【weak】

weak 实现原理：
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 指针的地址）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。

Q: 一个weak修饰的变量时怎么被加入到弱引用表中的？来看一个代码块：

```
{
    id __weak obj1 = obj;
}
// 编译后
{
    id obj1;
    objc_initWeak(&obj1, obj);
}
// 在这个过程中，发生了什么？
```

我们先来看看`objc_initWeak`调用了什么方法。

我们从源码可以得知其中的调用顺序。看名称可以得知，具体的注册弱引用的步骤是在`weak_register_no_lock`内部的。现在我们具体分析一下每一步的函数都做了什么。

```
// 这个方法传递了2个参数值，一个是要指向弱引用对象的对象，一个是需要被弱引用的对象。
objc_initWeak(id *location/* 弱引用指针*/, id newObj/*被弱引用的对象*/)
{
    // 这个方法内部就做了一个非空判断，然后直接走到storeWeak方法中
    if (!newObj) {
        *location = nil;
        return nil;
    }
    // 这里使用了C++的模板，DontHaveOld(无老对象)，DoHaveNew(有新对象)，DoCrashIfDeallocating(销毁过程中不Crash)
    return storeWeak<DontHaveOld, DoHaveNew, DoCrashIfDeallocating>
        (location, (objc_object*)newObj);
}
```

接下来我们看看`storeWeak`方法的实现，这里因为我们在上面的objc_initWeak传入的参数是无老对象，有新对象，所以我们按照上面传参的逻辑分析下面的代码。

```
enum CrashIfDeallocating {
    DontCrashIfDeallocating = false, DoCrashIfDeallocating = true
};
```

```
template <HaveOld haveOld, HaveNew haveNew,
          CrashIfDeallocating crashIfDeallocating>
static id
storeWeak(id *location, objc_object *newObj)
{
    // 这里做了一些值判断
    assert(haveOld  ||  haveNew);
    if (!haveNew) assert(newObj == nil);
    // 声明局部变量
    Class previouslyInitializedClass = nil;
    id oldObj;
    SideTable *oldTable;
    SideTable *newTable;
    // Acquire locks for old and new values.
    // Order by lock address to prevent lock ordering problems.
    // Retry if the old value changes underneath us.
 retry:
    if (haveOld) {  // 我们没有old所以这里直接过
        oldObj = *location;
        oldTable = &SideTables()[oldObj];
    } else {
        oldTable = nil;
    }
    if (haveNew) {  // 有新对象，走这里
        // 从SideTables当中，拿到newObj所在的表，赋值给newTable
        newTable = &SideTables()[newObj];
    } else {
        newTable = nil;
    }

    SideTable::lockTwo<haveOld, haveNew>(oldTable, newTable);

    if (haveOld  &&  *location != oldObj) { // 我们没有old所以这里直接过
        SideTable::unlockTwo<haveOld, haveNew>(oldTable, newTable);
        goto retry;
    }

    // Prevent a deadlock between the weak reference machinery
    // and the +initialize machinery by ensuring that no
```

```
    // weakly-referenced object has an un-+initialized isa.
    if (haveNew  &&  newObj) {  // 有新对象，且传递进来的newObj是有值的
        Class cls = newObj->getIsa();   // 根据newObj的isa指针 找到类对象
        if (cls != previouslyInitializedClass  &&
            !((objc_class *)cls)->isInitialized())  // 判断类是否已经初
始化
        {   // 已经初始化过了 这里面的内容不影响注册weak
            SideTable::unlockTwo<haveOld, haveNew>(oldTable,
newTable);
            _class_initialize(_class_getNonMetaClass(cls,
(id)newObj));
            previouslyInitializedClass = cls;
            goto retry;
        }
    }

    // Clean up old value, if any.
    if (haveOld) { // 我们没有old所以这里直接过
        weak_unregister_no_lock(&oldTable->weak_table, oldObj,
location);
    }

    // Assign new value, if any.
    if (haveNew) {
        /* 这里就是我们在上图中看到的weak_register_no_lockg方法，这个函数接收
4个参数
            1. weak_table_t *weak_table,     弱引用表
            2. id referent_id,               需要被引用的对象
            3. id *referrer_id,              弱引用指针
            4. bool crashIfDeallocating,     对象在废弃的过程中，Crash的一
个标志位
         */
        newObj = (objc_object *)
            weak_register_no_lock(&newTable->weak_table, (id)newObj,
location,
                                  crashIfDeallocating);
        // weak_register_no_lock returns nil if weak store should be
rejected
        // Set is-weakly-referenced bit in refcount table.
```

```
        if (newObj  &&  !newObj->isTaggedPointer()) {
            // 新对象有值 且不是小对象的指针类型 就设置这个对象有弱引用的标志位
            newObj->setWeaklyReferenced_nolock();
        }
        // Do not set *location anywhere else. That would introduce a
race.
        *location = (id)newObj;
    }
    else {
        // No new value. The storage is not changed.
    }
    SideTable::unlockTwo<haveOld, haveNew>(oldTable, newTable);
    return (id)newObj;
```

到这里，已经可以大致了解弱引用大致的注册流程了，我再来看看`weak_register_no_lock`中所做的操作

```
...
// 我们重点看这里
if ((entry = weak_entry_for_referent(weak_table, referent))) {
    // 将新的弱引用指针添加到弱引用数组当中
    append_referrer(entry, referrer);
}
else {  // 如果没有获取到弱引用数组，则重新创建，然后添加
    weak_entry_t new_entry(referent, referrer);
    weak_grow_maybe(weak_table);
    weak_entry_insert(weak_table, &new_entry);
}
...
```

现在再看看系统是如何查找到弱引用表中的弱引用数组的

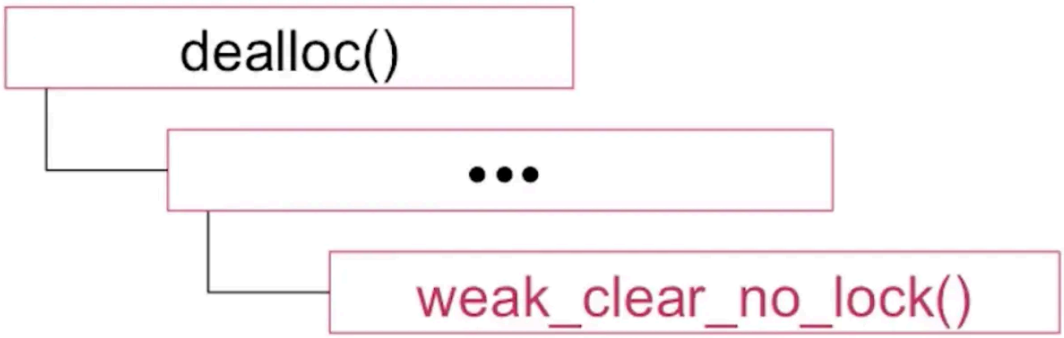
```
static weak_entry_t *
weak_entry_for_referent(weak_table_t *weak_table, objc_object
*referent)
{
    assert(referent);
    // 拿到弱引用结构体数组
    weak_entry_t *weak_entries = weak_table->weak_entries;
    if (!weak_entries) return nil;
    // 通过Hash算法根据原对象地址找到对应的索引位置
```

```
    size_t begin = hash_pointer(referent) & weak_table->mask;
    size_t index = begin;
    size_t hash_displacement = 0;
    // 这个while用来解决Hash冲突，如果找到的位置不是当前要查找的对象，会根据冲突算法来移动索引位置，直到找到要查找的对象
    while (weak_table->weak_entries[index].referent != referent) {
        index = (index+1) & weak_table->mask;
        if (index == begin) bad_weak_table(weak_table->weak_entries);
        hash_displacement++;
        if (hash_displacement > weak_table->max_hash_displacement) {
            return nil;
        }
    }
    // 找到了就返回弱引用表
    return &weak_table->weak_entries[index];
}
```

总结一下这个流程:::::

被weak修饰的变量，系统会在编译时调用objc_initWeak方法，然后调用storeWeak，再调用weak_register_no_lock，在这个方法中，会根据对象的地址通过Hash算法计算出位置，然后插入到弱引用表中。

Q: 当一个对象释放，weak变量是怎么处理的?



我们之前已经知道了大致的调用流程，现在我们看看weak_clear_no_lock方法是怎么实现的

```
// 这个对象有2个参数 一个是弱引用表 一个是需要被清除引用的对象
```

```
weak_clear_no_lock(weak_table_t *weak_table, id referent_id)
{
    // 定义一个局部变量 用referent_id赋值
    objc_object *referent = (objc_object *)referent_id;
    // 找到对应的弱引用数组
    weak_entry_t *entry = weak_entry_for_referent(weak_table,
referent);
    if (entry == nil) { // 如果没有 则当前对象没有弱引用 不用处理 直接返回
        /// XXX shouldn't happen, but does with mismatched CF/objc
        //printf("XXX no entry for clear deallocating %p\n",
referent);
        return;
    }

    // zero out references
    weak_referrer_t *referrers;
    size_t count;

    if (entry->out_of_line()) { // 如果弱引用列表元素个数大于4走这里
        referrers = entry->referrers;
        count = TABLE_SIZE(entry);
    }
    else { // 如果弱引用列表元素个数小于4走这里
        referrers = entry->inline_referrers;
        count = WEAK_INLINE_COUNT;
    }
    // 到这里 referrers 就取到了当前对象对应的弱引用列表
    for (size_t i = 0; i < count; ++i) {
        objc_object **referrer = referrers[i];
        if (referrer) {// 如果弱引用指针存在
            if (*referrer == referent) { // 这个弱引用代表的地址就是当前对
象的地址
                *referrer = nil;    // 弱引用指针置为nil
            }
            else if (*referrer) {
                _objc_inform("__weak variable at %p holds %p instead
of %p. "
                             "This is probably incorrect use of "
                             "objc_storeWeak() and objc_loadWeak(). "
```

```
                                "Break on objc_weak_error to debug.\n",
                                referrer, (void*)*referrer,
(void*)referent);
                objc_weak_error();
            }
        }
    }

    weak_entry_remove(weak_table, entry);
}
```

这里总结一下

当一个对象被dealloc，在dealloc的内部实现中，会调用weak_clear_no_lock方法。这个方法会在弱引用表中找到要被销毁的对象，然后把当前对象相对应的弱引用都取出来。置为nil。