (! entry->out_of_line()) {
        // Try to insert inline.
        for (size_t i = 0; i < WEAK_INLINE_COUNT; i++) {
            if (entry->inline_referrers[i] == nil) {
                entry->inline_referrers[i] = new_referrer;
                return;
            }
        }

        // Couldn't insert inline. Allocate out of line.
        weak_referrer_t *new_referrers = (weak_referrer_t *)
            calloc(WEAK_INLINE_COUNT, sizeof(weak_referrer_t));
        // This constructed table is invalid, but grow_refs_and_insert
        // will fix it and rehash it.
        for (size_t i = 0; i < WEAK_INLINE_COUNT; i++) {
            new_referrers[i] = entry->inline_referrers[i];
        }
        entry->referrers = new_referrers;
        entry->num_refs = WEAK_INLINE_COUNT;
        entry->out_of_line_ness = REFERRERS_OUT_OF_LINE;
        entry->mask = WEAK_INLINE_COUNT-1;
        entry->max_hash_displacement = 0;
    }

    assert(entry->out_of_line());

    if (entry->num_refs >= TABLE_SIZE(entry) * 3/4) {
        return grow_refs_and_insert(entry, new_referrer);
    }
    size_t begin = w_hash_pointer(new_referrer) & (entry->mask);
    size_t index = begin;
    size_t hash_displacement = 0;
    while (entry->referrers[index] != nil) {
        hash_displacement++;
        index = (index+1) & entry->mask;
        if (index == begin) bad_weak_table(entry);
    }
    if (hash_displacement > entry->max_hash_displacement) {
        entry->max_hash_displacement = hash_displacement;
    }
    weak_referrer_t &ref = entry->referrers[index];
    ref = new_referrer;
    entry->num_refs++;
}
```

- `append_referrer` 函数首先处理 `weak_entry_t` 还在使用 `inline` 数组的情况。首先尝试像 `inline` 数组中插入一个新的弱引用，如果 `inline` 数组已满，那就创建一个 `WEAK_INLINE_COUNT` 大小的新数组，改用 `outline` 的方式，将 `inline` 数组中的元素依次拷贝过来。
- 函数的后半部分处理使用 `outline` 数组的情况，如果 `outline` 数组的使用率在75%及以上，那么调用 `grow_refs_and_insert` 函数进

行扩充，并且插入新的弱引用。否则就直接进行 `hash` 运算插入，过程和 `weak_table_t` 的插入过程基本相同

grow_refs_and_insert

> ***扩充定长数组***
> `grow_refs_and_insert` *函数首先对outline数组进行扩充，容量是原来的两倍。而后依次将老数组中的元素hash插入到新数组中，最终hash插入新的引用。*

```
__attribute__((noinline, used))
static void grow_refs_and_insert(weak_entry_t *entry,
                                 objc_object **new_referrer)
{
    assert(entry->out_of_line());

    size_t old_size = TABLE_SIZE(entry);
    size_t new_size = old_size ? old_size * 2 : 8;

    size_t num_refs = entry->num_refs;
    weak_referrer_t *old_refs = entry->referrers;
    entry->mask = new_size - 1;

    entry->referrers = (weak_referrer_t *)
        calloc(TABLE_SIZE(entry), sizeof(weak_referrer_t));
    entry->num_refs = 0;
    entry->max_hash_displacement = 0;

    for (size_t i = 0; i < old_size && num_refs > 0; i++) {
        if (old_refs[i] != nil) {
            append_referrer(entry, old_refs[i]);
            num_refs--;
        }
    }
    // Insert
    append_referrer(entry, new_referrer);
    if (old_refs) free(old_refs);
}
```

remove_referrer

> `remove_referrer` *函数负责删除一个弱引用。*
> *函数首先处理inline数组的情况，直接将对应的弱引用项置空。如果使用了outline数组，则通过hash找到要删除的项，并直接删除*

```
static void remove_referrer(weak_entry_t *entry, objc_object **old_referrer)
{
    if (! entry->out_of_line()) {
        for (size_t i = 0; i < WEAK_INLINE_COUNT; i++) {
            if (entry->inline_referrers[i] == old_referrer) {
                entry->inline_referrers[i] = nil;
                return;
            }
        }
        _objc_inform("Attempted to unregister unknown __weak variable "
                     "at %p. This is probably incorrect use of "
                     "objc_storeWeak() and objc_loadWeak(). "
                     "Break on objc_weak_error to debug.\n",
                     old_referrer);
        objc_weak_error();
        return;
    }

    size_t begin = w_hash_pointer(old_referrer) & (entry->mask);
    size_t index = begin;
    size_t hash_displacement = 0;
    while (entry->referrers[index] != old_referrer) {
```

```
        index = (index+1) & entry->mask;
        if (index == begin) bad_weak_table(entry);
        hash_displacement++;
        if (hash_displacement > entry->max_hash_displacement) {
            _objc_inform("Attempted to unregister unknown __weak variable "
                         "at %p. This is probably incorrect use of "
                         "objc_storeWeak() and objc_loadWeak(). "
                         "Break on objc_weak_error to debug.\n",
                         old_referrer);
            objc_weak_error();
            return;
        }
    }
    entry->referrers[index] = nil;
    entry->num_refs--;
}
```

weak引用实现流程

初始化： objc_initWeak

> *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*

```
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/*crash*/>
    (location, (objc_object*)newObj);
}
```

添加引用时：objc_initWeak函数会调用 objc_storeWeak() 函数.

添加引用时:objc_storeWeak()

> *objc_storeWeak() 的作用是更新指针指向，创建对应的弱引用表.*

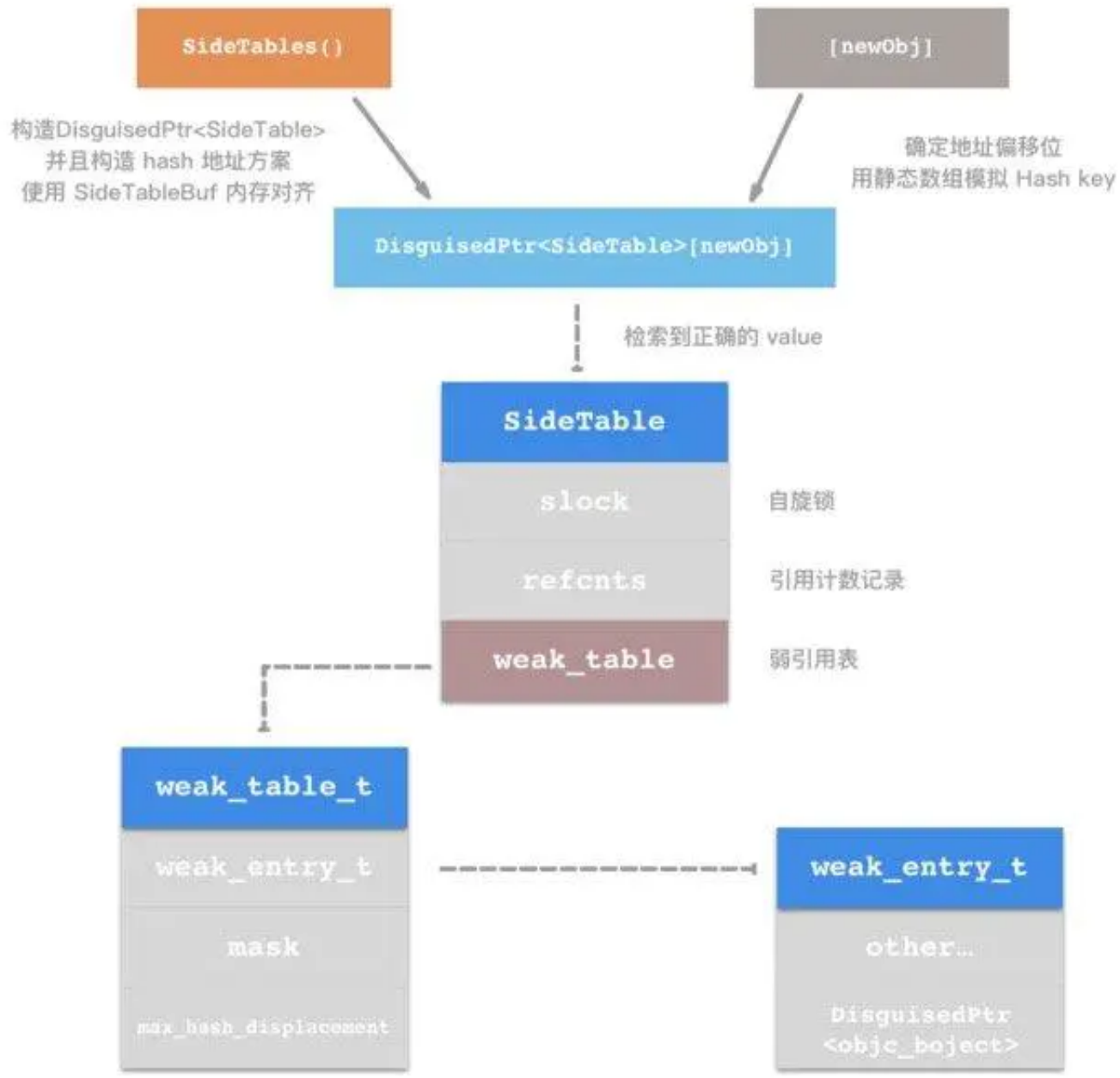
```
static id storeWeak(id *location, objc_object *newObj) {
    // 该过程用来更新弱引用指针的指向
    // 初始化 previouslyInitializedClass 指针
    Class previouslyInitializedClass = nil;
    id oldObj;
    // 声明两个 SideTable
    // ① 新旧散列创建
    SideTable *oldTable;
    SideTable *newTable;
    // 获得新值和旧值的锁存位置（用地址作为唯一标示）
    // 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, newTable);
    // 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 location 已经处理过 oldObj 可是又被其他线程所修改
    if (HaveOld  &&  *location != oldObj) {
        SideTable::unlockTwoHaveOld, HaveNew>(oldTable, newTable);
        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 非空指向
    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>(oldTable, newTable);
            // 对其 isa 指针进行初始化
            _class_initialize(_class_getNonMetaClass(cls, (id)newObj));
            // 如果该类已经完成执行 +initialize 方法是最理想情况
            // 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 previouslyInitializedClass 指针进行标记
            previouslyInitializedClass = cls;
            // 重新尝试
            goto retry;
        }
    }
    // ② 清除旧值
    if (HaveOld) {
        weak_unregister_no_lock(&oldTable->weak_table, oldObj, location);
    }
    // ③ 分配新值
    if (HaveNew) {
        newObj = (objc_object *)weak_register_no_lock(&newTable->weak_table,
                                                      (id)newObj, location,
                                                      CrashIfDeallocating);
        // 如果弱引用被释放 weak_register_no_lock 方法返回 nil
        // 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weak引用
            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, newTable);
    return (id)newObj;
}
```

weak指针进行的一些操作：

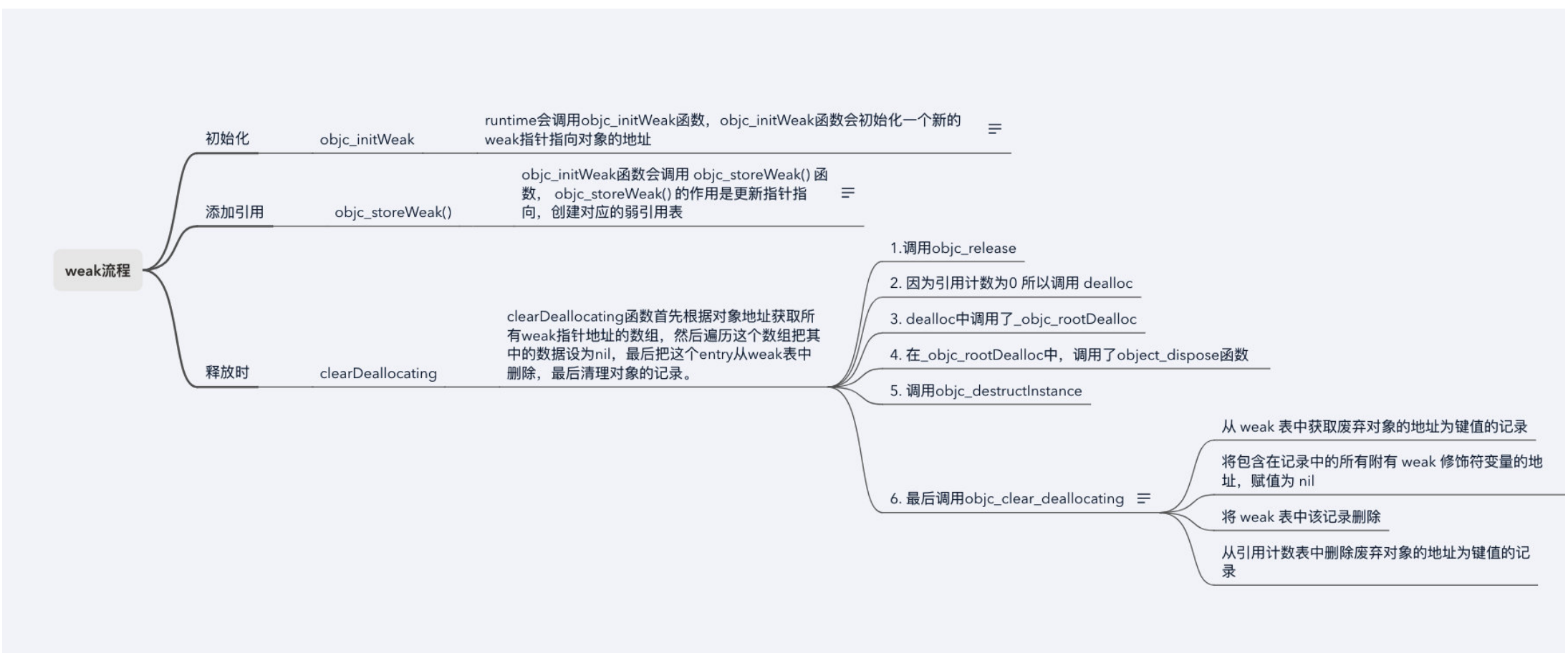
1. 从全局 `SideTables()` 中获取对象所在的 `SideTable`
2. `isa` 的非空校验，如果isa没有被初始化，则执行 `class_initialize(cls, (id)newObj)` 方法
3. 如果 `weak` 指针之前指向了别的对象，就**解除对旧对象的引用**
4. 注册新对象的弱引用

5. 设置新对象的弱引用标志符为 YES

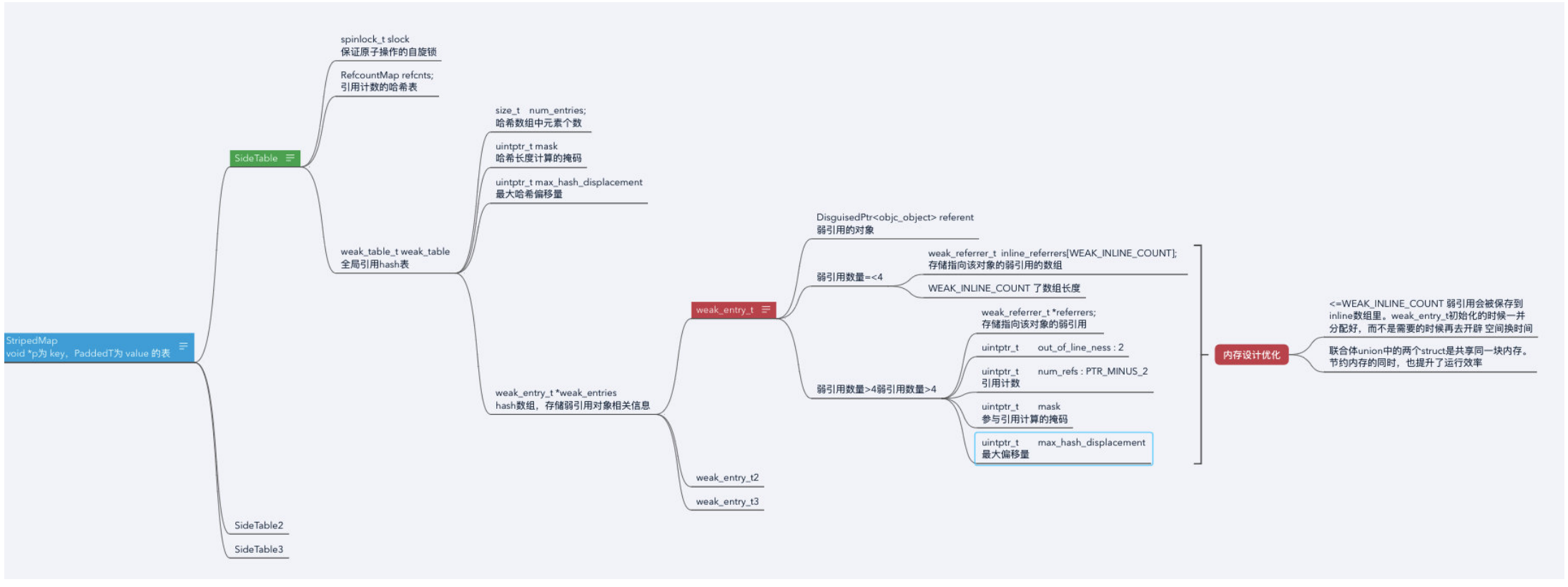


销毁：dealloc

`objc_release` -->
`_objc_rootDealloc` -->
`object_dispose` -->
`objc_destructInstance` -->
`objc_clear_deallocating` (原weak修饰符修饰的变量地址置为nil)



总结与疑问


StripedMap
void *p为 key，PaddedT为 value 的表
SideTable
spinlock_t slock
保证原子操作的自旋锁
RefcountMap refcnts;
引用计数的哈希表
weak_table_t weak_table
全局引用hash表
size_t num_entries;
哈希数组中元素个数
uintptr_t mask
哈希长度计算的掩码
uintptr_t max_hash_displacement
最大哈希偏移量
weak_entry_t *weak_entries
hash数组，存储弱引用对象相关信息
weak_entry_t
DisguisedPtr<objc_object> referent
弱引用的对象
弱引用数量=<4
weak_referrer_t inline_referrers[WEAK_INLINE_COUNT];
存储指向该对象的弱引用的数组
WEAK_INLINE_COUNT 了数组长度
弱引用数量>4弱引用数量>4
weak_referrer_t *referrers;
存储指向该对象的弱引用
uintptr_t out_of_line_ness : 2
uintptr_t num_refs : PTR_MINUS_2
引用计数
uintptr_t mask
参与引用计算的掩码
uintptr_t max_hash_displacement
最大偏移量
内存设计优化
<=WEAK_INLINE_COUNT 弱引用会被保存到inline数组里。weak_entry_t初始化的时候一并分配好，而不是需要的时候再去开辟 空间换时间
联合体union中的两个struct是共享同一块内存。节约内存的同时，也提升了运行效率
weak_entry_t2
weak_entry_t3
SideTable2
SideTable3