iOS (http://lib.csdn.net/base/ios)

iOS (http://lib.csdn.net/base/ios) - Objective-C (http://lib.csdn.net/ios/node/671) - category (http://lib.csdn.net/ios/knowledge/1468)

669 2

Objective-C的Category与关联对象实现原理

作者：zyx196 (http://my.csdn.net/zyx196)

1、什么是Category

category是Objective-C 2.0之后添加的语言特性，category的主要作用是为已经存在的类添加方法。除此之外，apple还推荐了category的另外两个使用场景1 (https://developer.apple.com/library/ios/documentation/General/Conceptual/DevPedia-CocoaCore/Category.html)

> 可以把类的实现分开在几个不同的文件里面。这样做有几个显而易见的好处，a)可以减少单个文件的体积 b)可以把不同的功能组织到不同的category里 c)可以由多个开发者共同完成一个类 d)可以按需加载想要的category 等等。
> 声明私有方法

不过除了apple推荐的使用场景，广大开发者脑洞大开，还衍生出了category的其他几个使用场景：

> 模拟多继承
> 把framework的私有方法公开

extension看起来很像一个匿名的category，但是extension和有名字的category几乎完全是两个东西。 extension在编译期决议，它就是类的一部分，在编译期和头文件里的@interface以及实现文件里的@implement一起形成一个完整的类，它伴随类的产生而产生，亦随之一起消亡。extension一般用来隐藏类的私有信息，你必须有一个类的源码才能为一个类添加extension，所以你无法为系统的类比如NSString添加extension。

但是category则完全不一样，它是在运行期决议的。
就category和extension的区别来看，我们可以推导出一个明显的事实，extension可以添加实例变量，而category是无法添加实例变量的（因为在运行期，对象的内存布局已经确定，如果添加实例变量就会破坏类的内部布局，这对编译型语言来说是灾难性的）。

需要注意的有两点:

1)、category的方法没有“完全替换掉”原来类已经有的方法，也就是说如果category和原来类都有methodA，那么category附加完成之后，类的方法列表里会有两个methodA

2)、category的方法被放到了新方法列表的前面，而原来类的方法被放到了新方法列表的后面，这也就是我们平常所说的category的方法会“覆盖”掉原来类的同名方法，这是因为运行时在查找方法的时候是顺着方法列表的顺序查找的，它只要一找到对应名字的方法，就会罢休^_^，殊不知后面可能还有一样名字的方法。

2、运行时如何load的类别与类:

APP启动主要流程: 点击icon -> 加载动态链接库等 -> 映像文件加载imageLoader -> runtime -> load -> main -> delegate.

runtime 的初始化函数在 objc-os.mm 中的 _objc_init 中, 这个方法在old-ABI中是由 dylb 在初始化动态链接库的时候调用的，现在是由libSystem在动态链接库初始化之前调用的:

```
/********************************************************************
* _objc_init
* Bootstrap initialization. Registers our image notifier with dyld.
* Old ABI: called by dyld as a library initializer
* New ABI: called by libSystem BEFORE library initialization time
********************************************************************/

void _objc_init(void)
{
    ......
    // Register for unmap first, in case some +load unmaps something
    _dyld_register_func_for_remove_image(&unmap_image);
    dyld_register_image_state_change_handler(dyld_image_state_bound,
                                             1/*batch*/, &map_2_images);
    dyld_register_image_state_change_handler(dyld_image_state_dependents_initialized, 0/*
}
```

在image（映像文件）加载完成后，会回调运行时的load_images方法：

```
const char *load_images(enum dyld_image_states state, uint32_t infoCount, const stru
{
    bool found;
    // Return without taking locks if there are no +load methods here.
    found = false;
    ......
    // Discover load methods
    {
        rwlock_writer_t lock2(runtimeLock);
        //先load images
        found = load_images_nolock(state, infoCount, infoList);
    }

    // Call +load methods (without runtimeLock - re-entrant)
    if (found) {
            //然后调用类的+load方法
        call_load_methods();
    }
    return nil;
}
```

load_images 在这个方法里先是调用 load_images_nolock 方法, 在这个方法里会调用 prepare_load_methods 方法去准备好要被调用的 +load 方法，我们先来看下 prepare_load_methods 方法的实现：

```
void prepare_load_methods(const headerType *mhdr)
{
    size_t count, i;

    runtimeLock.assertWriting();

    classref_t *classlist =
        _getObjc2NonlazyClassList(mhdr, &count);
    for (i = 0; i < count; i++) {
        schedule_class_load(remapClass(classlist[i]));
    }

    category_t **categorylist = _getObjc2NonlazyCategoryList(mhdr, &count);
    for (i = 0; i < count; i++) {
        category_t *cat = categorylist[i];
        Class cls = remapClass(cat->cls);
        if (!cls) continue;  // category for ignored weak-linked class
        realizeClass(cls);
        assert(cls->ISA()->isRealized());
        add_category_to_loadable_list(cat);
    }
}
```

在这里，先是 schedule_class_load(Class cls) 方法去准备好所有满足 +load 方法调用条件的类，这个方法会对入参的父类进行递归调用，以确保父类优先的顺序：

```
static void schedule_class_load(Class cls)
{
    if (!cls) return;
    assert(cls->isRealized());  // _read_images should realize

    if (cls->data()->flags & RW_LOADED) return;

    // Ensure superclass-first ordering
    schedule_class_load(cls->superclass);

    add_class_to_loadable_list(cls);
    cls->setInfo(RW_LOADED);
}
```

当 prepare_load_methods 函数执行完之后，所有满足 +load 方法调用条件的类和分类就被分别保存在全局变量 loadable_classes 和 loadable_categories 中了。

准备好类和分类之后，接下来就是对他们的 +load 方法进行调用了，找到 call_load_methods 方法:

```
void call_load_methods(void)
{
    static bool loading = NO;
    bool more_categories;

    loadMethodLock.assertLocked();

    // Re-entrant calls do nothing; the outermost call will finish the job.
    if (loading) return;
    loading = YES;

    void *pool = objc_autoreleasePoolPush();

    do {
        // 1. Repeatedly call class +loads until there aren't any more
        while (loadable_classes_used > 0) {
            call_class_loads();
        }

        // 2. Call category +loads ONCE
        more_categories = call_category_loads();

        // 3. Run more +loads if there are classes OR more untried categories
    } while (loadable_classes_used > 0  ||  more_categories);

    objc_autoreleasePoolPop(pool);

    loading = NO;
}
```

在这个方法里，会以类优先于分类的顺序调用 +load 方法，这里有两个关键的函数 call_class_loads() 和 call_category_loads ,这两个函数会遍历上一步中准备好的 loadable_classes 和 loadable_categories 的 +load 方法，需要注意的是他们都是以函数内存地址的方

式 ((*load_method)(cls, SEL_load)) 对 +load 方法进行调用的，而不是使用发送消息 objc_msgSend 的方式.

这样，类和类别都实现加载，且load方法只要实现（不管分别在类、类别，或同时在类、类别里）都会被执行，而且因为直接调用的函数地址，因此如果子类未实现load方法，是不会调用父类的方法的。

那如果多个类别和类本身实现了load方法，执行顺序是：
1、类本身；
2、类别，按编译顺序，越前面的越先执行，查看编译顺序可以xcode - >project ->buld phases -> compile sources

3、扩展initialize

+initialize 方法是在类或类的子类收到第一条消息之前被调用的，这里所指的消息包括实例方法和类方法的调用。也就是说 +initialize 方法是以懒加载的方式被调用的，如果一直没有给一个类或他的子类发送消息，那么这个类的 +initialize 方法是永远不会调用的。

当我们向某个类发送消息时， runtime 会调用 IMP lookUpImpOrForward(...) 这个函数在类中查找相应方法的实现或进行消息转发，打开 objc-runtime-new.h 找到这个函数：

```
IMP lookUpImpOrForward(Class cls, SEL sel, id inst,
                       bool initialize, bool cache, bool resolver)
{
    Class curClass;
    IMP imp = nil;
    ...
    if (initialize  &&  !cls->isInitialized()) {
            // 类没有初始化时，对类进行初始化
        _class_initialize (_class_getNonMetaClass(cls, inst));
        // If sel == initialize, _class_initialize will send +initialize and
        // then the messenger will send +initialize again after this
        // procedure finishes. Of course, if this is not being called
        // from the messenger then it won't happen. 2778172
    }

    ...
}
```

从中可以看到当类没有初始化时，会调用 _class_initialize(Class cls) 对类进行初始化：

```
void _class_initialize(Class cls)
{
    assert(!cls->isMetaClass());

    Class supercls;
    BOOL reallyInitialize = NO;

    // Make sure super is done initializing BEFORE beginning to initialize cls.
    // See note about deadlock above.
    supercls = cls->superclass;
    //递归调用，对父类进行_class_initialize调用，确保父类的initialize方法比子类先调用
    if (supercls  &&  !supercls->isInitialized()) {
        _class_initialize(supercls);
    }

    ......

    if (reallyInitialize) {
        // We successfully set the CLS_INITIALIZING bit. Initialize the class.

        // Record that we're initializing this class so we can message it.
        _setThisThreadIsInitializingClass(cls);

        // Send the +initialize message.
        // Note that +initialize is sent to the superclass (again) if
        // this class doesn't implement +initialize. 2157218
        if (PrintInitializing) {
            _objc_inform("INITIALIZE: calling +[%s initialize]",
                         cls->nameForLogging());
        }
        //发送调用类方法initialize的消息
        ((void(*)(Class, SEL))objc_msgSend)(cls, SEL_initialize);

        ......
}
```

在这里，先是对入参的父类进行递归调用，以确保父类优先于子类初始化，还有一个关键的地方： runtime 使用了发送消息 objc_msgSend 的方式对 +initialize 方法进行调用，这样， +initialize 方法的调用就与普通方法的调用是一致的，都是走的发送消息的流程，所以，如果子类没有实现 +initialize 方法，将会沿着继承链去调用父类的 +initialize 方法，同理，分类中的 +initialize 方法会覆盖原本类的方法。

虽然对每个类只会调用一次 _class_initialize(Class cls) 方法，但是由于 +initialize 方法的调用走的是消息发送的流程，当某个类有多个子类时，这个类的 +initialize 方法有可能会被多次调用，这时，可能需要在 +initialize 方法中判断是否是由子类调用的:

```
+ (void)initialize{
    if (self == [ClassName class]) {
        ......
    }
}
```

4、Category原理

_objc_init里面的调用的map_images最终会调用objc-runtime-new.mm里面的_read_images方法，而在_read_images方法的结尾，有以下的代码片段：

```
// Discover categories.
    for (EACH_HEADER) {
        category_t **catlist =
            _getObjc2CategoryList(hi, &count);
        for (i = 0; i < count; i++) {
            category_t *cat = catlist[i];
            class_t *cls = remapClass(cat->cls);

            if (!cls) {
                // Category's target class is missing (probably weak-linked).
                // Disavow any knowledge of this category.
                catlist[i] = NULL;
                if (PrintConnecting) {
                    _objc_inform("CLASS: IGNORING category \?\?\?(%s) %p with "
                                 "missing weak-linked target class",
                                 cat->name, cat);
                }
                continue;
            }

            // Process this category.
            // First, register the category with its target class.
            // Then, rebuild the class's method lists (etc) if
            // the class is realized.
            BOOL classExists = NO;
            if (cat->instanceMethods ||  cat->protocols
                ||  cat->instanceProperties)
            {
                addUnattachedCategoryForClass(cat, cls, hi);
                if (isRealized(cls)) {
                    remethodizeClass(cls);
                    classExists = YES;
                }
                if (PrintConnecting) {
                    _objc_inform("CLASS: found category -%s(%s) %s",
                                 getName(cls), cat->name,
                                 classExists ? "on existing class" : "");
                }
            }

            if (cat->classMethods  ||  cat->protocols
                /* ||  cat->classProperties */)
            {
                addUnattachedCategoryForClass(cat, cls->isa, hi);
                if (isRealized(cls->isa)) {
                    remethodizeClass(cls->isa);
                }
                if (PrintConnecting) {
                    _objc_inform("CLASS: found category +%s(%s)",
                                 getName(cls), cat->name);
                }
            }
        }
    }
```

这段代码主要是：

1)、把category的实例方法、协议以及属性添加到类上
2)、把category的类方法和协议添加到类的metaclass上

category的各种列表是怎么最终添加到类上的，就拿实例方法列表来说吧：
在上述的代码片段里，addUnattachedCategoryForClass只是把类和category做一个关联映射，而remethodizeClass才是真正去处理添加事宜的功臣。

```
static void remethodizeClass(class_t *cls)
{
    category_list *cats;
    BOOL isMeta;

    rwlock_assert_writing(&runtimeLock);

    isMeta = isMetaClass(cls);

    // Re-methodizing: check for more categories
    if ((cats = unattachedCategoriesForClass(cls))) {
        chained_property_list *newproperties;
        const protocol_list_t **newprotos;

        if (PrintConnecting) {
            _objc_inform("CLASS: attaching categories to class '%s' %s",
                         getName(cls), isMeta ? "(meta)" : "");
        }

        // Update methods, properties, protocols

        BOOL vtableAffected = NO;
        attachCategoryMethods(cls, cats, &vtableAffected);

        newproperties = buildPropertyList(NULL, cats, isMeta);
        if (newproperties) {
            newproperties->next = cls->data()->properties;
            cls->data()->properties = newproperties;
        }

        newprotos = buildProtocolList(cats, NULL, cls->data()->protocols);
        if (cls->data()->protocols  &&  cls->data()->protocols != newprotos) {
            _free_internal(cls->data()->protocols);
        }
        cls->data()->protocols = newprotos;

        _free_internal(cats);

        // Update method caches and vtables
        flushCaches(cls);
        if (vtableAffected) flushVtables(cls);
    }
}
```

而对于添加类的实例方法而言，又会去调用attachCategoryMethods这个方法，我们去看下attachCategoryMethods：

```
static void
attachCategoryMethods(class_t *cls, category_list *cats,
                      BOOL *inoutVtablesAffected)
{
    if (!cats) return;
    if (PrintReplacedMethods) printReplacements(cls, cats);

    BOOL isMeta = isMetaClass(cls);
    method_list_t **mlists = (method_list_t **)
        _malloc_internal(cats->count * sizeof(*mlists));

    // Count backwards through cats to get newest categories first
    int mcount = 0;
    int i = cats->count;
    BOOL fromBundle = NO;
    while (i--) {
        method_list_t *mlist = cat_method_list(cats->list[i].cat, isMeta);
        if (mlist) {
            mlists[mcount++] = mlist;
            fromBundle |= cats->list[i].fromBundle;
        }
    }

    attachMethodLists(cls, mlists, mcount, NO, fromBundle, inoutVtablesAffected);

    _free_internal(mlists);

}
```

attachCategoryMethods做的工作相对比较简单，它只是把所有category的实例方法列表拼成了一个大的实例方法列表，然后转交给了attachMethodLists方法（我发誓，这是本节我们看的最后一段代码了^_^），这个方法有点长，我们只看一小段：

```
for (uint32_t m = 0;
             (scanForCustomRR || scanForCustomAWZ)  &&  m < mlist->count;
             m++)
        {
            SEL sel = method_list_nth(mlist, m)->name;
            if (scanForCustomRR  &&  isRRSelector(sel)) {
                cls->setHasCustomRR();
                scanForCustomRR = false;
            } else if (scanForCustomAWZ  &&  isAWZSelector(sel)) {
                cls->setHasCustomAWZ();
                scanForCustomAWZ = false;
            }
        }

        // Fill method list array
        newLists[newCount++] = mlist;
    .
    .
    .

    // Copy old methods to the method list array
    for (i = 0; i < oldCount; i++) {
        newLists[newCount++] = oldLists[i];
    }
```

需要注意的有两点：

1)、category的方法没有“完全替换掉”原来类已经有的方法，也就是说如果category和原来类都有methodA，那么category附加完成之后，类的方法列表里会有两个methodA

2)、category的方法被放到了新方法列表的前面，而原来类的方法被放到了新方法列表的后面，这也就是我们平常所说的category的方法会“覆盖”掉原来类的同名方法，这是因为运行时在查找方法的时候是顺着方法列表的顺序查找的，它只要一找到对应名字的方法，就会罢休^_^，殊不知后面可能还有一样名字的方法。

5、属性追加

我们知道，类别是不能添加成员变量的（property本质也是成员变量 = var + setter、getter），原因是因为一个类的内存大小是固定的，一个雷在load方法执行前就已经加载在内存之中，大小已固定，那如果我们要追加成员变量，我们就可以通过属性追加的方法：

```
- (void)setName:(NSString *)name
{
    objc_setAssociatedObject(self,
                             "name",
                             name,
                             OBJC_ASSOCIATION_COPY);
}

- (NSString*)name
{
    NSString *nameObject = objc_getAssociatedObject(self, "name");
    return nameObject;
}
```

但是关联对象又是存在什么地方呢？ 如何存储？ 对象销毁时候如何处理关联对象呢？
我们去翻一下runtime的源码，在objc-references.mm文件中有个方法_object_set_associative_reference：

```
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);
                _class_setInstancesHaveAssociatedObjects(_object_getClass(object));
            }
        } 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);
}
```

我们可以看到所有的关联对象都由AssociationsManager管理，而AssociationsManager定义如下：

```
class AssociationsManager {
    static OSSpinLock _lock;
    static AssociationsHashMap *_map;               // associative references:  object p
public:
    AssociationsManager()   { OSSpinLockLock(&_lock); }
    ~AssociationsManager()  { OSSpinLockUnlock(&_lock); }

    AssociationsHashMap &associations() {
        if (_map == NULL)
            _map = new AssociationsHashMap();
        return *_map;
    }
};
```

AssociationsManager里面是由一个静态AssociationsHashMap来存储所有的关联对象的。这相当于把所有对象的关联对象都存在一个全局map里面。而map的的key是这个对象的指针地址（任意两个不同对象的指针地址一定是不同的），而这个map的value又是另外一个AssociationsHashMap，里面保存了关联对象的kv对。
而在对象的销毁逻辑里面，见objc-runtime-new.mm:

```
void *objc_destructInstance(id obj)
{
    if (obj) {
        Class isa_gen = _object_getClass(obj);
        class_t *isa = newcls(isa_gen);

        // Read all of the flags at once for performance.
        bool cxx = hasCxxStructors(isa);
        bool assoc = !UseGC && _class_instancesHaveAssociatedObjects(isa_gen);

        // This order is important.
        if (cxx) object_cxxDestruct(obj);
        if (assoc) _object_remove_assocations(obj);

        if (!UseGC) objc_clear_deallocating(obj);
    }

    return obj;
}
```

嗯，runtime的销毁对象函数objc_destructInstance里面会判断这个对象有没有关联对象，如果有，会调用_object_remove_assocations做关联对象的清理工作。

查看原文>> (http://blog.csdn.net/zyx196/article/details/50816976)

1

看过本文的人也看了：

- iOS知识结构图 (http://lib.csdn.net/base/ios/structure)
- 深入理解Objective-C：Category (原文... (http://lib.csdn.net/article/ios/43339)
- objective-c---分类(category)、类的深... (http://lib.csdn.net/article/ios/43342)
- Objective-C相关Category的收集 (http://lib.csdn.net/article/ios/43349)
- Objective-C——Category、Extension、Pr... (http://lib.csdn.net/article/ios/53045)
- Objective-C 【Category-非正式协议-延... (http://lib.csdn.net/article/ios/43345)

发表评论

输入评论内容

发表

2个评论

 (http://my.csdn.net/newjcj)

newjcj (http://my.csdn.net/newjcj)

1

2016-11-03 15:45:27 回复

 (http://my.csdn.net/laowantongliulizhi)

laowantongliulizhi (http://my.csdn.net/laowantongliulizhi)

不错的内容，挺好的

2016-11-01 15:29:32 回复

论坛反馈 (http://bbs.csdn.net/forums/Service)

网站客服 杂志客服 (http://wpa.qq.com/msgrd?v=3&uin=2251809102&site=qq&menu=yes)

微博客服 (http://e.weibo.com/csdnsupport/profile) webmaster@csdn.net (mailto:webmaster@csdn.net) 400-600-2320 |