我们知道，在 Objective-C 中可以通过 Category 给一个现有的类添加属性，但是却不能添加实例变量，这似乎成为了 Objective-C 的一个明显短板。然而值得庆幸的是，我们可以通过 Associated Objects 来弥补这一不足。本文将结合 runtime 源码深入探究 Objective-C 中 Associated Objects 的实现原理。

在阅读本文的过程中，读者需要着重关注以下三个问题：

1. 关联对象被存储在什么地方，是不是存放在被关联对象本身的内存中?
2. 关联对象的五种关联策略有什么区别，有什么坑?
3. 关联对象的生命周期是怎样的，什么时候被释放，什么时候被移除?

这是我写这篇文章的初衷，也是本文的价值所在。

使用场景

按照 Mattt Thompson 大神的文章 Associated Objects 中的说法，Associated Objects 主要有以下三个使用场景：

1. 为现有的类添加私有变量以帮助实现细节；
2. 为现有的类添加公有属性；
3. 为 KVO 创建一个关联的观察者。

从本质上看，第 1 、2 个场景其实是一个意思，唯一的区别就在于新添加的这个属性是公有的还是私有的而已。就目前来说，我在实际工作中使用得最多的是第 2 个场景，而第 3 个场景我还没有使用过。

相关函数

与 Associated Objects 相关的函数主要有三个，我们可以在 runtime 源码的 runtime.h 文件中找到它们的声明：

```
1  void objc_setAssociatedObject(id object, const void *key, id value, objc_AssociationPoli
2  id objc_getAssociatedObject(id object, const void *key);
3  void objc_removeAssociatedObjects(id object);
```

这三个函数的命名对程序员非常友好，可以让我们一眼就看出函数的作用：

1. objc_setAssociatedObject 用于给对象添加关联对象，传入 nil 则可以移除已有的关联对象；
2. objc_getAssociatedObject 用于获取关联对象；
3. objc_removeAssociatedObjects 用于移除一个对象的所有关联对象。

注：objc_removeAssociatedObjects 函数我们一般是用不上的，因为这个函数会移除一个对象的所有关联对象，将该对象恢复成“原始”状态。这样做就很有可能把别人添加的关联对象也一并移除，这并不是我们所希望的。所以一般的做法是通过给 objc_setAssociatedObject 函数传入 nil 来移除某个已有的关联对象。

key 值

关于前两个函数中的 key 值是我们需要重点关注的一个点，这个 key 值必须保证是一个对象级别（为什么是对象级别？看完下面的章节你就会明白了）的唯一常量。一般来说，有以下三种推荐的 key 值：

1. 声明 static char kAssociatedObjectKey; ，使用 &kAssociatedObjectKey 作为 key 值；
2. 声明 static void *kAssociatedObjectKey = &kAssociatedObjectKey; ，使用 kAssociatedObjectKey 作为 key 值；
3. 用 selector ，使用 getter 方法的名称作为 key 值。

我个人最喜欢的（没有之一）是第 3 种方式，因为它省掉了一个变量名，非常优雅地解决了计算科学中的两大世界难题之一（命名）。

关联策略

在给一个对象添加关联对象时有五种关联策略可供选择：

关联策略	等价属性	说明
OBJC_ASSOCIATION_ASSIGN	@property (assign) or @property (unsafe_unretained)	弱引用关联对象
OBJC_ASSOCIATION_RETAIN_NONATOMIC	@property (strong, nonatomic)	强引用关联对象，且为非原子操作
OBJC_ASSOCIATION_COPY_NONATOMIC	@property (copy, nonatomic)	复制关联对象，且为非原子操作
OBJC_ASSOCIATION_RETAIN	@property (strong, atomic)	强引用关联对象，且为原子操作
OBJC_ASSOCIATION_COPY	@property (copy, atomic)	复制关联对象，且为原子操作

其中，第 2 种与第 4 种、第 3 种与第 5 种关联策略的唯一差别就在于操作是否具有原子性。由于操作的原子性不在本文的讨论范围内，所以下面的实验和讨论就以前三种以例进行展开。

实现原理

在探究 Associated Objects 的实现原理前，我们还是先来动手做一个小实验，研究一下关联对象什么时候会被释放。本实验主要涉及 ViewController 类和它的分类 ViewController+AssociatedObjects 。注：本实验的完整代码可以在这里 AssociatedObjects 找到，其中关键代码如下：

```
@interface ViewController (AssociatedObjects)
@property (assign, nonatomic) NSString *associatedObject_assign;
@property (strong, nonatomic) NSString *associatedObject_retain;
@property (copy,   nonatomic) NSString *associatedObject_copy;
@end
@implementation ViewController (AssociatedObjects)
- (NSString *)associatedObject_assign {
    return objc_getAssociatedObject(self, _cmd);
}
- (void)setAssociatedObject_assign:(NSString *)associatedObject_assign {
    objc_setAssociatedObject(self, @selector(associatedObject_assign), associatedObject
}
```

```
- (NSString *)associatedObject_retain {
    return objc_getAssociatedObject(self, _cmd);
}
- (void)setAssociatedObject_retain:(NSString *)associatedObject_retain {
    objc_setAssociatedObject(self, @selector(associatedObject_retain), associatedObject
}
- (NSString *)associatedObject_copy {
    return objc_getAssociatedObject(self, _cmd);
}
- (void)setAssociatedObject_copy:(NSString *)associatedObject_copy {
    objc_setAssociatedObject(self, @selector(associatedObject_copy), associatedObject_c
}
@end
```

在 ViewController+AssociatedObjects.h 中声明了三个属性，限定符分别为 assign, nonatomic 、strong, nonatomic 和 copy, nonatomic ， 而在 ViewController+AssociatedObjects.m 中相应的分别用 OBJC_ASSOCIATION_ASSIGN 、OBJC_ASSOCIATION_RETAIN_NONATOMIC 、OBJC_ASSOCIATION_COPY_NONATOMIC 三种关联策略为这三个属性添加“实例变量”。

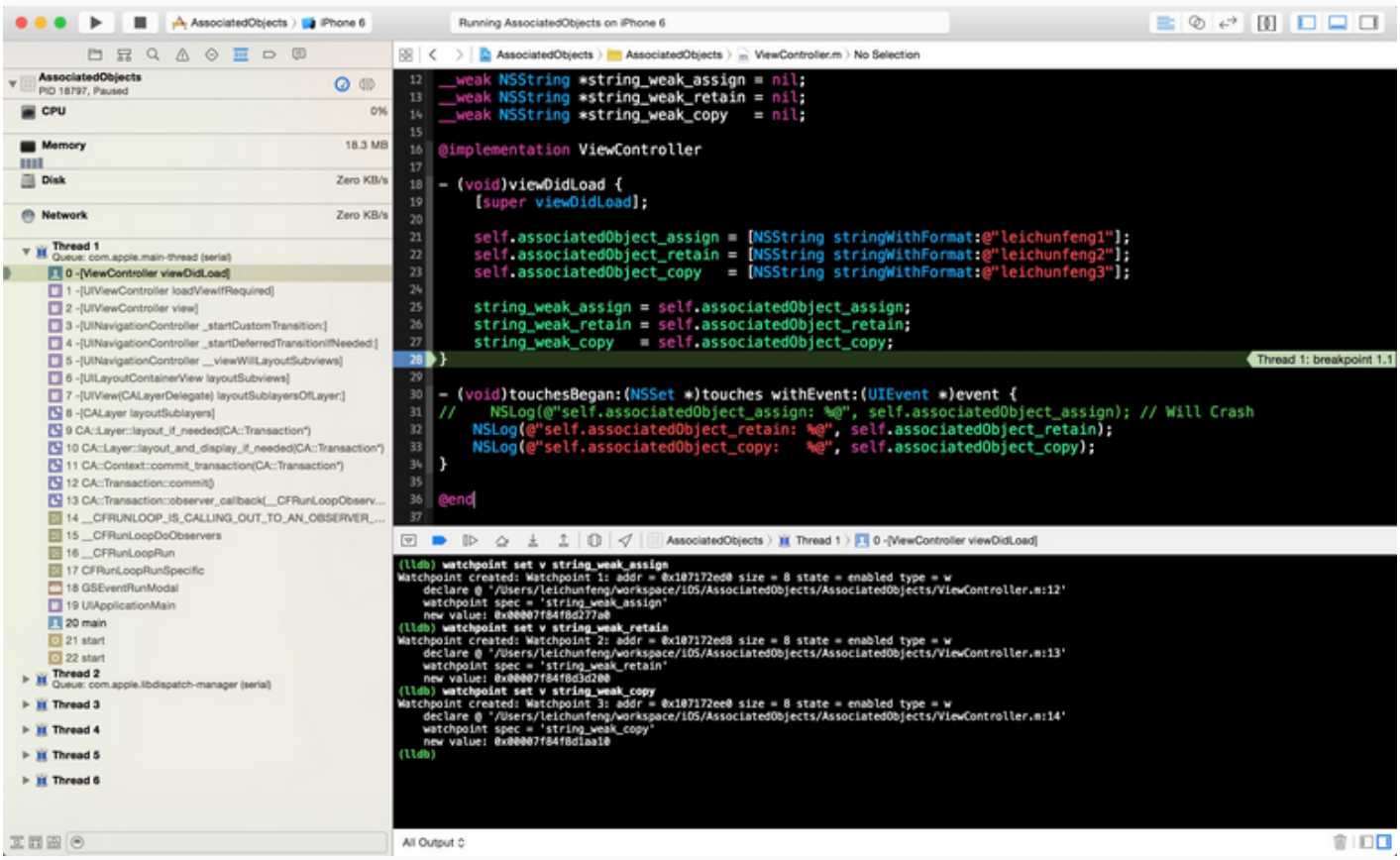
```
__weak NSString *string_weak_assign = nil;
__weak NSString *string_weak_retain = nil;
__weak NSString *string_weak_copy   = nil;
@implementation ViewController
- (void)viewDidLoad {
    [super viewDidLoad];
    self.associatedObject_assign = [NSString stringWithFormat:@"leichunfeng1"];
    self.associatedObject_retain = [NSString stringWithFormat:@"leichunfeng2"];
    self.associatedObject_copy   = [NSString stringWithFormat:@"leichunfeng3"];
    string_weak_assign = self.associatedObject_assign;
    string_weak_retain = self.associatedObject_retain;
    string_weak_copy   = self.associatedObject_copy;
}
- (void)touchesBegan:(NSSet *)touches withEvent:(UIEvent *)event {
//    NSLog(@"self.associatedObject_assign: %@", self.associatedObject_assign); // Will
    NSLog(@"self.associatedObject_retain: %@", self.associatedObject_retain);
    NSLog(@"self.associatedObject_copy:   %@", self.associatedObject_copy);
}
@end
```

在 ViewController 的 viewDidLoad 方法中，我们对三个属性进行了赋值，并声明了三个全局的 __weak 变量来观察相应对象的释放时机。此外，我们重写了 touchesBegan:withEvent: 方法，在方法中分别打印了这三个属性的当前值。

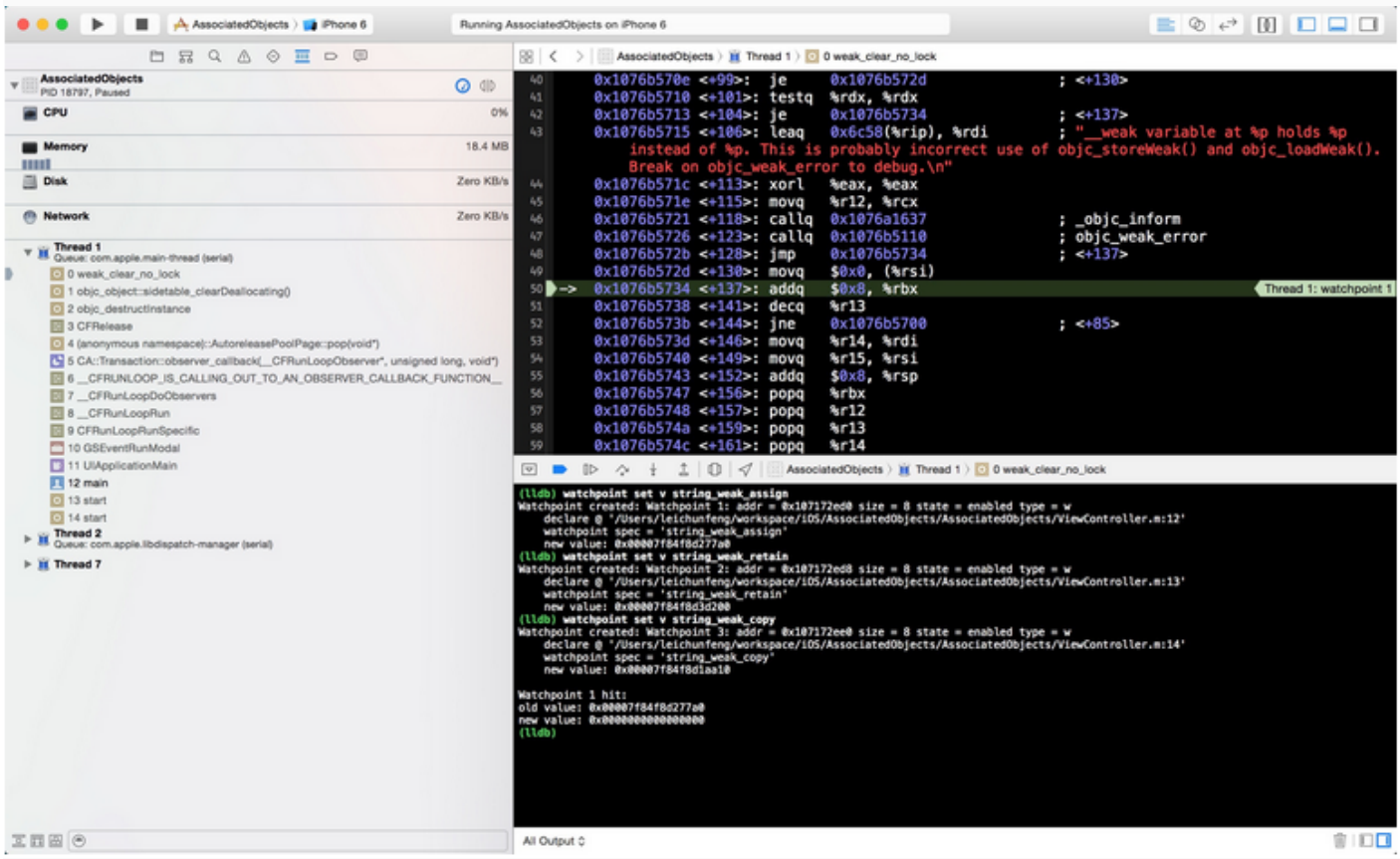
在继续阅读下面章节前，建议读者先自行思考一下 self.associatedObject_assign 、self.associatedObject_retain 和 self.associatedObject_copy 指向的对象分别会在什么时候被释放，以加深理解。

实验

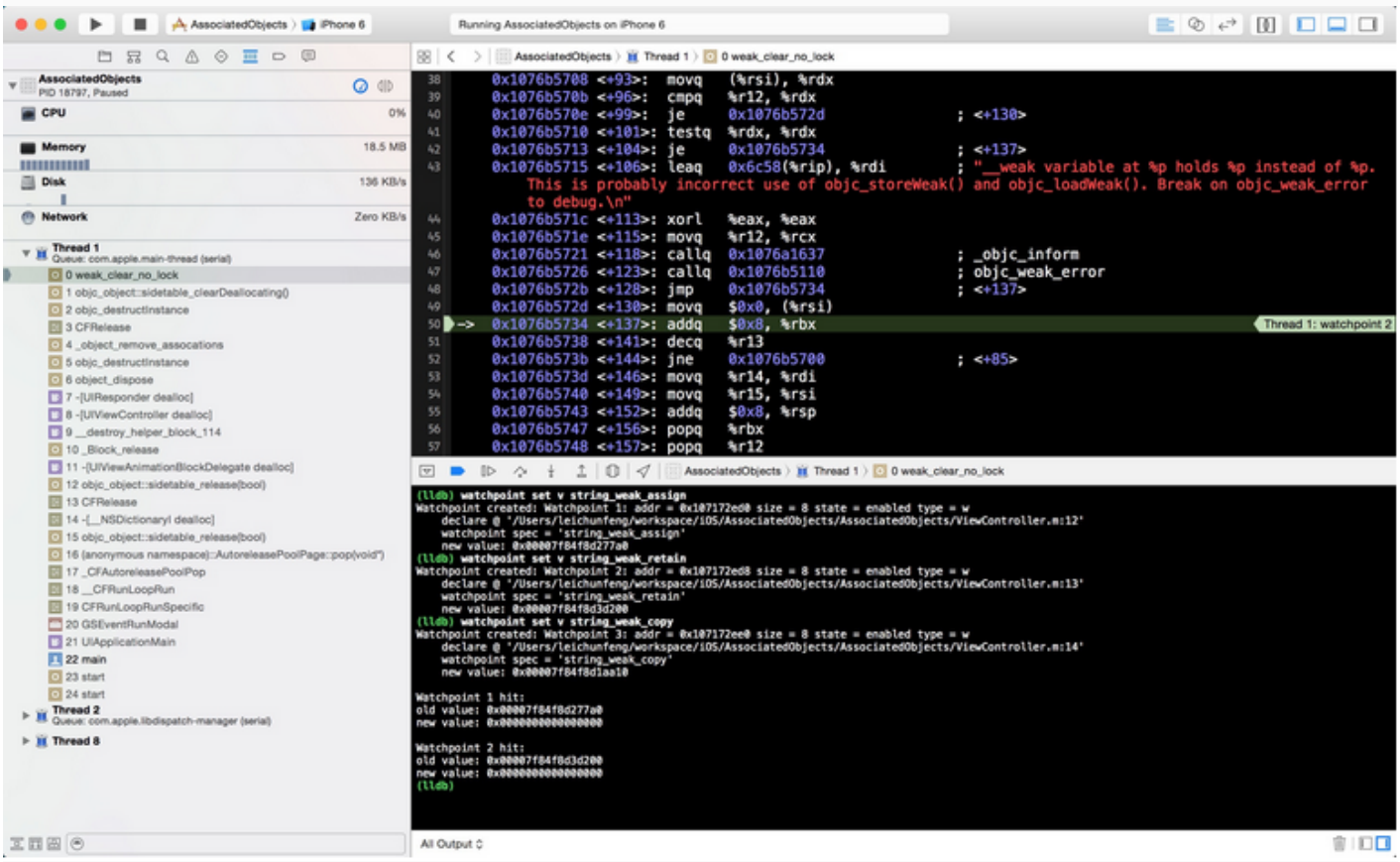
我们先在 viewDidLoad 方法的第 28 行打上断点，然后运行程序，点击导航栏右上角的按钮 Push 到 ViewController 界面，程序将停在断点处。接着，我们使用 lldb 的 watchpoint 命令来设置观察点，观察全局变量 string_weak_assign 、string_weak_retain 和 string_weak_copy 的值的变化。正确设置好观察点后，将会在 console 中看到如下的类似输出：



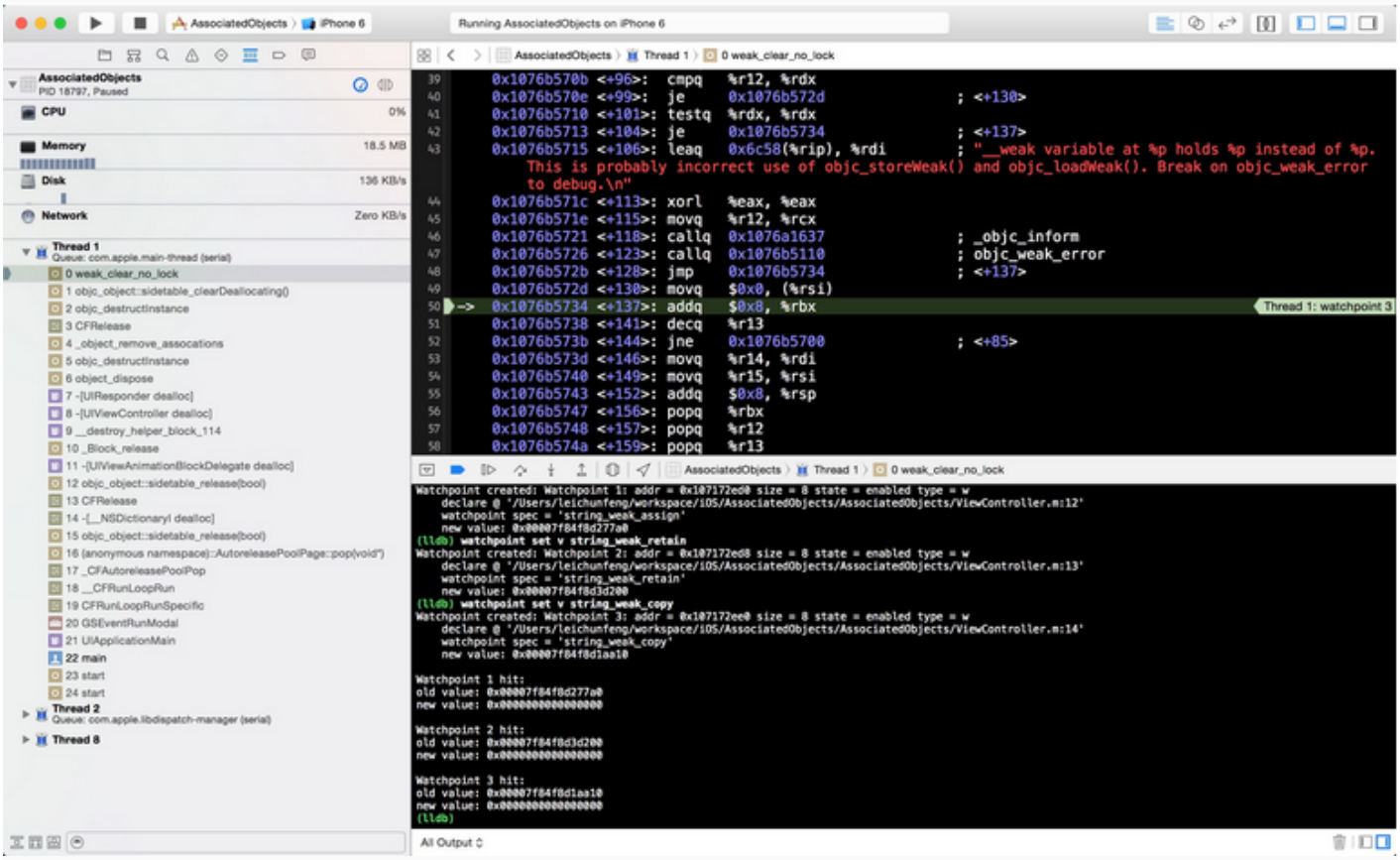
点击继续运行按钮，有一个观察点将被命中。我们先查看 console 中的输出，通过将这一步打印的 old value 和上一步的 new value 进行对比，我们可以知道本次命中的观察点是 string_weak_assign ，string_weak_assign 的值变成了 0x0000000000000000 ，也就是 nil 。换句话说 self.associatedObject_assign 指向的对象已经被释放了，而通过查看左侧调用栈我们可以知道，这个对象是由于其所在的 autoreleasepool 被 drain 而被释放的，这与我前面的文章《Objective-C Autorelease Pool 的实现原理》中的表述是一致的。提示，待会你也可以放开 touchesBegan:withEvent: 中第 31 行的注释，在 ViewController 出现后，点击一下它的 view ，进一步验证一下这个结论。



接下来，我们点击 ViewController 导航栏左上角的按钮，返回前一个界面，此时，又将有一个观察点被命中。同理，我们可以知道这个观察点是 string_weak_retain 。我们查看左侧的调用栈，将会发现一个非常敏感的函数调用 _object_remove_assocations ，调用这个函数后 ViewController 的所有关联对象被全部移除。最终，self.associatedObject_retain 指向的对象被释放。



点击继续运行按钮，最后一个观察点 string_weak_copy 被命中。同理，self.associatedObject_copy 指向的对象也由于关联对象的移除被最终释放。



结论

由这个实验，我们可以得出以下结论：

1. 关联对象的释放时机与被移除的时机并不总是一致的，比如上面的 self.associatedObject_assign 所指向的对象在 ViewController 出现后就被释放了，但是 self.associatedObject_assign 仍然有值，还是保存的原对象的地址。如果之后再使用 self.associatedObject_assign 就会造成 Crash ，所以我们在使用弱引用的关联对象时要非常小心；
2. 一个对象的所有关联对象是在这个对象被释放时调用的 _object_remove_assocations 函数中被移除的。

接下来，我们就一起看看 runtime 中的源码，来验证下我们的实验结论。

objc_setAssociatedObject

我们可以在 objc-references.mm 文件中找到 objc_setAssociatedObject 函数最终调用的函数：

```
void _object_set_associative_reference(id object, void *key, id value, uintptr_t policy
    // retain the new value (if any) outside the lock.
    ObjcAssociation old_association(0, nil);
    id new_value = value ? acquireValue(value, policy) : nil;
    {
        AssociationsManager manager;
        AssociationsHashMap &associations(manager.associations());
```

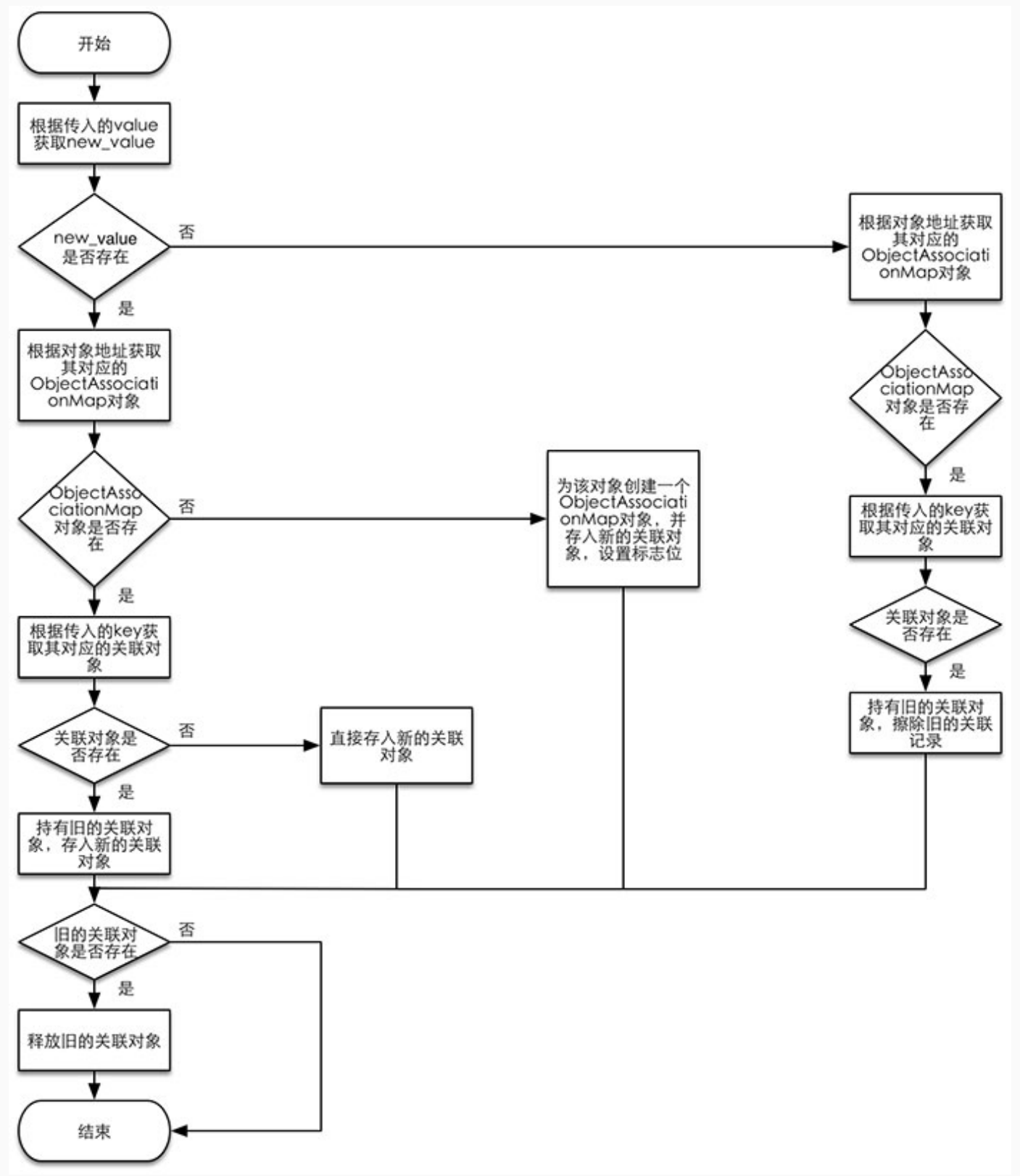
```
        disguised_ptr_t disguised_object = DISGUISE(object);
        if (new_value) {
            // break any existing association.
            AssociationsHashMap::iterator i = associations.find(disguised_object);
            if (i != associations.end()) {
                // secondary table exists
                ObjectAssociationMap *refs = i->second;
                ObjectAssociationMap::iterator j = refs->find(key);
                if (j != refs->end()) {
                    old_association = j->second;
                    j->second = ObjcAssociation(policy, new_value);
                } else {
                    (*refs)[key] = ObjcAssociation(policy, new_value);
                }
            } else {
                // create the new association (first time).
                ObjectAssociationMap *refs = new ObjectAssociationMap;
                associations[disguised_object] = refs;
                (*refs)[key] = ObjcAssociation(policy, new_value);
                object->setHasAssociatedObjects();
            }
        } else {
            // setting the association to nil breaks the association.
            AssociationsHashMap::iterator i = associations.find(disguised_object);
            if (i !=  associations.end()) {
                ObjectAssociationMap *refs = i->second;
                ObjectAssociationMap::iterator j = refs->find(key);
                if (j != refs->end()) {
                    old_association = j->second;
                    refs->erase(j);
                }
            }
        }
    }
    // release the old value (outside of the lock).
    if (old_association.hasValue()) ReleaseValue()(old_association);
}
```

在看这段代码前，我们需要先了解一下几个数据结构以及它们之间的关系：

1. AssociationsManager 是顶级的对象，维护了一个从 spinlock_t 锁到 AssociationsHashMap 哈希表的单例键值对映射；
2. AssociationsHashMap 是一个无序的哈希表，维护了从对象地址到 ObjectAssociationMap 的映射；
3. ObjectAssociationMap 是一个 C++ 中的 map ，维护了从 key 到 ObjcAssociation 的映射，即关联记录；
4. ObjcAssociation 是一个 C++ 的类，表示一个具体的关联结构，主要包括两个实例变量，_policy 表示关联策略，_value 表示关联对象。

每一个对象地址对应一个 ObjectAssociationMap 对象，而一个 ObjectAssociationMap 对象保存着这个对象的若干个关联记录。

弄清楚这些数据结构之间的关系后，再回过头来看上面的代码就不难了。我们发现，在苹果的底层代码中一般都会充斥着各种 if else ，可见写好 if else 后我们就距离成为高手不远了。开个玩笑，我们来看下面的流程图，一图胜千言：



objc_getAssociatedObject

同样的，我们也可以在 objc-references.mm 文件中找到 objc_getAssociatedObject 函数最终调用的函数：

```
id _object_get_associative_reference(id object, void *key) {
    id value = nil;
    uintptr_t policy = OBJC_ASSOCIATION_ASSIGN;
    {
        AssociationsManager manager;
        AssociationsHashMap &associations(manager.associations());
        disguised_ptr_t disguised_object = DISGUISE(object);
```

```
        AssociationsHashMap::iterator i = associations.find(disguised_object);
        if (i != associations.end()) {
            ObjectAssociationMap *refs = i->second;
            ObjectAssociationMap::iterator j = refs->find(key);
            if (j != refs->end()) {
                ObjcAssociation &entry = j->second;
                value = entry.value();
                policy = entry.policy();
                if (policy & OBJC_ASSOCIATION_GETTER_RETAIN) ((id(*)(id, SEL))objc_msgS
            }
        }
    }
    if (value && (policy & OBJC_ASSOCIATION_GETTER_AUTORELEASE)) {
        ((id(*)(id, SEL))objc_msgSend)(value, SEL_autorelease);
    }
    return value;
}
```

看懂了 objc_setAssociatedObject 函数后，objc_getAssociatedObject 函数对我们来说就是小菜一碟了。这个函数先根据对象地址在 AssociationsHashMap 中查找其对应的 ObjectAssociationMap 对象，如果能找到则进一步根据 key 在 ObjectAssociationMap 对象中查找这个 key 所对应的关联结构 ObjcAssociation ，如果能找到则返回 ObjcAssociation 对象的 value 值，否则返回 nil 。

objc_removeAssociatedObjects

同理，我们也可以在 objc-references.mm 文件中找到 objc_removeAssociatedObjects 函数最终调用的函数：

```
void _object_remove_assocations(id object) {
    vector< ObjcAssociation,ObjcAllocator > elements;
    {
        AssociationsManager manager;
        AssociationsHashMap &associations(manager.associations());
        if (associations.size() == 0) return;
        disguised_ptr_t disguised_object = DISGUISE(object);
        AssociationsHashMap::iterator i = associations.find(disguised_object);
        if (i != associations.end()) {
            // copy all of the associations that need to be removed.
            ObjectAssociationMap *refs = i->second;
            for (ObjectAssociationMap::iterator j = refs->begin(), end = refs->end(); j
                elements.push_back(j->second);
            }
            // remove the secondary table.
            delete refs;
            associations.erase(i);
        }
    }
    // the calls to releaseValue() happen outside of the lock.
    for_each(elements.begin(), elements.end(), ReleaseValue());
}
```

这个函数负责移除一个对象的所有关联对象，具体实现也是先根据对象的地址获取其对应的 ObjectAssociationMap 对象，然后将所有的关联结构保存到一个 vector 中，最终释放 vector 中保存的所有关联对象。根据前面的实验观察到的情况，在一个对象被释放时，也正是调用的这个函数来移除其所有的关联对象。

给类对象添加关联对象

看完源代码后，我们知道对象地址与 AssociationsHashMap 哈希表是一一对应的。那么我们可能就会思考这样一个

问题，是否可以给类对象添加关联对象呢？答案是肯定的。我们完全可以用同样的方式给类对象添加关联对象，只不过我们一般情况下不会这样做，因为更多时候我们可以通过 static 变量来实现类级别的变量。我在分类 ViewController+AssociatedObjects 中给 ViewController 类对象添加了一个关联对象 associatedObject ，读者可以亲自在 viewDidLoad 方法中调用一下以下两个方法验证一下：

```
+ (NSString *)associatedObject;
+ (void)setAssociatedObject:(NSString *)associatedObject;
```

总结

读到这里，相信你对开篇的那三个问题已经有了一定的认识，下面我们再梳理一下：

1. 关联对象与被关联对象本身的存储并没有直接的关系，它是存储在单独的哈希表中的；
2. 关联对象的五种关联策略与属性的限定符非常类似，在绝大多数情况下，我们都会使用 OBJC_ASSOCIATION_RETAIN_NONATOMIC 的关联策略，这可以保证我们持有关联对象；
3. 关联对象的释放时机与移除时机并不总是一致，比如实验中用关联策略 OBJC_ASSOCIATION_ASSIGN 进行关联的对象，很早就已经被释放了，但是并没有被移除，而再使用这个关联对象时就会造成 Crash 。

在弄懂 Associated Objects 的实现原理后，可以帮助我们更好地使用它，在出现问题时也能尽快地定位问题，最后希望本文能够对你有所帮助。